Akka Scala Documentation

Release 2.4.14

Lightbend Inc

Nov 21, 2016

1 Introduction **1**
1.1 What is Akka? . . . 1
1.2 Why Akka? . . . 2
1.3 Getting Started . . . 3
1.4 The Obligatory Hello World . . . 7
1.5 Use-case and Deployment Scenarios . . . 7
1.6 Examples of use-cases for Akka . . . 8

2 General **10**
2.1 Terminology, Concepts . . . 10
2.2 Actor Systems . . . 12
2.3 What is an Actor? . . . 14
2.4 Supervision and Monitoring . . . 16
2.5 Actor References, Paths and Addresses . . . 21
2.6 Location Transparency . . . 27
2.7 Akka and the Java Memory Model . . . 28
2.8 Message Delivery Reliability . . . 30
2.9 Configuration . . . 35

3 Actors **101**
3.1 Actors . . . 101
3.2 Akka Typed . . . 121
3.3 Fault Tolerance . . . 126
3.4 Dispatchers . . . 137
3.5 Mailboxes . . . 141
3.6 Routing . . . 149
3.7 FSM . . . 166
3.8 Persistence . . . 175
3.9 Persistence - Schema Evolution . . . 206
3.10 Persistence Query . . . 218
3.11 Persistence Query for LevelDB . . . 227
3.12 Testing Actor Systems . . . 230
3.13 Actor DSL . . . 247
3.14 Typed Actors . . . 249

4 Futures and Agents **256**
4.1 Futures . . . 256
4.2 Agents . . . 263

5 Networking **266**
5.1 Cluster Specification . . . 266
5.2 Cluster Usage . . . 271
5.3 Cluster Singleton . . . 290
5.4 Distributed Publish Subscribe in Cluster . . . 293
5.5 Cluster Client . . . 297

5.6 Cluster Sharding . . . 303
5.7 Cluster Metrics Extension . . . 312
5.8 Distributed Data . . . 319
5.9 Remoting . . . 333
5.10 Remoting (codename Artery) . . . 344
5.11 Serialization . . . 358
5.12 I/O . . . 363
5.13 Using TCP . . . 365
5.14 Using UDP . . . 374
5.15 Camel . . . 376

6 Utilities **387**
6.1 Event Bus . . . 387
6.2 Logging . . . 393
6.3 Scheduler . . . 401
6.4 Duration . . . 405
6.5 Circuit Breaker . . . 406
6.6 Akka Extensions . . . 410

7 Streams **414**
7.1 Introduction . . . 414
7.2 Quick Start Guide . . . 415
7.3 Reactive Tweets . . . 417
7.4 Design Principles behind Akka Streams . . . 421
7.5 Basics and working with Flows . . . 424
7.6 Working with Graphs . . . 431
7.7 Modularity, Composition and Hierarchy . . . 444
7.8 Buffers and working with rate . . . 454
7.9 Dynamic stream handling . . . 458
7.10 Custom stream processing . . . 462
7.11 Integration . . . 480
7.12 Error Handling . . . 492
7.13 Working with streaming IO . . . 494
7.14 Pipelining and Parallelism . . . 497
7.15 Testing streams . . . 500
7.16 Overview of built-in stages and their semantics . . . 503
7.17 Streams Cookbook . . . 527
7.18 Configuration . . . 539
7.19 Migration Guide 1.0 to 2.x . . . 541
7.20 Migration Guide 2.0.x to 2.4.x . . . 541

8 Akka HTTP Documentation (Scala) moved! **545**

9 HowTo: Common Patterns **546**
9.1 Throttling Messages . . . 546
9.2 Balancing Workload Across Nodes . . . 546
9.3 Work Pulling Pattern to throttle and distribute work, and prevent mailbox overflow . . . 546
9.4 Ordered Termination . . . 546
9.5 Akka AMQP Proxies . . . 547
9.6 Shutdown Patterns in Akka 2 . . . 547
9.7 Distributed (in-memory) graph processing with Akka . . . 547
9.8 Case Study: An Auto-Updating Cache Using Actors . . . 547
9.9 Discovering message flows in actor systems with the Spider Pattern . . . 547
9.10 Scheduling Periodic Messages . . . 548

10 Experimental Modules **549**
10.1 Multi Node Testing . . . 549
10.2 Actors (Java with Lambda Support) . . . 554
10.3 FSM (Java with Lambda Support) . . . 574

10.4 External Contributions . . . 584

11 Information for Akka Developers **607**
11.1 Building Akka . . . 607
11.2 Multi JVM Testing . . . 609
11.3 I/O Layer Design . . . 612
11.4 Developer Guidelines . . . 614
11.5 Documentation Guidelines . . . 615

12 Project Information **618**
12.1 Migration Guides . . . 618
12.2 Issue Tracking . . . 636
12.3 Licenses . . . 637
12.4 Sponsors . . . 637
12.5 Project . . . 638

13 Additional Information **640**
13.1 Binary Compatibility Rules . . . 640
13.2 Frequently Asked Questions . . . 642
13.3 Books . . . 645
13.4 Videos . . . 645
13.5 Akka in OSGi . . . 645

CHAPTER

ONE

INTRODUCTION

1.1 What is Akka?

«resilient elastic distributed real-time transaction processing»

We believe that writing correct distributed, concurrent, fault-tolerant and scalable applications is too hard. Most of the time it's because we are using the wrong tools and the wrong level of abstraction. Akka is here to change that. Using the Actor Model we raise the abstraction level and provide a better platform to build scalable, resilient and responsive applications—see the Reactive Manifesto for more details. For fault-tolerance we adopt the "let it crash" model which the telecom industry has used with great success to build applications that self-heal and systems that never stop. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

Akka is Open Source and available under the Apache 2 License.

Download from http://akka.io/downloads.

Please note that all code samples compile, so if you want direct access to the sources, have a look over at the Akka Docs subproject on github: for Java and Scala.

1.1.1 Akka implements a unique hybrid

Actors

Actors give you:

- Simple and high-level abstractions for distribution, concurrency and parallelism.
- Asynchronous, non-blocking and highly performant message-driven programming model.
- Very lightweight event-driven processes (several million actors per GB of heap memory).

See the chapter for *Scala* or Java.

Fault Tolerance

- Supervisor hierarchies with "let-it-crash" semantics.
- Actor systems can span over multiple JVMs to provide truly fault-tolerant systems.
- Excellent for writing highly fault-tolerant systems that self-heal and never stop.

See *Fault Tolerance (Scala)* and Fault Tolerance (Java).

Location Transparency

Everything in Akka is designed to work in a distributed environment: all interactions of actors use pure message passing and everything is asynchronous.

For an overview of the cluster support see the Java and *Scala* documentation chapters.

Persistence

State changes experienced by an actor can optionally be persisted and replayed when the actor is started or restarted. This allows actors to recover their state, even after JVM crashes or when being migrated to another node.

You can find more details in the respective chapter for Java or *Scala*.

1.1.2 Scala and Java APIs

Akka has both a *Scala Documentation* and a java-api.

1.1.3 Akka can be used in different ways

Akka is a toolkit, not a framework: you integrate it into your build like any other library without having to follow a particular source code layout. When expressing your systems as collaborating Actors you may feel pushed more towards proper encapsulation of internal state, you may find that there is a natural separation between business logic and inter-component communication.

Akka applications are typically deployed as follows:

- as a library: used as a regular JAR on the classpath or in a web app.
- packaged with sbt-native-packager.
- packaged and deployed using Lightbend ConductR.

1.1.4 Commercial Support

Akka is available from Lightbend Inc. under a commercial license which includes development or production support, read more here.

1.2 Why Akka?

1.2.1 What features can the Akka platform offer, over the competition?

Akka provides scalable real-time transaction processing.

Akka is an unified runtime and programming model for:

- Scale up (Concurrency)
- Scale out (Remoting)
- Fault tolerance

One thing to learn and admin, with high cohesion and coherent semantics.

Akka is a very scalable piece of software, not only in the context of performance but also in the size of applications it is useful for. The core of Akka, akka-actor, is very small and easily dropped into an existing project where you need asynchronicity and lockless concurrency without hassle.

You can choose to include only the parts of Akka you need in your application. With CPUs growing more and more cores every cycle, Akka is the alternative that provides outstanding performance even if you're only running it on one machine. Akka also supplies a wide array of concurrency-paradigms, allowing users to choose the right tool for the job.

1.2.2 What's a good use-case for Akka?

We see Akka being adopted by many large organizations in a big range of industries:

- Investment and Merchant Banking
- Retail
- Social Media
- Simulation
- Gaming and Betting
- Automobile and Traffic Systems
- Health Care
- Data Analytics

and much more. Any system with the need for high-throughput and low latency is a good candidate for using Akka.

Actors let you manage service failures (Supervisors), load management (back-off strategies, timeouts and processing-isolation), as well as both horizontal and vertical scalability (add more cores and/or add more machines).

Here's what some of the Akka users have to say about how they are using Akka: http://stackoverflow.com/questions/4493001/good-use-case-for-akka

All this in the ApacheV2-licensed open source project.

1.3 Getting Started

1.3.1 Prerequisites

Akka requires that you have Java 8 or later installed on your machine.

Lightbend Inc. provides a commercial build of Akka and related projects such as Scala or Play as part of the Lightbend Reactive Platform which is made available for Java 6 in case your project can not upgrade to Java 8 just yet. It also includes additional commercial features or libraries.

1.3.2 Getting Started Guides and Template Projects

The best way to start learning Akka is to download Lightbend Activator and try out one of Akka Template Projects.

1.3.3 Download

There are several ways to download Akka. You can download it as part of the Lightbend Platform (as described above). You can download the full distribution, which includes all modules. Or you can use a build tool like Maven or SBT to download dependencies from the Akka Maven repository.

1.3.4 Modules

Akka is very modular and consists of several JARs containing different features.

- `akka-actor` – Classic Actors, Typed Actors, IO Actor etc.
- `akka-agent` – Agents, integrated with Scala STM
- `akka-camel` – Apache Camel integration
- `akka-cluster` – Cluster membership management, elastic routers.
- `akka-osgi` – Utilities for using Akka in OSGi containers
- `akka-osgi-aries` – Aries blueprint for provisioning actor systems
- `akka-remote` – Remote Actors
- `akka-slf4j` – SLF4J Logger (event bus listener)
- `akka-stream` – Reactive stream processing
- `akka-testkit` – Toolkit for testing Actor systems

In addition to these stable modules there are several which are on their way into the stable core but are still marked "experimental" at this point. This does not mean that they do not function as intended, it primarily means that their API has not yet solidified enough in order to be considered frozen. You can help accelerating this process by giving feedback on these modules on our mailing list.

- `akka-contrib` – an assortment of contributions which may or may not be moved into core modules, see *External Contributions* for more details.

The filename of the actual JAR is for example `akka-actor_2.11-2.4.14.jar` (and analog for the other modules).

How to see the JARs dependencies of each Akka module is described in the *Dependencies* section.

1.3.5 Using a release distribution

Download the release you need from http://akka.io/downloads and unzip it.

1.3.6 Using a snapshot version

The Akka nightly snapshots are published to http://repo.akka.io/snapshots/ and are versioned with both `SNAPSHOT` and timestamps. You can choose a timestamped version to work with and can decide when to update to a newer version.

> **Warning:** The use of Akka SNAPSHOTs, nightlies and milestone releases is discouraged unless you know what you are doing.

1.3.7 Using a build tool

Akka can be used with build tools that support Maven repositories.

1.3.8 Maven repositories

For Akka version 2.1-M2 and onwards:

Maven Central

For previous Akka versions:

Akka Repo

1.3.9 Using Akka with Maven

The simplest way to get started with Akka and Maven is to check out the Lightbend Activator tutorial named Akka Main in Java.

Since Akka is published to Maven Central (for versions since 2.1-M2), it is enough to add the Akka dependencies to the POM. For example, here is the dependency for akka-actor:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

For snapshot versions, the snapshot repository needs to be added as well:

```
<repositories>
  <repository>
    <id>akka-snapshots</id>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    <url>http://repo.akka.io/snapshots/</url>
  </repository>
</repositories>
```

Note: for snapshot versions both `SNAPSHOT` and timestamped versions are published.

1.3.10 Using Akka with SBT

The simplest way to get started with Akka and SBT is to use Lightbend Activator with one of the SBT templates.

Summary of the essential parts for using Akka with SBT:

SBT installation instructions on http://www.scala-sbt.org/release/tutorial/Setup.html

`build.sbt` file:

```
name := "My Project"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies +=
  "com.typesafe.akka" %% "akka-actor" % "2.4.14"
```

Note: the libraryDependencies setting above is specific to SBT v0.12.x and higher. If you are using an older version of SBT, the libraryDependencies should look like this:

```
libraryDependencies +=
  "com.typesafe.akka" % "akka-actor_2.11" % "2.4.14"
```

For snapshot versions, the snapshot repository needs to be added as well:

```
resolvers += "Akka Snapshot Repository" at "http://repo.akka.io/snapshots/"
```

1.3.11 Using Akka with Gradle

Requires at least Gradle 1.4 Uses the Scala plugin

```
apply plugin: 'scala'

repositories {
  mavenCentral()
}

dependencies {
  compile 'org.scala-lang:scala-library:2.11.8'
}

tasks.withType(ScalaCompile) {
  scalaCompileOptions.useAnt = false
}

dependencies {
  compile group: 'com.typesafe.akka', name: 'akka-actor_2.11', version: '2.4.14'
  compile group: 'org.scala-lang', name: 'scala-library', version: '2.11.8'
}
```

For snapshot versions, the snapshot repository needs to be added as well:

```
repositories {
  mavenCentral()
  maven {
    url "http://repo.akka.io/snapshots/"
  }
}
```

1.3.12 Using Akka with Eclipse

Setup SBT project and then use sbteclipse to generate an Eclipse project.

1.3.13 Using Akka with IntelliJ IDEA

Setup SBT project and then use sbt-idea to generate an IntelliJ IDEA project.

1.3.14 Using Akka with NetBeans

Setup SBT project and then use nbsbt to generate a NetBeans project.

You should also use nbscala for general scala support in the IDE.

1.3.15 Do not use -optimize Scala compiler flag

> **Warning:** Akka has not been compiled or tested with -optimize Scala compiler flag. Strange behavior has been reported by users that have tried it.

1.3.16 Build from sources

Akka uses Git and is hosted at Github.

- Akka: clone the Akka repository from https://github.com/akka/akka

Continue reading the page on *Building Akka*

1.3.17 Need help?

If you have questions you can get help on the Akka Mailing List.

You can also ask for commercial support.

Thanks for being a part of the Akka community.

1.4 The Obligatory Hello World

The actor based version of the tough problem of printing a well-known greeting to the console is introduced in a Lightbend Activator tutorial named Akka Main in Scala.

The tutorial illustrates the generic launcher class `akka.Main` which expects only one command line argument: the class name of the application's main actor. This main method will then create the infrastructure needed for running the actors, start the given main actor and arrange for the whole application to shut down once the main actor terminates.

There is also another Lightbend Activator tutorial in the same problem domain that is named Hello Akka!. It describes the basics of Akka in more depth.

1.5 Use-case and Deployment Scenarios

1.5.1 How can I use and deploy Akka?

Akka can be used in different ways:

- As a library: used as a regular JAR on the classpath and/or in a web app, to be put into `WEB-INF/lib`
- Package with sbt-native-packager
- Package and deploy using Lightbend ConductR.

1.5.2 Native Packager

sbt-native-packager is a tool for creating distributions of any type of application, including an Akka applications.

Define sbt version in `project/build.properties` file:

```
sbt.version=0.13.7
```

Add sbt-native-packager in `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.0.0-RC1")
```

Use the package settings and optionally specify the mainClass in `build.sbt` file:

```
import NativePackagerHelper._

name := "akka-sample-main-scala"

version := "2.4.14"

scalaVersion := "2.11.7"
```

```
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.4.14"
)

enablePlugins(JavaServerAppPackaging)

mainClass in Compile := Some("sample.hello.Main")

mappings in Universal ++= {
  // optional example illustrating how to copy additional directory
  directory("scripts") ++
  // copy configuration files to config directory
  contentOf("src/main/resources").toMap.mapValues("config/" + _)
}

// add 'config' directory first in the classpath of the start script,
// an alternative is to set the config file locations via CLI parameters
// when starting the application
scriptClasspath := Seq("../config/") ++ scriptClasspath.value

licenses := Seq(("CC0", url("http://creativecommons.org/publicdomain/zero/1.0")))
```

Note: Use the `JavaServerAppPackaging`. Don't use the deprecated `AkkaAppPackaging` (previously named `packageArchetype.akka_application`), since it doesn't have the same flexibility and quality as the `JavaServerAppPackaging`.

Use sbt task `dist` package the application.

To start the application (on a unix-based system):

```
cd target/universal/
unzip akka-sample-main-scala-2.4.14.zip
chmod u+x akka-sample-main-scala-2.4.14/bin/akka-sample-main-scala
akka-sample-main-scala-2.4.14/bin/akka-sample-main-scala sample.hello.Main
```

Use `Ctrl-C` to interrupt and exit the application.

On a Windows machine you can also use the `bin\akka-sample-main-scala.bat` script.

1.5.3 In a Docker container

You can use both Akka remoting and Akka Cluster inside of Docker containers. But note that you will need to take special care with the network configuration when using Docker, described here: *Akka behind NAT or in a Docker container*

For an example of how to set up a project using Akka Cluster and Docker take a look at the "akka-docker-cluster" activator template.

1.6 Examples of use-cases for Akka

We see Akka being adopted by many large organizations in a big range of industries all from investment and merchant banking, retail and social media, simulation, gaming and betting, automobile and traffic systems, health care, data analytics and much more. Any system that have the need for high-throughput and low latency is a good candidate for using Akka.

There is a great discussion on use-cases for Akka with some good write-ups by production users here

1.6.1 Here are some of the areas where Akka is being deployed into production

Transaction processing (Online Gaming, Finance/Banking, Trading, Statistics, Betting, Social Media, Telecom)

Scale up, scale out, fault-tolerance / HA

Service backend (any industry, any app)

Service REST, SOAP, Cometd, WebSockets etc Act as message hub / integration layer Scale up, scale out, fault-tolerance / HA

Concurrency/parallelism (any app)

Correct Simple to work with and understand Just add the jars to your existing JVM project (use Scala, Java, Groovy or JRuby)

Simulation

Master/Worker, Compute Grid, MapReduce etc.

Batch processing (any industry)

Camel integration to hook up with batch data sources Actors divide and conquer the batch workloads

Communications Hub (Telecom, Web media, Mobile media)

Scale up, scale out, fault-tolerance / HA

Gaming and Betting (MOM, online gaming, betting)

Scale up, scale out, fault-tolerance / HA

Business Intelligence/Data Mining/general purpose crunching

Scale up, scale out, fault-tolerance / HA

Complex Event Stream Processing

Scale up, scale out, fault-tolerance / HA

CHAPTER

TWO

GENERAL

2.1 Terminology, Concepts

In this chapter we attempt to establish a common terminology to define a solid ground for communicating about concurrent, distributed systems which Akka targets. Please note that, for many of these terms, there is no single agreed definition. We simply seek to give working definitions that will be used in the scope of the Akka documentation.

2.1.1 Concurrency vs. Parallelism

Concurrency and parallelism are related concepts, but there are small differences. *Concurrency* means that two or more tasks are making progress even though they might not be executing simultaneously. This can for example be realized with time slicing where parts of tasks are executed sequentially and mixed with parts of other tasks. *Parallelism* on the other hand arise when the execution can be truly simultaneous.

2.1.2 Asynchronous vs. Synchronous

A method call is considered *synchronous* if the caller cannot make progress until the method returns a value or throws an exception. On the other hand, an *asynchronous* call allows the caller to progress after a finite number of steps, and the completion of the method may be signalled via some additional mechanism (it might be a registered callback, a Future, or a message).

A synchronous API may use blocking to implement synchrony, but this is not a necessity. A very CPU intensive task might give a similar behavior as blocking. In general, it is preferred to use asynchronous APIs, as they guarantee that the system is able to progress. Actors are asynchronous by nature: an actor can progress after a message send without waiting for the actual delivery to happen.

2.1.3 Non-blocking vs. Blocking

We talk about *blocking* if the delay of one thread can indefinitely delay some of the other threads. A good example is a resource which can be used exclusively by one thread using mutual exclusion. If a thread holds on to the resource indefinitely (for example accidentally running an infinite loop) other threads waiting on the resource can not progress. In contrast, *non-blocking* means that no thread is able to indefinitely delay others.

Non-blocking operations are preferred to blocking ones, as the overall progress of the system is not trivially guaranteed when it contains blocking operations.

2.1.4 Deadlock vs. Starvation vs. Live-lock

Deadlock arises when several participants are waiting on each other to reach a specific state to be able to progress. As none of them can progress without some other participant to reach a certain state (a "Catch-22" problem) all affected subsystems stall. Deadlock is closely related to *blocking*, as it is necessary that a participant thread be able to delay the progression of other threads indefinitely.

In the case of *deadlock*, no participants can make progress, while in contrast *Starvation* happens, when there are participants that can make progress, but there might be one or more that cannot. Typical scenario is the case of a naive scheduling algorithm that always selects high-priority tasks over low-priority ones. If the number of incoming high-priority tasks is constantly high enough, no low-priority ones will be ever finished.

Livelock is similar to *deadlock* as none of the participants make progress. The difference though is that instead of being frozen in a state of waiting for others to progress, the participants continuously change their state. An example scenario when two participants have two identical resources available. They each try to get the resource, but they also check if the other needs the resource, too. If the resource is requested by the other participant, they try to get the other instance of the resource. In the unfortunate case it might happen that the two participants "bounce" between the two resources, never acquiring it, but always yielding to the other.

2.1.5 Race Condition

We call it a *Race condition* when an assumption about the ordering of a set of events might be violated by external non-deterministic effects. Race conditions often arise when multiple threads have a shared mutable state, and the operations of thread on the state might be interleaved causing unexpected behavior. While this is a common case, shared state is not necessary to have race conditions. One example could be a client sending unordered packets (e.g UDP datagrams) `P1`, `P2` to a server. As the packets might potentially travel via different network routes, it is possible that the server receives `P2` first and `P1` afterwards. If the messages contain no information about their sending order it is impossible to determine by the server that they were sent in a different order. Depending on the meaning of the packets this can cause race conditions.

Note: The only guarantee that Akka provides about messages sent between a given pair of actors is that their order is always preserved. see *Message Delivery Reliability*

2.1.6 Non-blocking Guarantees (Progress Conditions)

As discussed in the previous sections blocking is undesirable for several reasons, including the dangers of deadlocks and reduced throughput in the system. In the following sections we discuss various non-blocking properties with different strength.

Wait-freedom

A method is *wait-free* if every call is guaranteed to finish in a finite number of steps. If a method is *bounded wait-free* then the number of steps has a finite upper bound.

From this definition it follows that wait-free methods are never blocking, therefore deadlock can not happen. Additionally, as each participant can progress after a finite number of steps (when the call finishes), wait-free methods are free of starvation.

Lock-freedom

Lock-freedom is a weaker property than *wait-freedom*. In the case of lock-free calls, infinitely often some method finishes in a finite number of steps. This definition implies that no deadlock is possible for lock-free calls. On the other hand, the guarantee that *some call finishes* in a finite number of steps is not enough to guarantee that *all of them eventually finish*. In other words, lock-freedom is not enough to guarantee the lack of starvation.

Obstruction-freedom

Obstruction-freedom is the weakest non-blocking guarantee discussed here. A method is called *obstruction-free* if there is a point in time after which it executes in isolation (other threads make no steps, e.g.: become suspended), it finishes in a bounded number of steps. All lock-free objects are obstruction-free, but the opposite is generally not true.

Optimistic concurrency control (OCC) methods are usually obstruction-free. The OCC approach is that every participant tries to execute its operation on the shared object, but if a participant detects conflicts from others, it rolls back the modifications, and tries again according to some schedule. If there is a point in time, where one of the participants is the only one trying, the operation will succeed.

2.1.7 Recommended literature

- The Art of Multiprocessor Programming, M. Herlihy and N Shavit, 2008. ISBN 978-0123705914
- Java Concurrency in Practice, B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, 2006. ISBN 978-0321349606

2.2 Actor Systems

Actors are objects which encapsulate state and behavior, they communicate exclusively by exchanging messages which are placed into the recipient's mailbox. In a sense, actors are the most stringent form of object-oriented programming, but it serves better to view them as persons: while modeling a solution with actors, envision a group of people and assign sub-tasks to them, arrange their functions into an organizational structure and think about how to escalate failure (all with the benefit of not actually dealing with people, which means that we need not concern ourselves with their emotional state or moral issues). The result can then serve as a mental scaffolding for building the software implementation.

Note: An ActorSystem is a heavyweight structure that will allocate 1…N Threads, so create one per logical application.

2.2.1 Hierarchical Structure

Like in an economic organization, actors naturally form hierarchies. One actor, which is to oversee a certain function in the program might want to split up its task into smaller, more manageable pieces. For this purpose it starts child actors which it supervises. While the details of supervision are explained *here*, we shall concentrate on the underlying concepts in this section. The only prerequisite is to know that each actor has exactly one supervisor, which is the actor that created it.

The quintessential feature of actor systems is that tasks are split up and delegated until they become small enough to be handled in one piece. In doing so, not only is the task itself clearly structured, but the resulting actors can be reasoned about in terms of which messages they should process, how they should react normally and how failure should be handled. If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help. The recursive structure then allows to handle failure at the right level.

Compare this to layered software design which easily devolves into defensive programming with the aim of not leaking any failure out: if the problem is communicated to the right person, a better solution can be found than if trying to keep everything "under the carpet".

Now, the difficulty in designing such a system is how to decide who should supervise what. There is of course no single best solution, but there are a few guidelines which might be helpful:

- If one actor manages the work another actor is doing, e.g. by passing on sub-tasks, then the manager should supervise the child. The reason is that the manager knows which kind of failures are expected and how to handle them.
- If one actor carries very important data (i.e. its state shall not be lost if avoidable), this actor should source out any possibly dangerous sub-tasks to children it supervises and handle failures of these children as appropriate. Depending on the nature of the requests, it may be best to create a new child for each request, which simplifies state management for collecting the replies. This is known as the "Error Kernel Pattern" from Erlang.

- If one actor depends on another actor for carrying out its duty, it should watch that other actor's liveness and act upon receiving a termination notice. This is different from supervision, as the watching party has no influence on the supervisor strategy, and it should be noted that a functional dependency alone is not a criterion for deciding where to place a certain child actor in the hierarchy.

There are of course always exceptions to these rules, but no matter whether you follow the rules or break them, you should always have a reason.

2.2.2 Configuration Container

The actor system as a collaborating ensemble of actors is the natural unit for managing shared facilities like scheduling services, configuration, logging, etc. Several actor systems with different configuration may co-exist within the same JVM without problems, there is no global shared state within Akka itself. Couple this with the transparent communication between actor systems—within one node or across a network connection—to see that actor systems themselves can be used as building blocks in a functional hierarchy.

2.2.3 Actor Best Practices

1. Actors should be like nice co-workers: do their job efficiently without bothering everyone else needlessly and avoid hogging resources. Translated to programming this means to process events and generate responses (or more requests) in an event-driven manner. Actors should not block (i.e. passively wait while occupying a Thread) on some external entity—which might be a lock, a network socket, etc.—unless it is unavoidable; in the latter case see below.
2. Do not pass mutable objects between actors. In order to ensure that, prefer immutable messages. If the encapsulation of actors is broken by exposing their mutable state to the outside, you are back in normal Java concurrency land with all the drawbacks.
3. Actors are made to be containers for behavior and state, embracing this means to not routinely send behavior within messages (which may be tempting using Scala closures). One of the risks is to accidentally share mutable state between actors, and this violation of the actor model unfortunately breaks all the properties which make programming in actors such a nice experience.
4. Top-level actors are the innermost part of your Error Kernel, so create them sparingly and prefer truly hierarchical systems. This has benefits with respect to fault-handling (both considering the granularity of configuration and the performance) and it also reduces the strain on the guardian actor, which is a single point of contention if over-used.

2.2.4 Blocking Needs Careful Management

In some cases it is unavoidable to do blocking operations, i.e. to put a thread to sleep for an indeterminate time, waiting for an external event to occur. Examples are legacy RDBMS drivers or messaging APIs, and the underlying reason is typically that (network) I/O occurs under the covers. When facing this, you may be tempted to just wrap the blocking call inside a `Future` and work with that instead, but this strategy is too simple: you are quite likely to find bottlenecks or run out of memory or threads when the application runs under increased load.

The non-exhaustive list of adequate solutions to the "blocking problem" includes the following suggestions:

- Do the blocking call within an actor (or a set of actors managed by a router [Java, *Scala*]), making sure to configure a thread pool which is either dedicated for this purpose or sufficiently sized.
- Do the blocking call within a `Future`, ensuring an upper bound on the number of such calls at any point in time (submitting an unbounded number of tasks of this nature will exhaust your memory or thread limits).
- Do the blocking call within a `Future`, providing a thread pool with an upper limit on the number of threads which is appropriate for the hardware on which the application runs.
- Dedicate a single thread to manage a set of blocking resources (e.g. a NIO selector driving multiple channels) and dispatch events as they occur as actor messages.

The first possibility is especially well-suited for resources which are single-threaded in nature, like database handles which traditionally can only execute one outstanding query at a time and use internal synchronization to ensure this. A common pattern is to create a router for N actors, each of which wraps a single DB connection and handles queries as sent to the router. The number N must then be tuned for maximum throughput, which will vary depending on which DBMS is deployed on what hardware.

Note: Configuring thread pools is a task best delegated to Akka, simply configure in the `application.conf` and instantiate through an `ActorSystem` [Java, *Scala*]

2.2.5 What you should not concern yourself with

An actor system manages the resources it is configured to use in order to run the actors which it contains. There may be millions of actors within one such system, after all the mantra is to view them as abundant and they weigh in at an overhead of only roughly 300 bytes per instance. Naturally, the exact order in which messages are processed in large systems is not controllable by the application author, but this is also not intended. Take a step back and relax while Akka does the heavy lifting under the hood.

2.3 What is an Actor?

The previous section about *Actor Systems* explained how actors form hierarchies and are the smallest unit when building an application. This section looks at one such actor in isolation, explaining the concepts you encounter while implementing it. For a more in depth reference with all the details please refer to *Actors (Scala)* and Untyped Actors (Java).

An actor is a container for *State*, *Behavior*, a *Mailbox*, *Child Actors* and a *Supervisor Strategy*. All of this is encapsulated behind an *Actor Reference*. One noteworthy aspect is that actors have an explicit lifecycle, they are not automatically destroyed when no longer referenced; after having created one, it is your responsibility to make sure that it will eventually be terminated as well—which also gives you control over how resources are released *When an Actor Terminates*.

2.3.1 Actor Reference

As detailed below, an actor object needs to be shielded from the outside in order to benefit from the actor model. Therefore, actors are represented to the outside using actor references, which are objects that can be passed around freely and without restriction. This split into inner and outer object enables transparency for all the desired operations: restarting an actor without needing to update references elsewhere, placing the actual actor object on remote hosts, sending messages to actors in completely different applications. But the most important aspect is that it is not possible to look inside an actor and get hold of its state from the outside, unless the actor unwisely publishes this information itself.

2.3.2 State

Actor objects will typically contain some variables which reflect possible states the actor may be in. This can be an explicit state machine (e.g. using the *FSM* module), or it could be a counter, set of listeners, pending requests, etc. These data are what make an actor valuable, and they must be protected from corruption by other actors. The good news is that Akka actors conceptually each have their own light-weight thread, which is completely shielded from the rest of the system. This means that instead of having to synchronize access using locks you can just write your actor code without worrying about concurrency at all.

Behind the scenes Akka will run sets of actors on sets of real threads, where typically many actors share one thread, and subsequent invocations of one actor may end up being processed on different threads. Akka ensures that this implementation detail does not affect the single-threadedness of handling the actor's state.

Because the internal state is vital to an actor's operations, having inconsistent state is fatal. Thus, when the actor fails and is restarted by its supervisor, the state will be created from scratch, like upon first creating the actor. This is to enable the ability of self-healing of the system.

Optionally, an actor's state can be automatically recovered to the state before a restart by persisting received messages and replaying them after restart (see *Persistence*).

2.3.3 Behavior

Every time a message is processed, it is matched against the current behavior of the actor. Behavior means a function which defines the actions to be taken in reaction to the message at that point in time, say forward a request if the client is authorized, deny it otherwise. This behavior may change over time, e.g. because different clients obtain authorization over time, or because the actor may go into an "out-of-service" mode and later come back. These changes are achieved by either encoding them in state variables which are read from the behavior logic, or the function itself may be swapped out at runtime, see the `become` and `unbecome` operations. However, the initial behavior defined during construction of the actor object is special in the sense that a restart of the actor will reset its behavior to this initial one.

2.3.4 Mailbox

An actor's purpose is the processing of messages, and these messages were sent to the actor from other actors (or from outside the actor system). The piece which connects sender and receiver is the actor's mailbox: each actor has exactly one mailbox to which all senders enqueue their messages. Enqueuing happens in the time-order of send operations, which means that messages sent from different actors may not have a defined order at runtime due to the apparent randomness of distributing actors across threads. Sending multiple messages to the same target from the same actor, on the other hand, will enqueue them in the same order.

There are different mailbox implementations to choose from, the default being a FIFO: the order of the messages processed by the actor matches the order in which they were enqueued. This is usually a good default, but applications may need to prioritize some messages over others. In this case, a priority mailbox will enqueue not always at the end but at a position as given by the message priority, which might even be at the front. While using such a queue, the order of messages processed will naturally be defined by the queue's algorithm and in general not be FIFO.

An important feature in which Akka differs from some other actor model implementations is that the current behavior must always handle the next dequeued message, there is no scanning the mailbox for the next matching one. Failure to handle a message will typically be treated as a failure, unless this behavior is overridden.

2.3.5 Child Actors

Each actor is potentially a supervisor: if it creates children for delegating sub-tasks, it will automatically supervise them. The list of children is maintained within the actor's context and the actor has access to it. Modifications to the list are done by creating (`context.actorOf(...)`) or stopping (`context.stop(child)`) children and these actions are reflected immediately. The actual creation and termination actions happen behind the scenes in an asynchronous way, so they do not "block" their supervisor.

2.3.6 Supervisor Strategy

The final piece of an actor is its strategy for handling faults of its children. Fault handling is then done transparently by Akka, applying one of the strategies described in *Supervision and Monitoring* for each incoming failure. As this strategy is fundamental to how an actor system is structured, it cannot be changed once an actor has been created.

Considering that there is only one such strategy for each actor, this means that if different strategies apply to the various children of an actor, the children should be grouped beneath intermediate supervisors with matching strategies, preferring once more the structuring of actor systems according to the splitting of tasks into sub-tasks.

2.3.7 When an Actor Terminates

Once an actor terminates, i.e. fails in a way which is not handled by a restart, stops itself or is stopped by its supervisor, it will free up its resources, draining all remaining messages from its mailbox into the system's "dead letter mailbox" which will forward them to the EventStream as DeadLetters. The mailbox is then replaced within the actor reference with a system mailbox, redirecting all new messages to the EventStream as DeadLetters. This is done on a best effort basis, though, so do not rely on it in order to construct "guaranteed delivery".

The reason for not just silently dumping the messages was inspired by our tests: we register the TestEventListener on the event bus to which the dead letters are forwarded, and that will log a warning for every dead letter received—this has been very helpful for deciphering test failures more quickly. It is conceivable that this feature may also be of use for other purposes.

2.4 Supervision and Monitoring

This chapter outlines the concept behind supervision, the primitives offered and their semantics. For details on how that translates into real code, please refer to the corresponding chapters for Scala and Java APIs.

2.4.1 What Supervision Means

As described in *Actor Systems* supervision describes a dependency relationship between actors: the supervisor delegates tasks to subordinates and therefore must respond to their failures. When a subordinate detects a failure (i.e. throws an exception), it suspends itself and all its subordinates and sends a message to its supervisor, signaling failure. Depending on the nature of the work to be supervised and the nature of the failure, the supervisor has a choice of the following four options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Stop the subordinate permanently
4. Escalate the failure, thereby failing itself

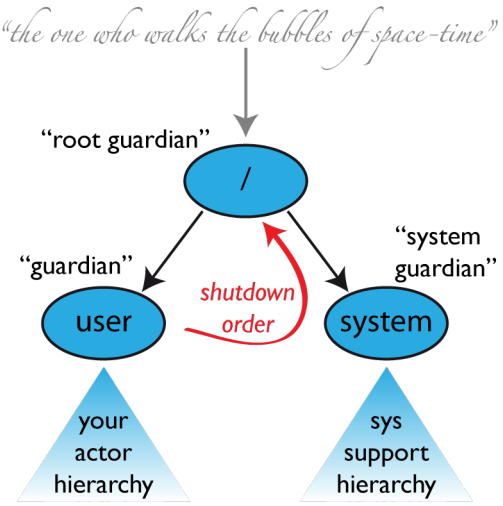
It is important to always view an actor as part of a supervision hierarchy, which explains the existence of the fourth choice (as a supervisor also is subordinate to another supervisor higher up) and has implications on the first three: resuming an actor resumes all its subordinates, restarting an actor entails restarting all its subordinates (but see below for more details), similarly terminating an actor will also terminate all its subordinates. It should be noted that the default behavior of the `preRestart` hook of the `Actor` class is to terminate all its children before restarting, but this hook can be overridden; the recursive restart applies to all children left after this hook has been executed.

Each supervisor is configured with a function translating all possible failure causes (i.e. exceptions) into one of the four choices given above; notably, this function does not take the failed actor's identity as an input. It is quite easy to come up with examples of structures where this might not seem flexible enough, e.g. wishing for different strategies to be applied to different subordinates. At this point it is vital to understand that supervision is about forming a recursive fault handling structure. If you try to do too much at one level, it will become hard to reason about, hence the recommended way in this case is to add a level of supervision.

Akka implements a specific form called "parental supervision". Actors can only be created by other actors—where the top-level actor is provided by the library—and each created actor is supervised by its parent. This restriction makes the formation of actor supervision hierarchies implicit and encourages sound design decisions. It should be noted that this also guarantees that actors cannot be orphaned or attached to supervisors from the outside, which might otherwise catch them unawares. In addition, this yields a natural and clean shutdown procedure for (sub-trees of) actor applications.

Warning: Supervision related parent-child communication happens by special system messages that have their own mailboxes separate from user messages. This implies that supervision related events are not deterministically ordered relative to ordinary messages. In general, the user cannot influence the order of normal messages and failure notifications. For details and example see the *Discussion: Message Ordering* section.

2.4.2 The Top-Level Supervisors



An actor system will during its creation start at least three actors, shown in the image above. For more information about the consequences for actor paths see *Top-Level Scopes for Actor Paths*.

`/user`: The Guardian Actor

The actor which is probably most interacted with is the parent of all user-created actors, the guardian named `"/user"`. Actors created using `system.actorOf()` are children of this actor. This means that when this guardian terminates, all normal actors in the system will be shutdown, too. It also means that this guardian's supervisor strategy determines how the top-level normal actors are supervised. Since Akka 2.1 it is possible to configure this using the setting `akka.actor.guardian-supervisor-strategy`, which takes the fully-qualified class-name of a `SupervisorStrategyConfigurator`. When the guardian escalates a failure, the root guardian's response will be to terminate the guardian, which in effect will shut down the whole actor system.

`/system`: The System Guardian

This special guardian has been introduced in order to achieve an orderly shut-down sequence where logging remains active while all normal actors terminate, even though logging itself is implemented using actors. This is realized by having the system guardian watch the user guardian and initiate its own shut-down upon reception of the `Terminated` message. The top-level system actors are supervised using a strategy which will restart indefinitely upon all types of `Exception` except for `ActorInitializationException` and `ActorKilledException`, which will terminate the child in question. All other throwables are escalated, which will shut down the whole actor system.

`/`: The Root Guardian

The root guardian is the grand-parent of all so-called "top-level" actors and supervises all the special actors mentioned in *Top-Level Scopes for Actor Paths* using the `SupervisorStrategy.stoppingStrategy`, whose purpose is to terminate the child upon any type of `Exception`. All other throwables will be escalated ... but to whom? Since every real actor has a supervisor, the supervisor of the root guardian cannot be a real

actor. And because this means that it is "outside of the bubble", it is called the "bubble-walker". This is a synthetic `ActorRef` which in effect stops its child upon the first sign of trouble and sets the actor system's `isTerminated` status to `true` as soon as the root guardian is fully terminated (all children recursively stopped).

2.4.3 What Restarting Means

When presented with an actor which failed while processing a certain message, causes for the failure fall into three categories:

- Systematic (i.e. programming) error for the specific message received
- (Transient) failure of some external resource used during processing the message
- Corrupt internal state of the actor

Unless the failure is specifically recognizable, the third cause cannot be ruled out, which leads to the conclusion that the internal state needs to be cleared out. If the supervisor decides that its other children or itself is not affected by the corruption—e.g. because of conscious application of the error kernel pattern—it is therefore best to restart the child. This is carried out by creating a new instance of the underlying `Actor` class and replacing the failed instance with the fresh one inside the child's `ActorRef`; the ability to do this is one of the reasons for encapsulating actors within special references. The new actor then resumes processing its mailbox, meaning that the restart is not visible outside of the actor itself with the notable exception that the message during which the failure occurred is not re-processed.

The precise sequence of events during a restart is the following:

1. suspend the actor (which means that it will not process normal messages until resumed), and recursively suspend all children
2. call the old instance's `preRestart` hook (defaults to sending termination requests to all children and calling `postStop`)
3. wait for all children which were requested to terminate (using `context.stop()`) during `preRestart` to actually terminate; this—like all actor operations—is non-blocking, the termination notice from the last killed child will effect the progression to the next step
4. create new actor instance by invoking the originally provided factory again
5. invoke `postRestart` on the new instance (which by default also calls `preStart`)
6. send restart request to all children which were not killed in step 3; restarted children will follow the same process recursively, from step 2
7. resume the actor

2.4.4 What Lifecycle Monitoring Means

Note: Lifecycle Monitoring in Akka is usually referred to as `DeathWatch`

In contrast to the special relationship between parent and child described above, each actor may monitor any other actor. Since actors emerge from creation fully alive and restarts are not visible outside of the affected supervisors, the only state change available for monitoring is the transition from alive to dead. Monitoring is thus used to tie one actor to another so that it may react to the other actor's termination, in contrast to supervision which reacts to failure.

Lifecycle monitoring is implemented using a `Terminated` message to be received by the monitoring actor, where the default behavior is to throw a special `DeathPactException` if not otherwise handled. In order to start listening for `Terminated` messages, invoke `ActorContext.watch(targetActorRef)`. To stop listening, invoke `ActorContext.unwatch(targetActorRef)`. One important property is that the message will be delivered irrespective of the order in which the monitoring request and target's termination occur, i.e. you still get the message even if at the time of registration the target is already dead.

Monitoring is particularly useful if a supervisor cannot simply restart its children and has to terminate them, e.g. in case of errors during actor initialization. In that case it should monitor those children and re-create them or schedule itself to retry this at a later time.

Another common use case is that an actor needs to fail in the absence of an external resource, which may also be one of its own children. If a third party terminates a child by way of the `system.stop(child)` method or sending a `PoisonPill`, the supervisor might well be affected.

Delayed restarts with the BackoffSupervisor pattern

Provided as a built-in pattern the `akka.pattern.BackoffSupervisor` implements the so-called *exponential backoff supervision strategy*, starting a child actor again when it fails, each time with a growing time delay between restarts.

This pattern is useful when the started actor fails [1] because some external resource is not available, and we need to give it some time to start-up again. One of the prime examples when this is useful is when a *PersistentActor* fails (by stopping) with a persistence failure - which indicates that the database may be down or overloaded, in such situations it makes most sense to give it a little bit of time to recover before the peristent actor is started.

The following Scala snippet shows how to create a backoff supervisor which will start the given echo actor after it has stopped because of a failure, in increasing intervals of 3, 6, 12, 24 and finally 30 seconds:

```
val childProps = Props(classOf[EchoActor])

val supervisor = BackoffSupervisor.props(
  Backoff.onStop(
    childProps,
    childName = "myEcho",
    minBackoff = 3.seconds,
    maxBackoff = 30.seconds,
    randomFactor = 0.2 // adds 20% "noise" to vary the intervals slightly
  ))

system.actorOf(supervisor, name = "echoSupervisor")
```

The above is equivalent to this Java code:

```
import scala.concurrent.duration.Duration;
```

```
final Props childProps = Props.create(EchoActor.class);

final Props  supervisorProps = BackoffSupervisor.props(
  Backoff.onStop(
    childProps,
    "myEcho",
    Duration.create(3, TimeUnit.SECONDS),
    Duration.create(30, TimeUnit.SECONDS),
    0.2)); // adds 20% "noise" to vary the intervals slightly

system.actorOf(supervisorProps, "echoSupervisor");
```

Using a `randomFactor` to add a little bit of additional variance to the backoff intervals is highly recommended, in order to avoid multiple actors re-start at the exact same point in time, for example because they were stopped due to a shared resource such as a database going down and re-starting after the same configured interval. By adding additional randomness to the re-start intervals the actors will start in slightly different points in time, thus avoiding large spikes of traffic hitting the recovering shared database or other resource that they all need to contact.

The `akka.pattern.BackoffSupervisor` actor can also be configured to restart the actor after a delay when the actor crashes and the supervision strategy decides that it should restart.

[1] A failure can be indicated in two different ways; by an actor stopping or crashing.

The following Scala snippet shows how to create a backoff supervisor which will start the given echo actor after it has crashed because of some exception, in increasing intervals of 3, 6, 12, 24 and finally 30 seconds:

```
val childProps = Props(classOf[EchoActor])

val supervisor = BackoffSupervisor.props(
  Backoff.onFailure(
    childProps,
    childName = "myEcho",
    minBackoff = 3.seconds,
    maxBackoff = 30.seconds,
    randomFactor = 0.2 // adds 20% "noise" to vary the intervals slightly
  ))

system.actorOf(supervisor, name = "echoSupervisor")
```

The above is equivalent to this Java code:

```
import scala.concurrent.duration.Duration;
```

```
final Props childProps = Props.create(EchoActor.class);

final Props  supervisorProps = BackoffSupervisor.props(
  Backoff.onFailure(
    childProps,
    "myEcho",
    Duration.create(3, TimeUnit.SECONDS),
    Duration.create(30, TimeUnit.SECONDS),
    0.2)); // adds 20% "noise" to vary the intervals slightly

system.actorOf(supervisorProps, "echoSupervisor");
```

The `akka.pattern.BackoffOptions` can be used to customize the behavior of the back-off supervisor actor, below are some examples:

```
val supervisor = BackoffSupervisor.props(
  Backoff.onStop(
  childProps,
  childName = "myEcho",
  minBackoff = 3.seconds,
  maxBackoff = 30.seconds,
  randomFactor = 0.2 // adds 20% "noise" to vary the intervals slightly
).withManualReset // the child must send BackoffSupervisor.Reset to its parent
  .withDefaultStoppingStrategy // Stop at any Exception thrown
)
```

The above code sets up a back-off supervisor that requires the child actor to send a `akka.pattern.BackoffSupervisor.Reset` message to its parent when a message is successfully processed, resetting the back-off. It also uses a default stopping strategy, any exception will cause the child to stop.

```
val supervisor = BackoffSupervisor.props(
  Backoff.onFailure(
    childProps,
    childName = "myEcho",
    minBackoff = 3.seconds,
    maxBackoff = 30.seconds,
    randomFactor = 0.2 // adds 20% "noise" to vary the intervals slightly
  ).withAutoReset(10.seconds) // the child must send BackoffSupervisor.Reset to␣
↪its parent
    .withSupervisorStrategy(
      OneForOneStrategy() {
```

```
        case _: MyException => SupervisorStrategy.Restart
        case _              => SupervisorStrategy.Escalate
      }))
```

The above code sets up a back-off supervisor that restarts the child after back-off if MyException is thrown, any other exception will be escalated. The back-off is automatically reset if the child does not throw any errors within 10 seconds.

2.4.5 One-For-One Strategy vs. All-For-One Strategy

There are two classes of supervision strategies which come with Akka: `OneForOneStrategy` and `AllForOneStrategy`. Both are configured with a mapping from exception type to supervision directive (see *above*) and limits on how often a child is allowed to fail before terminating it. The difference between them is that the former applies the obtained directive only to the failed child, whereas the latter applies it to all siblings as well. Normally, you should use the `OneForOneStrategy`, which also is the default if none is specified explicitly.

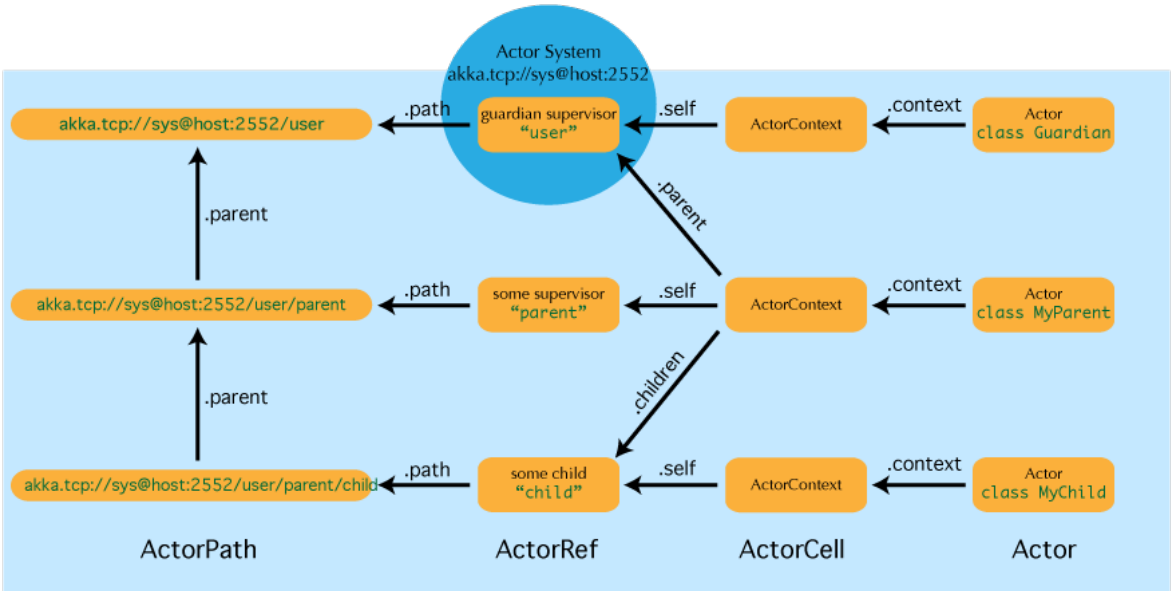
The `AllForOneStrategy` is applicable in cases where the ensemble of children has such tight dependencies among them, that a failure of one child affects the function of the others, i.e. they are inextricably linked. Since a restart does not clear out the mailbox, it often is best to terminate the children upon failure and re-create them explicitly from the supervisor (by watching the children's lifecycle); otherwise you have to make sure that it is no problem for any of the actors to receive a message which was queued before the restart but processed afterwards.

Normally stopping a child (i.e. not in response to a failure) will not automatically terminate the other children in an all-for-one strategy; this can easily be done by watching their lifecycle: if the `Terminated` message is not handled by the supervisor, it will throw a `DeathPactException` which (depending on its supervisor) will restart it, and the default `preRestart` action will terminate all children. Of course this can be handled explicitly as well.

Please note that creating one-off actors from an all-for-one supervisor entails that failures escalated by the temporary actor will affect all the permanent ones. If this is not desired, install an intermediate supervisor; this can very easily be done by declaring a router of size 1 for the worker, see *Routing* or routing-java.

2.5 Actor References, Paths and Addresses

This chapter describes how actors are identified and located within a possibly distributed actor system. It ties into the central idea that *Actor Systems* form intrinsic supervision hierarchies as well as that communication between actors is transparent with respect to their placement across multiple network nodes.



The above image displays the relationship between the most important entities within an actor system, please read on for the details.

2.5.1 What is an Actor Reference?

An actor reference is a subtype of `ActorRef`, whose foremost purpose is to support sending messages to the actor it represents. Each actor has access to its canonical (local) reference through the `self` field; this reference is also included as sender reference by default for all messages sent to other actors. Conversely, during message processing the actor has access to a reference representing the sender of the current message through the `sender` method.

There are several different types of actor references that are supported depending on the configuration of the actor system:

- Purely local actor references are used by actor systems which are not configured to support networking functions. These actor references will not function if sent across a network connection to a remote JVM.
- Local actor references when remoting is enabled are used by actor systems which support networking functions for those references which represent actors within the same JVM. In order to also be reachable when sent to other network nodes, these references include protocol and remote addressing information.
- There is a subtype of local actor references which is used for routers (i.e. actors mixing in the `Router` trait). Its logical structure is the same as for the aforementioned local references, but sending a message to them dispatches to one of their children directly instead.
- Remote actor references represent actors which are reachable using remote communication, i.e. sending messages to them will serialize the messages transparently and send them to the remote JVM.
- There are several special types of actor references which behave like local actor references for all practical purposes:
  - `PromiseActorRef` is the special representation of a `Promise` for the purpose of being completed by the response from an actor. `akka.pattern.ask` creates this actor reference.
  - `DeadLetterActorRef` is the default implementation of the dead letters service to which Akka routes all messages whose destinations are shut down or non-existent.
  - `EmptyLocalActorRef` is what Akka returns when looking up a non-existent local actor path: it is equivalent to a `DeadLetterActorRef`, but it retains its path so that Akka can send it over the network and compare it to other existing actor references for that path, some of which might have been obtained before the actor died.
- And then there are some one-off internal implementations which you should never really see:
  - There is an actor reference which does not represent an actor but acts only as a pseudo-supervisor for the root guardian, we call it “the one who walks the bubbles of space-time”.
  - The first logging service started before actually firing up actor creation facilities is a fake actor reference which accepts log events and prints them directly to standard output; it is `Logging.StandardOutLogger`.

2.5.2 What is an Actor Path?

Since actors are created in a strictly hierarchical fashion, there exists a unique sequence of actor names given by recursively following the supervision links between child and parent down towards the root of the actor system. This sequence can be seen as enclosing folders in a file system, hence we adopted the name “path” to refer to it, although actor hierarchy has some fundamental difference from file system hierarchy.

An actor path consists of an anchor, which identifies the actor system, followed by the concatenation of the path elements, from root guardian to the designated actor; the path elements are the names of the traversed actors and are separated by slashes.

What is the Difference Between Actor Reference and Path?

An actor reference designates a single actor and the life-cycle of the reference matches that actor's life-cycle; an actor path represents a name which may or may not be inhabited by an actor and the path itself does not have a life-cycle, it never becomes invalid. You can create an actor path without creating an actor, but you cannot create an actor reference without creating corresponding actor.

You can create an actor, terminate it, and then create a new actor with the same actor path. The newly created actor is a new incarnation of the actor. It is not the same actor. An actor reference to the old incarnation is not valid for the new incarnation. Messages sent to the old actor reference will not be delivered to the new incarnation even though they have the same path.

Actor Path Anchors

Each actor path has an address component, describing the protocol and location by which the corresponding actor is reachable, followed by the names of the actors in the hierarchy from the root up. Examples are:

```
"akka://my-sys/user/service-a/worker1"                   // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote
```

Here, `akka.tcp` is the default remote transport for the 2.4 release; other transports are pluggable. The interpretation of the host and port part (i.e. `host.example.com:5678` in the example) depends on the transport mechanism used, but it must abide by the URI structural rules.

Logical Actor Paths

The unique path obtained by following the parental supervision links towards the root guardian is called the logical actor path. This path matches exactly the creation ancestry of an actor, so it is completely deterministic as soon as the actor system's remoting configuration (and with it the address component of the path) is set.

Physical Actor Paths

While the logical actor path describes the functional location within one actor system, configuration-based remote deployment means that an actor may be created on a different network host than its parent, i.e. within a different actor system. In this case, following the actor path from the root guardian up entails traversing the network, which is a costly operation. Therefore, each actor also has a physical path, starting at the root guardian of the actor system where the actual actor object resides. Using this path as sender reference when querying other actors will let them reply directly to this actor, minimizing delays incurred by routing.

One important aspect is that a physical actor path never spans multiple actor systems or JVMs. This means that the logical path (supervision hierarchy) and the physical path (actor deployment) of an actor may diverge if one of its ancestors is remotely supervised.

Actor path alias or symbolic link?

As in some real file-systems you might think of a "path alias" or "symbolic link" for an actor, i.e. one actor may be reachable using more than one path. However, you should note that actor hierarchy is different from file system hierarchy. You cannot freely create actor paths like symbolic links to refer to arbitrary actors. As described in the above logical and physical actor path sections, an actor path must be either logical path which represents supervision hierarchy, or physical path which represents actor deployment.

2.5.3 How are Actor References obtained?

There are two general categories to how actor references may be obtained: by creating actors or by looking them up, where the latter functionality comes in the two flavours of creating actor references from concrete actor paths and querying the logical actor hierarchy.

Creating Actors

An actor system is typically started by creating actors beneath the guardian actor using the `ActorSystem.actorOf` method and then using `ActorContext.actorOf` from within the created actors to spawn the actor tree. These methods return a reference to the newly created actor. Each actor has direct access (through its `ActorContext`) to references for its parent, itself and its children. These references may be sent within messages to other actors, enabling those to reply directly.

Looking up Actors by Concrete Path

In addition, actor references may be looked up using the `ActorSystem.actorSelection` method. The selection can be used for communicating with said actor and the actor corresponding to the selection is looked up when delivering each message.

To acquire an `ActorRef` that is bound to the life-cycle of a specific actor you need to send a message, such as the built-in `Identify` message, to the actor and use the `sender()` reference of a reply from the actor.

Absolute vs. Relative Paths

In addition to `ActorSystem.actorSelection` there is also `ActorContext.actorSelection`, which is available inside any actor as `context.actorSelection`. This yields an actor selection much like its twin on `ActorSystem`, but instead of looking up the path starting from the root of the actor tree it starts out on the current actor. Path elements consisting of two dots (`".."`) may be used to access the parent actor. You can for example send a message to a specific sibling:

```
context.actorSelection("../brother") ! msg
```

Absolute paths may of course also be looked up on *context* in the usual way, i.e.

```
context.actorSelection("/user/serviceA") ! msg
```

will work as expected.

Querying the Logical Actor Hierarchy

Since the actor system forms a file-system like hierarchy, matching on paths is possible in the same way as supported by Unix shells: you may replace (parts of) path element names with wildcards (*«*»* and *«?»*) to formulate a selection which may match zero or more actual actors. Because the result is not a single actor reference, it has a different type `ActorSelection` and does not support the full set of operations an `ActorRef` does. Selections may be formulated using the `ActorSystem.actorSelection` and `ActorContext.actorSelection` methods and do support sending messages:

```
context.actorSelection("../*") ! msg
```

will send *msg* to all siblings including the current actor. As for references obtained using *actorSelection*, a traversal of the supervision hierarchy is done in order to perform the message send. As the exact set of actors which match a selection may change even while a message is making its way to the recipients, it is not possible to watch a selection for liveliness changes. In order to do that, resolve the uncertainty by sending a request and gathering all answers, extracting the sender references, and then watch all discovered concrete actors. This scheme of resolving a selection may be improved upon in a future release.

Summary: `actorOf` vs. `actorSelection`

Note: What the above sections described in some detail can be summarized and memorized easily as follows:

- `actorOf` only ever creates a new actor, and it creates it as a direct child of the context on which this method is invoked (which may be any actor or actor system).
- `actorSelection` only ever looks up existing actors when messages are delivered, i.e. does not create actors, or verify existence of actors when the selection is created.

2.5.4 Actor Reference and Path Equality

Equality of `ActorRef` match the intention that an `ActorRef` corresponds to the target actor incarnation. Two actor references are compared equal when they have the same path and point to the same actor incarnation. A reference pointing to a terminated actor does not compare equal to a reference pointing to another (re-created) actor with the same path. Note that a restart of an actor caused by a failure still means that it is the same actor incarnation, i.e. a restart is not visible for the consumer of the `ActorRef`.

If you need to keep track of actor references in a collection and do not care about the exact actor incarnation you can use the `ActorPath` as key, because the identifier of the target actor is not taken into account when comparing actor paths.

2.5.5 Reusing Actor Paths

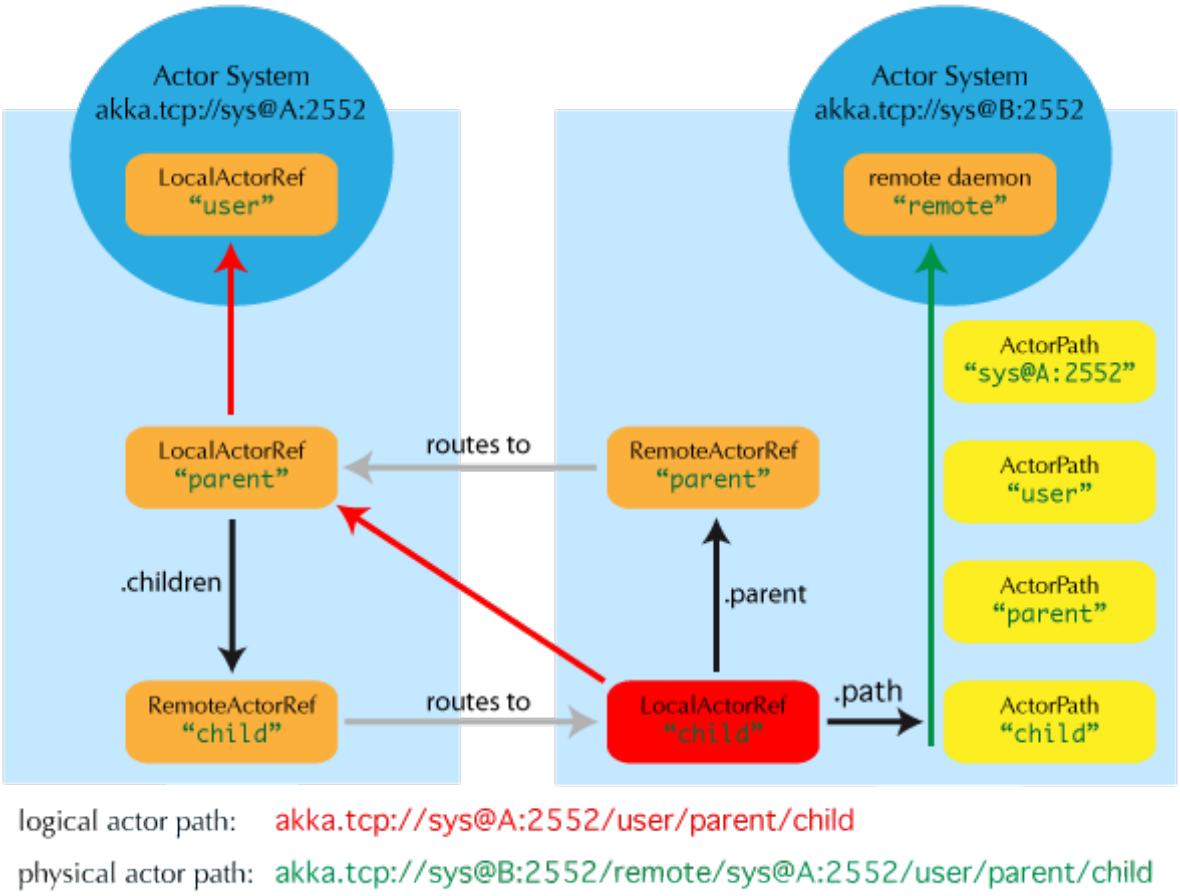
When an actor is terminated, its reference will point to the dead letter mailbox, DeathWatch will publish its final transition and in general it is not expected to come back to life again (since the actor life cycle does not allow this). While it is possible to create an actor at a later time with an identical path—simply due to it being impossible to enforce the opposite without keeping the set of all actors ever created available—this is not good practice: messages sent with `actorSelection` to an actor which "died" suddenly start to work again, but without any guarantee of ordering between this transition and any other event, hence the new inhabitant of the path may receive messages which were destined for the previous tenant.

It may be the right thing to do in very specific circumstances, but make sure to confine the handling of this precisely to the actor's supervisor, because that is the only actor which can reliably detect proper deregistration of the name, before which creation of the new child will fail.

It may also be required during testing, when the test subject depends on being instantiated at a specific path. In that case it is best to mock its supervisor so that it will forward the Terminated message to the appropriate point in the test procedure, enabling the latter to await proper deregistration of the name.

2.5.6 The Interplay with Remote Deployment

When an actor creates a child, the actor system's deployer will decide whether the new actor resides in the same JVM or on another node. In the second case, creation of the actor will be triggered via a network connection to happen in a different JVM and consequently within a different actor system. The remote system will place the new actor below a special path reserved for this purpose and the supervisor of the new actor will be a remote actor reference (representing that actor which triggered its creation). In this case, `context.parent` (the supervisor reference) and `context.path.parent` (the parent node in the actor's path) do not represent the same actor. However, looking up the child's name within the supervisor will find it on the remote node, preserving logical structure e.g. when sending to an unresolved actor reference.



2.5.7 What is the Address part used for?

When sending an actor reference across the network, it is represented by its path. Hence, the path must fully encode all information necessary to send messages to the underlying actor. This is achieved by encoding protocol, host and port in the address part of the path string. When an actor system receives an actor path from a remote node, it checks whether that path's address matches the address of this actor system, in which case it will be resolved to the actor's local reference. Otherwise, it will be represented by a remote actor reference.

2.5.8 Top-Level Scopes for Actor Paths

At the root of the path hierarchy resides the root guardian above which all other actors are found; its name is `"/"`. The next level consists of the following:

- `"/user"` is the guardian actor for all user-created top-level actors; actors created using `ActorSystem.actorOf` are found below this one.
- `"/system"` is the guardian actor for all system-created top-level actors, e.g. logging listeners or actors automatically deployed by configuration at the start of the actor system.
- `"/deadLetters"` is the dead letter actor, which is where all messages sent to stopped or non-existing actors are re-routed (on a best-effort basis: messages may be lost even within the local JVM).
- `"/temp"` is the guardian for all short-lived system-created actors, e.g. those which are used in the implementation of `ActorRef.ask`.
- `"/remote"` is an artificial path below which all actors reside whose supervisors are remote actor references

The need to structure the name space for actors like this arises from a central and very simple design goal: everything in the hierarchy is an actor, and all actors function in the same way. Hence you can not only look up the actors you created, you can also look up the system guardian and send it a message (which it will dutifully

discard in this case). This powerful principle means that there are no quirks to remember, it makes the whole system more uniform and consistent.

If you want to read more about the top-level structure of an actor system, have a look at *The Top-Level Supervisors*.

2.6 Location Transparency

The previous section describes how actor paths are used to enable location transparency. This special feature deserves some extra explanation, because the related term "transparent remoting" was used quite differently in the context of programming languages, platforms and technologies.

2.6.1 Distributed by Default

Everything in Akka is designed to work in a distributed setting: all interactions of actors use purely message passing and everything is asynchronous. This effort has been undertaken to ensure that all functions are available equally when running within a single JVM or on a cluster of hundreds of machines. The key for enabling this is to go from remote to local by way of optimization instead of trying to go from local to remote by way of generalization. See this classic paper for a detailed discussion on why the second approach is bound to fail.

2.6.2 Ways in which Transparency is Broken

What is true of Akka need not be true of the application which uses it, since designing for distributed execution poses some restrictions on what is possible. The most obvious one is that all messages sent over the wire must be serializable. While being a little less obvious this includes closures which are used as actor factories (i.e. within `Props`) if the actor is to be created on a remote node.

Another consequence is that everything needs to be aware of all interactions being fully asynchronous, which in a computer network might mean that it may take several minutes for a message to reach its recipient (depending on configuration). It also means that the probability for a message to be lost is much higher than within one JVM, where it is close to zero (still: no hard guarantee!).

2.6.3 How is Remoting Used?

We took the idea of transparency to the limit in that there is nearly no API for the remoting layer of Akka: it is purely driven by configuration. Just write your application according to the principles outlined in the previous sections, then specify remote deployment of actor sub-trees in the configuration file. This way, your application can be scaled out without having to touch the code. The only piece of the API which allows programmatic influence on remote deployment is that `Props` contain a field which may be set to a specific `Deploy` instance; this has the same effect as putting an equivalent deployment into the configuration file (if both are given, configuration file wins).

2.6.4 Peer-to-Peer vs. Client-Server

Akka Remoting is a communication module for connecting actor systems in a peer-to-peer fashion, and it is the foundation for Akka Clustering. The design of remoting is driven by two (related) design decisions:

1. Communication between involved systems is symmetric: if a system A can connect to a system B then system B must also be able to connect to system A independently.
2. The role of the communicating systems are symmetric in regards to connection patterns: there is no system that only accepts connections, and there is no system that only initiates connections.

The consequence of these decisions is that it is not possible to safely create pure client-server setups with predefined roles (violates assumption 2). For client-server setups it is better to use HTTP or Akka I/O.

Important: Using setups involving Network Address Translation, Load Balancers or Docker containers violates assumption 1, unless additional steps are taken in the network configuration to allow symmetric communication between involved systems. In such situations Akka can be configured to bind to a different network address than the one used for establishing connections between Akka nodes. See *Akka behind NAT or in a Docker container*.

2.6.5 Marking Points for Scaling Up with Routers

In addition to being able to run different parts of an actor system on different nodes of a cluster, it is also possible to scale up onto more cores by multiplying actor sub-trees which support parallelization (think for example a search engine processing different queries in parallel). The clones can then be routed to in different fashions, e.g. round-robin. The only thing necessary to achieve this is that the developer needs to declare a certain actor as "withRouter", then—in its stead—a router actor will be created which will spawn up a configurable number of children of the desired type and route to them in the configured fashion. Once such a router has been declared, its configuration can be freely overridden from the configuration file, including mixing it with the remote deployment of (some of) the children. Read more about this in *Routing (Scala)* and Routing (Java).

2.7 Akka and the Java Memory Model

A major benefit of using the Lightbend Platform, including Scala and Akka, is that it simplifies the process of writing concurrent software. This article discusses how the Lightbend Platform, and Akka in particular, approaches shared memory in concurrent applications.

2.7.1 The Java Memory Model

Prior to Java 5, the Java Memory Model (JMM) was ill defined. It was possible to get all kinds of strange results when shared memory was accessed by multiple threads, such as:

- a thread not seeing values written by other threads: a visibility problem
- a thread observing 'impossible' behavior of other threads, caused by instructions not being executed in the order expected: an instruction reordering problem.

With the implementation of JSR 133 in Java 5, a lot of these issues have been resolved. The JMM is a set of rules based on the "happens-before" relation, which constrain when one memory access must happen before another, and conversely, when they are allowed to happen out of order. Two examples of these rules are:

- **The monitor lock rule:** a release of a lock happens before every subsequent acquire of the same lock.
- **The volatile variable rule:** a write of a volatile variable happens before every subsequent read of the same volatile variable

Although the JMM can seem complicated, the specification tries to find a balance between ease of use and the ability to write performant and scalable concurrent data structures.

2.7.2 Actors and the Java Memory Model

With the Actors implementation in Akka, there are two ways multiple threads can execute actions on shared memory:

- if a message is sent to an actor (e.g. by another actor). In most cases messages are immutable, but if that message is not a properly constructed immutable object, without a "happens before" rule, it would be possible for the receiver to see partially initialized data structures and possibly even values out of thin air (longs/doubles).
- if an actor makes changes to its internal state while processing a message, and accesses that state while processing another message moments later. It is important to realize that with the actor model you don't get any guarantee that the same thread will be executing the same actor for different messages.

To prevent visibility and reordering problems on actors, Akka guarantees the following two "happens before" rules:

- **The actor send rule:** the send of the message to an actor happens before the receive of that message by the same actor.
- **The actor subsequent processing rule:** processing of one message happens before processing of the next message by the same actor.

Note: In layman's terms this means that changes to internal fields of the actor are visible when the next message is processed by that actor. So fields in your actor need not be volatile or equivalent.

Both rules only apply for the same actor instance and are not valid if different actors are used.

2.7.3 Futures and the Java Memory Model

The completion of a Future "happens before" the invocation of any callbacks registered to it are executed.

We recommend not to close over non-final fields (final in Java and val in Scala), and if you *do* choose to close over non-final fields, they must be marked *volatile* in order for the current value of the field to be visible to the callback.

If you close over a reference, you must also ensure that the instance that is referred to is thread safe. We highly recommend staying away from objects that use locking, since it can introduce performance problems and in the worst case, deadlocks. Such are the perils of synchronized.

2.7.4 Actors and shared mutable state

Since Akka runs on the JVM there are still some rules to be followed.

- Closing over internal Actor state and exposing it to other threads

```
class MyActor extends Actor {
 var state = ...
 def receive = {
    case _ =>
      //Wrongs

    // Very bad, shared mutable state,
    // will break your application in weird ways
      Future { state = NewState }
      anotherActor ? message onSuccess { r => state = r }

    // Very bad, "sender" changes for every message,
    // shared mutable state bug
      Future { expensiveCalculation(sender()) }

      //Rights

    // Completely safe, "self" is OK to close over
    // and it's an ActorRef, which is thread-safe
      Future { expensiveCalculation() } onComplete { f => self ! f.value.get }

    // Completely safe, we close over a fixed value
    // and it's an ActorRef, which is thread-safe
      val currentSender = sender()
      Future { expensiveCalculation(currentSender) }
 }
}
```

- Messages **should** be immutable, this is to avoid the shared mutable state trap.

2.8 Message Delivery Reliability

Akka helps you build reliable applications which make use of multiple processor cores in one machine ("scaling up") or distributed across a computer network ("scaling out"). The key abstraction to make this work is that all interactions between your code units—actors—happen via message passing, which is why the precise semantics of how messages are passed between actors deserve their own chapter.

In order to give some context to the discussion below, consider an application which spans multiple network hosts. The basic mechanism for communication is the same whether sending to an actor on the local JVM or to a remote actor, but of course there will be observable differences in the latency of delivery (possibly also depending on the bandwidth of the network link and the message size) and the reliability. In case of a remote message send there are obviously more steps involved which means that more can go wrong. Another aspect is that local sending will just pass a reference to the message inside the same JVM, without any restrictions on the underlying object which is sent, whereas a remote transport will place a limit on the message size.

Writing your actors such that every interaction could possibly be remote is the safe, pessimistic bet. It means to only rely on those properties which are always guaranteed and which are discussed in detail below. This has of course some overhead in the actor's implementation. If you are willing to sacrifice full location transparency—for example in case of a group of closely collaborating actors—you can place them always on the same JVM and enjoy stricter guarantees on message delivery. The details of this trade-off are discussed further below.

As a supplementary part we give a few pointers at how to build stronger reliability on top of the built-in ones. The chapter closes by discussing the role of the "Dead Letter Office".

2.8.1 The General Rules

These are the rules for message sends (i.e. the `tell` or `!` method, which also underlies the `ask` pattern):

- **at-most-once delivery**, i.e. no guaranteed delivery
- **message ordering per sender–receiver pair**

The first rule is typically found also in other actor implementations while the second is specific to Akka.

Discussion: What does "at-most-once" mean?

When it comes to describing the semantics of a delivery mechanism, there are three basic categories:

- **at-most-once** delivery means that for each message handed to the mechanism, that message is delivered zero or one times; in more casual terms it means that messages may be lost.
- **at-least-once** delivery means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.
- **exactly-once** delivery means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

The first one is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism. The second one requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. The third is most expensive—and has consequently worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries.

Discussion: Why No Guaranteed Delivery?

At the core of the problem lies the question what exactly this guarantee shall mean:

1. The message is sent out on the network?
2. The message is received by the other host?
3. The message is put into the target actor's mailbox?
4. The message is starting to be processed by the target actor?
5. The message is processed successfully by the target actor?

Each one of these have different challenges and costs, and it is obvious that there are conditions under which any message passing library would be unable to comply; think for example about configurable mailbox types and how a bounded mailbox would interact with the third point, or even what it would mean to decide upon the "successfully" part of point five.

Along those same lines goes the reasoning in Nobody Needs Reliable Messaging. The only meaningful way for a sender to know whether an interaction was successful is by receiving a business-level acknowledgement message, which is not something Akka could make up on its own (neither are we writing a "do what I mean" framework nor would you want us to).

Akka embraces distributed computing and makes the fallibility of communication explicit through message passing, therefore it does not try to lie and emulate a leaky abstraction. This is a model that has been used with great success in Erlang and requires the users to design their applications around it. You can read more about this approach in the Erlang documentation (section 10.9 and 10.10), Akka follows it closely.

Another angle on this issue is that by providing only basic guarantees those use cases which do not need stronger reliability do not pay the cost of their implementation; it is always possible to add stronger reliability on top of basic ones, but it is not possible to retro-actively remove reliability in order to gain more performance.

Discussion: Message Ordering

The rule more specifically is that *for a given pair of actors, messages sent directly from the first to the second will not be received out-of-order.* The word *directly* emphasizes that this guarantee only applies when sending with the *tell* operator to the final destination, not when employing mediators or other message dissemination features (unless stated otherwise).

The guarantee is illustrated in the following:

Actor `A1` sends messages `M1`, `M2`, `M3` to `A2`

Actor `A3` sends messages `M4`, `M5`, `M6` to `A2`

This means that:

1. If `M1` is delivered it must be delivered before `M2` and `M3`
2. If `M2` is delivered it must be delivered before `M3`
3. If `M4` is delivered it must be delivered before `M5` and `M6`
4. If `M5` is delivered it must be delivered before `M6`
5. `A2` can see messages from `A1` interleaved with messages from `A3`
6. Since there is no guaranteed delivery, any of the messages may be dropped, i.e. not arrive at `A2`

Note: It is important to note that Akka's guarantee applies to the order in which messages are enqueued into the recipient's mailbox. If the mailbox implementation does not respect FIFO order (e.g. a `PriorityMailbox`), then the order of processing by the actor can deviate from the enqueueing order.

Please note that this rule is **not transitive**:

Actor `A` sends message `M1` to actor `C`

Actor `A` then sends message `M2` to actor `B`

Actor `B` forwards message `M2` to actor `C`

Actor `C` may receive `M1` and `M2` in any order

Causal transitive ordering would imply that `M2` is never received before `M1` at actor `C` (though any of them might be lost). This ordering can be violated due to different message delivery latencies when `A`, `B` and `C` reside on different network hosts, see more below.

Note: Actor creation is treated as a message sent from the parent to the child, with the same semantics as discussed above. Sending a message to an actor in a way which could be reordered with this initial creation message means that the message might not arrive because the actor does not exist yet. An example where the message might arrive too early would be to create a remote-deployed actor R1, send its reference to another remote actor R2 and have R2 send a message to R1. An example of well-defined ordering is a parent which creates an actor and immediately sends a message to it.

Communication of failure

Please note, that the ordering guarantees discussed above only hold for user messages between actors. Failure of a child of an actor is communicated by special system messages that are not ordered relative to ordinary user messages. In particular:

Child actor `C` sends message `M` to its parent `P`

Child actor fails with failure `F`

Parent actor `P` might receive the two events either in order `M`, `F` or `F`, `M`

The reason for this is that internal system messages has their own mailboxes therefore the ordering of enqueue calls of a user and system message cannot guarantee the ordering of their dequeue times.

2.8.2 The Rules for In-JVM (Local) Message Sends

Be careful what you do with this section!

Relying on the stronger reliability in this section is not recommended since it will bind your application to local-only deployment: an application may have to be designed differently (as opposed to just employing some message exchange patterns local to some actors) in order to be fit for running on a cluster of machines. Our credo is "design once, deploy any way you wish", and to achieve this you should only rely on *The General Rules*.

Reliability of Local Message Sends

The Akka test suite relies on not losing messages in the local context (and for non-error condition tests also for remote deployment), meaning that we actually do apply the best effort to keep our tests stable. A local `tell` operation can however fail for the same reasons as a normal method call can on the JVM:

- `StackOverflowError`
- `OutOfMemoryError`
- other `VirtualMachineError`

In addition, local sends can fail in Akka-specific ways:

- if the mailbox does not accept the message (e.g. full BoundedMailbox)
- if the receiving actor fails while processing the message or is already terminated

While the first is clearly a matter of configuration the second deserves some thought: the sender of a message does not get feedback if there was an exception while processing, that notification goes to the supervisor instead. This is in general not distinguishable from a lost message for an outside observer.

Ordering of Local Message Sends

Assuming strict FIFO mailboxes the aforementioned caveat of non-transitivity of the message ordering guarantee is eliminated under certain conditions. As you will note, these are quite subtle as it stands, and it is even possible that future performance optimizations will invalidate this whole paragraph. The possibly non-exhaustive list of counter-indications is:

- Before receiving the first reply from a top-level actor, there is a lock which protects an internal interim queue, and this lock is not fair; the implication is that enqueue requests from different senders which arrive during the actor's construction (figuratively, the details are more involved) may be reordered depending on low-level thread scheduling. Since completely fair locks do not exist on the JVM this is unfixable.
- The same mechanism is used during the construction of a Router, more precisely the routed ActorRef, hence the same problem exists for actors deployed with Routers.
- As mentioned above, the problem occurs anywhere a lock is involved during enqueueing, which may also apply to custom mailboxes.

This list has been compiled carefully, but other problematic scenarios may have escaped our analysis.

How does Local Ordering relate to Network Ordering

The rule that *for a given pair of actors, messages sent directly from the first to the second will not be received out-of-order* holds for messages sent over the network with the TCP based Akka remote transport protocol.

As explained in the previous section local message sends obey transitive causal ordering under certain conditions. This ordering can be violated due to different message delivery latencies. For example:

> Actor `A` on node-1 sends message `M1` to actor `C` on node-3
>
> Actor `A` on node-1 then sends message `M2` to actor `B` on node-2
>
> Actor `B` on node-2 forwards message `M2` to actor `C` on node-3
>
> Actor `C` may receive `M1` and `M2` in any order

It might take longer time for `M1` to "travel" to node-3 than it takes for `M2` to "travel" to node-3 via node-2.

2.8.3 Higher-level abstractions

Based on a small and consistent tool set in Akka's core, Akka also provides powerful, higher-level abstractions on top it.

Messaging Patterns

As discussed above a straight-forward answer to the requirement of reliable delivery is an explicit ACK–RETRY protocol. In its simplest form this requires

- a way to identify individual messages to correlate message with acknowledgement
- a retry mechanism which will resend messages if not acknowledged in time
- a way for the receiver to detect and discard duplicates

The third becomes necessary by virtue of the acknowledgements not being guaranteed to arrive either. An ACK-RETRY protocol with business-level acknowledgements is supported by *At-Least-Once Delivery* of the Akka Persistence module. Duplicates can be detected by tracking the identifiers of messages sent via *At-Least-Once*

Delivery. Another way of implementing the third part would be to make processing the messages idempotent on the level of the business logic.

Another example of implementing all three requirements is shown at *Reliable Proxy Pattern* (which is now superseded by *At-Least-Once Delivery*).

Event Sourcing

Event sourcing (and sharding) is what makes large websites scale to billions of users, and the idea is quite simple: when a component (think actor) processes a command it will generate a list of events representing the effect of the command. These events are stored in addition to being applied to the component's state. The nice thing about this scheme is that events only ever are appended to the storage, nothing is ever mutated; this enables perfect replication and scaling of consumers of this event stream (i.e. other components may consume the event stream as a means to replicate the component's state on a different continent or to react to changes). If the component's state is lost—due to a machine failure or by being pushed out of a cache—it can easily be reconstructed by replaying the event stream (usually employing snapshots to speed up the process). *Event sourcing* is supported by Akka Persistence.

Mailbox with Explicit Acknowledgement

By implementing a custom mailbox type it is possible to retry message processing at the receiving actor's end in order to handle temporary failures. This pattern is mostly useful in the local communication context where delivery guarantees are otherwise sufficient to fulfill the application's requirements.

Please note that the caveats for *The Rules for In-JVM (Local) Message Sends* do apply.

An example implementation of this pattern is shown at *Mailbox with Explicit Acknowledgement*.

2.8.4 Dead Letters

Messages which cannot be delivered (and for which this can be ascertained) will be delivered to a synthetic actor called `/deadLetters`. This delivery happens on a best-effort basis; it may fail even within the local JVM (e.g. during actor termination). Messages sent via unreliable network transports will be lost without turning up as dead letters.

What Should I Use Dead Letters For?

The main use of this facility is for debugging, especially if an actor send does not arrive consistently (where usually inspecting the dead letters will tell you that the sender or recipient was set wrong somewhere along the way). In order to be useful for this purpose it is good practice to avoid sending to deadLetters where possible, i.e. run your application with a suitable dead letter logger (see more below) from time to time and clean up the log output. This exercise—like all else—requires judicious application of common sense: it may well be that avoiding to send to a terminated actor complicates the sender's code more than is gained in debug output clarity.

The dead letter service follows the same rules with respect to delivery guarantees as all other message sends, hence it cannot be used to implement guaranteed delivery.

How do I Receive Dead Letters?

An actor can subscribe to class `akka.actor.DeadLetter` on the event stream, see event-stream-java (Java) or *Event Stream* (Scala) for how to do that. The subscribed actor will then receive all dead letters published in the (local) system from that point onwards. Dead letters are not propagated over the network, if you want to collect them in one place you will have to subscribe one actor per network node and forward them manually. Also consider that dead letters are generated at that node which can determine that a send operation is failed, which for a remote send can be the local system (if no network connection can be established) or the remote one (if the actor you are sending to does not exist at that point in time).

Dead Letters Which are (Usually) not Worrisome

Every time an actor does not terminate by its own decision, there is a chance that some messages which it sends to itself are lost. There is one which happens quite easily in complex shutdown scenarios that is usually benign: seeing a `akka.dispatch.Terminate` message dropped means that two termination requests were given, but of course only one can succeed. In the same vein, you might see `akka.actor.Terminated` messages from children while stopping a hierarchy of actors turning up in dead letters if the parent is still watching the child when the parent terminates.

2.9 Configuration

You can start using Akka without defining any configuration, since sensible default values are provided. Later on you might need to amend the settings to change the default behavior or adapt for specific runtime environments. Typical examples of settings that you might amend:

- log level and logger backend
- enable remoting
- message serializers
- definition of routers
- tuning of dispatchers

Akka uses the Typesafe Config Library, which might also be a good choice for the configuration of your own application or library built with or without Akka. This library is implemented in Java with no external dependencies; you should have a look at its documentation (in particular about ConfigFactory), which is only summarized in the following.

> **Warning:** If you use Akka from the Scala REPL from the 2.9.x series, and you do not provide your own ClassLoader to the ActorSystem, start the REPL with "-Yrepl-sync" to work around a deficiency in the REPLs provided Context ClassLoader.

2.9.1 Where configuration is read from

All configuration for Akka is held within instances of `ActorSystem`, or put differently, as viewed from the outside, `ActorSystem` is the only consumer of configuration information. While constructing an actor system, you can either pass in a `Config` object or not, where the second case is equivalent to passing `ConfigFactory.load()` (with the right class loader). This means roughly that the default is to parse all `application.conf`, `application.json` and `application.properties` found at the root of the class path—please refer to the aforementioned documentation for details. The actor system then merges in all `reference.conf` resources found at the root of the class path to form the fallback configuration, i.e. it internally uses

```
appConfig.withFallback(ConfigFactory.defaultReference(classLoader))
```

The philosophy is that code never contains default values, but instead relies upon their presence in the `reference.conf` supplied with the library in question.

Highest precedence is given to overrides given as system properties, see the HOCON specification (near the bottom). Also noteworthy is that the application configuration—which defaults to `application`—may be overridden using the `config.resource` property (there are more, please refer to the Config docs).

> **Note:** If you are writing an Akka application, keep you configuration in `application.conf` at the root of the class path. If you are writing an Akka-based library, keep its configuration in `reference.conf` at the root

of the JAR file.

2.9.2 When using JarJar, OneJar, Assembly or any jar-bundler

Warning: Akka's configuration approach relies heavily on the notion of every module/jar having its own reference.conf file, all of these will be discovered by the configuration and loaded. Unfortunately this also means that if you put/merge multiple jars into the same jar, you need to merge all the reference.confs as well. Otherwise all defaults will be lost and Akka will not function.

If you are using Maven to package your application, you can also make use of the Apache Maven Shade Plugin support for Resource Transformers to merge all the reference.confs on the build classpath into one.

The plugin configuration might look like this:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-shade-plugin</artifactId>
 <version>1.5</version>
 <executions>
  <execution>
   <phase>package</phase>
   <goals>
    <goal>shade</goal>
   </goals>
   <configuration>
    <shadedArtifactAttached>true</shadedArtifactAttached>
    <shadedClassifierName>allinone</shadedClassifierName>
    <artifactSet>
     <includes>
      <include>*:*</include>
     </includes>
    </artifactSet>
    <transformers>
      <transformer
       implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer
↪">
       <resource>reference.conf</resource>
      </transformer>
      <transformer
       implementation="org.apache.maven.plugins.shade.resource.
↪ManifestResourceTransformer">
       <manifestEntries>
        <Main-Class>akka.Main</Main-Class>
       </manifestEntries>
      </transformer>
    </transformers>
   </configuration>
  </execution>
 </executions>
</plugin>
```

2.9.3 Custom application.conf

A custom `application.conf` might look like this:

```
# In this file you can override any option defined in the reference files.
# Copy in parts of the reference files and modify as you please.
```

```
akka {

  # Loggers to register at boot time (akka.event.Logging$DefaultLogger logs
  # to STDOUT)
  loggers = ["akka.event.slf4j.Slf4jLogger"]

  # Log level used by the configured loggers (see "loggers") as soon
  # as they have been started; before that, see "stdout-loglevel"
  # Options: OFF, ERROR, WARNING, INFO, DEBUG
  loglevel = "DEBUG"

  # Log level for the very basic logger activated during ActorSystem startup.
  # This logger prints the log messages to stdout (System.out).
  # Options: OFF, ERROR, WARNING, INFO, DEBUG
  stdout-loglevel = "DEBUG"

  # Filter of log events that is used by the LoggingAdapter before
  # publishing log events to the eventStream.
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"

  actor {
    provider = "cluster"

    default-dispatcher {
      # Throughput for default Dispatcher, set to 1 for as fair as possible
      throughput = 10
    }
  }

  remote {
    # The port clients should connect to. Default is 2552.
    netty.tcp.port = 4711
  }
}
```

2.9.4 Including files

Sometimes it can be useful to include another configuration file, for example if you have one `application.conf` with all environment independent settings and then override some settings for specific environments.

Specifying system property with `-Dconfig.resource=/dev.conf` will load the `dev.conf` file, which includes the `application.conf`

dev.conf:

```
include "application"

akka {
  loglevel = "DEBUG"
}
```

More advanced include and substitution mechanisms are explained in the HOCON specification.

2.9.5 Logging of Configuration

If the system or config property `akka.log-config-on-start` is set to `on`, then the complete configuration is logged at INFO level when the actor system is started. This is useful when you are uncertain of what configuration is used.

If in doubt, you can also easily and nicely inspect configuration objects before or after using them to construct an actor system:

```
Welcome to Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import com.typesafe.config._
import com.typesafe.config._

scala> ConfigFactory.parseString("a.b=12")
res0: com.typesafe.config.Config = Config(SimpleConfigObject({"a" : {"b" : 12}}))

scala> res0.root.render
res1: java.lang.String =
{
    # String: 1
    "a" : {
        # String: 1
        "b" : 12
    }
}
```

The comments preceding every item give detailed information about the origin of the setting (file & line number) plus possible comments which were present, e.g. in the reference configuration. The settings as merged with the reference and parsed by the actor system can be displayed like this:

```
final ActorSystem system = ActorSystem.create();
System.out.println(system.settings());
// this is a shortcut for system.settings().config().root().render()
```

2.9.6 A Word About ClassLoaders

In several places of the configuration file it is possible to specify the fully-qualified class name of something to be instantiated by Akka. This is done using Java reflection, which in turn uses a `ClassLoader`. Getting the right one in challenging environments like application containers or OSGi bundles is not always trivial, the current approach of Akka is that each `ActorSystem` implementation stores the current thread's context class loader (if available, otherwise just its own loader as in `this.getClass.getClassLoader`) and uses that for all reflective accesses. This implies that putting Akka on the boot class path will yield `NullPointerException` from strange places: this is simply not supported.

2.9.7 Application specific settings

The configuration can also be used for application specific settings. A good practice is to place those settings in an Extension, as described in:

- Scala API: *Application specific settings*
- Java API: extending-akka-java.settings

2.9.8 Configuring multiple ActorSystem

If you have more than one `ActorSystem` (or you're writing a library and have an `ActorSystem` that may be separate from the application's) you may want to separate the configuration for each system.

Given that `ConfigFactory.load()` merges all resources with matching name from the whole class path, it is easiest to utilize that functionality and differentiate actor systems within the hierarchy of the configuration:

```
myapp1 {
  akka.loglevel = "WARNING"
  my.own.setting = 43
}
myapp2 {
  akka.loglevel = "ERROR"
  app2.setting = "appname"
}
my.own.setting = 42
my.other.setting = "hello"
```

```
val config = ConfigFactory.load()
val app1 = ActorSystem("MyApp1", config.getConfig("myapp1").withFallback(config))
val app2 = ActorSystem("MyApp2",
  config.getConfig("myapp2").withOnlyPath("akka").withFallback(config))
```

These two samples demonstrate different variations of the "lift-a-subtree" trick: in the first case, the configuration accessible from within the actor system is this

```
akka.loglevel = "WARNING"
my.own.setting = 43
my.other.setting = "hello"
// plus myapp1 and myapp2 subtrees
```

while in the second one, only the "akka" subtree is lifted, with the following result

```
akka.loglevel = "ERROR"
my.own.setting = 42
my.other.setting = "hello"
// plus myapp1 and myapp2 subtrees
```

Note: The configuration library is really powerful, explaining all features exceeds the scope affordable here. In particular not covered are how to include other configuration files within other files (see a small example at *Including files*) and copying parts of the configuration tree by way of path substitutions.

You may also specify and parse the configuration programmatically in other ways when instantiating the `ActorSystem`.

```
import akka.actor.ActorSystem
import com.typesafe.config.ConfigFactory
    val customConf = ConfigFactory.parseString("""
      akka.actor.deployment {
        /my-service {
          router = round-robin-pool
          nr-of-instances = 3
        }
      }
      """)
    // ConfigFactory.load sandwiches customConfig between default reference
    // config and default overrides, and then resolves it.
    val system = ActorSystem("MySystem", ConfigFactory.load(customConf))
```

2.9.9 Reading configuration from a custom location

You can replace or supplement `application.conf` either in code or using system properties.

If you're using `ConfigFactory.load()` (which Akka does by default) you can replace `application.conf` by defining `-Dconfig.resource=whatever`, `-Dconfig.file=whatever`, or `-Dconfig.url=whatever`.

From inside your replacement file specified with `-Dconfig.resource` and friends, you can `include "application"` if you still want to use `application.{conf,json,properties}` as well. Settings specified before `include "application"` would be overridden by the included file, while those after would override the included file.

In code, there are many customization options.

There are several overloads of `ConfigFactory.load()`; these allow you to specify something to be sandwiched between system properties (which override) and the defaults (from `reference.conf`), replacing the usual `application.{conf,json,properties}` and replacing `-Dconfig.file` and friends.

The simplest variant of `ConfigFactory.load()` takes a resource basename (instead of `application`); `myname.conf`, `myname.json`, and `myname.properties` would then be used instead of `application.{conf,json,properties}`.

The most flexible variant takes a `Config` object, which you can load using any method in `ConfigFactory`. For example you could put a config string in code using `ConfigFactory.parseString()` or you could make a map and `ConfigFactory.parseMap()`, or you could load a file.

You can also combine your custom config with the usual config, that might look like:

```
// make a Config with just your special setting
Config myConfig =
  ConfigFactory.parseString("something=somethingElse");
// load the normal config stack (system props,
// then application.conf, then reference.conf)
Config regularConfig =
  ConfigFactory.load();
// override regular stack with myConfig
Config combined =
  myConfig.withFallback(regularConfig);
// put the result in between the overrides
// (system props) and defaults again
Config complete =
  ConfigFactory.load(combined);
// create ActorSystem
ActorSystem system =
  ActorSystem.create("myname", complete);
```

When working with `Config` objects, keep in mind that there are three "layers" in the cake:

- `ConfigFactory.defaultOverrides()` (system properties)
- the app's settings
- `ConfigFactory.defaultReference()` (reference.conf)

The normal goal is to customize the middle layer while leaving the other two alone.

- `ConfigFactory.load()` loads the whole stack
- the overloads of `ConfigFactory.load()` let you specify a different middle layer
- the `ConfigFactory.parse()` variations load single files or resources

To stack two layers, use `override.withFallback(fallback)`; try to keep system props (`defaultOverrides()`) on top and `reference.conf` (`defaultReference()`) on the bottom.

Do keep in mind, you can often just add another `include` statement in `application.conf` rather than writing code. Includes at the top of `application.conf` will be overridden by the rest of `application.conf`, while those at the bottom will override the earlier stuff.

2.9.10 Actor Deployment Configuration

Deployment settings for specific actors can be defined in the `akka.actor.deployment` section of the configuration. In the deployment section it is possible to define things like dispatcher, mailbox, router settings, and

remote deployment. Configuration of these features are described in the chapters detailing corresponding topics. An example may look like this:

```
akka.actor.deployment {

  # '/user/actorA/actorB' is a remote deployed actor
  /actorA/actorB {
    remote = "akka.tcp://sampleActorSystem@127.0.0.1:2553"
  }

  # all direct children of '/user/actorC' have a dedicated dispatcher
  "/actorC/*" {
    dispatcher = my-dispatcher
  }

  # all descendants of '/user/actorC' (direct children, and their children␣
↪recursively)
  # have a dedicated dispatcher
  "/actorC/**" {
    dispatcher = my-dispatcher
  }

  # '/user/actorD/actorE' has a special priority mailbox
  /actorD/actorE {
    mailbox = prio-mailbox
  }

  # '/user/actorF/actorG/actorH' is a random pool
  /actorF/actorG/actorH {
    router = random-pool
    nr-of-instances = 5
  }
}

my-dispatcher {
  fork-join-executor.parallelism-min = 10
  fork-join-executor.parallelism-max = 10
}
prio-mailbox {
  mailbox-type = "a.b.MyPrioMailbox"
}
```

Note: The deployment section for a specific actor is identified by the path of the actor relative to `/user`.

You can use asterisks as wildcard matches for the actor path sections, so you could specify: `/*/sampleActor` and that would match all `sampleActor` on that level in the hierarchy. In addition, please note:

- you can also use wildcards in the last position to match all actors at a certain level: `/someParent/*`
- you can use double-wildcards in the last position to match all child actors and their children recursively: `/someParent/**`
- non-wildcard matches always have higher priority to match than wildcards, and single wildcard matches have higher priority than double-wildcards, so: `/foo/bar` is considered **more specific** than `/foo/*`, which is considered **more specific** than `/foo/**`. Only the highest priority match is used
- wildcards **cannot** be used to partially match section, like this: `/foo*/bar`, `/f*o/bar` etc.

Note: Double-wildcards can only be placed in the last position.

2.9.11 Listing of the Reference Configuration

Each Akka module has a reference configuration file with the default values.

akka-actor

```
####################################
# Akka Actor Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# Akka version, checked against the runtime version of Akka. Loaded from generated␣
↪conf file.
include "version"

akka {
  # Home directory of Akka, modules in the deploy directory will be loaded
  home = ""

  # Loggers to register at boot time (akka.event.Logging$DefaultLogger logs
  # to STDOUT)
  loggers = ["akka.event.Logging$DefaultLogger"]

  # Filter of log events that is used by the LoggingAdapter before
  # publishing log events to the eventStream. It can perform
  # fine grained filtering based on the log source. The default
  # implementation filters on the `loglevel`.
  # FQCN of the LoggingFilter. The Class of the FQCN must implement
  # akka.event.LoggingFilter and have a public constructor with
  # (akka.actor.ActorSystem.Settings, akka.event.EventStream) parameters.
  logging-filter = "akka.event.DefaultLoggingFilter"

  # Specifies the default loggers dispatcher
  loggers-dispatcher = "akka.actor.default-dispatcher"

  # Loggers are created and registered synchronously during ActorSystem
  # start-up, and since they are actors, this timeout is used to bound the
  # waiting time
  logger-startup-timeout = 5s

  # Log level used by the configured loggers (see "loggers") as soon
  # as they have been started; before that, see "stdout-loglevel"
  # Options: OFF, ERROR, WARNING, INFO, DEBUG
  loglevel = "INFO"

  # Log level for the very basic logger activated during ActorSystem startup.
  # This logger prints the log messages to stdout (System.out).
  # Options: OFF, ERROR, WARNING, INFO, DEBUG
  stdout-loglevel = "WARNING"

  # Log the complete configuration at INFO level when the actor system is started.
  # This is useful when you are uncertain of what configuration is used.
  log-config-on-start = off

  # Log at info level when messages are sent to dead letters.
  # Possible values:
  # on: all dead letters are logged
  # off: no logging of dead letters
  # n: positive integer, number of dead letters that will be logged
  log-dead-letters = 10
```

```
  # Possibility to turn off logging of dead letters while the actor system
  # is shutting down. Logging is only done when enabled by 'log-dead-letters'
  # setting.
  log-dead-letters-during-shutdown = on

  # List FQCN of extensions which shall be loaded at actor system startup.
  # Library extensions are regular extensions that are loaded at startup and are
  # available for third party library authors to enable auto-loading of extensions␣
↪when
  # present on the classpath. This is done by appending entries:
  # 'library-extensions += "Extension"' in the library `reference.conf`.
  #
  # Should not be set by end user applications in 'application.conf', use the␣
↪extensions property for that
  #
  library-extensions = ${?akka.library-extensions} []

  # List FQCN of extensions which shall be loaded at actor system startup.
  # Should be on the format: 'extensions = ["foo", "bar"]' etc.
  # See the Akka Documentation for more info about Extensions
  extensions = []

  # Toggles whether threads created by this ActorSystem should be daemons or not
  daemonic = off

  # JVM shutdown, System.exit(-1), in case of a fatal error,
  # such as OutOfMemoryError
  jvm-exit-on-fatal-error = on

  actor {

    # Either one of "local", "remote" or "cluster" or the
    # FQCN of the ActorRefProvider to be used; the below is the built-in default,
    # note that "remote" and "cluster" requires the akka-remote and akka-cluster
    # artifacts to be on the classpath.
    provider = "local"

    # The guardian "/user" will use this class to obtain its supervisorStrategy.
    # It needs to be a subclass of akka.actor.SupervisorStrategyConfigurator.
    # In addition to the default there is akka.actor.StoppingSupervisorStrategy.
    guardian-supervisor-strategy = "akka.actor.DefaultSupervisorStrategy"

    # Timeout for ActorSystem.actorOf
    creation-timeout = 20s

    # Serializes and deserializes (non-primitive) messages to ensure immutability,
    # this is only intended for testing.
    serialize-messages = off

    # Serializes and deserializes creators (in Props) to ensure that they can be
    # sent over the network, this is only intended for testing. Purely local␣
↪deployments
    # as marked with deploy.scope == LocalScope are exempt from verification.
    serialize-creators = off

    # Timeout for send operations to top-level actors which are in the process
    # of being started. This is only relevant if using a bounded mailbox or the
    # CallingThreadDispatcher for a top-level actor.
    unstarted-push-timeout = 10s

    typed {
      # Default timeout for typed actor methods with non-void return type
```

```
    timeout = 5s
  }

  # Mapping between ´deployment.router' short names to fully qualified class␣
↪names
  router.type-mapping {
    from-code = "akka.routing.NoRouter"
    round-robin-pool = "akka.routing.RoundRobinPool"
    round-robin-group = "akka.routing.RoundRobinGroup"
    random-pool = "akka.routing.RandomPool"
    random-group = "akka.routing.RandomGroup"
    balancing-pool = "akka.routing.BalancingPool"
    smallest-mailbox-pool = "akka.routing.SmallestMailboxPool"
    broadcast-pool = "akka.routing.BroadcastPool"
    broadcast-group = "akka.routing.BroadcastGroup"
    scatter-gather-pool = "akka.routing.ScatterGatherFirstCompletedPool"
    scatter-gather-group = "akka.routing.ScatterGatherFirstCompletedGroup"
    tail-chopping-pool = "akka.routing.TailChoppingPool"
    tail-chopping-group = "akka.routing.TailChoppingGroup"
    consistent-hashing-pool = "akka.routing.ConsistentHashingPool"
    consistent-hashing-group = "akka.routing.ConsistentHashingGroup"
  }

  deployment {

    # deployment id pattern - on the format: /parent/child etc.
    default {

      # The id of the dispatcher to use for this actor.
      # If undefined or empty the dispatcher specified in code
      # (Props.withDispatcher) is used, or default-dispatcher if not
      # specified at all.
      dispatcher = ""

      # The id of the mailbox to use for this actor.
      # If undefined or empty the default mailbox of the configured dispatcher
      # is used or if there is no mailbox configuration the mailbox specified
      # in code (Props.withMailbox) is used.
      # If there is a mailbox defined in the configured dispatcher then that
      # overrides this setting.
      mailbox = ""

      # routing (load-balance) scheme to use
      # - available: "from-code", "round-robin", "random", "smallest-mailbox",
      #              "scatter-gather", "broadcast"
      # - or:        Fully qualified class name of the router class.
      #              The class must extend akka.routing.CustomRouterConfig and
      #              have a public constructor with com.typesafe.config.Config
      #              and optional akka.actor.DynamicAccess parameter.
      # - default is "from-code";
      # Whether or not an actor is transformed to a Router is decided in code
      # only (Props.withRouter). The type of router can be overridden in the
      # configuration; specifying "from-code" means that the values specified
      # in the code shall be used.
      # In case of routing, the actors to be routed to can be specified
      # in several ways:
      # - nr-of-instances: will create that many children
      # - routees.paths: will route messages to these paths using ActorSelection,
      #   i.e. will not create children
      # - resizer: dynamically resizable number of routees as specified in
      #   resizer below
      router = "from-code"
```

```
        # number of children to create in case of a router;
        # this setting is ignored if routees.paths is given
        nr-of-instances = 1

        # within is the timeout used for routers containing future calls
        within = 5 seconds

        # number of virtual nodes per node for consistent-hashing router
        virtual-nodes-factor = 10

        tail-chopping-router {
          # interval is duration between sending message to next routee
          interval = 10 milliseconds
        }

        routees {
          # Alternatively to giving nr-of-instances you can specify the full
          # paths of those actors which should be routed to. This setting takes
          # precedence over nr-of-instances
          paths = []
        }

        # To use a dedicated dispatcher for the routees of the pool you can
        # define the dispatcher configuration inline with the property name
        # 'pool-dispatcher' in the deployment section of the router.
        # For example:
        # pool-dispatcher {
        #   fork-join-executor.parallelism-min = 5
        #   fork-join-executor.parallelism-max = 5
        # }

        # Routers with dynamically resizable number of routees; this feature is
        # enabled by including (parts of) this section in the deployment
        resizer {

          enabled = off

          # The fewest number of routees the router should ever have.
          lower-bound = 1

          # The most number of routees the router should ever have.
          # Must be greater than or equal to lower-bound.
          upper-bound = 10

          # Threshold used to evaluate if a routee is considered to be busy
          # (under pressure). Implementation depends on this value (default is 1).
          # 0:   number of routees currently processing a message.
          # 1:   number of routees currently processing a message has
          #      some messages in mailbox.
          # > 1: number of routees with at least the configured pressure-threshold
          #      messages in their mailbox. Note that estimating mailbox size of
          #      default UnboundedMailbox is O(N) operation.
          pressure-threshold = 1

          # Percentage to increase capacity whenever all routees are busy.
          # For example, 0.2 would increase 20% (rounded up), i.e. if current
          # capacity is 6 it will request an increase of 2 more routees.
          rampup-rate = 0.2

          # Minimum fraction of busy routees before backing off.
          # For example, if this is 0.3, then we'll remove some routees only when
          # less than 30% of routees are busy, i.e. if current capacity is 10 and
          # 3 are busy then the capacity is unchanged, but if 2 or less are busy
```

```
          # the capacity is decreased.
          # Use 0.0 or negative to avoid removal of routees.
          backoff-threshold = 0.3

          # Fraction of routees to be removed when the resizer reaches the
          # backoffThreshold.
          # For example, 0.1 would decrease 10% (rounded up), i.e. if current
          # capacity is 9 it will request an decrease of 1 routee.
          backoff-rate = 0.1

          # Number of messages between resize operation.
          # Use 1 to resize before each message.
          messages-per-resize = 10
        }

        # Routers with dynamically resizable number of routees based on
        # performance metrics.
        # This feature is enabled by including (parts of) this section in
        # the deployment, cannot be enabled together with default resizer.
        optimal-size-exploring-resizer {

          enabled = off

          # The fewest number of routees the router should ever have.
          lower-bound = 1

          # The most number of routees the router should ever have.
          # Must be greater than or equal to lower-bound.
          upper-bound = 10

          # probability of doing a ramping down when all routees are busy
          # during exploration.
          chance-of-ramping-down-when-full = 0.2

          # Interval between each resize attempt
          action-interval = 5s

          # If the routees have not been fully utilized (i.e. all routees busy)
          # for such length, the resizer will downsize the pool.
          downsize-after-underutilized-for = 72h

          # Duration exploration, the ratio between the largest step size and
          # current pool size. E.g. if the current pool size is 50, and the
          # explore-step-size is 0.1, the maximum pool size change during
          # exploration will be +- 5
          explore-step-size = 0.1

          # Probabily of doing an exploration v.s. optmization.
          chance-of-exploration = 0.4

          # When downsizing after a long streak of underutilization, the resizer
          # will downsize the pool to the highest utiliziation multiplied by a
          # a downsize rasio. This downsize ratio determines the new pools size
          # in comparison to the highest utilization.
          # E.g. if the highest utilization is 10, and the down size ratio
          # is 0.8, the pool will be downsized to 8
          downsize-ratio = 0.8

          # When optimizing, the resizer only considers the sizes adjacent to the
          # current size. This number indicates how many adjacent sizes to␣
↪consider.
          optimization-range = 16
```

```
          # The weight of the latest metric over old metrics when collecting
          # performance metrics.
          # E.g. if the last processing speed is 10 millis per message at pool
          # size 5, and if the new processing speed collected is 6 millis per
          # message at pool size 5. Given a weight of 0.3, the metrics
          # representing pool size 5 will be 6 * 0.3 + 10 * 0.7, i.e. 8.8 millis
          # Obviously, this number should be between 0 and 1.
          weight-of-latest-metric = 0.5
        }
      }

      /IO-DNS/inet-address {
        mailbox = "unbounded"
        router = "consistent-hashing-pool"
        nr-of-instances = 4
      }
    }

    default-dispatcher {
      # Must be one of the following
      # Dispatcher, PinnedDispatcher, or a FQCN to a class inheriting
      # MessageDispatcherConfigurator with a public constructor with
      # both com.typesafe.config.Config parameter and
      # akka.dispatch.DispatcherPrerequisites parameters.
      # PinnedDispatcher must be used together with executor=thread-pool-executor.
      type = "Dispatcher"

      # Which kind of ExecutorService to use for this dispatcher
      # Valid options:
      #  - "default-executor" requires a "default-executor" section
      #  - "fork-join-executor" requires a "fork-join-executor" section
      #  - "thread-pool-executor" requires a "thread-pool-executor" section
      #  - A FQCN of a class extending ExecutorServiceConfigurator
      executor = "default-executor"

      # This will be used if you have set "executor = "default-executor"".
      # If an ActorSystem is created with a given ExecutionContext, this
      # ExecutionContext will be used as the default executor for all
      # dispatchers in the ActorSystem configured with
      # executor = "default-executor". Note that "default-executor"
      # is the default value for executor, and therefore used if not
      # specified otherwise. If no ExecutionContext is given,
      # the executor configured in "fallback" will be used.
      default-executor {
        fallback = "fork-join-executor"
      }

      # This will be used if you have set "executor = "fork-join-executor""
      # Underlying thread pool implementation is scala.concurrent.forkjoin.
↪ForkJoinPool
      fork-join-executor {
        # Min number of threads to cap factor-based parallelism number to
        parallelism-min = 8

        # The parallelism factor is used to determine thread pool size using the
        # following formula: ceil(available processors * factor). Resulting size
        # is then bounded by the parallelism-min and parallelism-max values.
        parallelism-factor = 3.0

        # Max number of threads to cap factor-based parallelism number to
        parallelism-max = 64

        # Setting to "FIFO" to use queue like peeking mode which "poll" or "LIFO"␣
↪to use stack
```

```
        # like peeking mode which "pop".
        task-peeking-mode = "FIFO"
      }

      # This will be used if you have set "executor = "thread-pool-executor""
      # Underlying thread pool implementation is java.util.concurrent.
↪ThreadPoolExecutor
      thread-pool-executor {
        # Keep alive time for threads
        keep-alive-time = 60s

        # Define a fixed thread pool size with this property. The corePoolSize
        # and the maximumPoolSize of the ThreadPoolExecutor will be set to this
        # value, if it is defined. Then the other pool-size properties will not
        # be used.
        #
        # Valid values are: `off` or a positive integer.
        fixed-pool-size = off

        # Min number of threads to cap factor-based corePoolSize number to
        core-pool-size-min = 8

        # The core-pool-size-factor is used to determine corePoolSize of the
        # ThreadPoolExecutor using the following formula:
        # ceil(available processors * factor).
        # Resulting size is then bounded by the core-pool-size-min and
        # core-pool-size-max values.
        core-pool-size-factor = 3.0

        # Max number of threads to cap factor-based corePoolSize number to
        core-pool-size-max = 64

        # Minimum number of threads to cap factor-based maximumPoolSize number to
        max-pool-size-min = 8

        # The max-pool-size-factor is used to determine maximumPoolSize of the
        # ThreadPoolExecutor using the following formula:
        # ceil(available processors * factor)
        # The maximumPoolSize will not be less than corePoolSize.
        # It is only used if using a bounded task queue.
        max-pool-size-factor  = 3.0

        # Max number of threads to cap factor-based maximumPoolSize number to
        max-pool-size-max = 64

        # Specifies the bounded capacity of the task queue (< 1 == unbounded)
        task-queue-size = -1

        # Specifies which type of task queue will be used, can be "array" or
        # "linked" (default)
        task-queue-type = "linked"

        # Allow core threads to time out
        allow-core-timeout = on
      }

      # How long time the dispatcher will wait for new actors until it shuts down
      shutdown-timeout = 1s

      # Throughput defines the number of messages that are processed in a batch
      # before the thread is returned to the pool. Set to 1 for as fair as␣
↪possible.
      throughput = 5
```

```
    # Throughput deadline for Dispatcher, set to 0 or negative for no deadline
    throughput-deadline-time = 0ms

    # For BalancingDispatcher: If the balancing dispatcher should attempt to
    # schedule idle actors using the same dispatcher when a message comes in,
    # and the dispatchers ExecutorService is not fully busy already.
    attempt-teamwork = on

    # If this dispatcher requires a specific type of mailbox, specify the
    # fully-qualified class name here; the actually created mailbox will
    # be a subtype of this type. The empty string signifies no requirement.
    mailbox-requirement = ""
  }

  default-mailbox {
    # FQCN of the MailboxType. The Class of the FQCN must have a public
    # constructor with
    # (akka.actor.ActorSystem.Settings, com.typesafe.config.Config) parameters.
    mailbox-type = "akka.dispatch.UnboundedMailbox"

    # If the mailbox is bounded then it uses this setting to determine its
    # capacity. The provided value must be positive.
    # NOTICE:
    # Up to version 2.1 the mailbox type was determined based on this setting;
    # this is no longer the case, the type must explicitly be a bounded mailbox.
    mailbox-capacity = 1000

    # If the mailbox is bounded then this is the timeout for enqueueing
    # in case the mailbox is full. Negative values signify infinite
    # timeout, which should be avoided as it bears the risk of dead-lock.
    mailbox-push-timeout-time = 10s

    # For Actor with Stash: The default capacity of the stash.
    # If negative (or zero) then an unbounded stash is used (default)
    # If positive then a bounded stash is used and the capacity is set using
    # the property
    stash-capacity = -1
  }

  mailbox {
    # Mapping between message queue semantics and mailbox configurations.
    # Used by akka.dispatch.RequiresMessageQueue[T] to enforce different
    # mailbox types on actors.
    # If your Actor implements RequiresMessageQueue[T], then when you create
    # an instance of that actor its mailbox type will be decided by looking
    # up a mailbox configuration via T in this mapping
    requirements {
      "akka.dispatch.UnboundedMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-queue-based
      "akka.dispatch.BoundedMessageQueueSemantics" =
        akka.actor.mailbox.bounded-queue-based
      "akka.dispatch.DequeBasedMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-deque-based
      "akka.dispatch.UnboundedDequeBasedMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-deque-based
      "akka.dispatch.BoundedDequeBasedMessageQueueSemantics" =
        akka.actor.mailbox.bounded-deque-based
      "akka.dispatch.MultipleConsumerSemantics" =
        akka.actor.mailbox.unbounded-queue-based
      "akka.dispatch.ControlAwareMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-control-aware-queue-based
      "akka.dispatch.UnboundedControlAwareMessageQueueSemantics" =
```

```
        akka.actor.mailbox.unbounded-control-aware-queue-based
      "akka.dispatch.BoundedControlAwareMessageQueueSemantics" =
        akka.actor.mailbox.bounded-control-aware-queue-based
      "akka.event.LoggerMessageQueueSemantics" =
        akka.actor.mailbox.logger-queue
    }

    unbounded-queue-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.UnboundedMailbox"
    }

    bounded-queue-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.BoundedMailbox"
    }

    unbounded-deque-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
    }

    bounded-deque-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.BoundedDequeBasedMailbox"
    }

    unbounded-control-aware-queue-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.UnboundedControlAwareMailbox"
    }

    bounded-control-aware-queue-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.BoundedControlAwareMailbox"
    }

    # The LoggerMailbox will drain all messages in the mailbox
    # when the system is shutdown and deliver them to the StandardOutLogger.
    # Do not change this unless you know what you are doing.
    logger-queue {
      mailbox-type = "akka.event.LoggerMailboxType"
    }
  }

  debug {
    # enable function of Actor.loggable(), which is to log any received message
    # at DEBUG level, see the "Testing Actor Systems" section of the Akka
    # Documentation at http://akka.io/docs
    receive = off
```

```
      # enable DEBUG logging of all AutoReceiveMessages (Kill, PoisonPill et.c.)
      autoreceive = off

      # enable DEBUG logging of actor lifecycle changes
      lifecycle = off

      # enable DEBUG logging of all LoggingFSMs for events, transitions and timers
      fsm = off

      # enable DEBUG logging of subscription changes on the eventStream
      event-stream = off

      # enable DEBUG logging of unhandled messages
      unhandled = off

      # enable WARN logging of misconfigured routers
      router-misconfiguration = off
    }

    # Entries for pluggable serializers and their bindings.
    serializers {
      java = "akka.serialization.JavaSerializer"
      bytes = "akka.serialization.ByteArraySerializer"
    }

    # Class to Serializer binding. You only need to specify the name of an
    # interface or abstract base class of the messages. In case of ambiguity it
    # is using the most specific configured class, or giving a warning and
    # choosing the "first" one.
    #
    # To disable one of the default serializers, assign its class to "none", like
    # "java.io.Serializable" = none
    serialization-bindings {
      "[B" = bytes
      "java.io.Serializable" = java
    }

    # Set this to on to enable serialization-bindings define in
    # additional-serialization-bindings. Those are by default not included
    # for backwards compatibility reasons. They are enabled by default if
    # akka.remote.artery.enabled=on.
    enable-additional-serialization-bindings = off

    # Additional serialization-bindings that are replacing Java serialization are
    # defined in this section and not included by default for backwards␣
↪compatibility
    # reasons. They can be enabled with enable-additional-serialization-
↪bindings=on.
    # They are enabled by default if akka.remote.artery.enabled=on.
    additional-serialization-bindings {
    }

    # Log warnings when the default Java serialization is used to serialize␣
↪messages.
    # The default serializer uses Java serialization which is not very performant␣
↪and should not
    # be used in production environments unless you don't care about performance.␣
↪In that case
    # you can turn this off.
    warn-about-java-serializer-usage = on

    # To be used with the above warn-about-java-serializer-usage
    # When warn-about-java-serializer-usage = on, and this warn-on-no-
↪serialization-verification = off,
```

```
    # warnings are suppressed for classes extending␣
↪NoSerializationVerificationNeeded
    # to reduce noize.
    warn-on-no-serialization-verification = on

    # Configuration namespace of serialization identifiers.
    # Each serializer implementation must have an entry in the following format:
    # `akka.actor.serialization-identifiers."FQCN" = ID`
    # where `FQCN` is fully qualified class name of the serializer implementation
    # and `ID` is globally unique serializer identifier number.
    # Identifier values from 0 to 16 are reserved for Akka internal usage.
    serialization-identifiers {
      "akka.serialization.JavaSerializer" = 1
      "akka.serialization.ByteArraySerializer" = 4
    }

    # Configuration items which are used by the akka.actor.ActorDSL._ methods
    dsl {
      # Maximum queue size of the actor created by newInbox(); this protects
      # against faulty programs which use select() and consistently miss messages
      inbox-size = 1000

      # Default timeout to assume for operations like Inbox.receive et al
      default-timeout = 5s
    }
  }

  # Used to set the behavior of the scheduler.
  # Changing the default values may change the system behavior drastically so make
  # sure you know what you're doing! See the Scheduler section of the Akka
  # Documentation for more details.
  scheduler {
    # The LightArrayRevolverScheduler is used as the default scheduler in the
    # system. It does not execute the scheduled tasks on exact time, but on every
    # tick, it will run everything that is (over)due. You can increase or decrease
    # the accuracy of the execution timing by specifying smaller or larger tick
    # duration. If you are scheduling a lot of tasks you should consider increasing
    # the ticks per wheel.
    # Note that it might take up to 1 tick to stop the Timer, so setting the
    # tick-duration to a high value will make shutting down the actor system
    # take longer.
    tick-duration = 10ms

    # The timer uses a circular wheel of buckets to store the timer tasks.
    # This should be set such that the majority of scheduled timeouts (for high
    # scheduling frequency) will be shorter than one rotation of the wheel
    # (ticks-per-wheel * ticks-duration)
    # THIS MUST BE A POWER OF TWO!
    ticks-per-wheel = 512

    # This setting selects the timer implementation which shall be loaded at
    # system start-up.
    # The class given here must implement the akka.actor.Scheduler interface
    # and offer a public constructor which takes three arguments:
    #  1) com.typesafe.config.Config
    #  2) akka.event.LoggingAdapter
    #  3) java.util.concurrent.ThreadFactory
    implementation = akka.actor.LightArrayRevolverScheduler

    # When shutting down the scheduler, there will typically be a thread which
    # needs to be stopped, and this timeout determines how long to wait for
    # that to happen. In case of timeout the shutdown of the actor system will
    # proceed without running possibly still enqueued tasks.
```

```
    shutdown-timeout = 5s
  }

  io {

    # By default the select loops run on dedicated threads, hence using a
    # PinnedDispatcher
    pinned-dispatcher {
      type = "PinnedDispatcher"
      executor = "thread-pool-executor"
      thread-pool-executor.allow-core-timeout = off
    }

    tcp {

      # The number of selectors to stripe the served channels over; each of
      # these will use one select loop on the selector-dispatcher.
      nr-of-selectors = 1

      # Maximum number of open channels supported by this TCP module; there is
      # no intrinsic general limit, this setting is meant to enable DoS
      # protection by limiting the number of concurrently connected clients.
      # Also note that this is a "soft" limit; in certain cases the implementation
      # will accept a few connections more or a few less than the number configured
      # here. Must be an integer > 0 or "unlimited".
      max-channels = 256000

      # When trying to assign a new connection to a selector and the chosen
      # selector is at full capacity, retry selector choosing and assignment
      # this many times before giving up
      selector-association-retries = 10

      # The maximum number of connection that are accepted in one go,
      # higher numbers decrease latency, lower numbers increase fairness on
      # the worker-dispatcher
      batch-accept-limit = 10

      # The number of bytes per direct buffer in the pool used to read or write
      # network data from the kernel.
      direct-buffer-size = 128 KiB

      # The maximal number of direct buffers kept in the direct buffer pool for
      # reuse.
      direct-buffer-pool-limit = 1000

      # The duration a connection actor waits for a `Register` message from
      # its commander before aborting the connection.
      register-timeout = 5s

      # The maximum number of bytes delivered by a `Received` message. Before
      # more data is read from the network the connection actor will try to
      # do other work.
      # The purpose of this setting is to impose a smaller limit than the
      # configured receive buffer size. When using value 'unlimited' it will
      # try to read all from the receive buffer.
      max-received-message-size = unlimited

      # Enable fine grained logging of what goes on inside the implementation.
      # Be aware that this may log more than once per message sent to the actors
      # of the tcp implementation.
      trace-logging = off

      # Fully qualified config path which holds the dispatcher configuration
```

```
      # to be used for running the select() calls in the selectors
      selector-dispatcher = "akka.io.pinned-dispatcher"

      # Fully qualified config path which holds the dispatcher configuration
      # for the read/write worker actors
      worker-dispatcher = "akka.actor.default-dispatcher"

      # Fully qualified config path which holds the dispatcher configuration
      # for the selector management actors
      management-dispatcher = "akka.actor.default-dispatcher"

      # Fully qualified config path which holds the dispatcher configuration
      # on which file IO tasks are scheduled
      file-io-dispatcher = "akka.actor.default-dispatcher"

      # The maximum number of bytes (or "unlimited") to transfer in one batch
      # when using `WriteFile` command which uses `FileChannel.transferTo` to
      # pipe files to a TCP socket. On some OS like Linux `FileChannel.transferTo`
      # may block for a long time when network IO is faster than file IO.
      # Decreasing the value may improve fairness while increasing may improve
      # throughput.
      file-io-transferTo-limit = 512 KiB

      # The number of times to retry the `finishConnect` call after being notified␣
↪about
      # OP_CONNECT. Retries are needed if the OP_CONNECT notification doesn't␣
↪imply that
      # `finishConnect` will succeed, which is the case on Android.
      finish-connect-retries = 5

      # On Windows connection aborts are not reliably detected unless an OP_READ is
      # registered on the selector _after_ the connection has been reset. This
      # workaround enables an OP_CONNECT which forces the abort to be visible on␣
↪Windows.
      # Enabling this setting on other platforms than Windows will cause various␣
↪failures
      # and undefined behavior.
      # Possible values of this key are on, off and auto where auto will enable the
      # workaround if Windows is detected automatically.
      windows-connection-abort-workaround-enabled = off
    }

    udp {

      # The number of selectors to stripe the served channels over; each of
      # these will use one select loop on the selector-dispatcher.
      nr-of-selectors = 1

      # Maximum number of open channels supported by this UDP module Generally
      # UDP does not require a large number of channels, therefore it is
      # recommended to keep this setting low.
      max-channels = 4096

      # The select loop can be used in two modes:
      # - setting "infinite" will select without a timeout, hogging a thread
      # - setting a positive timeout will do a bounded select call,
      #   enabling sharing of a single thread between multiple selectors
      #   (in this case you will have to use a different configuration for the
      #   selector-dispatcher, e.g. using "type=Dispatcher" with size 1)
      # - setting it to zero means polling, i.e. calling selectNow()
      select-timeout = infinite

      # When trying to assign a new connection to a selector and the chosen
```

```
    # selector is at full capacity, retry selector choosing and assignment
    # this many times before giving up
    selector-association-retries = 10

    # The maximum number of datagrams that are read in one go,
    # higher numbers decrease latency, lower numbers increase fairness on
    # the worker-dispatcher
    receive-throughput = 3

    # The number of bytes per direct buffer in the pool used to read or write
    # network data from the kernel.
    direct-buffer-size = 128 KiB

    # The maximal number of direct buffers kept in the direct buffer pool for
    # reuse.
    direct-buffer-pool-limit = 1000

    # Enable fine grained logging of what goes on inside the implementation.
    # Be aware that this may log more than once per message sent to the actors
    # of the tcp implementation.
    trace-logging = off

    # Fully qualified config path which holds the dispatcher configuration
    # to be used for running the select() calls in the selectors
    selector-dispatcher = "akka.io.pinned-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # for the read/write worker actors
    worker-dispatcher = "akka.actor.default-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # for the selector management actors
    management-dispatcher = "akka.actor.default-dispatcher"
  }

  udp-connected {

    # The number of selectors to stripe the served channels over; each of
    # these will use one select loop on the selector-dispatcher.
    nr-of-selectors = 1

    # Maximum number of open channels supported by this UDP module Generally
    # UDP does not require a large number of channels, therefore it is
    # recommended to keep this setting low.
    max-channels = 4096

    # The select loop can be used in two modes:
    # - setting "infinite" will select without a timeout, hogging a thread
    # - setting a positive timeout will do a bounded select call,
    #   enabling sharing of a single thread between multiple selectors
    #   (in this case you will have to use a different configuration for the
    #   selector-dispatcher, e.g. using "type=Dispatcher" with size 1)
    # - setting it to zero means polling, i.e. calling selectNow()
    select-timeout = infinite

    # When trying to assign a new connection to a selector and the chosen
    # selector is at full capacity, retry selector choosing and assignment
    # this many times before giving up
    selector-association-retries = 10

    # The maximum number of datagrams that are read in one go,
    # higher numbers decrease latency, lower numbers increase fairness on
    # the worker-dispatcher
```

```
      receive-throughput = 3

      # The number of bytes per direct buffer in the pool used to read or write
      # network data from the kernel.
      direct-buffer-size = 128 KiB

      # The maximal number of direct buffers kept in the direct buffer pool for
      # reuse.
      direct-buffer-pool-limit = 1000

      # Enable fine grained logging of what goes on inside the implementation.
      # Be aware that this may log more than once per message sent to the actors
      # of the tcp implementation.
      trace-logging = off

      # Fully qualified config path which holds the dispatcher configuration
      # to be used for running the select() calls in the selectors
      selector-dispatcher = "akka.io.pinned-dispatcher"

      # Fully qualified config path which holds the dispatcher configuration
      # for the read/write worker actors
      worker-dispatcher = "akka.actor.default-dispatcher"

      # Fully qualified config path which holds the dispatcher configuration
      # for the selector management actors
      management-dispatcher = "akka.actor.default-dispatcher"
    }

    dns {
      # Fully qualified config path which holds the dispatcher configuration
      # for the manager and resolver router actors.
      # For actual router configuration see akka.actor.deployment./IO-DNS/*
      dispatcher = "akka.actor.default-dispatcher"

      # Name of the subconfig at path akka.io.dns, see inet-address below
      resolver = "inet-address"

      inet-address {
        # Must implement akka.io.DnsProvider
        provider-object = "akka.io.InetAddressDnsProvider"

        # These TTLs are set to default java 6 values
        positive-ttl = 30s
        negative-ttl = 10s

        # How often to sweep out expired cache entries.
        # Note that this interval has nothing to do with TTLs
        cache-cleanup-interval = 120s
      }
    }
  }


}
```

akka-agent

```
#####################################
# Akka Agent Reference Config File #
#####################################
```

```
# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  agent {

    # The dispatcher used for agent-send-off actor
    send-off-dispatcher {
      executor = thread-pool-executor
      type = PinnedDispatcher
    }

    # The dispatcher used for agent-alter-off actor
    alter-off-dispatcher {
      executor = thread-pool-executor
      type = PinnedDispatcher
    }
  }
}
```

akka-camel

```
####################################
# Akka Camel Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  camel {
    # FQCN of the ContextProvider to be used to create or locate a CamelContext
    # it must implement akka.camel.ContextProvider and have a no-arg constructor
    # the built-in default create a fresh DefaultCamelContext
    context-provider = akka.camel.DefaultContextProvider

    # Whether JMX should be enabled or disabled for the Camel Context
    jmx = off
    # enable/disable streaming cache on the Camel Context
    streamingCache = on
    consumer {
      # Configured setting which determines whether one-way communications
      # between an endpoint and this consumer actor
      # should be auto-acknowledged or application-acknowledged.
      # This flag has only effect when exchange is in-only.
      auto-ack = on

      # When endpoint is out-capable (can produce responses) reply-timeout is the
      # maximum time the endpoint can take to send the response before the message
      # exchange fails. This setting is used for out-capable, in-only,
      # manually acknowledged communication.
      reply-timeout = 1m

      # The duration of time to await activation of an endpoint.
      activation-timeout = 10s
    }

    #Scheme to FQCN mappings for CamelMessage body conversions
    conversions {
      "file" = "java.io.InputStream"
    }
```

```
  }
}
```

akka-cluster

```
#######################################
# Akka Cluster Reference Config File #
#######################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {

  cluster {
    # Initial contact points of the cluster.
    # The nodes to join automatically at startup.
    # Comma separated full URIs defined by a string on the form of
    # "akka.tcp://system@hostname:port"
    # Leave as empty if the node is supposed to be joined manually.
    seed-nodes = []

    # how long to wait for one of the seed nodes to reply to initial join request
    seed-node-timeout = 5s

    # If a join request fails it will be retried after this period.
    # Disable join retry by specifying "off".
    retry-unsuccessful-join-after = 10s

    # Should the 'leader' in the cluster be allowed to automatically mark
    # unreachable nodes as DOWN after a configured time of unreachability?
    # Using auto-down implies that two separate clusters will automatically be
    # formed in case of network partition.
    #
    # Don't enable this in production, see 'Auto-downing (DO NOT USE)' section
    # of Akka Cluster documentation.
    #
    # Disable with "off" or specify a duration to enable auto-down.
    # If a downing-provider-class is configured this setting is ignored.
    auto-down-unreachable-after = off

    # Time margin after which shards or singletons that belonged to a downed/
↪removed
    # partition are created in surviving partition. The purpose of this margin is␣
↪that
    # in case of a network partition the persistent actors in the non-surviving␣
↪partitions
    # must be stopped before corresponding persistent actors are started somewhere␣
↪else.
    # This is useful if you implement downing strategies that handle network␣
↪partitions,
    # e.g. by keeping the larger side of the partition and shutting down the␣
↪smaller side.
    # It will not add any extra safety for auto-down-unreachable-after, since that␣
↪is not
    # handling network partitions.
    # Disable with "off" or specify a duration to enable.
    down-removal-margin = off

    # Pluggable support for downing of nodes in the cluster.
    # If this setting is left empty behaviour will depend on 'auto-down-unreachable
↪' in the following ways:
```

```
    # * if it is 'off' the `NoDowning` provider is used and no automatic downing␣
↪will be performed
    # * if it is set to a duration the `AutoDowning` provider is with the␣
↪configured downing duration
    #
    # If specified the value must be the fully qualified class name of a subclass␣
↪of
    # `akka.cluster.DowningProvider` having a public one argument constructor␣
↪accepting an `ActorSystem`
    downing-provider-class = ""

    # Artery only setting
    # When a node has been gracefully removed, let this time pass (to allow for␣
↪example
    # cluster singleton handover to complete) and then quarantine the removed node.
    quarantine-removed-node-after=30s

    # By default, the leader will not move 'Joining' members to 'Up' during a␣
↪network
    # split. This feature allows the leader to accept 'Joining' members to be
↪'WeaklyUp'
    # so they become part of the cluster even during a network split. The leader␣
↪will
    # move 'WeaklyUp' members to 'Up' status once convergence has been reached.␣
↪This
    # feature must be off if some members are running Akka 2.3.X.
    # WeaklyUp is an EXPERIMENTAL feature.
    allow-weakly-up-members = off

    # The roles of this member. List of strings, e.g. roles = ["A", "B"].
    # The roles are part of the membership information and can be used by
    # routers or other services to distribute work to certain member types,
    # e.g. front-end and back-end nodes.
    roles = []

    role {
      # Minimum required number of members of a certain role before the leader
      # changes member status of 'Joining' members to 'Up'. Typically used together
      # with 'Cluster.registerOnMemberUp' to defer some action, such as starting
      # actors, until the cluster has reached a certain size.
      # E.g. to require 2 nodes with role 'frontend' and 3 nodes with role 'backend
↪':
      #   frontend.min-nr-of-members = 2
      #   backend.min-nr-of-members = 3
      #<role-name>.min-nr-of-members = 1
    }

    # Minimum required number of members before the leader changes member status
    # of 'Joining' members to 'Up'. Typically used together with
    # 'Cluster.registerOnMemberUp' to defer some action, such as starting actors,
    # until the cluster has reached a certain size.
    min-nr-of-members = 1

    # Enable/disable info level logging of cluster events
    log-info = on

    # Enable or disable JMX MBeans for management of the cluster
    jmx.enabled = on

    # how long should the node wait before starting the periodic tasks
    # maintenance tasks?
    periodic-tasks-initial-delay = 1s
```

```
    # how often should the node send out gossip information?
    gossip-interval = 1s

    # discard incoming gossip messages if not handled within this duration
    gossip-time-to-live = 2s

    # how often should the leader perform maintenance tasks?
    leader-actions-interval = 1s

    # how often should the node move nodes, marked as unreachable by the failure
    # detector, out of the membership ring?
    unreachable-nodes-reaper-interval = 1s

    # How often the current internal stats should be published.
    # A value of 0s can be used to always publish the stats, when it happens.
    # Disable with "off".
    publish-stats-interval = off

    # The id of the dispatcher to use for cluster actors. If not specified
    # default dispatcher is used.
    # If specified you need to define the settings of the actual dispatcher.
    use-dispatcher = ""

    # Gossip to random node with newer or older state information, if any with
    # this probability. Otherwise Gossip to any random live node.
    # Probability value is between 0.0 and 1.0. 0.0 means never, 1.0 means always.
    gossip-different-view-probability = 0.8

    # Reduced the above probability when the number of nodes in the cluster
    # greater than this value.
    reduce-gossip-different-view-probability = 400

    # Settings for the Phi accrual failure detector (http://www.jaist.ac.jp/~
↪defago/files/pdf/IS_RR_2004_010.pdf
    # [Hayashibara et al]) used by the cluster subsystem to detect unreachable
    # members.
    # The default PhiAccrualFailureDetector will trigger if there are no␣
↪heartbeats within
    # the duration heartbeat-interval + acceptable-heartbeat-pause + threshold_
↪adjustment,
    # i.e. around 5.5 seconds with default settings.
    failure-detector {

      # FQCN of the failure detector implementation.
      # It must implement akka.remote.FailureDetector and have
      # a public constructor with a com.typesafe.config.Config and
      # akka.actor.EventStream parameter.
      implementation-class = "akka.remote.PhiAccrualFailureDetector"

      # How often keep-alive heartbeat messages should be sent to each connection.
      heartbeat-interval = 1 s

      # Defines the failure detector threshold.
      # A low threshold is prone to generate many wrong suspicions but ensures
      # a quick detection in the event of a real crash. Conversely, a high
      # threshold generates fewer mistakes but needs more time to detect
      # actual crashes.
      threshold = 8.0

      # Number of the samples of inter-heartbeat arrival times to adaptively
      # calculate the failure timeout for connections.
      max-sample-size = 1000
```

```
    # Minimum standard deviation to use for the normal distribution in
    # AccrualFailureDetector. Too low standard deviation might result in
    # too much sensitivity for sudden, but normal, deviations in heartbeat
    # inter arrival times.
    min-std-deviation = 100 ms

    # Number of potentially lost/delayed heartbeats that will be
    # accepted before considering it to be an anomaly.
    # This margin is important to be able to survive sudden, occasional,
    # pauses in heartbeat arrivals, due to for example garbage collect or
    # network drop.
    acceptable-heartbeat-pause = 3 s

    # Number of member nodes that each member will send heartbeat messages to,
    # i.e. each node will be monitored by this number of other nodes.
    monitored-by-nr-of-members = 5

    # After the heartbeat request has been sent the first failure detection
    # will start after this period, even though no heartbeat message has
    # been received.
    expected-response-after = 1 s

  }

  metrics {
    # Enable or disable metrics collector for load-balancing nodes.
    enabled = on

    # FQCN of the metrics collector implementation.
    # It must implement akka.cluster.MetricsCollector and
    # have public constructor with akka.actor.ActorSystem parameter.
    # The default SigarMetricsCollector uses JMX and Hyperic SIGAR, if SIGAR
    # is on the classpath, otherwise only JMX.
    collector-class = "akka.cluster.SigarMetricsCollector"

    # How often metrics are sampled on a node.
    # Shorter interval will collect the metrics more often.
    collect-interval = 3s

    # How often a node publishes metrics information.
    gossip-interval = 3s

    # How quickly the exponential weighting of past data is decayed compared to
    # new data. Set lower to increase the bias toward newer values.
    # The relevance of each data sample is halved for every passing half-life
    # duration, i.e. after 4 times the half-life, a data sample's relevance is
    # reduced to 6% of its original relevance. The initial relevance of a data
    # sample is given by 1 - 0.5 ^ (collect-interval / half-life).
    # See http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average
    moving-average-half-life = 12s
  }

  # If the tick-duration of the default scheduler is longer than the
  # tick-duration configured here a dedicated scheduler will be used for
  # periodic tasks of the cluster, otherwise the default scheduler is used.
  # See akka.scheduler settings for more details.
  scheduler {
    tick-duration = 33ms
    ticks-per-wheel = 512
  }

  debug {
    # log heartbeat events (very verbose, useful mostly when debugging␣
↪heartbeating issues)
```

```
    verbose-heartbeat-logging = off
  }

}

# Default configuration for routers
actor.deployment.default {
  # MetricsSelector to use
  # - available: "mix", "heap", "cpu", "load"
  # - or: Fully qualified class name of the MetricsSelector class.
  #       The class must extend akka.cluster.routing.MetricsSelector
  #       and have a public constructor with com.typesafe.config.Config
  #       parameter.
  # - default is "mix"
  metrics-selector = mix
}
actor.deployment.default.cluster {
  # enable cluster aware router that deploys to nodes in the cluster
  enabled = off

  # Maximum number of routees that will be deployed on each cluster
  # member node.
  # Note that max-total-nr-of-instances defines total number of routees, but
  # number of routees per node will not be exceeded, i.e. if you
  # define max-total-nr-of-instances = 50 and max-nr-of-instances-per-node = 2
  # it will deploy 2 routees per new member in the cluster, up to
  # 25 members.
  max-nr-of-instances-per-node = 1

  # Maximum number of routees that will be deployed, in total
  # on all nodes. See also description of max-nr-of-instances-per-node.
  # For backwards compatibility reasons, nr-of-instances
  # has the same purpose as max-total-nr-of-instances for cluster
  # aware routers and nr-of-instances (if defined by user) takes
  # precedence over max-total-nr-of-instances.
  max-total-nr-of-instances = 10000

  # Defines if routees are allowed to be located on the same node as
  # the head router actor, or only on remote nodes.
  # Useful for master-worker scenario where all routees are remote.
  allow-local-routees = on

  # Use members with specified role, or all members if undefined or empty.
  use-role = ""

}

# Protobuf serializer for cluster messages
actor {
  serializers {
    akka-cluster = "akka.cluster.protobuf.ClusterMessageSerializer"
  }

  serialization-bindings {
    "akka.cluster.ClusterMessage" = akka-cluster
  }

  serialization-identifiers {
    "akka.cluster.protobuf.ClusterMessageSerializer" = 5
  }

  router.type-mapping {
    adaptive-pool = "akka.cluster.routing.AdaptiveLoadBalancingPool"
```

```
      adaptive-group = "akka.cluster.routing.AdaptiveLoadBalancingGroup"
    }
  }

}
```

akka-multi-node-testkit

```
############################################
# Akka Remote Testing Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  testconductor {

    # Timeout for joining a barrier: this is the maximum time any participants
    # waits for everybody else to join a named barrier.
    barrier-timeout = 30s

    # Timeout for interrogation of TestConductor's Controller actor
    query-timeout = 10s

    # Threshold for packet size in time unit above which the failure injector will
    # split the packet and deliver in smaller portions; do not give value smaller
    # than HashedWheelTimer resolution (would not make sense)
    packet-split-threshold = 100ms

    # amount of time for the ClientFSM to wait for the connection to the conductor
    # to be successful
    connect-timeout = 20s

    # Number of connect attempts to be made to the conductor controller
    client-reconnects = 30

    # minimum time interval which is to be inserted between reconnect attempts
    reconnect-backoff = 1s

    netty {
      # (I&O) Used to configure the number of I/O worker threads on server sockets
      server-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 1

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 1.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 2
      }

      # (I&O) Used to configure the number of I/O worker threads on client sockets
      client-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 1
```

```
        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 1.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 2
      }
    }
  }
}
```

akka-persistence

```
#####################################################################
# Akka Persistence Extension Reference Configuration File #
#####################################################################

# This is the reference config file that contains all the default settings.
# Make your edits in your application.conf in order to override these settings.

# Directory of persistence journal and snapshot store plugins is available at the
# Akka Community Projects page http://akka.io/community/

# Default persistence extension settings.
akka.persistence {
    # Fully qualified class name providing a default internal stash overflow␣
↪strategy.
    # It needs to be a subclass of akka.persistence.
↪StashOverflowStrategyConfigurator.
    # The default strategy throws StashOverflowException.
    internal-stash-overflow-strategy = "akka.persistence.ThrowExceptionConfigurator
↪"
    journal {
        # Absolute path to the journal plugin configuration entry used by
        # persistent actor or view by default.
        # Persistent actor or view can override `journalPluginId` method
        # in order to rely on a different journal plugin.
        plugin = ""
        # List of journal plugins to start automatically. Use "" for the default␣
↪journal plugin.
        auto-start-journals = []
    }
    snapshot-store {
        # Absolute path to the snapshot plugin configuration entry used by
        # persistent actor or view by default.
        # Persistent actor or view can override `snapshotPluginId` method
        # in order to rely on a different snapshot plugin.
        # It is not mandatory to specify a snapshot store plugin.
        # If you don't use snapshots you don't have to configure it.
        # Note that Cluster Sharding is using snapshots, so if you
        # use Cluster Sharding you need to define a snapshot store plugin.
        plugin = ""
        # List of snapshot stores to start automatically. Use "" for the default␣
↪snapshot store.
        auto-start-snapshot-stores = []
    }
    # used as default-snapshot store if no plugin configured
    # (see `akka.persistence.snapshot-store`)
    no-snapshot-store {
```

```
      class = "akka.persistence.snapshot.NoSnapshotStore"
    }
    # Default persistent view settings.
    view {
        # Automated incremental view update.
        auto-update = on
        # Interval between incremental updates.
        auto-update-interval = 5s
        # Maximum number of messages to replay per incremental view update.
        # Set to -1 for no upper limit.
        auto-update-replay-max = -1
    }
    # Default reliable delivery settings.
    at-least-once-delivery {
        # Interval between re-delivery attempts.
        redeliver-interval = 5s
        # Maximum number of unconfirmed messages that will be sent in one
        # re-delivery burst.
        redelivery-burst-limit = 10000
        # After this number of delivery attempts a
        # `ReliableRedelivery.UnconfirmedWarning`, message will be sent to the␣
↪actor.
        warn-after-number-of-unconfirmed-attempts = 5
        # Maximum number of unconfirmed messages that an actor with
        # AtLeastOnceDelivery is allowed to hold in memory.
        max-unconfirmed-messages = 100000
    }
    # Default persistent extension thread pools.
    dispatchers {
        # Dispatcher used by every plugin which does not declare explicit
        # `plugin-dispatcher` field.
        default-plugin-dispatcher {
            type = PinnedDispatcher
            executor = "thread-pool-executor"
        }
        # Default dispatcher for message replay.
        default-replay-dispatcher {
            type = Dispatcher
            executor = "fork-join-executor"
            fork-join-executor {
                parallelism-min = 2
                parallelism-max = 8
            }
        }
        # Default dispatcher for streaming snapshot IO
        default-stream-dispatcher {
            type = Dispatcher
            executor = "fork-join-executor"
            fork-join-executor {
                parallelism-min = 2
                parallelism-max = 8
            }
        }
    }

    # Fallback settings for journal plugin configurations.
    # These settings are used if they are not defined in plugin config section.
    journal-plugin-fallback {

      # Fully qualified class name providing journal plugin api implementation.
      # It is mandatory to specify this property.
      # The class must have a constructor without parameters or constructor with
      # one `com.typesafe.config.Config` parameter.
```

```
      class = ""

      # Dispatcher for the plugin actor.
      plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"

      # Dispatcher for message replay.
      replay-dispatcher = "akka.persistence.dispatchers.default-replay-dispatcher"

      # Removed: used to be the Maximum size of a persistent message batch written␣
↪to the journal.
      # Now this setting is without function, PersistentActor will write as many␣
↪messages
      # as it has accumulated since the last write.
      max-message-batch-size = 200

      # If there is more time in between individual events gotten from the journal
      # recovery than this the recovery will fail.
      # Note that it also affects reading the snapshot before replaying events on
      # top of it, even though it is configured for the journal.
      recovery-event-timeout = 30s

      circuit-breaker {
        max-failures = 10
        call-timeout = 10s
        reset-timeout = 30s
      }

      # The replay filter can detect a corrupt event stream by inspecting
      # sequence numbers and writerUuid when replaying events.
      replay-filter {
        # What the filter should do when detecting invalid events.
        # Supported values:
        # `repair-by-discard-old` : discard events from old writers,
        #                           warning is logged
        # `fail` : fail the replay, error is logged
        # `warn` : log warning but emit events untouched
        # `off` : disable this feature completely
        mode = repair-by-discard-old

        # It uses a look ahead buffer for analyzing the events.
        # This defines the size (in number of events) of the buffer.
        window-size = 100

        # How many old writerUuid to remember
        max-old-writers = 10

        # Set this to `on` to enable detailed debug logging of each
        # replayed event.
        debug = off
      }
    }

    # Fallback settings for snapshot store plugin configurations
    # These settings are used if they are not defined in plugin config section.
    snapshot-store-plugin-fallback {

      # Fully qualified class name providing snapshot store plugin api
      # implementation. It is mandatory to specify this property if
      # snapshot store is enabled.
      # The class must have a constructor without parameters or constructor with
      # one `com.typesafe.config.Config` parameter.
      class = ""
```

```
      # Dispatcher for the plugin actor.
      plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"

      circuit-breaker {
        max-failures = 5
        call-timeout = 20s
        reset-timeout = 60s
      }
    }
}

# Protobuf serialization for the persistent extension messages.
akka.actor {
    serializers {
        akka-persistence-message = "akka.persistence.serialization.
↪MessageSerializer"
        akka-persistence-snapshot = "akka.persistence.serialization.
↪SnapshotSerializer"
    }
    serialization-bindings {
        "akka.persistence.serialization.Message" = akka-persistence-message
        "akka.persistence.serialization.Snapshot" = akka-persistence-snapshot
    }
    serialization-identifiers {
        "akka.persistence.serialization.MessageSerializer" = 7
        "akka.persistence.serialization.SnapshotSerializer" = 8
    }
}


####################################################
# Persistence plugins included with the extension #
####################################################

# In-memory journal plugin.
akka.persistence.journal.inmem {
    # Class name of the plugin.
    class = "akka.persistence.journal.inmem.InmemJournal"
    # Dispatcher for the plugin actor.
    plugin-dispatcher = "akka.actor.default-dispatcher"
}

# Local file system snapshot store plugin.
akka.persistence.snapshot-store.local {
    # Class name of the plugin.
    class = "akka.persistence.snapshot.local.LocalSnapshotStore"
    # Dispatcher for the plugin actor.
    plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
    # Dispatcher for streaming snapshot IO.
    stream-dispatcher = "akka.persistence.dispatchers.default-stream-dispatcher"
    # Storage location of snapshot files.
    dir = "snapshots"
    # Number load attempts when recovering from the latest snapshot fails
    # yet older snapshot files are available. Each recovery attempt will try
    # to recover using an older than previously failed-on snapshot file
    # (if any are present). If all attempts fail the recovery will fail and
    # the persistent actor will be stopped.
    max-load-attempts = 3
}

# LevelDB journal plugin.
# Note: this plugin requires explicit LevelDB dependency, see below.
akka.persistence.journal.leveldb {
```

```
    # Class name of the plugin.
    class = "akka.persistence.journal.leveldb.LeveldbJournal"
    # Dispatcher for the plugin actor.
    plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
    # Dispatcher for message replay.
    replay-dispatcher = "akka.persistence.dispatchers.default-replay-dispatcher"
    # Storage location of LevelDB files.
    dir = "journal"
    # Use fsync on write.
    fsync = on
    # Verify checksum on read.
    checksum = off
    # Native LevelDB (via JNI) or LevelDB Java port.
    native = on
}

# Shared LevelDB journal plugin (for testing only).
# Note: this plugin requires explicit LevelDB dependency, see below.
akka.persistence.journal.leveldb-shared {
    # Class name of the plugin.
    class = "akka.persistence.journal.leveldb.SharedLeveldbJournal"
    # Dispatcher for the plugin actor.
    plugin-dispatcher = "akka.actor.default-dispatcher"
    # Timeout for async journal operations.
    timeout = 10s
    store {
        # Dispatcher for shared store actor.
        store-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
        # Dispatcher for message replay.
        replay-dispatcher = "akka.persistence.dispatchers.default-replay-dispatcher
↪"
        # Storage location of LevelDB files.
        dir = "journal"
        # Use fsync on write.
        fsync = on
        # Verify checksum on read.
        checksum = off
        # Native LevelDB (via JNI) or LevelDB Java port.
        native = on
    }
}

akka.persistence.journal.proxy {
  # Class name of the plugin.
  class = "akka.persistence.journal.PersistencePluginProxy"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.actor.default-dispatcher"
  # Set this to on in the configuration of the ActorSystem
  # that will host the target journal
  start-target-journal = off
  # The journal plugin config path to use for the target journal
  target-journal-plugin = ""
  # The address of the proxy to connect to from other nodes. Optional setting.
  target-journal-address = ""
  # Initialization timeout of target lookup
  init-timeout = 10s
}

akka.persistence.snapshot-store.proxy {
  # Class name of the plugin.
  class = "akka.persistence.journal.PersistencePluginProxy"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.actor.default-dispatcher"
```

```
  # Set this to on in the configuration of the ActorSystem
  # that will host the target snapshot-store
  start-target-snapshot-store = off
  # The journal plugin config path to use for the target snapshot-store
  target-snapshot-store-plugin = ""
  # The address of the proxy to connect to from other nodes. Optional setting.
  target-snapshot-store-address = ""
  # Initialization timeout of target lookup
  init-timeout = 10s
}

# LevelDB persistence requires the following dependency declarations:
#
# SBT:
#       "org.iq80.leveldb"            % "leveldb"          % "0.7"
#       "org.fusesource.leveldbjni"   % "leveldbjni-all"   % "1.8"
#
# Maven:
#        <dependency>
#            <groupId>org.iq80.leveldb</groupId>
#            <artifactId>leveldb</artifactId>
#            <version>0.7</version>
#        </dependency>
#        <dependency>
#            <groupId>org.fusesource.leveldbjni</groupId>
#            <artifactId>leveldbjni-all</artifactId>
#            <version>1.8</version>
#        </dependency>
```

akka-remote

```
####################################
# Akka Remote Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# comments about akka.actor settings left out where they are already in akka-
# actor.jar, because otherwise they would be repeated in config rendering.
#
# For the configuration of the new remoting implementation (Artery) please look
# at the bottom section of this file as it is listed separately.

akka {

  actor {

    serializers {
      akka-containers = "akka.remote.serialization.MessageContainerSerializer"
      akka-misc = "akka.remote.serialization.MiscMessageSerializer"
      artery = "akka.remote.serialization.ArteryMessageSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      daemon-create = "akka.remote.serialization.DaemonMsgCreateSerializer"
      primitive-long = "akka.remote.serialization.LongSerializer"
      primitive-int = "akka.remote.serialization.IntSerializer"
      primitive-string = "akka.remote.serialization.StringSerializer"
      primitive-bytestring = "akka.remote.serialization.ByteStringSerializer"
      akka-system-msg = "akka.remote.serialization.SystemMessageSerializer"
    }
```

```
    serialization-bindings {
      "akka.actor.ActorSelectionMessage" = akka-containers

      "akka.remote.DaemonMsgCreate" = daemon-create

      "akka.remote.artery.ArteryMessage" = artery

      # Since akka.protobuf.Message does not extend Serializable but
      # GeneratedMessage does, need to use the more specific one here in order
      # to avoid ambiguity.
      "akka.protobuf.GeneratedMessage" = proto

      # Since com.google.protobuf.Message does not extend Serializable but
      # GeneratedMessage does, need to use the more specific one here in order
      # to avoid ambiguity.
      # This com.google.protobuf serialization binding is only used if the class␣
↪can be loaded,
      # i.e. com.google.protobuf dependency has been added in the application␣
↪project.
      "com.google.protobuf.GeneratedMessage" = proto
    }

    # For the purpose of preserving protocol backward compatibility these bindings␣
↪are not
    # included by default. They can be enabled with enable-additional-
↪serialization-bindings=on.
    # They are enabled by default if akka.remote.artery.enabled=on.
    additional-serialization-bindings {
      "akka.actor.Identify" = akka-misc
      "akka.actor.ActorIdentity" = akka-misc
      "scala.Some" = akka-misc
      "scala.None$" = akka-misc
      "akka.actor.Status$Success" = akka-misc
      "akka.actor.Status$Failure" = akka-misc
      "akka.actor.ActorRef" = akka-misc
      "akka.actor.PoisonPill$" = akka-misc
      "akka.actor.Kill$" = akka-misc
      "akka.remote.RemoteWatcher$Heartbeat$" = akka-misc
      "akka.remote.RemoteWatcher$HeartbeatRsp" = akka-misc
      "akka.actor.ActorInitializationException" = akka-misc

      "akka.dispatch.sysmsg.SystemMessage" = akka-system-msg

      "java.lang.String" = primitive-string
      "akka.util.ByteString$ByteString1C" = primitive-bytestring
      "akka.util.ByteString$ByteString1" = primitive-bytestring
      "akka.util.ByteString$ByteStrings" = primitive-bytestring
      "java.lang.Long" = primitive-long
      "scala.Long" = primitive-long
      "java.lang.Integer" = primitive-int
      "scala.Int" = primitive-int

      # Java Serializer is by default used for exceptions.
      # It's recommended that you implement custom serializer for exceptions that␣
↪are
      # sent remotely, e.g. in akka.actor.Status.Failure for ask replies. You can␣
↪add
      # binding to akka-misc (MiscMessageSerializerSpec) for the exceptions that␣
↪have
      # a constructor with single message String or constructor with message␣
↪String as
      # first parameter and cause Throwable as second parameter. Note that it's not
      # safe to add this binding for general exceptions such as␣
↪IllegalArgumentException
```

```
      # because it may have a subclass without required constructor.
      "java.lang.Throwable" = java
      "akka.actor.IllegalActorStateException" = akka-misc
      "akka.actor.ActorKilledException" = akka-misc
      "akka.actor.InvalidActorNameException" = akka-misc
      "akka.actor.InvalidMessageException" = akka-misc
    }

    serialization-identifiers {
      "akka.remote.serialization.ProtobufSerializer" = 2
      "akka.remote.serialization.DaemonMsgCreateSerializer" = 3
      "akka.remote.serialization.MessageContainerSerializer" = 6
      "akka.remote.serialization.MiscMessageSerializer" = 16
      "akka.remote.serialization.ArteryMessageSerializer" = 17
      "akka.remote.serialization.LongSerializer" = 18
      "akka.remote.serialization.IntSerializer" = 19
      "akka.remote.serialization.StringSerializer" = 20
      "akka.remote.serialization.ByteStringSerializer" = 21
      "akka.remote.serialization.SystemMessageSerializer" = 22
    }

    deployment {

      default {

        # if this is set to a valid remote address, the named actor will be
        # deployed at that node e.g. "akka.tcp://sys@host:port"
        remote = ""

        target {

          # A list of hostnames and ports for instantiating the children of a
          # router
          #   The format should be on "akka.tcp://sys@host:port", where:
          #    - sys is the remote actor system name
          #    - hostname can be either hostname or IP address the remote actor
          #      should connect to
          #    - port should be the port for the remote server on the other node
          # The number of actor instances to be spawned is still taken from the
          # nr-of-instances setting as for local routers; the instances will be
          # distributed round-robin among the given nodes.
          nodes = []

        }
      }
    }
  }

  remote {
    ### Settings shared by classic remoting and Artery (the new implementation of␣
↪remoting)

    # If set to a nonempty string remoting will use the given dispatcher for
    # its internal actors otherwise the default dispatcher is used. Please note
    # that since remoting can load arbitrary 3rd party drivers (see
    # "enabled-transport" and "adapters" entries) it is not guaranteed that
    # every module will respect this setting.
    use-dispatcher = "akka.remote.default-remote-dispatcher"

    # Settings for the failure detector to monitor connections.
    # For TCP it is not important to have fast failure detection, since
    # most connection failures are captured by TCP itself.
    # The default DeadlineFailureDetector will trigger if there are no heartbeats␣
↪within
```

```
    # the duration heartbeat-interval + acceptable-heartbeat-pause, i.e. 20 seconds
    # with the default settings.
    transport-failure-detector {

      # FQCN of the failure detector implementation.
      # It must implement akka.remote.FailureDetector and have
      # a public constructor with a com.typesafe.config.Config and
      # akka.actor.EventStream parameter.
      implementation-class = "akka.remote.DeadlineFailureDetector"

      # How often keep-alive heartbeat messages should be sent to each connection.
      heartbeat-interval = 4 s

      # Number of potentially lost/delayed heartbeats that will be
      # accepted before considering it to be an anomaly.
      # A margin to the `heartbeat-interval` is important to be able to survive␣
↪sudden,
      # occasional, pauses in heartbeat arrivals, due to for example garbage␣
↪collect or
      # network drop.
      acceptable-heartbeat-pause = 16 s
    }

    # Settings for the Phi accrual failure detector (http://www.jaist.ac.jp/~
↪defago/files/pdf/IS_RR_2004_010.pdf
    # [Hayashibara et al]) used for remote death watch.
    # The default PhiAccrualFailureDetector will trigger if there are no␣
↪heartbeats within
    # the duration heartbeat-interval + acceptable-heartbeat-pause + threshold_
↪adjustment,
    # i.e. around 12.5 seconds with default settings.
    watch-failure-detector {

      # FQCN of the failure detector implementation.
      # It must implement akka.remote.FailureDetector and have
      # a public constructor with a com.typesafe.config.Config and
      # akka.actor.EventStream parameter.
      implementation-class = "akka.remote.PhiAccrualFailureDetector"

      # How often keep-alive heartbeat messages should be sent to each connection.
      heartbeat-interval = 1 s

      # Defines the failure detector threshold.
      # A low threshold is prone to generate many wrong suspicions but ensures
      # a quick detection in the event of a real crash. Conversely, a high
      # threshold generates fewer mistakes but needs more time to detect
      # actual crashes.
      threshold = 10.0

      # Number of the samples of inter-heartbeat arrival times to adaptively
      # calculate the failure timeout for connections.
      max-sample-size = 200

      # Minimum standard deviation to use for the normal distribution in
      # AccrualFailureDetector. Too low standard deviation might result in
      # too much sensitivity for sudden, but normal, deviations in heartbeat
      # inter arrival times.
      min-std-deviation = 100 ms

      # Number of potentially lost/delayed heartbeats that will be
      # accepted before considering it to be an anomaly.
      # This margin is important to be able to survive sudden, occasional,
      # pauses in heartbeat arrivals, due to for example garbage collect or
```

```
      # network drop.
      acceptable-heartbeat-pause = 10 s


      # How often to check for nodes marked as unreachable by the failure
      # detector
      unreachable-nodes-reaper-interval = 1s

      # After the heartbeat request has been sent the first failure detection
      # will start after this period, even though no heartbeat mesage has
      # been received.
      expected-response-after = 1 s

    }

    # remote deployment configuration section
    deployment {
      # If true, will only allow specific classes to be instanciated on this␣
↪system via remote deployment
      enable-whitelist = off

      whitelist = []
    }

    ### Configuration for classic remoting

    # Timeout after which the startup of the remoting subsystem is considered
    # to be failed. Increase this value if your transport drivers (see the
    # enabled-transports section) need longer time to be loaded.
    startup-timeout = 10 s

    # Timout after which the graceful shutdown of the remoting subsystem is
    # considered to be failed. After the timeout the remoting system is
    # forcefully shut down. Increase this value if your transport drivers
    # (see the enabled-transports section) need longer time to stop properly.
    shutdown-timeout = 10 s

    # Before shutting down the drivers, the remoting subsystem attempts to flush
    # all pending writes. This setting controls the maximum time the remoting is
    # willing to wait before moving on to shut down the drivers.
    flush-wait-on-shutdown = 2 s

    # Reuse inbound connections for outbound messages
    use-passive-connections = on

    # Controls the backoff interval after a refused write is reattempted.
    # (Transports may refuse writes if their internal buffer is full)
    backoff-interval = 5 ms

    # Acknowledgment timeout of management commands sent to the transport stack.
    command-ack-timeout = 30 s

    # The timeout for outbound associations to perform the handshake.
    # If the transport is akka.remote.netty.tcp or akka.remote.netty.ssl
    # the configured connection-timeout for the transport will be used instead.
    handshake-timeout = 15 s

    ### Security settings

    # Enable untrusted mode for full security of server managed actors, prevents
    # system messages to be send by clients, e.g. messages like 'Create',
    # 'Suspend', 'Resume', 'Terminate', 'Supervise', 'Link' etc.
    untrusted-mode = off
```

```
    # When 'untrusted-mode=on' inbound actor selections are by default discarded.
    # Actors with paths defined in this white list are granted permission to␣
↪receive actor
    # selections messages.
    # E.g. trusted-selection-paths = ["/user/receptionist", "/user/namingService"]
    trusted-selection-paths = []

    # Should the remote server require that its peers share the same
    # secure-cookie (defined in the 'remote' section)? Secure cookies are passed
    # between during the initial handshake. Connections are refused if the initial
    # message contains a mismatching cookie or the cookie is missing.
    require-cookie = off

    # Deprecated since 2.4-M1
    secure-cookie = ""

    ### Logging

    # If this is "on", Akka will log all inbound messages at DEBUG level,
    # if off then they are not logged
    log-received-messages = off

    # If this is "on", Akka will log all outbound messages at DEBUG level,
    # if off then they are not logged
    log-sent-messages = off

    # Sets the log granularity level at which Akka logs remoting events. This␣
↪setting
    # can take the values OFF, ERROR, WARNING, INFO, DEBUG, or ON. For␣
↪compatibility
    # reasons the setting "on" will default to "debug" level. Please note that the␣
↪effective
    # logging level is still determined by the global logging level of the actor␣
↪system:
    # for example debug level remoting events will be only logged if the system
    # is running with debug level logging.
    # Failures to deserialize received messages also fall under this flag.
    log-remote-lifecycle-events = on

    # Logging of message types with payload size in bytes larger than
    # this value. Maximum detected size per message type is logged once,
    # with an increase threshold of 10%.
    # By default this feature is turned off. Activate it by setting the property to
    # a value in bytes, such as 1000b. Note that for all messages larger than this
    # limit there will be extra performance and scalability cost.
    log-frame-size-exceeding = off

    # Log warning if the number of messages in the backoff buffer in the endpoint
    # writer exceeds this limit. It can be disabled by setting the value to off.
    log-buffer-size-exceeding = 50000



    # After failed to establish an outbound connection, the remoting will mark the
    # address as failed. This configuration option controls how much time should
    # be elapsed before reattempting a new connection. While the address is
    # gated, all messages sent to the address are delivered to dead-letters.
    # Since this setting limits the rate of reconnects setting it to a
    # very short interval (i.e. less than a second) may result in a storm of
    # reconnect attempts.
    retry-gate-closed-for = 5 s
```

```
    # After catastrophic communication failures that result in the loss of system
    # messages or after the remote DeathWatch triggers the remote system gets
    # quarantined to prevent inconsistent behavior.
    # This setting controls how long the Quarantine marker will be kept around
    # before being removed to avoid long-term memory leaks.
    # WARNING: DO NOT change this to a small value to re-enable communication with
    # quarantined nodes. Such feature is not supported and any behavior between
    # the affected systems after lifting the quarantine is undefined.
    prune-quarantine-marker-after = 5 d

    # If system messages have been exchanged between two systems (i.e. remote death
    # watch or remote deployment has been used) a remote system will be marked as
    # quarantined after the two system has no active association, and no
    # communication happens during the time configured here.
    # The only purpose of this setting is to avoid storing system message␣
↪redelivery
    # data (sequence number state, etc.) for an undefined amount of time leading␣
↪to long
    # term memory leak. Instead, if a system has been gone for this period,
    # or more exactly
    # - there is no association between the two systems (TCP connection, if TCP␣
↪transport is used)
    # - neither side has been attempting to communicate with the other
    # - there are no pending system messages to deliver
    # for the amount of time configured here, the remote system will be␣
↪quarantined and all state
    # associated with it will be dropped.
    quarantine-after-silence = 5 d

    # This setting defines the maximum number of unacknowledged system messages
    # allowed for a remote system. If this limit is reached the remote system is
    # declared to be dead and its UID marked as tainted.
    system-message-buffer-size = 20000

    # This setting defines the maximum idle time after an individual
    # acknowledgement for system messages is sent. System message delivery
    # is guaranteed by explicit acknowledgement messages. These acks are
    # piggybacked on ordinary traffic messages. If no traffic is detected
    # during the time period configured here, the remoting will send out
    # an individual ack.
    system-message-ack-piggyback-timeout = 0.3 s

    # This setting defines the time after internal management signals
    # between actors (used for DeathWatch and supervision) that have not been
    # explicitly acknowledged or negatively acknowledged are resent.
    # Messages that were negatively acknowledged are always immediately
    # resent.
    resend-interval = 2 s

    # Maximum number of unacknowledged system messages that will be resent
    # each 'resend-interval'. If you watch many (> 1000) remote actors you can
    # increase this value to for example 600, but a too large limit (e.g. 10000)
    # may flood the connection and might cause false failure detection to trigger.
    # Test such a configuration by watching all actors at the same time and stop
    # all watched actors at the same time.
    resend-limit = 200

    # WARNING: this setting should not be not changed unless all of its␣
↪consequences
    # are properly understood which assumes experience with remoting internals
    # or expert advice.
    # This setting defines the time after redelivery attempts of internal␣
↪management
```

```
    # signals are stopped to a remote system that has been not confirmed to be␣
↪alive by
    # this system before.
    initial-system-message-delivery-timeout = 3 m

    ### Transports and adapters

    # List of the transport drivers that will be loaded by the remoting.
    # A list of fully qualified config paths must be provided where
    # the given configuration path contains a transport-class key
    # pointing to an implementation class of the Transport interface.
    # If multiple transports are provided, the address of the first
    # one will be used as a default address.
    enabled-transports = ["akka.remote.netty.tcp"]

    # Transport drivers can be augmented with adapters by adding their
    # name to the applied-adapters setting in the configuration of a
    # transport. The available adapters should be configured in this
    # section by providing a name, and the fully qualified name of
    # their corresponding implementation. The class given here
    # must implement akka.akka.remote.transport.TransportAdapterProvider
    # and have public constructor without parameters.
    adapters {
      gremlin = "akka.remote.transport.FailureInjectorProvider"
      trttl = "akka.remote.transport.ThrottlerProvider"
    }

    ### Default configuration for the Netty based transport drivers

    netty.tcp {
      # The class given here must implement the akka.remote.transport.Transport
      # interface and offer a public constructor which takes two arguments:
      #  1) akka.actor.ExtendedActorSystem
      #  2) com.typesafe.config.Config
      transport-class = "akka.remote.transport.netty.NettyTransport"

      # Transport drivers can be augmented with adapters by adding their
      # name to the applied-adapters list. The last adapter in the
      # list is the adapter immediately above the driver, while
      # the first one is the top of the stack below the standard
      # Akka protocol
      applied-adapters = []

      transport-protocol = tcp

      # The default remote server port clients should connect to.
      # Default is 2552 (AKKA), use 0 if you want a random available port
      # This port needs to be unique for each actor system on the same machine.
      port = 2552

      # The hostname or ip clients should connect to.
      # InetAddress.getLocalHost.getHostAddress is used if empty
      hostname = ""

      # Use this setting to bind a network interface to a different port
      # than remoting protocol expects messages at. This may be used
      # when running akka nodes in a separated networks (under NATs or docker␣
↪containers).
      # Use 0 if you want a random available port. Examples:
      #
      # akka.remote.netty.tcp.port = 2552
      # akka.remote.netty.tcp.bind-port = 2553
      # Network interface will be bound to the 2553 port, but remoting protocol␣
↪will
```

```
      # expect messages sent to port 2552.
      #
      # akka.remote.netty.tcp.port = 0
      # akka.remote.netty.tcp.bind-port = 0
      # Network interface will be bound to a random port, and remoting protocol␣
↪will
      # expect messages sent to the bound port.
      #
      # akka.remote.netty.tcp.port = 2552
      # akka.remote.netty.tcp.bind-port = 0
      # Network interface will be bound to a random port, but remoting protocol␣
↪will
      # expect messages sent to port 2552.
      #
      # akka.remote.netty.tcp.port = 0
      # akka.remote.netty.tcp.bind-port = 2553
      # Network interface will be bound to the 2553 port, and remoting protocol␣
↪will
      # expect messages sent to the bound port.
      #
      # akka.remote.netty.tcp.port = 2552
      # akka.remote.netty.tcp.bind-port = ""
      # Network interface will be bound to the 2552 port, and remoting protocol␣
↪will
      # expect messages sent to the bound port.
      #
      # akka.remote.netty.tcp.port if empty
      bind-port = ""

      # Use this setting to bind a network interface to a different hostname or ip
      # than remoting protocol expects messages at.
      # Use "0.0.0.0" to bind to all interfaces.
      # akka.remote.netty.tcp.hostname if empty
      bind-hostname = ""

      # Enables SSL support on this transport
      enable-ssl = false

      # Sets the connectTimeoutMillis of all outbound connections,
      # i.e. how long a connect may take until it is timed out
      connection-timeout = 15 s

      # If set to "<id.of.dispatcher>" then the specified dispatcher
      # will be used to accept inbound connections, and perform IO. If "" then
      # dedicated threads will be used.
      # Please note that the Netty driver only uses this configuration and does
      # not read the "akka.remote.use-dispatcher" entry. Instead it has to be
      # configured manually to point to the same dispatcher if needed.
      use-dispatcher-for-io = ""

      # Sets the high water mark for the in and outbound sockets,
      # set to 0b for platform default
      write-buffer-high-water-mark = 0b

      # Sets the low water mark for the in and outbound sockets,
      # set to 0b for platform default
      write-buffer-low-water-mark = 0b

      # Sets the send buffer size of the Sockets,
      # set to 0b for platform default
      send-buffer-size = 256000b

      # Sets the receive buffer size of the Sockets,
```

```
      # set to 0b for platform default
      receive-buffer-size = 256000b

      # Maximum message size the transport will accept, but at least
      # 32000 bytes.
      # Please note that UDP does not support arbitrary large datagrams,
      # so this setting has to be chosen carefully when using UDP.
      # Both send-buffer-size and receive-buffer-size settings has to
      # be adjusted to be able to buffer messages of maximum size.
      maximum-frame-size = 128000b

      # Sets the size of the connection backlog
      backlog = 4096

      # Enables the TCP_NODELAY flag, i.e. disables Nagle's algorithm
      tcp-nodelay = on

      # Enables TCP Keepalive, subject to the O/S kernel's configuration
      tcp-keepalive = on

      # Enables SO_REUSEADDR, which determines when an ActorSystem can open
      # the specified listen port (the meaning differs between *nix and Windows)
      # Valid values are "on", "off" and "off-for-windows"
      # due to the following Windows bug: http://bugs.sun.com/bugdatabase/view_bug.
↪do?bug_id=4476378
      # "off-for-windows" of course means that it's "on" for all other platforms
      tcp-reuse-addr = off-for-windows

      # Used to configure the number of I/O worker threads on server sockets
      server-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 2

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 1.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 2
      }

      # Used to configure the number of I/O worker threads on client sockets
      client-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 2

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 1.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 2
      }


    }

    netty.udp = ${akka.remote.netty.tcp}
    netty.udp {
```

```
      transport-protocol = udp
    }

    netty.ssl = ${akka.remote.netty.tcp}
    netty.ssl = {
      # Enable SSL/TLS encryption.
      # This must be enabled on both the client and server to work.
      enable-ssl = true

      security {
        # This is the Java Key Store used by the server connection
        key-store = "keystore"

        # This password is used for decrypting the key store
        key-store-password = "changeme"

        # This password is used for decrypting the key
        key-password = "changeme"

        # This is the Java Key Store used by the client connection
        trust-store = "truststore"

        # This password is used for decrypting the trust store
        trust-store-password = "changeme"

        # Protocol to use for SSL encryption, choose from:
        # TLS 1.2 is available since JDK7, and default since JDK8:
        # https://blogs.oracle.com/java-platform-group/entry/java_8_will_use_tls
        protocol = "TLSv1.2"

        # Example: ["TLS_RSA_WITH_AES_128_CBC_SHA", "TLS_RSA_WITH_AES_256_CBC_SHA"]
        # You need to install the JCE Unlimited Strength Jurisdiction Policy
        # Files to use AES 256.
        # More info here:
        # http://docs.oracle.com/javase/7/docs/technotes/guides/security/
↪SunProviders.html#SunJCEProvider
        enabled-algorithms = ["TLS_RSA_WITH_AES_128_CBC_SHA"]

        # There are three options, in increasing order of security:
        # "" or SecureRandom => (default)
        # "SHA1PRNG" => Can be slow because of blocking issues on Linux
        # "AES128CounterSecureRNG" => fastest startup and based on AES encryption
        # algorithm
        # "AES256CounterSecureRNG"
        #
        # The following are deprecated in Akka 2.4. They use one of 3 possible
        # seed sources, depending on availability: /dev/random, random.org and
        # SecureRandom (provided by Java)
        # "AES128CounterInetRNG"
        # "AES256CounterInetRNG" (Install JCE Unlimited Strength Jurisdiction
        # Policy Files first)
        # Setting a value here may require you to supply the appropriate cipher
        # suite (see enabled-algorithms section above)
        random-number-generator = ""

        # Require mutual authentication between TLS peers
        #
        # Without mutual authentication only the peer that actively establishes a␣
↪connection (TLS client side)
        # checks if the passive side (TLS server side) sends over a trusted␣
↪certificate. With the flag turned on,
        # the passive side will also request and verify a certificate from the␣
↪connecting peer.
```

```
        #
        # To prevent man-in-the-middle attacks you should enable this setting. For␣
↪compatibility reasons it is
        # still set to 'off' per default.
        #
        # Note: Nodes that are configured with this setting to 'on' might not be␣
↪able to receive messages from nodes that
        # run on older versions of akka-remote. This is because in older versions␣
↪of Akka the active side of the remoting
        # connection will not send over certificates.
        #
        # However, starting from the version this setting was added, even with␣
↪this setting "off", the active side
        # (TLS client side) will use the given key-store to send over a␣
↪certificate if asked. A rolling upgrades from
        # older versions of Akka can therefore work like this:
        #   - upgrade all nodes to an Akka version supporting this flag, keeping␣
↪it off
        #   - then switch the flag on and do again a rolling upgrade of all nodes
        # The first step ensures that all nodes will send over a certificate when␣
↪asked to. The second
        # step will ensure that all nodes finally enforce the secure checking of␣
↪client certificates.
        require-mutual-authentication = off
      }
    }

    ### Default configuration for the failure injector transport adapter

    gremlin {
      # Enable debug logging of the failure injector transport adapter
      debug = off
    }

    ### Default dispatcher for the remoting subsystem

    default-remote-dispatcher {
      type = Dispatcher
      executor = "fork-join-executor"
      fork-join-executor {
        parallelism-min = 2
        parallelism-factor = 0.5
        parallelism-max = 16
      }
      throughput = 10
    }

    backoff-remote-dispatcher {
      type = Dispatcher
      executor = "fork-join-executor"
      fork-join-executor {
        # Min number of threads to cap factor-based parallelism number to
        parallelism-min = 2
        parallelism-max = 2
      }
    }
  }
}
```

akka-remote (artery)

```
####################################
# Akka Remote Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# comments about akka.actor settings left out where they are already in akka-
# actor.jar, because otherwise they would be repeated in config rendering.
#
# For the configuration of the new remoting implementation (Artery) please look
# at the bottom section of this file as it is listed separately.

akka {

  actor {

    serializers {
      akka-containers = "akka.remote.serialization.MessageContainerSerializer"
      akka-misc = "akka.remote.serialization.MiscMessageSerializer"
      artery = "akka.remote.serialization.ArteryMessageSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      daemon-create = "akka.remote.serialization.DaemonMsgCreateSerializer"
      primitive-long = "akka.remote.serialization.LongSerializer"
      primitive-int = "akka.remote.serialization.IntSerializer"
      primitive-string = "akka.remote.serialization.StringSerializer"
      primitive-bytestring = "akka.remote.serialization.ByteStringSerializer"
      akka-system-msg = "akka.remote.serialization.SystemMessageSerializer"
    }

    serialization-bindings {
      "akka.actor.ActorSelectionMessage" = akka-containers

      "akka.remote.DaemonMsgCreate" = daemon-create

      "akka.remote.artery.ArteryMessage" = artery

      # Since akka.protobuf.Message does not extend Serializable but
      # GeneratedMessage does, need to use the more specific one here in order
      # to avoid ambiguity.
      "akka.protobuf.GeneratedMessage" = proto

      # Since com.google.protobuf.Message does not extend Serializable but
      # GeneratedMessage does, need to use the more specific one here in order
      # to avoid ambiguity.
      # This com.google.protobuf serialization binding is only used if the class␣
↪can be loaded,
      # i.e. com.google.protobuf dependency has been added in the application␣
↪project.
      "com.google.protobuf.GeneratedMessage" = proto
    }

    # For the purpose of preserving protocol backward compatibility these bindings␣
↪are not
    # included by default. They can be enabled with enable-additional-
↪serialization-bindings=on.
    # They are enabled by default if akka.remote.artery.enabled=on.
    additional-serialization-bindings {
      "akka.actor.Identify" = akka-misc
      "akka.actor.ActorIdentity" = akka-misc
      "scala.Some" = akka-misc
```

```
      "scala.None$" = akka-misc
      "akka.actor.Status$Success" = akka-misc
      "akka.actor.Status$Failure" = akka-misc
      "akka.actor.ActorRef" = akka-misc
      "akka.actor.PoisonPill$" = akka-misc
      "akka.actor.Kill$" = akka-misc
      "akka.remote.RemoteWatcher$Heartbeat$" = akka-misc
      "akka.remote.RemoteWatcher$HeartbeatRsp" = akka-misc
      "akka.actor.ActorInitializationException" = akka-misc

      "akka.dispatch.sysmsg.SystemMessage" = akka-system-msg

      "java.lang.String" = primitive-string
      "akka.util.ByteString$ByteString1C" = primitive-bytestring
      "akka.util.ByteString$ByteString1" = primitive-bytestring
      "akka.util.ByteString$ByteStrings" = primitive-bytestring
      "java.lang.Long" = primitive-long
      "scala.Long" = primitive-long
      "java.lang.Integer" = primitive-int
      "scala.Int" = primitive-int

      # Java Serializer is by default used for exceptions.
      # It's recommended that you implement custom serializer for exceptions that␣
↪are
      # sent remotely, e.g. in akka.actor.Status.Failure for ask replies. You can␣
↪add
      # binding to akka-misc (MiscMessageSerializerSpec) for the exceptions that␣
↪have
      # a constructor with single message String or constructor with message␣
↪String as
      # first parameter and cause Throwable as second parameter. Note that it's not
      # safe to add this binding for general exceptions such as␣
↪IllegalArgumentException
      # because it may have a subclass without required constructor.
      "java.lang.Throwable" = java
      "akka.actor.IllegalActorStateException" = akka-misc
      "akka.actor.ActorKilledException" = akka-misc
      "akka.actor.InvalidActorNameException" = akka-misc
      "akka.actor.InvalidMessageException" = akka-misc
    }

    serialization-identifiers {
      "akka.remote.serialization.ProtobufSerializer" = 2
      "akka.remote.serialization.DaemonMsgCreateSerializer" = 3
      "akka.remote.serialization.MessageContainerSerializer" = 6
      "akka.remote.serialization.MiscMessageSerializer" = 16
      "akka.remote.serialization.ArteryMessageSerializer" = 17
      "akka.remote.serialization.LongSerializer" = 18
      "akka.remote.serialization.IntSerializer" = 19
      "akka.remote.serialization.StringSerializer" = 20
      "akka.remote.serialization.ByteStringSerializer" = 21
      "akka.remote.serialization.SystemMessageSerializer" = 22
    }

    deployment {

      default {

        # if this is set to a valid remote address, the named actor will be
        # deployed at that node e.g. "akka.tcp://sys@host:port"
        remote = ""

        target {
```

```
          # A list of hostnames and ports for instantiating the children of a
          # router
          #   The format should be on "akka.tcp://sys@host:port", where:
          #    - sys is the remote actor system name
          #    - hostname can be either hostname or IP address the remote actor
          #      should connect to
          #    - port should be the port for the remote server on the other node
          # The number of actor instances to be spawned is still taken from the
          # nr-of-instances setting as for local routers; the instances will be
          # distributed round-robin among the given nodes.
          nodes = []

        }
      }
    }
  }

  remote {
    ### Settings shared by classic remoting and Artery (the new implementation of␣
↪remoting)

    # If set to a nonempty string remoting will use the given dispatcher for
    # its internal actors otherwise the default dispatcher is used. Please note
    # that since remoting can load arbitrary 3rd party drivers (see
    # "enabled-transport" and "adapters" entries) it is not guaranteed that
    # every module will respect this setting.
    use-dispatcher = "akka.remote.default-remote-dispatcher"

    # Settings for the failure detector to monitor connections.
    # For TCP it is not important to have fast failure detection, since
    # most connection failures are captured by TCP itself.
    # The default DeadlineFailureDetector will trigger if there are no heartbeats␣
↪within
    # the duration heartbeat-interval + acceptable-heartbeat-pause, i.e. 20 seconds
    # with the default settings.
    transport-failure-detector {

      # FQCN of the failure detector implementation.
      # It must implement akka.remote.FailureDetector and have
      # a public constructor with a com.typesafe.config.Config and
      # akka.actor.EventStream parameter.
      implementation-class = "akka.remote.DeadlineFailureDetector"

      # How often keep-alive heartbeat messages should be sent to each connection.
      heartbeat-interval = 4 s

      # Number of potentially lost/delayed heartbeats that will be
      # accepted before considering it to be an anomaly.
      # A margin to the `heartbeat-interval` is important to be able to survive␣
↪sudden,
      # occasional, pauses in heartbeat arrivals, due to for example garbage␣
↪collect or
      # network drop.
      acceptable-heartbeat-pause = 16 s
    }

    # Settings for the Phi accrual failure detector (http://www.jaist.ac.jp/~
↪defago/files/pdf/IS_RR_2004_010.pdf
    # [Hayashibara et al]) used for remote death watch.
    # The default PhiAccrualFailureDetector will trigger if there are no␣
↪heartbeats within
    # the duration heartbeat-interval + acceptable-heartbeat-pause + threshold_
↪adjustment,
```

```
    # i.e. around 12.5 seconds with default settings.
    watch-failure-detector {

      # FQCN of the failure detector implementation.
      # It must implement akka.remote.FailureDetector and have
      # a public constructor with a com.typesafe.config.Config and
      # akka.actor.EventStream parameter.
      implementation-class = "akka.remote.PhiAccrualFailureDetector"

      # How often keep-alive heartbeat messages should be sent to each connection.
      heartbeat-interval = 1 s

      # Defines the failure detector threshold.
      # A low threshold is prone to generate many wrong suspicions but ensures
      # a quick detection in the event of a real crash. Conversely, a high
      # threshold generates fewer mistakes but needs more time to detect
      # actual crashes.
      threshold = 10.0

      # Number of the samples of inter-heartbeat arrival times to adaptively
      # calculate the failure timeout for connections.
      max-sample-size = 200

      # Minimum standard deviation to use for the normal distribution in
      # AccrualFailureDetector. Too low standard deviation might result in
      # too much sensitivity for sudden, but normal, deviations in heartbeat
      # inter arrival times.
      min-std-deviation = 100 ms

      # Number of potentially lost/delayed heartbeats that will be
      # accepted before considering it to be an anomaly.
      # This margin is important to be able to survive sudden, occasional,
      # pauses in heartbeat arrivals, due to for example garbage collect or
      # network drop.
      acceptable-heartbeat-pause = 10 s


      # How often to check for nodes marked as unreachable by the failure
      # detector
      unreachable-nodes-reaper-interval = 1s

      # After the heartbeat request has been sent the first failure detection
      # will start after this period, even though no heartbeat mesage has
      # been received.
      expected-response-after = 1 s

    }

    # remote deployment configuration section
    deployment {
      # If true, will only allow specific classes to be instanciated on this␣
↪system via remote deployment
      enable-whitelist = off

      whitelist = []
    }

    ### Configuration for Artery, the reimplementation of remoting
    artery {

      # Enable the new remoting with this flag
      enabled = off
```

```
      # Canonical address is the address other clients should connect to.
      # Artery transport will expect messages to this address.
      canonical {

        # The default remote server port clients should connect to.
        # Default is 25520, use 0 if you want a random available port
        # This port needs to be unique for each actor system on the same machine.
        port = 25520

        # Hostname clients should connect to. Can be set to an ip, hostname
        # or one of the following special values:
        #   "<getHostAddress>"   InetAddress.getLocalHost.getHostAddress
        #   "<getHostName>"      InetAddress.getLocalHost.getHostName
        #
        hostname = "<getHostAddress>"
      }

      # Use these settings to bind a network interface to a different address
      # than artery expects messages at. This may be used when running Akka
      # nodes in a separated networks (under NATs or in containers). If canonical
      # and bind addresses are different, then network configuration that relays
      # communications from canonical to bind addresses is expected.
      bind {

        # Port to bind a network interface to. Can be set to a port number
        # of one of the following special values:
        #   0    random available port
        #   ""   akka.remote.artery.canonical.port
        #
        port = ""

        # Hostname to bind a network interface to. Can be set to an ip, hostname
        # or one of the following special values:
        #   "0.0.0.0"            all interfaces
        #   ""                   akka.remote.artery.canonical.hostname
        #   "<getHostAddress>"   InetAddress.getLocalHost.getHostAddress
        #   "<getHostName>"      InetAddress.getLocalHost.getHostName
        #
        hostname = ""
      }

      # Actor paths to use the large message stream for when a message
      # is sent to them over remoting. The large message stream dedicated
      # is separate from "normal" and system messages so that sending a
      # large message does not interfere with them.
      # Entries should be the full path to the actor. Wildcards in the form of "*"
      # can be supplied at any place and matches any name at that segment -
      # "/user/supervisor/actor/*" will match any direct child to actor,
      # while "/supervisor/*/child" will match any grandchild to "supervisor" that
      # has the name "child"
      # Messages sent to ActorSelections will not be passed through the large␣
↪message
      # stream, to pass such messages through the large message stream the␣
↪selections
      # but must be resolved to ActorRefs first.
      large-message-destinations = []

      # Enable untrusted mode, which discards inbound system messages,␣
↪PossiblyHarmful and
      # ActorSelection messages. E.g. remote watch and remote deployment will not␣
↪work.
      # ActorSelection messages can be enabled for specific paths with the trusted-
↪selection-paths
```

```
    untrusted-mode = off

    # When 'untrusted-mode=on' inbound actor selections are by default discarded.
    # Actors with paths defined in this white list are granted permission to␣
↪receive actor
    # selections messages.
    # E.g. trusted-selection-paths = ["/user/receptionist", "/user/namingService
↪"]
    trusted-selection-paths = []

    # If this is "on", all inbound remote messages will be logged at DEBUG level,
    # if off then they are not logged
    log-received-messages = off

    # If this is "on", all outbound remote messages will be logged at DEBUG␣
↪level,
    # if off then they are not logged
    log-sent-messages = off

    advanced {

      # Maximum serialized message size, including header data.
      maximum-frame-size = 256 KiB

      # Direct byte buffers are reused in a pool with this maximum size.
      # Each buffer has the size of 'maximum-frame-size'.
      # This is not a hard upper limit on number of created buffers. Additional
      # buffers will be created if needed, e.g. when using many outbound
      # associations at the same time. Such additional buffers will be garbage
      # collected, which is not as efficient as reusing buffers in the pool.
      buffer-pool-size = 128

      # Maximum serialized message size for the large messages, including header␣
↪data.
      # See 'large-message-destinations'.
      maximum-large-frame-size = 2 MiB

      # Direct byte buffers for the large messages are reused in a pool with␣
↪this maximum size.
      # Each buffer has the size of 'maximum-large-frame-size'.
      # See 'large-message-destinations'.
      # This is not a hard upper limit on number of created buffers. Additional
      # buffers will be created if needed, e.g. when using many outbound
      # associations at the same time. Such additional buffers will be garbage
      # collected, which is not as efficient as reusing buffers in the pool.
      large-buffer-pool-size = 32

      # For enabling testing features, such as blackhole in akka-remote-testkit.
      test-mode = off

      # Settings for the materializer that is used for the remote streams.
      materializer = ${akka.stream.materializer}

      # If set to a nonempty string artery will use the given dispatcher for
      # the ordinary and large message streams, otherwise the default dispatcher␣
↪is used.
      use-dispatcher = "akka.remote.default-remote-dispatcher"

      # If set to a nonempty string remoting will use the given dispatcher for
      # the control stream, otherwise the default dispatcher is used.
      # It can be good to not use the same dispatcher for the control stream as
      # the dispatcher for the ordinary message stream so that heartbeat messages
      # are not disturbed.
```

```
       use-control-stream-dispatcher = ""

       # Controls whether to start the Aeron media driver in the same JVM or use␣
↪external
       # process. Set to 'off' when using external media driver, and then also␣
↪set the
       # 'aeron-dir'.
       embedded-media-driver = on

       # Directory used by the Aeron media driver. It's mandatory to define the
↪'aeron-dir'
       # if using external media driver, i.e. when 'embedded-media-driver = off'.
       # Embedded media driver will use a this directory, or a temporary␣
↪directory if this
       # property is not defined (empty).
       aeron-dir = ""

       # Whether to delete aeron embeded driver directory upon driver stop.
       delete-aeron-dir = yes

       # Level of CPU time used, on a scale between 1 and 10, during backoff/idle.
       # The tradeoff is that to have low latency more CPU time must be used to be
       # able to react quickly on incoming messages or send as fast as possible␣
↪after
       # backoff backpressure.
       # Level 1 strongly prefer low CPU consumption over low latency.
       # Level 10 strongly prefer low latency over low CPU consumption.
       idle-cpu-level = 5

       # WARNING: This feature is not supported yet. Don't use other value than 1.
       # It requires more hardening and performance optimizations.
       # Number of outbound lanes for each outbound association. A value greater␣
↪than 1
       # means that serialization can be performed in parallel for different␣
↪destination
       # actors. The selection of lane is based on consistent hashing of the␣
↪recipient
       # ActorRef to preserve message ordering per receiver.
       outbound-lanes = 1

       # WARNING: This feature is not supported yet. Don't use other value than 1.
       # It requires more hardening and performance optimizations.
       # Total number of inbound lanes, shared among all inbound associations. A␣
↪value
       # greater than 1 means that deserialization can be performed in parallel␣
↪for
       # different destination actors. The selection of lane is based on␣
↪consistent
       # hashing of the recipient ActorRef to preserve message ordering per␣
↪receiver.
       inbound-lanes = 1

       # Size of the send queue for outgoing messages. Messages will be dropped if
       # the queue becomes full. This may happen if you send a burst of many␣
↪messages
       # without end-to-end flow control. Note that there is one such queue per
       # outbound association. The trade-off of using a larger queue size is that
       # it consumes more memory, since the queue is based on preallocated array␣
↪with
       # fixed size.
       outbound-message-queue-size = 3072

       # Size of the send queue for outgoing control messages, such as system␣
↪messages.
```

```
        # If this limit is reached the remote system is declared to be dead and␣
↪its UID
        # marked as quarantined.
        # The trade-off of using a larger queue size is that it consumes more␣
↪memory,
        # since the queue is based on preallocated array with fixed size.
        outbound-control-queue-size = 3072

        # Size of the send queue for outgoing large messages. Messages will be␣
↪dropped if
        # the queue becomes full. This may happen if you send a burst of many␣
↪messages
        # without end-to-end flow control. Note that there is one such queue per
        # outbound association. The trade-off of using a larger queue size is that
        # it consumes more memory, since the queue is based on preallocated array␣
↪with
        # fixed size.
        outbound-large-message-queue-size = 256

        # This setting defines the maximum number of unacknowledged system messages
        # allowed for a remote system. If this limit is reached the remote system␣
↪is
        # declared to be dead and its UID marked as quarantined.
        system-message-buffer-size = 20000

        # unacknowledged system messages are re-delivered with this interval
        system-message-resend-interval = 1 second

        # The timeout for outbound associations to perform the handshake.
        # This timeout must be greater than the 'image-liveness-timeout'.
        handshake-timeout = 20 s

        # incomplete handshake attempt is retried with this interval
        handshake-retry-interval = 1 second

        # handshake requests are performed periodically with this interval,
        # also after the handshake has been completed to be able to establish
        # a new session with a restarted destination system
        inject-handshake-interval = 1 second

        # messages that are not accepted by Aeron are dropped after retrying for␣
↪this period
        give-up-message-after = 60 seconds

        # System messages that are not acknowledged after re-sending for this␣
↪period are
        # dropped and will trigger quarantine. The value should be longer than the␣
↪length
        # of a network partition that you need to survive.
        give-up-system-message-after = 6 hours

        # during ActorSystem termination the remoting will wait this long for
        # an acknowledgment by the destination system that flushing of outstanding
        # remote messages has been completed
        shutdown-flush-timeout = 1 second

        # See 'inbound-max-restarts'
        inbound-restart-timeout = 5 seconds

        # Max number of restarts within 'inbound-restart-timeout' for the inbound␣
↪streams.
        # If more restarts occurs the ActorSystem will be terminated.
        inbound-max-restarts = 5
```

```
       # See 'outbound-max-restarts'
       outbound-restart-timeout = 5 seconds

       # Max number of restarts within 'outbound-restart-timeout' for the␣
↪outbound streams.
       # If more restarts occurs the ActorSystem will be terminated.
       outbound-max-restarts = 5

       # Stop outbound stream of a quarantined association after this idle␣
↪timeout, i.e.
       # when not used any more.
       stop-quarantined-after-idle = 3 seconds

       # Timeout after which aeron driver has not had keepalive messages
       # from a client before it considers the client dead.
       client-liveness-timeout = 20 seconds

       # Timeout for each the INACTIVE and LINGER stages an aeron image
       # will be retained for when it is no longer referenced.
       # This timeout must be less than the 'handshake-timeout'.
       image-liveness-timeout = 10 seconds

       # Timeout after which the aeron driver is considered dead
       # if it does not update its C'n'C timestamp.
       driver-timeout = 20 seconds

       flight-recorder {
         // FIXME it should be enabled by default when we have a good solution␣
↪for naming the files
         enabled = off
         # Controls where the flight recorder file will be written. There are␣
↪three options:
         # 1. Empty: a file will be generated in the temporary directory of the OS
         # 2. A relative or absolute path ending with ".afr": this file will be␣
↪used
         # 3. A relative or absolute path: this directory will be used, the file␣
↪will get a random file name
         destination = ""
       }

       # compression of common strings in remoting messages, like actor␣
↪destinations, serializers etc
       compression {

         actor-refs {
           # Max number of compressed actor-refs
           # Note that compression tables are "rolling" (i.e. a new table␣
↪replaces the old
           # compression table once in a while), and this setting is only about␣
↪the total number
           # of compressions within a single such table.
           # Must be a positive natural number.
           max = 256

           # interval between new table compression advertisements.
           # this means the time during which we collect heavy-hitter data and␣
↪then turn it into a compression table.
           advertisement-interval = 1 minute # TODO find good number as default,␣
↪for benchmarks trigger immediately
         }
         manifests {
           # Max number of compressed manifests
```

```
            # Note that compression tables are "rolling" (i.e. a new table␣
↪replaces the old
            # compression table once in a while), and this setting is only about␣
↪the total number
            # of compressions within a single such table.
            # Must be a positive natural number.
            max = 256

            # interval between new table compression advertisements.
            # this means the time during which we collect heavy-hitter data and␣
↪then turn it into a compression table.
            advertisement-interval = 1 minute # TODO find good number as default,␣
↪for benchmarks trigger immediately
          }
        }

        # List of fully qualified class names of remote instruments which should
        # be initialized and used for monitoring of remote messages.
        # The class must extend akka.remote.artery.RemoteInstrument and
        # have a public constructor with empty parameters or one␣
↪ExtendedActorSystem
        # parameter.
        # A new instance of RemoteInstrument will be created for each encoder and␣
↪decoder.
        # It's only called from the stage, so if it dosn't delegate to any shared␣
↪instance
        # it doesn't have to be thread-safe.
        # Refer to `akka.remote.artery.RemoteInstrument` for more information.
        instruments = ${?akka.remote.artery.advanced.instruments} []
      }
    }
  }

}
```

akka-testkit

```
#####################################
# Akka Testkit Reference Config File #
#####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  test {
    # factor by which to scale timeouts during tests, e.g. to account for shared
    # build system load
    timefactor =  1.0

    # duration of EventFilter.intercept waits after the block is finished until
    # all required messages are received
    filter-leeway = 3s

    # duration to wait in expectMsg and friends outside of within() block
    # by default
    single-expect-default = 3s

    # The timeout that is added as an implicit by DefaultTimeout trait
    default-timeout = 5s
```

```
    calling-thread-dispatcher {
      type = akka.testkit.CallingThreadDispatcherConfigurator
    }
  }

  actor.serialization-bindings {
    "akka.testkit.JavaSerializable" = java
  }
}
```

akka-cluster-metrics ~~~~~~~~~~~~~~

```
############################################
# Akka Cluster Metrics Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits in your application.conf in order to override these settings.

# Sigar provisioning:
#
#  User can provision sigar classes and native library in one of the following␣
↪ways:
#
#  1) Use https://github.com/kamon-io/sigar-loader Kamon sigar-loader as a project␣
↪dependency for the user project.
#  Metrics extension will extract and load sigar library on demand with help of␣
↪Kamon sigar provisioner.
#
#  2) Use https://github.com/kamon-io/sigar-loader Kamon sigar-loader as java␣
↪agent: `java -javaagent:/path/to/sigar-loader.jar`
#  Kamon sigar loader agent will extract and load sigar library during JVM start.
#
#  3) Place `sigar.jar` on the `classpath` and sigar native library for the o/s on␣
↪the `java.library.path`
#  User is required to manage both project dependency and library deployment␣
↪manually.

# Cluster metrics extension.
# Provides periodic statistics collection and publication throughout the cluster.
akka.cluster.metrics {
    # Full path of dispatcher configuration key.
    # Use "" for default key `akka.actor.default-dispatcher`.
    dispatcher = ""
    # How long should any actor wait before starting the periodic tasks.
    periodic-tasks-initial-delay = 1s
    # Sigar native library extract location.
    # Use per-application-instance scoped location, such as program working␣
↪directory.
    native-library-extract-folder = ${user.dir}"/native"
    # Metrics supervisor actor.
    supervisor {
        # Actor name. Example name space: /system/cluster-metrics
        name = "cluster-metrics"
        # Supervision strategy.
        strategy {
            #
            # FQCN of class providing `akka.actor.SupervisorStrategy`.
            # Must have a constructor with signature `<init>(com.typesafe.config.
↪Config)`.
            # Default metrics strategy provider is a configurable extension of␣
↪`OneForOneStrategy`.
            provider = "akka.cluster.metrics.ClusterMetricsStrategy"
```

```
            #
            # Configuration of the default strategy provider.
            # Replace with custom settings when overriding the provider.
            configuration = {
                # Log restart attempts.
                loggingEnabled = true
                # Child actor restart-on-failure window.
                withinTimeRange = 3s
                # Maximum number of restart attempts before child actor is stopped.
                maxNrOfRetries = 3
            }
        }
    }
    # Metrics collector actor.
    collector {
        # Enable or disable metrics collector for load-balancing nodes.
        # Metrics collection can also be controlled at runtime by sending control␣
↪messages
        # to /system/cluster-metrics actor: `akka.cluster.metrics.
↪{CollectionStartMessage,CollectionStopMessage}`
        enabled = on
        # FQCN of the metrics collector implementation.
        # It must implement `akka.cluster.metrics.MetricsCollector` and
        # have public constructor with akka.actor.ActorSystem parameter.
        # Will try to load in the following order of priority:
        # 1) configured custom collector 2) internal `SigarMetricsCollector` 3)␣
↪internal `JmxMetricsCollector`
        provider = ""
        # Try all 3 available collector providers, or else fail on the configured␣
↪custom collector provider.
        fallback = true
        # How often metrics are sampled on a node.
        # Shorter interval will collect the metrics more often.
        # Also controls frequency of the metrics publication to the node system␣
↪event bus.
        sample-interval = 3s
        # How often a node publishes metrics information to the other nodes in the␣
↪cluster.
        # Shorter interval will publish the metrics gossip more often.
        gossip-interval = 3s
        # How quickly the exponential weighting of past data is decayed compared to
        # new data. Set lower to increase the bias toward newer values.
        # The relevance of each data sample is halved for every passing half-life
        # duration, i.e. after 4 times the half-life, a data sample's relevance is
        # reduced to 6% of its original relevance. The initial relevance of a data
        # sample is given by 1 - 0.5 ^ (collect-interval / half-life).
        # See http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_
↪average
        moving-average-half-life = 12s
    }
}

# Cluster metrics extension serializers and routers.
akka.actor {
    # Protobuf serializer for remote cluster metrics messages.
    serializers {
       akka-cluster-metrics = "akka.cluster.metrics.protobuf.MessageSerializer"
    }
    # Interface binding for remote cluster metrics messages.
    serialization-bindings {
       "akka.cluster.metrics.ClusterMetricsMessage" = akka-cluster-metrics
    }
    # Globally unique metrics extension serializer identifier.
```

```
    serialization-identifiers {
      "akka.cluster.metrics.protobuf.MessageSerializer" = 10
    }
    #  Provide routing of messages based on cluster metrics.
    router.type-mapping {
        cluster-metrics-adaptive-pool  = "akka.cluster.metrics.
↪AdaptiveLoadBalancingPool"
        cluster-metrics-adaptive-group = "akka.cluster.metrics.
↪AdaptiveLoadBalancingGroup"
    }
}
```

akka-cluster-tools ~~~~~~~~~~~~~~

```
############################################
# Akka Cluster Tools Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# //#pub-sub-ext-config
# Settings for the DistributedPubSub extension
akka.cluster.pub-sub {
  # Actor name of the mediator actor, /system/distributedPubSubMediator
  name = distributedPubSubMediator

  # Start the mediator on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # The routing logic to use for 'Send'
  # Possible values: random, round-robin, broadcast
  routing-logic = random

  # How often the DistributedPubSubMediator should send out gossip information
  gossip-interval = 1s

  # Removed entries are pruned after this duration
  removed-time-to-live = 120s

  # Maximum number of elements to transfer in one message when synchronizing the␣
↪registries.
  # Next chunk will be transferred in next round of gossip.
  max-delta-elements = 3000

  # The id of the dispatcher to use for DistributedPubSubMediator actors.
  # If not specified default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  use-dispatcher = ""
}
# //#pub-sub-ext-config

# Protobuf serializer for cluster DistributedPubSubMeditor messages
akka.actor {
  serializers {
    akka-pubsub = "akka.cluster.pubsub.protobuf.DistributedPubSubMessageSerializer"
  }
  serialization-bindings {
    "akka.cluster.pubsub.DistributedPubSubMessage" = akka-pubsub
  }
  serialization-identifiers {
    "akka.cluster.pubsub.protobuf.DistributedPubSubMessageSerializer" = 9
```

```
  }
  # adds the protobuf serialization of pub sub messages to groups
  additional-serialization-bindings {
    "akka.cluster.pubsub.DistributedPubSubMediator$Internal$SendToOneSubscriber" =␣
↪akka-pubsub
  }
}


# //#receptionist-ext-config
# Settings for the ClusterClientReceptionist extension
akka.cluster.client.receptionist {
  # Actor name of the ClusterReceptionist actor, /system/receptionist
  name = receptionist

  # Start the receptionist on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # The receptionist will send this number of contact points to the client
  number-of-contacts = 3

  # The actor that tunnel response messages to the client will be stopped
  # after this time of inactivity.
  response-tunnel-receive-timeout = 30s

  # The id of the dispatcher to use for ClusterReceptionist actors.
  # If not specified default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  use-dispatcher = ""

  # How often failure detection heartbeat messages should be received for
  # each ClusterClient
  heartbeat-interval = 2s

  # Number of potentially lost/delayed heartbeats that will be
  # accepted before considering it to be an anomaly.
  # The ClusterReceptionist is using the akka.remote.DeadlineFailureDetector, which
  # will trigger if there are no heartbeats within the duration
  # heartbeat-interval + acceptable-heartbeat-pause, i.e. 15 seconds with
  # the default settings.
  acceptable-heartbeat-pause = 13s

  # Failure detection checking interval for checking all ClusterClients
  failure-detection-interval = 2s
}
# //#receptionist-ext-config

# //#cluster-client-config
# Settings for the ClusterClient
akka.cluster.client {
  # Actor paths of the ClusterReceptionist actors on the servers (cluster nodes)
  # that the client will try to contact initially. It is mandatory to specify
  # at least one initial contact.
  # Comma separated full actor paths defined by a string on the form of
  # "akka.tcp://system@hostname:port/system/receptionist"
  initial-contacts = []

  # Interval at which the client retries to establish contact with one of
  # ClusterReceptionist on the servers (cluster nodes)
  establishing-get-contacts-interval = 3s

  # Interval at which the client will ask the ClusterReceptionist for
```

```
  # new contact points to be used for next reconnect.
  refresh-contacts-interval = 60s

  # How often failure detection heartbeat messages should be sent
  heartbeat-interval = 2s

  # Number of potentially lost/delayed heartbeats that will be
  # accepted before considering it to be an anomaly.
  # The ClusterClient is using the akka.remote.DeadlineFailureDetector, which
  # will trigger if there are no heartbeats within the duration
  # heartbeat-interval + acceptable-heartbeat-pause, i.e. 15 seconds with
  # the default settings.
  acceptable-heartbeat-pause = 13s

  # If connection to the receptionist is not established the client will buffer
  # this number of messages and deliver them the connection is established.
  # When the buffer is full old messages will be dropped when new messages are sent
  # via the client. Use 0 to disable buffering, i.e. messages will be dropped
  # immediately if the location of the singleton is unknown.
  # Maximum allowed buffer size is 10000.
  buffer-size = 1000

  # If connection to the receiptionist is lost and the client has not been
  # able to acquire a new connection for this long the client will stop itself.
  # This duration makes it possible to watch the cluster client and react on a␣
↪more permanent
  # loss of connection with the cluster, for example by accessing some kind of
  # service registry for an updated set of initial contacts to start a new cluster␣
↪client with.
  # If this is not wanted it can be set to "off" to disable the timeout and retry
  # forever.
  reconnect-timeout = off
}
# //#cluster-client-config

# Protobuf serializer for ClusterClient messages
akka.actor {
  serializers {
    akka-cluster-client = "akka.cluster.client.protobuf.
↪ClusterClientMessageSerializer"
  }
  serialization-bindings {
    "akka.cluster.client.ClusterClientMessage" = akka-cluster-client
  }
  serialization-identifiers {
    "akka.cluster.client.protobuf.ClusterClientMessageSerializer" = 15
  }
}

# //#singleton-config
akka.cluster.singleton {
  # The actor name of the child singleton actor.
  singleton-name = "singleton"

  # Singleton among the nodes tagged with specified role.
  # If the role is not specified it's a singleton among all nodes in the cluster.
  role = ""

  # When a node is becoming oldest it sends hand-over request to previous oldest,
  # that might be leaving the cluster. This is retried with this interval until
  # the previous oldest confirms that the hand over has started or the previous
  # oldest member is removed from the cluster (+ akka.cluster.down-removal-margin).
  hand-over-retry-interval = 1s
```

```
  # The number of retries are derived from hand-over-retry-interval and
  # akka.cluster.down-removal-margin (or ClusterSingletonManagerSettings.
↪removalMargin),
  # but it will never be less than this property.
  min-number-of-hand-over-retries = 10
}
# //#singleton-config

# //#singleton-proxy-config
akka.cluster.singleton-proxy {
  # The actor name of the singleton actor that is started by the␣
↪ClusterSingletonManager
  singleton-name = ${akka.cluster.singleton.singleton-name}

  # The role of the cluster nodes where the singleton can be deployed.
  # If the role is not specified then any node will do.
  role = ""

  # Interval at which the proxy will try to resolve the singleton instance.
  singleton-identification-interval = 1s

  # If the location of the singleton is unknown the proxy will buffer this
  # number of messages and deliver them when the singleton is identified.
  # When the buffer is full old messages will be dropped when new messages are
  # sent via the proxy.
  # Use 0 to disable buffering, i.e. messages will be dropped immediately if
  # the location of the singleton is unknown.
  # Maximum allowed buffer size is 10000.
  buffer-size = 1000
}
# //#singleton-proxy-config

# Serializer for cluster ClusterSingleton messages
akka.actor {
  serializers {
    akka-singleton = "akka.cluster.singleton.protobuf.
↪ClusterSingletonMessageSerializer"
  }
  serialization-bindings {
    "akka.cluster.singleton.ClusterSingletonMessage" = akka-singleton
  }
  serialization-identifiers {
    "akka.cluster.singleton.protobuf.ClusterSingletonMessageSerializer" = 14
  }
}
```

akka-cluster-sharding ~~~~~~~~~~~~~~

```
############################################
# Akka Cluster Sharding Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.


# //#sharding-ext-config
# Settings for the ClusterShardingExtension
akka.cluster.sharding {

  # The extension creates a top level actor with this name in top level system␣
↪scope,
```

```
  # e.g. '/system/sharding'
  guardian-name = sharding

  # Specifies that entities runs on cluster nodes with a specific role.
  # If the role is not specified (or empty) all nodes in the cluster are used.
  role = ""

  # When this is set to 'on' the active entity actors will automatically be␣
↪restarted
  # upon Shard restart. i.e. if the Shard is started on a different ShardRegion
  # due to rebalance or crash.
  remember-entities = off

  # If the coordinator can't store state changes it will be stopped
  # and started again after this duration, with an exponential back-off
  # of up to 5 times this duration.
  coordinator-failure-backoff = 5 s

  # The ShardRegion retries registration and shard location requests to the
  # ShardCoordinator with this interval if it does not reply.
  retry-interval = 2 s

  # Maximum number of messages that are buffered by a ShardRegion actor.
  buffer-size = 100000

  # Timeout of the shard rebalancing process.
  handoff-timeout = 60 s

  # Time given to a region to acknowledge it's hosting a shard.
  shard-start-timeout = 10 s

  # If the shard is remembering entities and can't store state changes
  # will be stopped and then started again after this duration. Any messages
  # sent to an affected entity may be lost in this process.
  shard-failure-backoff = 10 s

  # If the shard is remembering entities and an entity stops itself without
  # using passivate. The entity will be restarted after this duration or when
  # the next message for it is received, which ever occurs first.
  entity-restart-backoff = 10 s

  # Rebalance check is performed periodically with this interval.
  rebalance-interval = 10 s

  # Absolute path to the journal plugin configuration entity that is to be
  # used for the internal persistence of ClusterSharding. If not defined
  # the default journal plugin is used. Note that this is not related to
  # persistence used by the entity actors.
  journal-plugin-id = ""

  # Absolute path to the snapshot plugin configuration entity that is to be
  # used for the internal persistence of ClusterSharding. If not defined
  # the default snapshot plugin is used. Note that this is not related to
  # persistence used by the entity actors.
  snapshot-plugin-id = ""

  # Parameter which determines how the coordinator will be store a state
  # valid values either "persistence" or "ddata"
  # The "ddata" mode is experimental, since it depends on the experimental
  # module akka-distributed-data-experimental.
  state-store-mode = "persistence"

  # The shard saves persistent snapshots after this number of persistent
```

```
  # events. Snapshots are used to reduce recovery times.
  snapshot-after = 1000

  # Setting for the default shard allocation strategy
  least-shard-allocation-strategy {
    # Threshold of how large the difference between most and least number of
    # allocated shards must be to begin the rebalancing.
    rebalance-threshold = 10

    # The number of ongoing rebalancing processes is limited to this number.
    max-simultaneous-rebalance = 3
  }

  # Timeout of waiting the initial distributed state (an initial state will be␣
↪queried again if the timeout happened)
  # works only for state-store-mode = "ddata"
  waiting-for-state-timeout = 5 s

  # Timeout of waiting for update the distributed state (update will be retried if␣
↪the timeout happened)
  # works only for state-store-mode = "ddata"
  updating-state-timeout = 5 s

  # The shard uses this strategy to determines how to recover the underlying␣
↪entity actors. The strategy is only used
  # by the persistent shard when rebalancing or restarting. The value can either␣
↪be "all" or "constant". The "all"
  # strategy start all the underlying entity actors at the same time. The constant␣
↪strategy will start the underlying
  # entity actors at a fix rate. The default strategy "all".
  entity-recovery-strategy = "all"

  # Default settings for the constant rate entity recovery strategy
  entity-recovery-constant-rate-strategy {
    # Sets the frequency at which a batch of entity actors is started.
    frequency = 100 ms
    # Sets the number of entity actors to be restart at a particular interval
    number-of-entities = 5
  }

  # Settings for the coordinator singleton. Same layout as akka.cluster.singleton.
  # The "role" of the singleton configuration is not used. The singleton role will
  # be the same as "akka.cluster.sharding.role".
  coordinator-singleton = ${akka.cluster.singleton}

  # The id of the dispatcher to use for ClusterSharding actors.
  # If not specified default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  # This dispatcher for the entity actors is defined by the user provided
  # Props, i.e. this dispatcher is not used for the entity actors.
  use-dispatcher = ""
}
# //#sharding-ext-config


# Protobuf serializer for Cluster Sharding messages
akka.actor {
  serializers {
    akka-sharding = "akka.cluster.sharding.protobuf.
↪ClusterShardingMessageSerializer"
  }
  serialization-bindings {
    "akka.cluster.sharding.ClusterShardingSerializable" = akka-sharding
```

```
  }
  serialization-identifiers {
    "akka.cluster.sharding.protobuf.ClusterShardingMessageSerializer" = 13
  }
}
```

akka-distributed-data ~~~~~~~~~~~~~~~~

```
############################################
# Akka Distributed DataReference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.


#//#distributed-data
# Settings for the DistributedData extension
akka.cluster.distributed-data {
  # Actor name of the Replicator actor, /system/ddataReplicator
  name = ddataReplicator

  # Replicas are running on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # How often the Replicator should send out gossip information
  gossip-interval = 2 s

  # How often the subscribers will be notified of changes, if any
  notify-subscribers-interval = 500 ms

  # Maximum number of entries to transfer in one gossip message when synchronizing
  # the replicas. Next chunk will be transferred in next round of gossip.
  max-delta-elements = 1000

  # The id of the dispatcher to use for Replicator actors. If not specified
  # default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  use-dispatcher = ""

  # How often the Replicator checks for pruning of data associated with
  # removed cluster nodes.
  pruning-interval = 30 s

  # How long time it takes (worst case) to spread the data to all other replica␣
↪nodes.
  # This is used when initiating and completing the pruning process of data␣
↪associated
  # with removed cluster nodes. The time measurement is stopped when any replica is
  # unreachable, so it should be configured to worst case in a healthy cluster.
  max-pruning-dissemination = 60 s

  # Serialized Write and Read messages are cached when they are sent to
  # several nodes. If no further activity they are removed from the cache
  # after this duration.
  serializer-cache-time-to-live = 10s

}
#//#distributed-data

# Protobuf serializer for cluster DistributedData messages
akka.actor {
```

```
  serializers {
    akka-data-replication = "akka.cluster.ddata.protobuf.
↪ReplicatorMessageSerializer"
    akka-replicated-data = "akka.cluster.ddata.protobuf.ReplicatedDataSerializer"
  }
  serialization-bindings {
    "akka.cluster.ddata.Replicator$ReplicatorMessage" = akka-data-replication
    "akka.cluster.ddata.ReplicatedDataSerialization" = akka-replicated-data
  }
  serialization-identifiers {
    "akka.cluster.ddata.protobuf.ReplicatedDataSerializer" = 11
    "akka.cluster.ddata.protobuf.ReplicatorMessageSerializer" = 12
  }
}
```

CHAPTER

THREE

ACTORS

3.1 Actors

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

The API of Akka's Actors is similar to Scala Actors which has borrowed some of its syntax from Erlang.

3.1.1 Creating Actors

Note: Since Akka enforces parental supervision every actor is supervised and (potentially) the supervisor of its children, it is advisable that you familiarize yourself with *Actor Systems* and *Supervision and Monitoring* and it may also help to read *Actor References, Paths and Addresses*.

Defining an Actor class

Actors are implemented by extending the `Actor` base trait and implementing the `receive` method. The `receive` method should define a series of case statements (which has the type `PartialFunction[Any,Unit]`) that defines which messages your Actor can handle, using standard Scala pattern matching, along with the implementation of how the messages should be processed.

Here is an example:

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

Please note that the Akka Actor `receive` message loop is exhaustive, which is different compared to Erlang and the late Scala Actors. This means that you need to provide a pattern match for all messages that it can accept and if you want to be able to handle unknown messages then you need to have a default case as in the example above. Otherwise an `akka.actor.UnhandledMessage(message,sender,recipient)` will be published to the `ActorSystem`'s `EventStream`.

Note further that the return type of the behavior defined above is `Unit`; if the actor shall reply to the received message then this must be done explicitly as explained below.

The result of the `receive` method is a partial function object, which is stored within the actor as its "initial behavior", see *Become/Unbecome* for further information on changing the behavior of an actor after its construction.

Props

`Props` is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information (e.g. which dispatcher to use, see more below). Here are some examples of how to create a `Props` instance.

```
import akka.actor.Props

val props1 = Props[MyActor]
val props2 = Props(new ActorWithArgs("arg")) // careful, see below
val props3 = Props(classOf[ActorWithArgs], "arg") // no support for value class␣
↪arguments
```

The second variant shows how to pass constructor arguments to the `Actor` being created, but it should only be used outside of actors as explained below.

The last line shows a possibility to pass constructor arguments regardless of the context it is being used in. The presence of a matching constructor is verified during construction of the `Props` object, resulting in an `IllegalArgumentException` if no or multiple matching constructors are found.

Note: The recommended approach to create the actor `Props` is not supported for cases when the actor constructor takes value classes as arguments.

Dangerous Variants

```
// NOT RECOMMENDED within another actor:
// encourages to close over enclosing class
val props7 = Props(new MyActor)
```

This method is not recommended to be used within another actor because it encourages to close over the enclosing scope, resulting in non-serializable `Props` and possibly race conditions (breaking the actor encapsulation). We will provide a macro-based solution in a future release which allows similar syntax without the headaches, at which point this variant will be properly deprecated. On the other hand using this variant in a `Props` factory in the actor's companion object as documented under "Recommended Practices" below is completely fine.

There were two use-cases for these methods: passing constructor arguments to the actor—which is solved by the newly introduced `Props.apply(clazz, args)` method above or the recommended practice below—and creating actors "on the spot" as anonymous classes. The latter should be solved by making these actors named classes instead (if they are not declared within a top-level `object` then the enclosing instance's `this` reference needs to be passed as the first argument).

Warning: Declaring one actor within another is very dangerous and breaks actor encapsulation. Never pass an actor's `this` reference into `Props`!

Edge cases

There are two edge cases in actor creation with `Props`:

- An actor with `AnyVal` arguments.

```
case class MyValueClass(v: Int) extends AnyVal
```

```
class ValueActor(value: MyValueClass) extends Actor {
  def receive = {
    case multiplier: Long => sender() ! (value.v * multiplier)
  }
}
val valueClassProp = Props(classOf[ValueActor], MyValueClass(5)) // Unsupported
```

- An actor with default constructor values.

```
class DefaultValueActor(a: Int, b: Int = 5) extends Actor {
  def receive = {
    case x: Int => sender() ! ((a + x) * b)
  }
}

val defaultValueProp1 = Props(classOf[DefaultValueActor], 2.0) // Unsupported

class DefaultValueActor2(b: Int = 5) extends Actor {
  def receive = {
    case x: Int => sender() ! (x * b)
  }
}
val defaultValueProp2 = Props[DefaultValueActor2] // Unsupported
val defaultValueProp3 = Props(classOf[DefaultValueActor2]) // Unsupported
```

In both cases an `IllegalArgumentException` will be thrown stating no matching constructor could be found.

The next section explains the recommended ways to create `Actor` props in a way, which simultaneously safeguards against these edge cases.

Recommended Practices

It is a good idea to provide factory methods on the companion object of each `Actor` which help keeping the creation of suitable `Props` as close to the actor definition as possible. This also avoids the pitfalls associated with using the `Props.apply(...)` method which takes a by-name argument, since within a companion object the given code block will not retain a reference to its enclosing scope:

```
object DemoActor {
  /**
   * Create Props for an actor of this type.
   *
   * @param magicNumber The magic number to be passed to this actor's constructor.
   * @return a Props for creating this actor, which can then be further configured
   *         (e.g. calling `.withDispatcher()` on it)
   */
  def props(magicNumber: Int): Props = Props(new DemoActor(magicNumber))
}

class DemoActor(magicNumber: Int) extends Actor {
  def receive = {
    case x: Int => sender() ! (x + magicNumber)
  }
}

class SomeOtherActor extends Actor {
  // Props(new DemoActor(42)) would not be safe
  context.actorOf(DemoActor.props(42), "demo")
```

```
  // ...
}
```

Another good practice is to declare what messages an Actor can receive in the companion object of the Actor, which makes easier to know what it can receive:

```
object MyActor {
  case class Greeting(from: String)
  case object Goodbye
}
class MyActor extends Actor with ActorLogging {
  import MyActor._
  def receive = {
    case Greeting(greeter) => log.info(s"I was greeted by $greeter.")
    case Goodbye           => log.info("Someone said goodbye to me.")
  }
}
```

Creating Actors with Props

Actors are created by passing a `Props` instance into the `actorOf` factory method which is available on `ActorSystem` and `ActorContext`.

```
import akka.actor.ActorSystem

// ActorSystem is a heavy object: create only one per application
val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor2")
```

Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

```
class FirstActor extends Actor {
  val child = context.actorOf(Props[MyActor], name = "myChild")
  // plus some behavior ...
}
```

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see *Actor Systems*.

The call to `actorOf` returns an instance of `ActorRef`. This is a handle to the actor instance and the only way to interact with it. The `ActorRef` is immutable and has a one to one relationship with the Actor it represents. The `ActorRef` is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same Actor on the original node, across the network.

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with `$`, but it may contain URL encoded characters (eg. `%20` for a blank space). If the given name is already in use by another child to the same parent an `InvalidActorNameException` is thrown.

Actors are automatically started asynchronously when created.

Value classes as constructor arguments

The recommended way to instantiate actor props uses reflection at runtime to determine the correct actor constructor to be invoked and due to technical limitations is not supported when said constructor takes arguments that are value classes. In these cases you should either unpack the arguments or create the props by calling the constructor manually:

```
class Argument(val value: String) extends AnyVal
class ValueClassActor(arg: Argument) extends Actor {
  def receive = { case _ => () }
}

object ValueClassActor {
  def props1(arg: Argument) = Props(classOf[ValueClassActor], arg) // fails at␣
↪runtime
  def props2(arg: Argument) = Props(classOf[ValueClassActor], arg.value) // ok
  def props3(arg: Argument) = Props(new ValueClassActor(arg)) // ok
}
```

Dependency Injection

If your Actor has a constructor that takes parameters then those need to be part of the `Props` as well, as described *above*. But there are cases when a factory method must be used, for example when the actual constructor arguments are determined by a dependency injection framework.

```
import akka.actor.IndirectActorProducer

class DependencyInjector(applicationContext: AnyRef, beanName: String)
  extends IndirectActorProducer {

  override def actorClass = classOf[Actor]
  override def produce =
    // obtain fresh Actor instance from DI framework ...
}

val actorRef = system.actorOf(
  Props(classOf[DependencyInjector], applicationContext, "hello"),
  "helloBean")
```

Warning: You might be tempted at times to offer an `IndirectActorProducer` which always returns the same instance, e.g. by using a `lazy val`. This is not supported, as it goes against the meaning of an actor restart, which is described here: *What Restarting Means*.

When using a dependency injection framework, actor beans *MUST NOT* have singleton scope.

Techniques for dependency injection and integration with dependency injection frameworks are described in more depth in the Using Akka with Dependency Injection guideline and the Akka Java Spring tutorial in Lightbend Activator.

The Inbox

When writing code outside of actors which shall communicate with actors, the `ask` pattern can be a solution (see below), but there are two things it cannot do: receiving multiple replies (e.g. by subscribing an `ActorRef` to a notification service) and watching other actors' lifecycle. For these purposes there is the `Inbox` class:

```
implicit val i = inbox()
echo ! "hello"
i.receive() should ===("hello")
```

There is an implicit conversion from inbox to actor reference which means that in this example the sender reference will be that of the actor hidden away within the inbox. This allows the reply to be received on the last line. Watching an actor is quite simple as well:

```
val target = // some actor
val i = inbox()
i watch target
```

3.1.2 Actor API

The `Actor` trait defines only one abstract method, the above mentioned `receive`, which implements the behavior of the actor.

If the current actor behavior does not match a received message, `unhandled` is called, which by default publishes an `akka.actor.UnhandledMessage(message,sender,recipient)` on the actor system's event stream (set configuration item `akka.actor.debug.unhandled` to `on` to have them converted into actual Debug messages).

In addition, it offers:

- `self` reference to the `ActorRef` of the actor
- `sender` reference sender Actor of the last received message, typically used as described in *Reply to messages*
- `supervisorStrategy` user overridable definition the strategy to use for supervising child actors

  This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the `sender` reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).
- `context` exposes contextual information for the actor and the current message, such as:
  - factory methods to create child actors (`actorOf`)
  - system that the actor belongs to
  - parent supervisor
  - supervised children
  - lifecycle monitoring
  - hotswap behavior stack as described in *Become/Unbecome*

You can import the members in the `context` to avoid prefixing access with `context.`

```
class FirstActor extends Actor {
  import context._
  val myActor = actorOf(Props[MyActor], name = "myactor")
  def receive = {
    case x => myActor ! x
  }
}
```

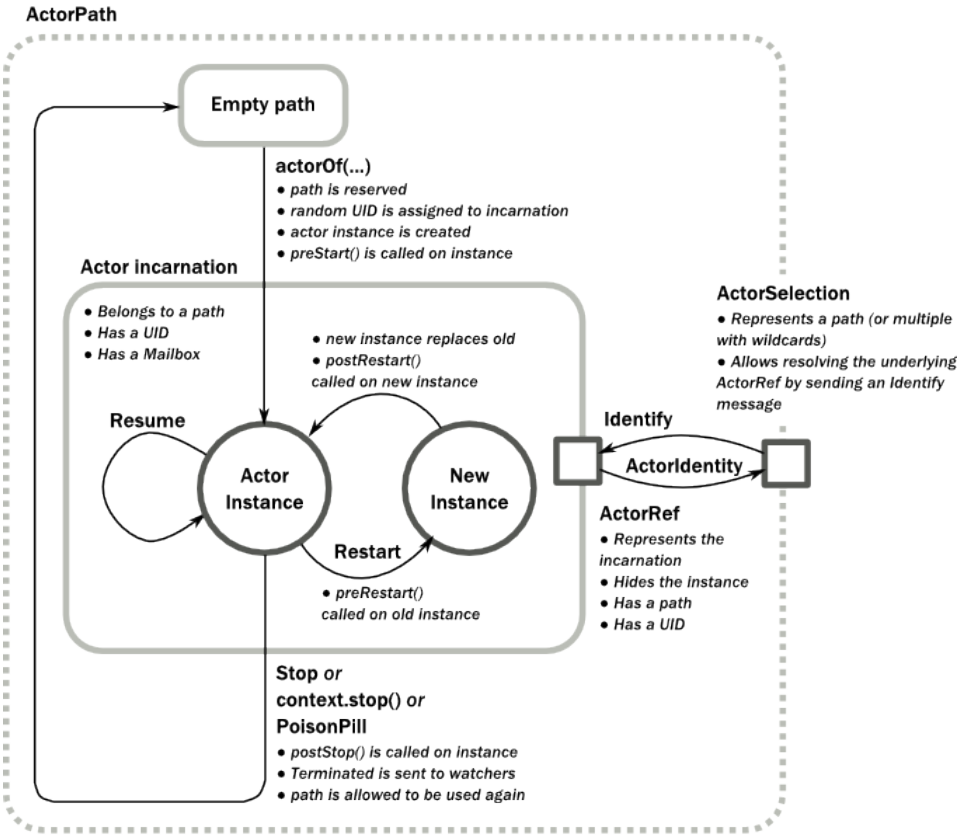
The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

```
def preStart(): Unit = ()

def postStop(): Unit = ()

def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  context.children foreach { child ⇒
    context.unwatch(child)
    context.stop(child)
  }
  postStop()
```

```
}

def postRestart(reason: Throwable): Unit = {
  preStart()
}
```

The implementations shown above are the defaults provided by the `Actor` trait.

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When `actorOf()` is called it assigns an *incarnation* of the actor described by the passed `Props` to the given path. An actor incarnation is identified by the path *and a UID*. A restart only swaps the `Actor` instance defined by the `Props` but the incarnation and hence the UID remains the same.

The lifecycle of an incarnation ends when the actor is stopped. At that point the appropriate lifecycle events are called and watching actors are notified of the termination. After the incarnation is stopped, the path can be reused again by creating an actor with `actorOf()`. In this case the name of the new incarnation will be the same as the previous one but the UIDs will differ. An actor can be stopped by the actor itself, another actor or the `ActorSystem` (see *Stopping actors*).

Note: It is important to note that Actors do not stop automatically when no longer referenced, every Actor that is created must also explicitly be destroyed. The only simplification is that stopping a parent Actor will also recursively stop all the child Actors that this parent has created.

An `ActorRef` always represents an incarnation (path and UID) not just a given path. Therefore if an actor is

stopped and a new one with the same name is created an `ActorRef` of the old incarnation will not point to the new one.

`ActorSelection` on the other hand points to the path (or multiple paths if wildcards are used) and is completely oblivious to which incarnation is currently occupying it. `ActorSelection` cannot be watched for this reason. It is possible to resolve the current incarnation's `ActorRef` living under the path by sending an `Identify` message to the `ActorSelection` which will be replied to with an `ActorIdentity` containing the correct reference (see *Identifying Actors via Actor Selection*). This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the `Terminated` message dispatched by the other actor upon termination (see *Stopping Actors*). This service is provided by the `DeathWatch` component of the actor system.

Registering a monitor is easy:

```
import akka.actor.{ Actor, Props, Terminated }

class WatchActor extends Actor {
  val child = context.actorOf(Props.empty, "child")
  context.watch(child) // <-- this is the only call needed for registration
  var lastSender = context.system.deadLetters

  def receive = {
    case "kill" =>
      context.stop(child); lastSender = sender()
    case Terminated(`child`) => lastSender ! "finished"
  }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. In particular, the watching actor will receive a `Terminated` message even if the watched actor has already been terminated at the time of registration.

Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the `Terminated` message.

It is also possible to deregister from watching another actor's liveliness using `context.unwatch(target)`. This works even if the `Terminated` message has already been enqueued in the mailbox; after calling `unwatch` no `Terminated` message for that actor will be processed anymore.

Start Hook

Right after starting the actor, its `preStart` method is invoked.

```
override def preStart() {
  child = context.actorOf(Props[MyActor], "child")
}
```

This method is called when the actor is first created. During restarts it is called by the default implementation of `postRestart`, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the actor's constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see *Supervision and Monitoring*). This restart involves the hooks mentioned above:

1. The old actor is informed by calling `preRestart` with the exception which caused the restart and the message which triggered that exception; the latter may be `None` if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling `sender`).

   This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc. By default it stops all children and calls `postStop`.

2. The initial factory from the `actorOf` call is used to produce the fresh instance.

3. The new actor's `postRestart` method is invoked with the exception which caused the restart. By default the `preStart` is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the `postRestart` hook returns. The message that triggered the exception will not be received again. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

Warning: Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See *Discussion: Message Ordering* for details.

Stop Hook

After stopping an actor, its `postStop` hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. messages sent to a stopped actor will be redirected to the `deadLetters` of the `ActorSystem`.

3.1.3 Identifying Actors via Actor Selection

As described in *Actor References, Paths and Addresses*, each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an `ActorSelection` with the result:

```
// will look up this absolute path
context.actorSelection("/user/serviceA/aggregator")
// will look up sibling beneath same supervisor
context.actorSelection("../joe")
```

Note: It is always preferable to communicate with other Actors using their ActorRef instead of relying upon ActorSelection. Exceptions are

- sending messages using the *At-Least-Once Delivery* facility
- initiating first contact with a remote system

In all other cases ActorRefs can be provided during Actor creation or initialization, passing them from parent to child or introducing Actors by sending their ActorRefs to other Actors within messages.

The supplied path is parsed as a `java.net.URI`, which basically means that it is split on `/` into path elements. If the path starts with `/`, it is absolute and the look-up starts at the root guardian (which is the parent of `"/user"`); otherwise it starts at the current actor. If a path element equals `..`, the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the `..` in actor paths here always means the logical structure, i.e. the supervisor.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

```
// will look all children to serviceB with names starting with worker
context.actorSelection("/user/serviceB/worker*")
// will look up all siblings beneath same supervisor
context.actorSelection("../*")
```

Messages can be sent via the `ActorSelection` and the path of the `ActorSelection` is looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender()` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

```
import akka.actor.{ Actor, Props, Identify, ActorIdentity, Terminated }

class Follower extends Actor {
  val identifyId = 1
  context.actorSelection("/user/another") ! Identify(identifyId)

  def receive = {
    case ActorIdentity(`identifyId`, Some(ref)) =>
      context.watch(ref)
      context.become(active(ref))
    case ActorIdentity(`identifyId`, None) => context.stop(self)

  }

  def active(another: ActorRef): Actor.Receive = {
    case Terminated(`another`) => context.stop(self)
  }
}
```

You can also acquire an `ActorRef` for an `ActorSelection` with the `resolveOne` method of the `ActorSelection`. It returns a `Future` of the matching `ActorRef` if such an actor exists. It is completed with failure [[akka.actor.ActorNotFound]] if no such actor exists or the identification didn't complete within the supplied *timeout*.

Remote actor addresses may also be looked up, if *remoting* is enabled:

```
context.actorSelection("akka.tcp://app@otherhost:1234/user/serviceB")
```

An example demonstrating actor look-up is given in *Remoting Sample*.

3.1.4 Messages and immutability

IMPORTANT: Messages can be any kind of object but have to be immutable. Scala can't enforce immutability (yet) so this has to be by convention. Primitives like String, Int, Boolean are always immutable. Apart from these the recommended approach is to use Scala case classes which are immutable (if you don't explicitly expose the state) and works great with pattern matching at the receiver side.

Here is an example:

```
// define the case class
case class Register(user: User)

// create a new case class message
val message = Register(user)
```

3.1.5 Send messages

Messages are sent to an Actor through one of the following methods.

- `!` means "fire-and-forget", e.g. send a message asynchronously and return immediately. Also known as `tell`.
- `?` sends a message asynchronously and returns a `Future` representing a possible reply. Also known as `ask`.

Message ordering is guaranteed on a per-sender basis.

Note: There are performance implications of using `ask` since something needs to keep track of when it times out, there needs to be something that bridges a `Promise` into an `ActorRef` and it also needs to be reachable through remoting. So always prefer `tell` for performance, and only `ask` if you must.

Tell: Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

```
actorRef ! message
```

If invoked from within an Actor, then the sending actor reference will be implicitly passed along with the message and available to the receiving Actor in its `sender(): ActorRef` member method. The target actor can use this to reply to the original sender, by using `sender() ! replyMsg`.

If invoked from an instance that is **not** an Actor the sender will be `deadLetters` actor reference by default.

Ask: Send-And-Receive-Future

The `ask` pattern involves actors as well as futures, hence it is offered as a use pattern rather than a method on `ActorRef`:

```
import akka.pattern.{ ask, pipe }
import system.dispatcher // The ExecutionContext that will be used
final case class Result(x: Int, s: String, d: Double)
case object Request

implicit val timeout = Timeout(5 seconds) // needed for `?` below

val f: Future[Result] =
  for {
    x <- ask(actorA, Request).mapTo[Int] // call pattern directly
    s <- (actorB ask Request).mapTo[String] // call by implicit conversion
    d <- (actorC ? Request).mapTo[Double] // call by symbolic name
  } yield Result(x, s, d)

f pipeTo actorD // .. or ..
pipe(f) to actorD
```

This example demonstrates `ask` together with the `pipeTo` pattern on futures, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: `ask` produces a `Future`, three of which are composed into a new future using the for-comprehension and then `pipeTo` installs an `onComplete`-handler on the future to affect the submission of the aggregated `Result` to another actor.

Using `ask` will send a message to the receiving Actor as with `tell`, and the receiving actor must reply with `sender() ! reply` in order to complete the returned `Future` with a value. The `ask` operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

Warning: To complete the future with an exception you need send a Failure message to the sender. This is *not done automatically* when an actor throws an exception while processing a message.

```
try {
  val result = operation()
  sender() ! result
} catch {
  case e: Exception =>
    sender() ! akka.actor.Status.Failure(e)
    throw e
}
```

If the actor does not complete the future, it will expire after the timeout period, completing it with an `AskTimeoutException`. The timeout is taken from one of the following locations in order of precedence:

1. explicitly given timeout as in:

```
import scala.concurrent.duration._
import akka.pattern.ask
val future = myActor.ask("hello")(5 seconds)
```

2. implicit argument of type `akka.util.Timeout`, e.g.

```
import scala.concurrent.duration._
import akka.util.Timeout
import akka.pattern.ask
implicit val timeout = Timeout(5 seconds)
val future = myActor ? "hello"
```

See *Futures* for more information on how to await or query a future.

The `onComplete`, `onSuccess`, or `onFailure` methods of the `Future` can be used to register a callback to get a notification when the Future completes, giving you a way to avoid blocking.

Warning: When using future callbacks, such as `onComplete`, `onSuccess`, and `onFailure`, inside actors you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time. See also: *Actors and shared mutable state*

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a 'mediator'. This can be useful when writing actors that work as routers, load-balancers, replicators etc.

```
target forward message
```

3.1.6 Receive messages

An Actor has to implement the `receive` method to receive messages:

```
type Receive = PartialFunction[Any, Unit]

def receive: Actor.Receive
```

This method returns a `PartialFunction`, e.g. a 'match/case' clause in which the message can be matched against the different case clauses using Scala pattern matching. Here is an example:

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

3.1.7 Reply to messages

If you want to have a handle for replying to a message, you can use `sender()`, which gives you an ActorRef. You can reply by sending to that ActorRef with `sender() ! replyMsg`. You can also store the ActorRef for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or future context) then the sender defaults to a 'dead-letter' actor ref.

```
case request =>
  val result = process(request)
  sender() ! result       // will have dead-letter actor as default
```

3.1.8 Receive timeout

The *ActorContext* `setReceiveTimeout` defines the inactivity timeout after which the sending of a *ReceiveTimeout* message is triggered. When specified, the receive function should be able to handle an *akka.actor.ReceiveTimeout* message. 1 millisecond is the minimum supported timeout.

Please note that the receive timeout might fire and enqueue the *ReceiveTimeout* message right after another message was enqueued; hence it is **not guaranteed** that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in *Duration.Undefined* to switch off this feature.

```
import akka.actor.ReceiveTimeout
import scala.concurrent.duration._
class MyActor extends Actor {
  // To set an initial delay
  context.setReceiveTimeout(30 milliseconds)
  def receive = {
    case "Hello" =>
```

```
      // To set in a response to a message
      context.setReceiveTimeout(100 milliseconds)
    case ReceiveTimeout =>
      // To turn it off
      context.setReceiveTimeout(Duration.Undefined)
      throw new RuntimeException("Receive timed out")
  }
}
```

Messages marked with `NotInfluenceReceiveTimeout` will not reset the timer. This can be useful when `ReceiveTimeout` should be fired by external inactivity but not influenced by internal activity, e.g. scheduled tick messages.

3.1.9 Stopping actors

Actors are stopped by invoking the `stop` method of a `ActorRefFactory`, i.e. `ActorContext` or `ActorSystem`. Typically the context is used for stopping the actor itself or child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. `stop` may return before the actor is stopped.

```
class MyActor extends Actor {

  val child: ActorRef = ???

  def receive = {
    case "interrupt-child" =>
      context stop child

    case "done" =>
      context stop self
  }

}
```

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the `deadLetters` of the `ActorSystem`, but that depends on the mailbox implementation.

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the internal termination notifications from its children until the last one is gone, finally terminating itself (invoking `postStop`, dumping mailbox, publishing `Terminated` on the *DeathWatch*, telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon `ActorSystem.terminate`, the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

The `postStop` hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

```
override def postStop() {
  // clean up some resources ...
}
```

Note: Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an `InvalidActorNameException`. Instead, `watch` the terminating actor and create its replacement in response to the `Terminated` message which will eventually arrive.

PoisonPill

You can also send an actor the `akka.actor.PoisonPill` message, which will stop the actor when the message is processed. `PoisonPill` is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Graceful Stop

`gracefulStop` is useful if you need to wait for termination or compose ordered termination of several actors:

```
import akka.pattern.gracefulStop
import scala.concurrent.Await

try {
  val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds, Manager.
↪Shutdown)
  Await.result(stopped, 6 seconds)
  // the actor has been stopped
} catch {
  // the actor wasn't stopped within 5 seconds
  case e: akka.pattern.AskTimeoutException =>
}
```

```
object Manager {
  case object Shutdown
}

class Manager extends Actor {
  import Manager._
  val worker = context.watch(context.actorOf(Props[Cruncher], "worker"))

  def receive = {
    case "job" => worker ! "crunch"
    case Shutdown =>
      worker ! PoisonPill
      context become shuttingDown
  }

  def shuttingDown: Receive = {
    case "job" => sender() ! "service unavailable, shutting down"
    case Terminated(`worker`) =>
      context stop self
  }
}
```

When `gracefulStop()` returns successfully, the actor's `postStop()` hook will have been executed: there exists a happens-before edge between the end of `postStop()` and the return of `gracefulStop()`.

In the above example a custom `Manager.Shutdown` message is sent to the target actor to initiate the process of stopping the actor. You can use `PoisonPill` for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in `postStop`.

Warning: Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after `gracefulStop()` returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a `Terminated` message, i.e. not for top-level actors.

3.1.10 Become/Unbecome

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime: invoke the `context.become` method from within the Actor. `become` takes a `PartialFunction[Any,Unit]` that implements the new message handler. The hotswapped code is kept in a Stack which can be pushed and popped.

> **Warning:** Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor behavior using `become`:

```
class HotSwapActor extends Actor {
  import context._
  def angry: Receive = {
    case "foo" => sender() ! "I am already angry?"
    case "bar" => become(happy)
  }

  def happy: Receive = {
    case "bar" => sender() ! "I am already happy :-)"
    case "foo" => become(angry)
  }

  def receive = {
    case "foo" => become(angry)
    case "bar" => become(happy)
  }
}
```

This variant of the `become` method is useful for many different things, such as to implement a Finite State Machine (FSM, for an example see Dining Hakkers). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use `unbecome`, instead always the next behavior is explicitly installed.

The other way of using `become` does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of "pop" operations (i.e. `unbecome`) matches the number of "push" ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

```
case object Swap
class Swapper extends Actor {
  import context._
  val log = Logging(system, this)

  def receive = {
    case Swap =>
      log.info("Hi")
      become({
        case Swap =>
          log.info("Ho")
          unbecome() // resets the latest 'become' (just for fun)
      }, discardOld = false) // push on top instead of replace
  }
}

object SwapperApp extends App {
  val system = ActorSystem("SwapperSystem")
  val swap = system.actorOf(Props[Swapper], name = "swapper")
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
  swap ! Swap // logs Hi
```

```
  swap ! Swap // logs Ho
}
```

Encoding Scala Actors nested receives without accidentally leaking memory

See this Unnested receive example.

3.1.11 Stash

The *Stash* trait enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking `context.become` or `context.unbecome`, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally.

Note: The trait `Stash` extends the marker trait `RequiresMessageQueue[DequeBasedMessageQueueSemantics]` which requests the system to automatically choose a deque based mailbox implementation for the actor. If you want more control over the mailbox, see the documentation on mailboxes: *Mailboxes*.

Here is an example of the `Stash` in action:

```
import akka.actor.Stash
class ActorWithProtocol extends Actor with Stash {
  def receive = {
    case "open" =>
      unstashAll()
      context.become({
        case "write" => // do writing...
        case "close" =>
          unstashAll()
          context.unbecome()
        case msg => stash()
      }, discardOld = false) // stack on top instead of replacing
    case msg => stash()
  }
}
```

Invoking `stash()` adds the current message (the message that the actor received last) to the actor's stash. It is typically invoked when handling the default case in the actor's message handler to stash messages that aren't handled by the other cases. It is illegal to stash the same message twice; to do so results in an `IllegalStateException` being thrown. The stash may also be bounded in which case invoking `stash()` may lead to a capacity violation, which results in a `StashOverflowException`. The capacity of the stash can be configured using the `stash-capacity` setting (an `Int`) of the mailbox's configuration.

Invoking `unstashAll()` enqueues messages from the stash to the actor's mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a `MessageQueueAppendFailedException` is thrown. The stash is guaranteed to be empty after calling `unstashAll()`.

The stash is backed by a `scala.collection.immutable.Vector`. As a result, even a very large number of messages may be stashed without a major impact on performance.

Warning: Note that the `Stash` trait must be mixed into (a subclass of) the `Actor` trait before any trait/class that overrides the `preRestart` callback. This means it's not possible to write `Actor with MyActor with Stash` if `MyActor` overrides `preRestart`.

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor's state which have the same property. The `Stash` trait's implementation of `preRestart` will call `unstashAll()`, which is usually the desired behavior.

Note: If you want to enforce that your actor can only work with an unbounded stash, then you should use the `UnboundedStash` trait instead.

3.1.12 Killing an Actor

You can kill an actor by sending a `Kill` message. This will cause the actor to throw a `ActorKilledException`, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See *What Supervision Means* for more information.

Use `Kill` like this:

```
// kill the 'victim' actor
victim ! Kill
```

3.1.13 Actors and exceptions

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

What happens to the Message

If an exception is thrown while a message is being processed (i.e. taken out of its mailbox and handed over to the current behavior), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don't want a system to livelock (so consuming a lot of cpu cycles without making progress). Another possibility would be to have a look at the *PeekMailbox pattern*.

What happens to the mailbox

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox will be there as well.

What happens to the actor

If code within an actor throws an exception, that actor is suspended and the supervision process is started (see *Supervision and Monitoring*). Depending on the supervisor's decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.

3.1.14 Extending Actors using PartialFunction chaining

Sometimes it can be useful to share common behavior among a few actors, or compose one actor's behavior from multiple smaller functions. This is possible because an actor's `receive` method returns an `Actor.Receive`, which is a type alias for `PartialFunction[Any,Unit]`, and partial functions can be chained together using the `PartialFunction#orElse` method. You can chain as many functions as you need, however you should keep in mind that "first match" wins - which may be important when combining functions that both can handle the same type of message.

For example, imagine you have a set of actors which are either `Producers` or `Consumers`, yet sometimes it makes sense to have an actor share both behaviors. This can be easily achieved without having to duplicate code by extracting the behaviors to traits and implementing the actor's `receive` as combination of these partial functions.

```
trait ProducerBehavior {
  this: Actor =>

  val producerBehavior: Receive = {
    case GiveMeThings =>
      sender() ! Give("thing")
  }
}

trait ConsumerBehavior {
  this: Actor with ActorLogging =>

  val consumerBehavior: Receive = {
    case ref: ActorRef =>
      ref ! GiveMeThings

    case Give(thing) =>
      log.info("Got a thing! It's {}", thing)
  }
}

class Producer extends Actor with ProducerBehavior {
  def receive = producerBehavior
}

class Consumer extends Actor with ActorLogging with ConsumerBehavior {
  def receive = consumerBehavior
}

class ProducerConsumer extends Actor with ActorLogging
  with ProducerBehavior with ConsumerBehavior {

  def receive = producerBehavior.orElse[Any, Unit](consumerBehavior)
}

// protocol
case object GiveMeThings
final case class Give(thing: Any)
```

Instead of inheritance the same pattern can be applied via composition - one would simply compose the receive method using partial functions from delegates.

3.1.15 Initialization patterns

The rich lifecycle hooks of Actors provide a useful toolkit to implement various initialization patterns. During the lifetime of an `ActorRef`, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the `ActorRef`.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the `ActorRef` is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use `val` fields to store any state that does not change during the life of the actor instance, making the implementation of the actor

more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via preStart

The method `preStart()` of an actor is only called once directly during the initialization of the first instance, that is, at creation of its `ActorRef`. In the case of restarts, `preStart()` is called from `postRestart()`, therefore if not overridden, `preStart()` is called on every incarnation. However, by overriding `postRestart()` one can disable this behavior, and ensure that there is only one call to `preStart()`.

One useful usage of this pattern is to disable creation of new `ActorRefs` for children during restarts. This can be achieved by overriding `preRestart()`:

```
override def preStart(): Unit = {
  // Initialize children here
}

// Overriding postRestart to disable the call to preStart()
// after restarts
override def postRestart(reason: Throwable): Unit = ()

// The default implementation of preRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override preRestart()
override def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  // Keep the call to postStop(), but no stopping of children
  postStop()
}
```

Please note, that the child actors are *still restarted*, but no new `ActorRef` is created. One can recursively apply the same principles for the children, ensuring that their `preStart()` method is called only at the creation of their refs.

For more information see *What Restarting Means*.

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use `become()` or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

```
var initializeMe: Option[String] = None

override def receive = {
  case "init" =>
    initializeMe = Some("Up and running")
    context.become(initialized, discardOld = true)

}

def initialized: Receive = {
  case "U OK?" => initializeMe foreach { sender() ! _ }
}
```

If the actor may receive messages before it has been initialized, a useful tool can be the `Stash` to save messages until the initialization finishes, and replaying them after the actor became initialized.

Warning: This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an `ActorRef` in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

3.2 Akka Typed

Warning: This module is currently experimental in the sense of being the subject of active research. This means that API or semantics can change without warning or deprecation period and it is not recommended to use this module in production just yet—you have been warned.

As discussed in *Actor Systems* (and following chapters) Actors are about sending messages between independent units of computation, but how does that look like? In all of the following these imports are assumed:

```
import akka.typed._
import akka.typed.ScalaDSL._
import akka.typed.AskPattern._
import scala.concurrent.Future
import scala.concurrent.duration._
import scala.concurrent.Await
```

With these in place we can define our first Actor, and of course it will say hello!

```
object HelloWorld {
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String)

  val greeter = Static[Greet] { msg =>
    println(s"Hello ${msg.whom}!")
    msg.replyTo ! Greeted(msg.whom)
  }
}
```

This small piece of code defines two message types, one for commanding the Actor to greet someone and one that the Actor will use to confirm that it has done so. The `Greet` type contains not only the information of whom to greet, it also holds an `ActorRef` that the sender of the message supplies so that the `HelloWorld` Actor can send back the confirmation message.

The behavior of the Actor is defined as the `greeter` value with the help of the `Static` behavior constructor—there are many different ways of formulating behaviors as we shall see in the following. The "static" behavior is not capable of changing in response to a message, it will stay the same until the Actor is stopped by its parent.

The type of the messages handled by this behavior is declared to be of class `Greet`, which implies that the supplied function's `msg` argument is also typed as such. This is why we can access the `whom` and `replyTo` members without needing to use a pattern match.

On the last line we see the `HelloWorld` Actor send a message to another Actor, which is done using the `!` operator (pronounced "tell"). Since the `replyTo` address is declared to be of type `ActorRef[Greeted]` the compiler will only permit us to send messages of this type, other usage will not be accepted.

The accepted message types of an Actor together with all reply types defines the protocol spoken by this Actor; in this case it is a simple request–reply protocol but Actors can model arbitrarily complex protocols when needed. The protocol is bundled together with the behavior that implements it in a nicely wrapped scope—the `HelloWorld` object.

Now we want to try out this Actor, so we must start an ActorSystem to host it:

```
import HelloWorld._
// using global pool since we want to run tasks after system.terminate
import scala.concurrent.ExecutionContext.Implicits.global

val system: ActorSystem[Greet] = ActorSystem("hello", greeter)

val future: Future[Greeted] = system ? (Greet("world", _))

for {
  greeting <- future.recover { case ex => ex.getMessage }
  done <- { println(s"result: $greeting"); system.terminate() }
} println("system terminated")
```

After importing the Actor's protocol definition we start an Actor system from the defined behavior.

As Carl Hewitt said, one Actor is no Actor—it would be quite lonely with nobody to talk to. In this sense the example is a little cruel because we only give the `HelloWorld` Actor a fake person to talk to—the "ask" pattern (represented by the `?` operator) can be used to send a message such that the reply fulfills a Promise to which we get back the corresponding Future.

Note that the `Future` that is returned by the "ask" operation is properly typed already, no type checks or casts needed. This is possible due to the type information that is part of the message protocol: the `?` operator takes as argument a function that accepts an `ActorRef[U]` (which explains the _ hole in the expression on line 7 above) and the `replyTo` parameter which we fill in is of type `ActorRef[Greeted]`, which means that the value that fulfills the `Promise` can only be of type `Greeted`.

We use this here to send the `Greet` command to the Actor and when the reply comes back we will print it out and tell the actor system to shut down. Once that is done as well we print the `"system terminated"` messages and the program ends. The `recovery` combinator on the original `Future` is needed in order to ensure proper system shutdown even in case something went wrong; the `flatMap` and `map` combinators that the `for` expression gets turned into care only about the "happy path" and if the `future` failed with a timeout then no `greeting` would be extracted and nothing would happen.

This shows that there are aspects of Actor messaging that can be type-checked by the compiler, but this ability is not unlimited, there are bounds to what we can statically express. Before we go on with a more complex (and realistic) example we make a small detour to highlight some of the theory behind this.

3.2.1 A Little Bit of Theory

The Actor Model as defined by Hewitt, Bishop and Steiger in 1973 is a computational model that expresses exactly what it means for computation to be distributed. The processing units—Actors—can only communicate by exchanging messages and upon reception of a message an Actor can do the following three fundamental actions:

1. send a finite number of messages to Actors it knows
2. create a finite number of new Actors
3. designate the behavior to be applied to the next message

The Akka Typed project expresses these actions using behaviors and addresses. Messages can be sent to an address and behind this façade there is a behavior that receives the message and acts upon it. The binding between address and behavior can change over time as per the third point above, but that is not visible on the outside.

With this preamble we can get to the unique property of this project, namely that it introduces static type checking to Actor interactions: addresses are parameterized and only messages that are of the specified type can be sent to them. The association between an address and its type parameter must be made when the address (and its Actor) is created. For this purpose each behavior is also parameterized with the type of messages it is able to process. Since the behavior can change behind the address façade, designating the next behavior is a constrained operation: the successor must handle the same type of messages as its predecessor. This is necessary in order to not invalidate the addresses that refer to this Actor.

What this enables is that whenever a message is sent to an Actor we can statically ensure that the type of the message is one that the Actor declares to handle—we can avoid the mistake of sending completely pointless

messages. What we cannot statically ensure, though, is that the behavior behind the address will be in a given state when our message is received. The fundamental reason is that the association between address and behavior is a dynamic runtime property, the compiler cannot know it while it translates the source code.

This is the same as for normal Java objects with internal variables: when compiling the program we cannot know what their value will be, and if the result of a method call depends on those variables then the outcome is uncertain to a degree—we can only be certain that the returned value is of a given type.

We have seen above that the return type of an Actor command is described by the type of reply-to address that is contained within the message. This allows a conversation to be described in terms of its types: the reply will be of type A, but it might also contain an address of type B, which then allows the other Actor to continue the conversation by sending a message of type B to this new address. While we cannot statically express the "current" state of an Actor, we can express the current state of a protocol between two Actors, since that is just given by the last message type that was received or sent.

In the next section we demonstrate this on a more realistic example.

3.2.2 A More Complex Example

Consider an Actor that runs a chat room: client Actors may connect by sending a message that contains their screen name and then they can post messages. The chat room Actor will disseminate all posted messages to all currently connected client Actors. The protocol definition could look like the following:

```
sealed trait Command
final case class GetSession(screenName: String, replyTo: ActorRef[SessionEvent])
  extends Command

sealed trait SessionEvent
final case class SessionGranted(handle: ActorRef[PostMessage]) extends SessionEvent
final case class SessionDenied(reason: String) extends SessionEvent
final case class MessagePosted(screenName: String, message: String) extends␣
↪SessionEvent

final case class PostMessage(message: String)
```

Initially the client Actors only get access to an `ActorRef[GetSession]` which allows them to make the first step. Once a client's session has been established it gets a `SessionGranted` message that contains a `handle` to unlock the next protocol step, posting messages. The `PostMessage` command will need to be sent to this particular address that represents the session that has been added to the chat room. The other aspect of a session is that the client has revealed its own address, via the `replyTo` argument, so that subsequent `MessagePosted` events can be sent to it.

This illustrates how Actors can express more than just the equivalent of method calls on Java objects. The declared message types and their contents describe a full protocol that can involve multiple Actors and that can evolve over multiple steps. The implementation of the chat room protocol would be as simple as the following:

```
private final case class PostSessionMessage(screenName: String, message: String)
  extends Command

val behavior: Behavior[GetSession] =
  ContextAware[Command] { ctx =>
    var sessions = List.empty[ActorRef[SessionEvent]]

    Static {
      case GetSession(screenName, client) =>
        sessions ::= client
        val wrapper = ctx.spawnAdapter {
          p: PostMessage => PostSessionMessage(screenName, p.message)
        }
        client ! SessionGranted(wrapper)
      case PostSessionMessage(screenName, message) =>
        val mp = MessagePosted(screenName, message)
```

```
        sessions foreach (_ ! mp)
    }
  }.narrow // only expose GetSession to the outside
```

The core of this behavior is again static, the chat room itself does not change into something else when sessions are established, but we introduce a variable that tracks the opened sessions. When a new `GetSession` command comes in we add that client to the list and then we need to create the session's `ActorRef` that will be used to post messages. In this case we want to create a very simple Actor that just repackages the `PostMessage` command into a `PostSessionMessage` command which also includes the screen name. Such a wrapper Actor can be created by using the `spawnAdapter` method on the `ActorContext`, so that we can then go on to reply to the client with the `SessionGranted` result.

The behavior that we declare here can handle both subtypes of `Command`. `GetSession` has been explained already and the `PostSessionMessage` commands coming from the wrapper Actors will trigger the dissemination of the contained chat room message to all connected clients. But we do not want to give the ability to send `PostSessionMessage` commands to arbitrary clients, we reserve that right to the wrappers we create—otherwise clients could pose as completely different screen names (imagine the `GetSession` protocol to include authentication information to further secure this). Therefore we narrow the behavior down to only accepting `GetSession` commands before exposing it to the world, hence the type of the `behavior` value is `Behavior[GetSession]` instead of `Behavior[Command]`.

Narrowing the type of a behavior is always a safe operation since it only restricts what clients can do. If we were to widen the type then clients could send other messages that were not foreseen while writing the source code for the behavior.

If we did not care about securing the correspondence between a session and a screen name then we could change the protocol such that `PostMessage` is removed and all clients just get an `ActorRef[PostSessionMessage]` to send to. In this case no wrapper would be needed and we could just use `ctx.self`. The type-checks work out in that case because `ActorRef[-T]` is contravariant in its type parameter, meaning that we can use a `ActorRef[Command]` wherever an `ActorRef[PostSessionMessage]` is needed—this makes sense because the former simply speaks more languages than the latter. The opposite would be problematic, so passing an `ActorRef[PostSessionMessage]` where `ActorRef[Command]` is required will lead to a type error.

The final piece of this behavior definition is the `ContextAware` decorator that we use in order to obtain access to the `ActorContext` within the `Static` behavior definition. This decorator invokes the provided function when the first message is received and thereby creates the real behavior that will be used going forward—the decorator is discarded after it has done its job.

Trying it out

In order to see this chat room in action we need to write a client Actor that can use it:

```
import ChatRoom._

val gabbler: Behavior[SessionEvent] =
  Total {
    case SessionDenied(reason) =>
      println(s"cannot start chat room session: $reason")
      Stopped
    case SessionGranted(handle) =>
      handle ! PostMessage("Hello World!")
      Same
    case MessagePosted(screenName, message) =>
      println(s"message has been posted by '$screenName': $message")
      Stopped
  }
```

From this behavior we can create an Actor that will accept a chat room session, post a message, wait to see it published, and then terminate. The last step requires the ability to change behavior, we need to transition

from the normal running behavior into the terminated state. This is why this Actor uses a different behavior constructor named `Total`. This constructor takes as argument a function from the handled message type, in this case `SessionEvent`, to the next behavior. That next behavior must again be of the same type as we discussed in the theory section above. Here we either stay in the very same behavior or we terminate, and both of these cases are so common that there are special values `Same` and `Stopped` that can be used. The behavior is named "total" (as opposed to "partial") because the declared function must handle all values of its input type. Since `SessionEvent` is a sealed trait the Scala compiler will warn us if we forget to handle one of the subtypes; in this case it reminded us that alternatively to `SessionGranted` we may also receive a `SessionDenied` event.

Now to try things out we must start both a chat room and a gabbler and of course we do this inside an Actor system. Since there can be only one guardian supervisor we could either start the chat room from the gabbler (which we don't want—it complicates its logic) or the gabbler from the chat room (which is nonsensical) or we start both of them from a third Actor—our only sensible choice:

```
val main: Behavior[akka.NotUsed] =
  Full {
    case Sig(ctx, PreStart) =>
      val chatRoom = ctx.spawn(ChatRoom.behavior, "chatroom")
      val gabblerRef = ctx.spawn(gabbler, "gabbler")
      ctx.watch(gabblerRef)
      chatRoom ! GetSession("ol' Gabbler", gabblerRef)
      Same
    case Sig(_, Terminated(ref)) =>
      Stopped
  }

val system = ActorSystem("ChatRoomDemo", main)
Await.result(system.whenTerminated, 1.second)
```

In good tradition we call the `main` Actor what it is, it directly corresponds to the `main` method in a traditional Java application. This Actor will perform its job on its own accord, we do not need to send messages from the outside, so we declare it to be of type `NotUsed`. Actors receive not only external messages, they also are notified of certain system events, so-called Signals. In order to get access to those we choose to implement this particular one using the `Full` behavior decorator. The name stems from the fact that within this we have full access to all aspects of the Actor. The provided function will be invoked for signals (wrapped in `Sig`) or user messages (wrapped in `Msg`) and the wrapper also contains a reference to the `ActorContext`.

This particular main Actor reacts to two signals: when it is started it will first receive the `PreStart` signal, upon which the chat room and the gabbler are created and the session between them is initiated, and when the gabbler is finished we will receive the `Terminated` event due to having called `ctx.watch` for it. This allows us to shut down the Actor system: when the main Actor terminates there is nothing more to do.

Therefore after creating the Actor system with the `main` Actor's `Props` we just await its termination.

3.2.3 Status of this Project and Relation to Akka Actors

Akka Typed is the result of many years of research and previous attempts (including Typed Channels in the 2.2.x series) and it is on its way to stabilization, but maturing such a profound change to the core concept of Akka will take a long time. We expect that this module will stay experimental for multiple major releases of Akka and the plain `akka.actor.Actor` will not be deprecated or go away anytime soon.

Being a research project also entails that the reference documentation is not as detailed as it will be for a final version, please refer to the API documentation for greater depth and finer detail.

Main Differences

The most prominent difference is the removal of the `sender()` functionality. This turned out to be the Achilles heel of the Typed Channels project, it is the feature that makes its type signatures and macros too complex to be viable. The solution chosen in Akka Typed is to explicitly include the properly typed reply-to address in the message, which both burdens the user with this task but also places this aspect of protocol design where it belongs.

The other prominent difference is the removal of the `Actor` trait. In order to avoid closing over unstable references from different execution contexts (e.g. Future transformations) we turned all remaining methods that were on this trait into messages: the behavior receives the `ActorContext` as an argument during processing and the lifecycle hooks have been converted into Signals.

A side-effect of this is that behaviors can now be tested in isolation without having to be packaged into an Actor, tests can run fully synchronously without having to worry about timeouts and spurious failures. Another side-effect is that behaviors can nicely be composed and decorated, see the `And`, `Or`, `Widened`, `ContextAware` combinators; nothing about these is special or internal, new combinators can be written as external libraries or tailor-made for each project.

3.3 Fault Tolerance

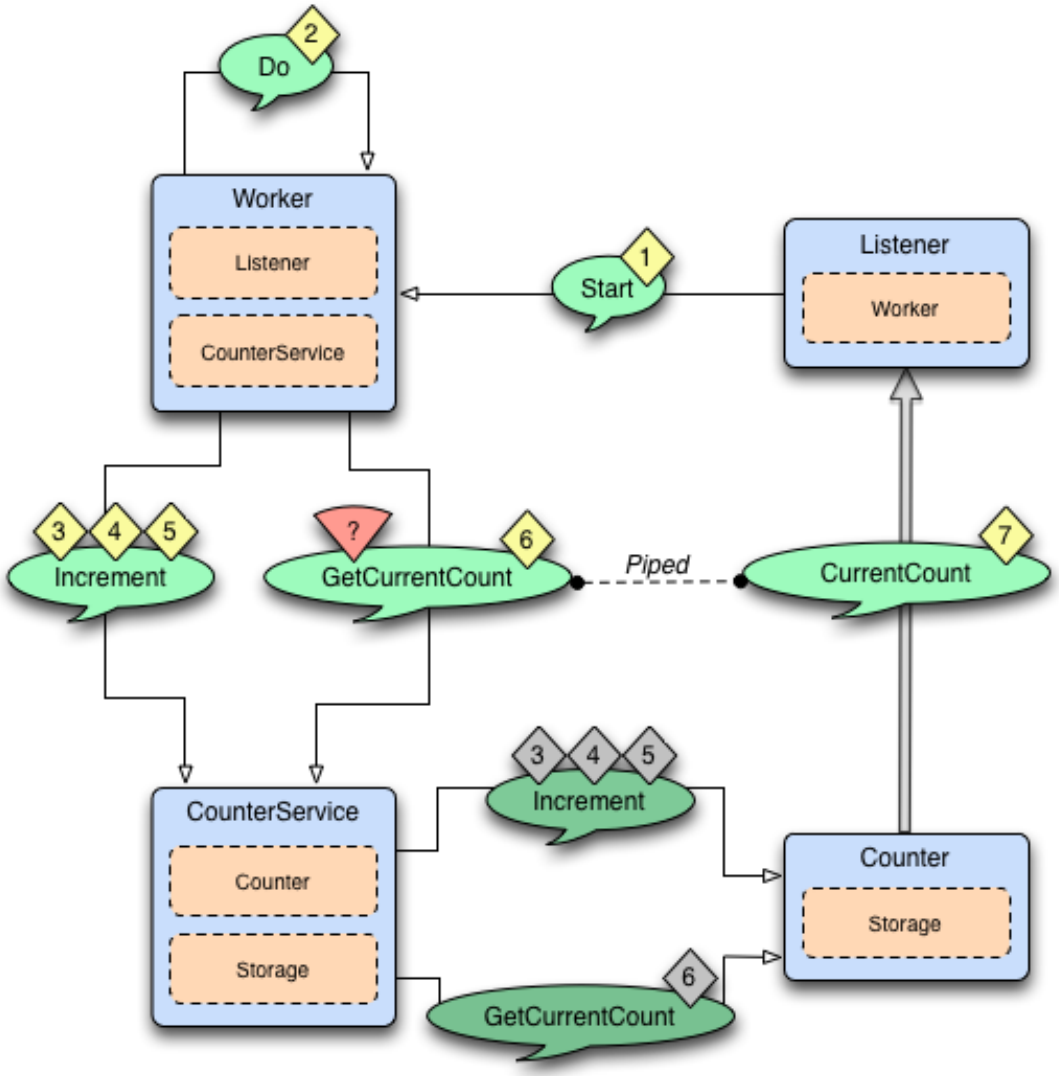
As explained in *Actor Systems* each actor is the supervisor of its children, and as such each actor defines fault handling supervisor strategy. This strategy cannot be changed afterwards as it is an integral part of the actor system's structure.

3.3.1 Fault Handling in Practice

First, let us look at a sample that illustrates one way to handle data store errors, which is a typical source of failure in real world applications. Of course it depends on the actual application what is possible to do when the data store is unavailable, but in this sample we use a best effort re-connect approach.

Read the following source code. The inlined comments explain the different pieces of the fault handling and why they are added. It is also highly recommended to run this sample as it is easy to follow the log output to understand what is happening at runtime.

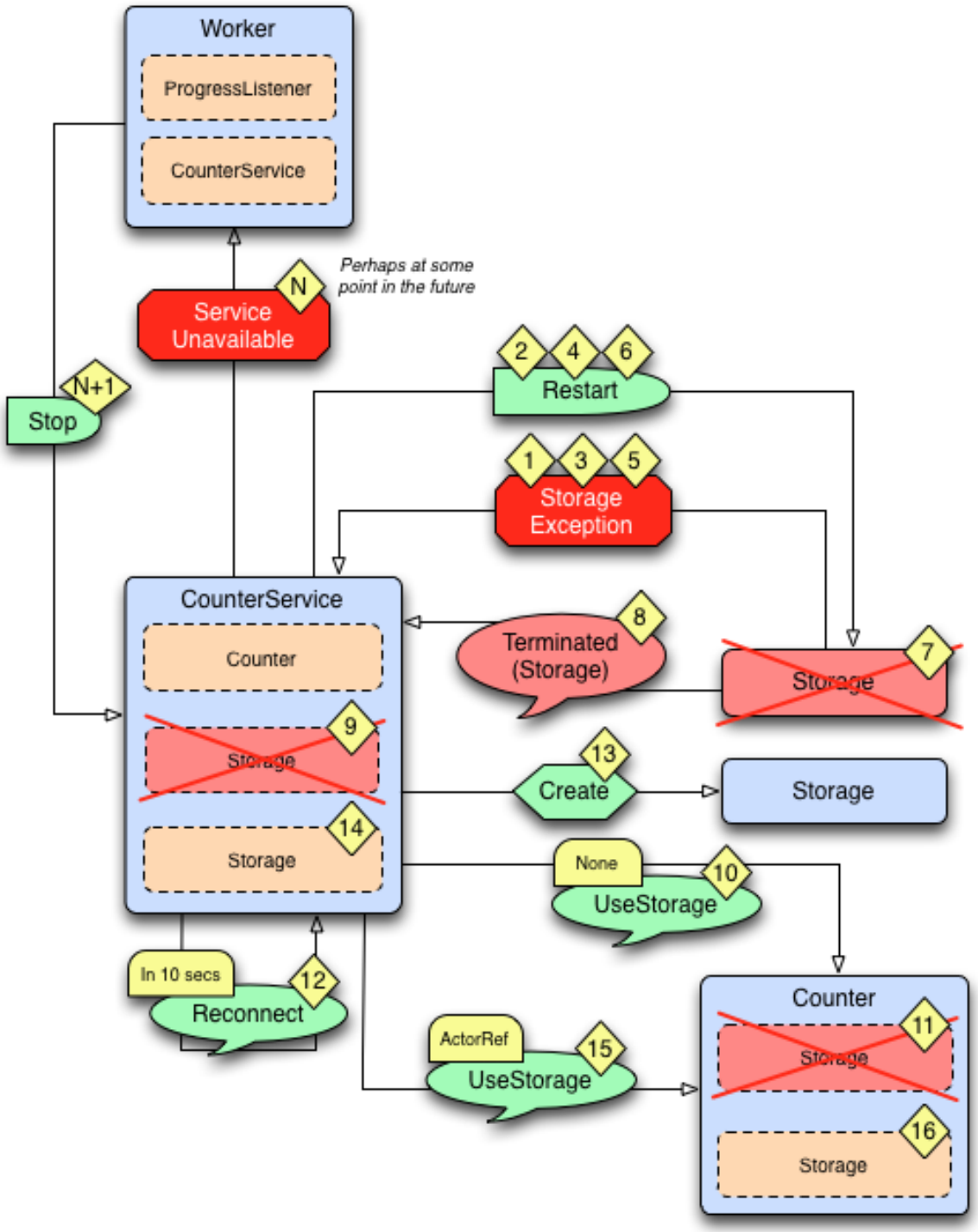
Diagrams of the Fault Tolerance Sample



The above diagram illustrates the normal message flow.

Normal flow:

Step	Description
1	The progress `Listener` starts the work.
2	The `Worker` schedules work by sending `Do` messages periodically to itself
3, 4, 5	When receiving `Do` the `Worker` tells the `CounterService` to increment the counter, three times. The `Increment` message is forwarded to the `Counter`, which updates its counter variable and sends current value to the `Storage`.
6, 7	The `Worker` asks the `CounterService` of current value of the counter and pipes the result back to the `Listener`.



The above diagram illustrates what happens in case of storage failure.

Failure flow:

Step	Description
1	The `Storage` throws `StorageException`.
2	The `CounterService` is supervisor of the `Storage` and restarts the `Storage` when `StorageException` is thrown.
3, 4, 5, 6	The `Storage` continues to fail and is restarted.
7	After 3 failures and restarts within 5 seconds the `Storage` is stopped by its supervisor, i.e. the `CounterService`.
8	The `CounterService` is also watching the `Storage` for termination and receives the `Terminated` message when the `Storage` has been stopped ...
9, 10, 11	and tells the `Counter` that there is no `Storage`.
12	The `CounterService` schedules a `Reconnect` message to itself.
13, 14	When it receives the `Reconnect` message it creates a new `Storage` ...
15, 16	and tells the `Counter` to use the new `Storage`

Full Source Code of the Fault Tolerance Sample

```
import akka.actor._
import akka.actor.SupervisorStrategy._
import scala.concurrent.duration._
import akka.util.Timeout
import akka.event.LoggingReceive
import akka.pattern.{ ask, pipe }
import com.typesafe.config.ConfigFactory

/**
 * Runs the sample
 */
object FaultHandlingDocSample extends App {
  import Worker._

  val config = ConfigFactory.parseString("""
    akka.loglevel = "DEBUG"
    akka.actor.debug {
      receive = on
      lifecycle = on
    }
    """)

  val system = ActorSystem("FaultToleranceSample", config)
  val worker = system.actorOf(Props[Worker], name = "worker")
  val listener = system.actorOf(Props[Listener], name = "listener")
  // start the work and listen on progress
  // note that the listener is used as sender of the tell,
  // i.e. it will receive replies from the worker
  worker.tell(Start, sender = listener)
}

/**
 * Listens on progress from the worker and shuts down the system when enough
 * work has been done.
 */
class Listener extends Actor with ActorLogging {
  import Worker._
  // If we don't get any progress within 15 seconds then the service is unavailable
  context.setReceiveTimeout(15 seconds)
```

```
  def receive = {
    case Progress(percent) =>
      log.info("Current progress: {} %", percent)
      if (percent >= 100.0) {
        log.info("That's all, shutting down")
        context.system.terminate()
      }

    case ReceiveTimeout =>
      // No progress within 15 seconds, ServiceUnavailable
      log.error("Shutting down due to unavailable service")
      context.system.terminate()
  }
}

object Worker {
  case object Start
  case object Do
  final case class Progress(percent: Double)
}

/**
 * Worker performs some work when it receives the `Start` message.
 * It will continuously notify the sender of the `Start` message
 * of current ``Progress``. The `Worker` supervise the `CounterService`.
 */
class Worker extends Actor with ActorLogging {
  import Worker._
  import CounterService._
  implicit val askTimeout = Timeout(5 seconds)

  // Stop the CounterService child if it throws ServiceUnavailable
  override val supervisorStrategy = OneForOneStrategy() {
    case _: CounterService.ServiceUnavailable => Stop
  }

  // The sender of the initial Start message will continuously be notified
  // about progress
  var progressListener: Option[ActorRef] = None
  val counterService = context.actorOf(Props[CounterService], name = "counter")
  val totalCount = 51
  import context.dispatcher // Use this Actors' Dispatcher as ExecutionContext

  def receive = LoggingReceive {
    case Start if progressListener.isEmpty =>
      progressListener = Some(sender())
      context.system.scheduler.schedule(Duration.Zero, 1 second, self, Do)

    case Do =>
      counterService ! Increment(1)
      counterService ! Increment(1)
      counterService ! Increment(1)

      // Send current progress to the initial sender
      counterService ? GetCurrentCount map {
        case CurrentCount(_, count) => Progress(100.0 * count / totalCount)
      } pipeTo progressListener.get
  }
}

object CounterService {
  final case class Increment(n: Int)
  sealed abstract class GetCurrentCount
```

```
  case object GetCurrentCount extends GetCurrentCount
  final case class CurrentCount(key: String, count: Long)
  class ServiceUnavailable(msg: String) extends RuntimeException(msg)

  private case object Reconnect
}

/**
 * Adds the value received in `Increment` message to a persistent
 * counter. Replies with `CurrentCount` when it is asked for `CurrentCount`.
 * `CounterService` supervise `Storage` and `Counter`.
 */
class CounterService extends Actor {
  import CounterService._
  import Counter._
  import Storage._

  // Restart the storage child when StorageException is thrown.
  // After 3 restarts within 5 seconds it will be stopped.
  override val supervisorStrategy = OneForOneStrategy(
    maxNrOfRetries = 3,
    withinTimeRange = 5 seconds) {
    case _: Storage.StorageException => Restart
  }

  val key = self.path.name
  var storage: Option[ActorRef] = None
  var counter: Option[ActorRef] = None
  var backlog = IndexedSeq.empty[(ActorRef, Any)]
  val MaxBacklog = 10000

  import context.dispatcher // Use this Actors' Dispatcher as ExecutionContext

  override def preStart() {
    initStorage()
  }

  /**
   * The child storage is restarted in case of failure, but after 3 restarts,
   * and still failing it will be stopped. Better to back-off than continuously
   * failing. When it has been stopped we will schedule a Reconnect after a delay.
   * Watch the child so we receive Terminated message when it has been terminated.
   */
  def initStorage() {
    storage = Some(context.watch(context.actorOf(Props[Storage], name = "storage
↪")))
    // Tell the counter, if any, to use the new storage
    counter foreach { _ ! UseStorage(storage) }
    // We need the initial value to be able to operate
    storage.get ! Get(key)
  }

  def receive = LoggingReceive {

    case Entry(k, v) if k == key && counter == None =>
      // Reply from Storage of the initial value, now we can create the Counter
      val c = context.actorOf(Props(classOf[Counter], key, v))
      counter = Some(c)
      // Tell the counter to use current storage
      c ! UseStorage(storage)
      // and send the buffered backlog to the counter
      for ((replyTo, msg) <- backlog) c.tell(msg, sender = replyTo)
      backlog = IndexedSeq.empty
```

```
    case msg: Increment       => forwardOrPlaceInBacklog(msg)

    case msg: GetCurrentCount => forwardOrPlaceInBacklog(msg)

    case Terminated(actorRef) if Some(actorRef) == storage =>
      // After 3 restarts the storage child is stopped.
      // We receive Terminated because we watch the child, see initStorage.
      storage = None
      // Tell the counter that there is no storage for the moment
      counter foreach { _ ! UseStorage(None) }
      // Try to re-establish storage after while
      context.system.scheduler.scheduleOnce(10 seconds, self, Reconnect)

    case Reconnect =>
      // Re-establish storage after the scheduled delay
      initStorage()
  }

  def forwardOrPlaceInBacklog(msg: Any) {
    // We need the initial value from storage before we can start delegate to
    // the counter. Before that we place the messages in a backlog, to be sent
    // to the counter when it is initialized.
    counter match {
      case Some(c) => c forward msg
      case None =>
        if (backlog.size >= MaxBacklog)
          throw new ServiceUnavailable(
            "CounterService not available, lack of initial value")
        backlog :+= (sender() -> msg)
    }
  }

}

object Counter {
  final case class UseStorage(storage: Option[ActorRef])
}

/**
 * The in memory count variable that will send current
 * value to the `Storage`, if there is any storage
 * available at the moment.
 */
class Counter(key: String, initialValue: Long) extends Actor {
  import Counter._
  import CounterService._
  import Storage._

  var count = initialValue
  var storage: Option[ActorRef] = None

  def receive = LoggingReceive {
    case UseStorage(s) =>
      storage = s
      storeCount()

    case Increment(n) =>
      count += n
      storeCount()

    case GetCurrentCount =>
      sender() ! CurrentCount(key, count)
```

```
  }

  def storeCount() {
    // Delegate dangerous work, to protect our valuable state.
    // We can continue without storage.
    storage foreach { _ ! Store(Entry(key, count)) }
  }

}

object Storage {
  final case class Store(entry: Entry)
  final case class Get(key: String)
  final case class Entry(key: String, value: Long)
  class StorageException(msg: String) extends RuntimeException(msg)
}

/**
 * Saves key/value pairs to persistent storage when receiving `Store` message.
 * Replies with current value when receiving `Get` message.
 * Will throw StorageException if the underlying data store is out of order.
 */
class Storage extends Actor {
  import Storage._

  val db = DummyDB

  def receive = LoggingReceive {
    case Store(Entry(key, count)) => db.save(key, count)
    case Get(key)                 => sender() ! Entry(key, db.load(key).
↪getOrElse(0L))
  }
}

object DummyDB {
  import Storage.StorageException
  private var db = Map[String, Long]()

  @throws(classOf[StorageException])
  def save(key: String, value: Long): Unit = synchronized {
    if (11 <= value && value <= 14)
      throw new StorageException("Simulated store failure " + value)
    db += (key -> value)
  }

  @throws(classOf[StorageException])
  def load(key: String): Option[Long] = synchronized {
    db.get(key)
  }
}
```

3.3.2 Creating a Supervisor Strategy

The following sections explain the fault handling mechanism and alternatives in more depth.

For the sake of demonstration let us consider the following strategy:

```
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._
import scala.concurrent.duration._
```

```
override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
    case _: ArithmeticException      => Resume
    case _: NullPointerException     => Restart
    case _: IllegalArgumentException => Stop
    case _: Exception                => Escalate
  }
```

I have chosen a few well-known exception types in order to demonstrate the application of the fault handling directives described in *Supervision and Monitoring*. First off, it is a one-for-one strategy, meaning that each child is treated separately (an all-for-one strategy works very similarly, the only difference is that any decision is applied to all children of the supervisor, not only the failing one). There are limits set on the restart frequency, namely maximum 10 restarts per minute; each of these settings could be left out, which means that the respective limit does not apply, leaving the possibility to specify an absolute upper limit on the restarts or to make the restarts work infinitely. The child actor is stopped if the limit is exceeded.

The match statement which forms the bulk of the body is of type `Decider`, which is a `PartialFunction[Throwable,Directive]`. This is the piece which maps child failure types to their corresponding directives.

Note: If the strategy is declared inside the supervising actor (as opposed to within a companion object) its decider has access to all internal state of the actor in a thread-safe fashion, including obtaining a reference to the currently failed child (available as the `sender` of the failure message).

Default Supervisor Strategy

`Escalate` is used if the defined strategy doesn't cover the exception that was thrown.

When the supervisor strategy is not defined for an actor the following exceptions are handled by default:

- `ActorInitializationException` will stop the failing child actor
- `ActorKilledException` will stop the failing child actor
- `Exception` will restart the failing child actor
- Other types of `Throwable` will be escalated to parent actor

If the exception escalate all the way up to the root guardian it will handle it in the same way as the default strategy defined above.

You can combine your own strategy with the default strategy:

```
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._
import scala.concurrent.duration._

override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
    case _: ArithmeticException => Resume
    case t =>
      super.supervisorStrategy.decider.applyOrElse(t, (_: Any) => Escalate)
  }
```

Stopping Supervisor Strategy

Closer to the Erlang way is the strategy to just stop children when they fail and then take corrective action in the supervisor when DeathWatch signals the loss of the child. This strategy is

also provided pre-packaged as `SupervisorStrategy.stoppingStrategy` with an accompanying `StoppingSupervisorStrategy` configurator to be used when you want the `"/user"` guardian to apply it.

Logging of Actor Failures

By default the `SupervisorStrategy` logs failures unless they are escalated. Escalated failures are supposed to be handled, and potentially logged, at a level higher in the hierarchy.

You can mute the default logging of a `SupervisorStrategy` by setting `loggingEnabled` to `false` when instantiating it. Customized logging can be done inside the `Decider`. Note that the reference to the currently failed child is available as the `sender` when the `SupervisorStrategy` is declared inside the supervising actor.

You may also customize the logging in your own `SupervisorStrategy` implementation by overriding the `logFailure` method.

3.3.3 Supervision of Top-Level Actors

Toplevel actors means those which are created using `system.actorOf()`, and they are children of the *User Guardian*. There are no special rules applied in this case, the guardian simply applies the configured strategy.

3.3.4 Test Application

The following section shows the effects of the different directives in practice, where a test setup is needed. First off, we need a suitable supervisor:

```
import akka.actor.Actor

class Supervisor extends Actor {
  import akka.actor.OneForOneStrategy
  import akka.actor.SupervisorStrategy._
  import scala.concurrent.duration._

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case _: ArithmeticException      => Resume
      case _: NullPointerException     => Restart
      case _: IllegalArgumentException => Stop
      case _: Exception                => Escalate
    }

  def receive = {
    case p: Props => sender() ! context.actorOf(p)
  }
}
```

This supervisor will be used to create a child, with which we can experiment:

```
import akka.actor.Actor

class Child extends Actor {
  var state = 0
  def receive = {
    case ex: Exception => throw ex
    case x: Int        => state = x
    case "get"         => sender() ! state
  }
}
```

The test is easier by using the utilities described in *Testing Actor Systems*.

```
import com.typesafe.config.{ Config, ConfigFactory }
import org.scalatest.{ FlatSpecLike, Matchers, BeforeAndAfterAll }
import akka.testkit.{ TestActors, TestKit, ImplicitSender, EventFilter }

class FaultHandlingDocSpec(_system: ActorSystem) extends TestKit(_system)
  with ImplicitSender with FlatSpecLike with Matchers with BeforeAndAfterAll {

  def this() = this(ActorSystem(
    "FaultHandlingDocSpec",
    ConfigFactory.parseString("""
      akka {
        loggers = ["akka.testkit.TestEventListener"]
        loglevel = "WARNING"
      }
      """)))

  override def afterAll {
    TestKit.shutdownActorSystem(system)
  }

  "A supervisor" must "apply the chosen strategy for its child" in {
    // code here
  }
}
```

Let us create actors:

```
val supervisor = system.actorOf(Props[Supervisor], "supervisor")

supervisor ! Props[Child]
val child = expectMsgType[ActorRef] // retrieve answer from TestKit's testActor
```

The first test shall demonstrate the `Resume` directive, so we try it out by setting some non-initial state in the actor and have it fail:

```
child ! 42 // set state to 42
child ! "get"
expectMsg(42)

child ! new ArithmeticException // crash it
child ! "get"
expectMsg(42)
```

As you can see the value 42 survives the fault handling directive. Now, if we change the failure to a more serious `NullPointerException`, that will no longer be the case:

```
child ! new NullPointerException // crash it harder
child ! "get"
expectMsg(0)
```

And finally in case of the fatal `IllegalArgumentException` the child will be terminated by the supervisor:

```
watch(child) // have testActor watch "child"
child ! new IllegalArgumentException // break it
expectMsgPF() { case Terminated(`child`) => () }
```

Up to now the supervisor was completely unaffected by the child's failure, because the directives set did handle it. In case of an `Exception`, this is not true anymore and the supervisor escalates the failure.

```
supervisor ! Props[Child] // create new child
val child2 = expectMsgType[ActorRef]
```

```
watch(child2)
child2 ! "get" // verify it is alive
expectMsg(0)

child2 ! new Exception("CRASH") // escalate failure
expectMsgPF() {
  case t @ Terminated(`child2`) if t.existenceConfirmed => ()
}
```

The supervisor itself is supervised by the top-level actor provided by the `ActorSystem`, which has the default policy to restart in case of all `Exception` cases (with the notable exceptions of `ActorInitializationException` and `ActorKilledException`). Since the default directive in case of a restart is to kill all children, we expected our poor child not to survive this failure.

In case this is not desired (which depends on the use case), we need to use a different supervisor which overrides this behavior.

```
class Supervisor2 extends Actor {
  import akka.actor.OneForOneStrategy
  import akka.actor.SupervisorStrategy._
  import scala.concurrent.duration._

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case _: ArithmeticException      => Resume
      case _: NullPointerException     => Restart
      case _: IllegalArgumentException => Stop
      case _: Exception                => Escalate
    }

  def receive = {
    case p: Props => sender() ! context.actorOf(p)
  }
  // override default to kill all children during restart
  override def preRestart(cause: Throwable, msg: Option[Any]) {}
}
```

With this parent, the child survives the escalated restart, as demonstrated in the last test:

```
val supervisor2 = system.actorOf(Props[Supervisor2], "supervisor2")

supervisor2 ! Props[Child]
val child3 = expectMsgType[ActorRef]

child3 ! 23
child3 ! "get"
expectMsg(23)

child3 ! new Exception("CRASH")
child3 ! "get"
expectMsg(0)
```

3.4 Dispatchers

An Akka `MessageDispatcher` is what makes Akka Actors "tick", it is the engine of the machine so to speak. All `MessageDispatcher` implementations are also an `ExecutionContext`, which means that they can be used to execute arbitrary code, for instance *Futures*.

3.4.1 Default dispatcher

Every `ActorSystem` will have a default dispatcher that will be used in case nothing else is configured for an `Actor`. The default dispatcher can be configured, and is by default a `Dispatcher` with the specified `default-executor`. If an ActorSystem is created with an ExecutionContext passed in, this ExecutionContext will be used as the default executor for all dispatchers in this ActorSystem. If no ExecutionContext is given, it will fallback to the executor specified in `akka.actor.default-dispatcher.default-executor.fallback`. By default this is a "fork-join-executor", which gives excellent performance in most cases.

3.4.2 Looking up a Dispatcher

Dispatchers implement the `ExecutionContext` interface and can thus be used to run `Future` invocations etc.

```
// for use with Futures, Scheduler, etc.
implicit val executionContext = system.dispatchers.lookup("my-dispatcher")
```

3.4.3 Setting the dispatcher for an Actor

So in case you want to give your `Actor` a different dispatcher than the default, you need to do two things, of which the first is to configure the dispatcher:

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

Note: Note that the `parallelism-max` does not set the upper bound on the total number of threads allocated by the ForkJoinPool. It is a setting specifically talking about the number of *hot* threads the pool keep running in order to reduce the latency of handling a new incoming task. You can read more about parallelism in the JDK's ForkJoinPool documentation.

And here's another example that uses the "thread-pool-executor":

```
my-thread-pool-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "thread-pool-executor"
  # Configuration for the thread pool
  thread-pool-executor {
    # minimum number of threads to cap factor-based core number to
```

```
    core-pool-size-min = 2
    # No of core threads ... ceil(available processors * factor)
    core-pool-size-factor = 2.0
    # maximum number of threads to cap factor-based number to
    core-pool-size-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

Note: The thread pool executor dispatcher is implemented using by a `java.util.concurrent.ThreadPoolExecutor`. You can read more about it in the JDK's ThreadPoolExecutor documentation.

For more options, see the default-dispatcher section of the *Configuration*.

Then you create the actor as usual and define the dispatcher in the deployment configuration.

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "myactor")
```

```
akka.actor.deployment {
  /myactor {
    dispatcher = my-dispatcher
  }
}
```

An alternative to the deployment configuration is to define the dispatcher in code. If you define the `dispatcher` in the deployment configuration then this value will be used instead of programmatically provided parameter.

```
import akka.actor.Props
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), "myactor1")
```

Note: The dispatcher you specify in `withDispatcher` and the `dispatcher` property in the deployment configuration is in fact a path into your configuration. So in this example it's a top-level section, but you could for instance put it as a sub-section, where you'd use periods to denote sub-sections, like this: `"foo.bar.my-dispatcher"`

3.4.4 Types of dispatchers

There are 3 different types of message dispatchers:

- Dispatcher
  - This is an event-based dispatcher that binds a set of Actors to a thread pool. It is the default dispatcher used if one is not specified.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor
  - Use cases: Default dispatcher, Bulkheading
  - **Driven by:** `java.util.concurrent.ExecutorService` specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`

- PinnedDispatcher
  - This dispatcher dedicates a unique thread for each actor using it; i.e. each actor will have its own thread pool with only one thread in the pool.
  - Sharability: None
  - Mailboxes: Any, creates one per Actor
  - Use cases: Bulkheading
  - **Driven by: Any `akka.dispatch.ThreadPoolExecutorConfigurator`** by default a "thread-pool-executor"
- BalancingDispatcher
  - This is an executor based event driven dispatcher that will try to redistribute work from busy actors to idle actors.
  - All the actors share a single Mailbox that they get their messages from.
  - It is assumed that all actors using the same instance of this dispatcher can process all messages that have been sent to one of the actors; i.e. the actors belong to a pool of actors, and to the client there is no guarantee about which actor instance actually processes a given message.
  - Sharability: Actors of the same type only
  - Mailboxes: Any, creates one for all Actors
  - Use cases: Work-sharing
  - **Driven by: `java.util.concurrent.ExecutorService`** specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`
  - Note that you can **not** use a `BalancingDispatcher` as a **Router Dispatcher**. (You can however use it for the **Routees**)
- CallingThreadDispatcher
  - This dispatcher runs invocations on the current thread only. This dispatcher does not create any new threads, but it can be used from different threads concurrently for the same actor. See *CallingThreadDispatcher* for details and restrictions.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor per Thread (on demand)
  - Use cases: Testing
  - Driven by: The calling thread (duh)

More dispatcher configuration examples

Configuring a dispatcher with fixed thread pool size, e.g. for actors that perform blocking IO:

```
blocking-io-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executor {
    fixed-pool-size = 32
  }
  throughput = 1
}
```

And then using it:

```
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("blocking-io-dispatcher"),
↪"myactor2")
```

Configuring a `PinnedDispatcher`:

```
my-pinned-dispatcher {
  executor = "thread-pool-executor"
  type = PinnedDispatcher
}
```

And then using it:

```
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-pinned-dispatcher"), "myactor3
↪")
```

Note that `thread-pool-executor` configuration as per the above `my-thread-pool-dispatcher` example is NOT applicable. This is because every actor will have its own thread pool when using `PinnedDispatcher`, and that pool will have only one thread.

Note that it's not guaranteed that the *same* thread is used over time, since the core pool timeout is used for `PinnedDispatcher` to keep resource usage down in case of idle actors. To use the same thread all the time you need to add `thread-pool-executor.allow-core-timeout=off` to the configuration of the `PinnedDispatcher`.

3.5 Mailboxes

An Akka `Mailbox` holds the messages that are destined for an `Actor`. Normally each `Actor` has its own mailbox, but with for example a `BalancingPool` all routees will share a single mailbox instance.

3.5.1 Mailbox Selection

Requiring a Message Queue Type for an Actor

It is possible to require a certain type of message queue for a certain type of actor by having that actor extend the parameterized trait `RequiresMessageQueue`. Here is an example:

```
import akka.dispatch.RequiresMessageQueue
import akka.dispatch.BoundedMessageQueueSemantics

class MyBoundedActor extends MyActor
  with RequiresMessageQueue[BoundedMessageQueueSemantics]
```

The type parameter to the `RequiresMessageQueue` trait needs to be mapped to a mailbox in configuration like this:

```
bounded-mailbox {
  mailbox-type = "akka.dispatch.BoundedMailbox"
  mailbox-capacity = 1000
  mailbox-push-timeout-time = 10s
}

akka.actor.mailbox.requirements {
  "akka.dispatch.BoundedMessageQueueSemantics" = bounded-mailbox
}
```

Now every time you create an actor of type `MyBoundedActor` it will try to get a bounded mailbox. If the actor has a different mailbox configured in deployment, either directly or via a dispatcher with a specified mailbox type, then that will override this mapping.

Note: The type of the queue in the mailbox created for an actor will be checked against the required type in the trait and if the queue doesn't implement the required type then actor creation will fail.

Requiring a Message Queue Type for a Dispatcher

A dispatcher may also have a requirement for the mailbox type used by the actors running on it. An example is the BalancingDispatcher which requires a message queue that is thread-safe for multiple concurrent consumers. Such a requirement is formulated within the dispatcher configuration section like this:

```
my-dispatcher {
  mailbox-requirement = org.example.MyInterface
}
```

The given requirement names a class or interface which will then be ensured to be a supertype of the message queue's implementation. In case of a conflict—e.g. if the actor requires a mailbox type which does not satisfy this requirement—then actor creation will fail.

How the Mailbox Type is Selected

When an actor is created, the `ActorRefProvider` first determines the dispatcher which will execute it. Then the mailbox is determined as follows:

1. If the actor's deployment configuration section contains a `mailbox` key then that names a configuration section describing the mailbox type to be used.
2. If the actor's `Props` contains a mailbox selection—i.e. `withMailbox` was called on it—then that names a configuration section describing the mailbox type to be used.
3. If the dispatcher's configuration section contains a `mailbox-type` key the same section will be used to configure the mailbox type.
4. If the actor requires a mailbox type as described above then the mapping for that requirement will be used to determine the mailbox type to be used; if that fails then the dispatcher's requirement—if any—will be tried instead.
5. If the dispatcher requires a mailbox type as described above then the mapping for that requirement will be used to determine the mailbox type to be used.
6. The default mailbox `akka.actor.default-mailbox` will be used.

Default Mailbox

When the mailbox is not specified as described above the default mailbox is used. By default it is an unbounded mailbox, which is backed by a `java.util.concurrent.ConcurrentLinkedQueue`.

`SingleConsumerOnlyUnboundedMailbox` is an even more efficient mailbox, and it can be used as the default mailbox, but it cannot be used with a BalancingDispatcher.

Configuration of `SingleConsumerOnlyUnboundedMailbox` as default mailbox:

```
akka.actor.default-mailbox {
  mailbox-type = "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
}
```

Which Configuration is passed to the Mailbox Type

Each mailbox type is implemented by a class which extends `MailboxType` and takes two constructor arguments: a `ActorSystem.Settings` object and a `Config` section. The latter is computed by obtaining the named configuration section from the actor system's configuration, overriding its `id` key with the configuration path of the mailbox type and adding a fall-back to the default mailbox configuration section.

3.5.2 Builtin Mailbox Implementations

Akka comes shipped with a number of mailbox implementations:

- **UnboundedMailbox** (default)
  - The default mailbox
  - Backed by a `java.util.concurrent.ConcurrentLinkedQueue`
  - Blocking: No
  - Bounded: No
  - Configuration name: `"unbounded"` or `"akka.dispatch.UnboundedMailbox"`
- **SingleConsumerOnlyUnboundedMailbox**

  This queue may or may not be faster than the default one depending on your use-case—be sure to benchmark properly!
  - Backed by a Multiple-Producer Single-Consumer queue, cannot be used with `BalancingDispatcher`
  - Blocking: No
  - Bounded: No
  - Configuration name: `"akka.dispatch.SingleConsumerOnlyUnboundedMailbox"`
- **NonBlockingBoundedMailbox**
  - Backed by a very efficient Multiple-Producer Single-Consumer queue
  - Blocking: No (discards overflowing messages into deadLetters)
  - Bounded: Yes
  - Configuration name: `"akka.dispatch.NonBlockingBoundedMailbox"`
- **UnboundedControlAwareMailbox**
  - Delivers messages that extend `akka.dispatch.ControlMessage` with higher priority
  - Backed by two `java.util.concurrent.ConcurrentLinkedQueue`
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedControlAwareMailbox"
- **UnboundedPriorityMailbox**
  - Backed by a `java.util.concurrent.PriorityBlockingQueue`
  - Delivery order for messages of equal priority is undefined - contrast with the UnboundedStablePriorityMailbox
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedPriorityMailbox"

- **UnboundedStablePriorityMailbox**
  - Backed by a `java.util.concurrent.PriorityBlockingQueue` wrapped in an `akka.util.PriorityQueueStabilizer`
  - FIFO order is preserved for messages of equal priority - contrast with the UnboundedPriorityMailbox
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedStablePriorityMailbox"

Other bounded mailbox implementations which will block the sender if the capacity is reached and configured with non-zero `mailbox-push-timeout-time`.

Note: The following mailboxes should only be used with zero `mailbox-push-timeout-time`.

- **BoundedMailbox**
  - Backed by a `java.util.concurrent.LinkedBlockingQueue`
  - Blocking: Yes if used with non-zero `mailbox-push-timeout-time`, otherwise No
  - Bounded: Yes
  - Configuration name: "bounded" or "akka.dispatch.BoundedMailbox"
- **BoundedPriorityMailbox**
  - Backed by a `java.util.PriorityQueue` wrapped in an `akka.util.BoundedBlockingQueue`
  - Delivery order for messages of equal priority is undefined - contrast with the `BoundedStablePriorityMailbox`
  - Blocking: Yes if used with non-zero `mailbox-push-timeout-time`, otherwise No
  - Bounded: Yes
  - Configuration name: `"akka.dispatch.BoundedPriorityMailbox"`
- **BoundedStablePriorityMailbox**
  - Backed by a `java.util.PriorityQueue` wrapped in an `akka.util.PriorityQueueStabilizer` and an `akka.util.BoundedBlockingQueue`
  - FIFO order is preserved for messages of equal priority - contrast with the BoundedPriorityMailbox
  - Blocking: Yes if used with non-zero `mailbox-push-timeout-time`, otherwise No
  - Bounded: Yes
  - Configuration name: "akka.dispatch.BoundedStablePriorityMailbox"
- **BoundedControlAwareMailbox**
  - Delivers messages that extend `akka.dispatch.ControlMessage` with higher priority
  - Backed by two `java.util.concurrent.ConcurrentLinkedQueue` and blocking on enqueue if capacity has been reached
  - Blocking: Yes if used with non-zero `mailbox-push-timeout-time`, otherwise No
  - Bounded: Yes
  - Configuration name: "akka.dispatch.BoundedControlAwareMailbox"

3.5.3 Mailbox configuration examples

PriorityMailbox

How to create a PriorityMailbox:

```
import akka.dispatch.PriorityGenerator
import akka.dispatch.UnboundedStablePriorityMailbox
import com.typesafe.config.Config

// We inherit, in this case, from UnboundedStablePriorityMailbox
// and seed it with the priority generator
class MyPrioMailbox(settings: ActorSystem.Settings, config: Config)
  extends UnboundedStablePriorityMailbox(
    // Create a new PriorityGenerator, lower prio means more important
    PriorityGenerator {
      // 'highpriority messages should be treated first if possible
      case 'highpriority => 0

      // 'lowpriority messages should be treated last if possible
      case 'lowpriority  => 2

      // PoisonPill when no other left
      case PoisonPill    => 3

      // We default to 1, which is in between high and low
      case otherwise     => 1
    })
```

And then add it to the configuration:

```
prio-dispatcher {
  mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
  //Other dispatcher configuration goes here
}
```

And then an example on how you would use it:

```
// We create a new Actor that just prints out what it processes
class Logger extends Actor {
  val log: LoggingAdapter = Logging(context.system, this)

  self ! 'lowpriority
  self ! 'lowpriority
  self ! 'highpriority
  self ! 'pigdog
  self ! 'pigdog2
  self ! 'pigdog3
  self ! 'highpriority
  self ! PoisonPill

  def receive = {
    case x => log.info(x.toString)
  }
}
val a = system.actorOf(Props(classOf[Logger], this).withDispatcher(
  "prio-dispatcher"))

/*
 * Logs:
 * 'highpriority
 * 'highpriority
 * 'pigdog
```

```
 * 'pigdog2
 * 'pigdog3
 * 'lowpriority
 * 'lowpriority
 */
```

It is also possible to configure a mailbox type directly like this:

```
prio-mailbox {
  mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
  //Other mailbox configuration goes here
}

akka.actor.deployment {
  /priomailboxactor {
    mailbox = prio-mailbox
  }
}
```

And then use it either from deployment like this:

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "priomailboxactor")
```

Or code like this:

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor].withMailbox("prio-mailbox"))
```

ControlAwareMailbox

A `ControlAwareMailbox` can be very useful if an actor needs to be able to receive control messages immediately no matter how many other messages are already in its mailbox.

It can be configured like this:

```
control-aware-dispatcher {
  mailbox-type = "akka.dispatch.UnboundedControlAwareMailbox"
  //Other dispatcher configuration goes here
}
```

Control messages need to extend the `ControlMessage` trait:

```
import akka.dispatch.ControlMessage

case object MyControlMessage extends ControlMessage
```

And then an example on how you would use it:

```
// We create a new Actor that just prints out what it processes
class Logger extends Actor {
  val log: LoggingAdapter = Logging(context.system, this)

  self ! 'foo
  self ! 'bar
  self ! MyControlMessage
  self ! PoisonPill

  def receive = {
    case x => log.info(x.toString)
  }
```

```
}
val a = system.actorOf(Props(classOf[Logger], this).withDispatcher(
  "control-aware-dispatcher"))

/*
 * Logs:
 * MyControlMessage
 * 'foo
 * 'bar
 */
```

3.5.4 Creating your own Mailbox type

An example is worth a thousand quacks:

```
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.dispatch.Envelope
import akka.dispatch.MailboxType
import akka.dispatch.MessageQueue
import akka.dispatch.ProducesMessageQueue
import com.typesafe.config.Config
import java.util.concurrent.ConcurrentLinkedQueue
import scala.Option

// Marker trait used for mailbox requirements mapping
trait MyUnboundedMessageQueueSemantics

object MyUnboundedMailbox {
  // This is the MessageQueue implementation
  class MyMessageQueue extends MessageQueue
    with MyUnboundedMessageQueueSemantics {

    private final val queue = new ConcurrentLinkedQueue[Envelope]()

    // these should be implemented; queue used as example
    def enqueue(receiver: ActorRef, handle: Envelope): Unit =
      queue.offer(handle)
    def dequeue(): Envelope = queue.poll()
    def numberOfMessages: Int = queue.size
    def hasMessages: Boolean = !queue.isEmpty
    def cleanUp(owner: ActorRef, deadLetters: MessageQueue) {
      while (hasMessages) {
        deadLetters.enqueue(owner, dequeue())
      }
    }
  }
}

// This is the Mailbox implementation
class MyUnboundedMailbox extends MailboxType
  with ProducesMessageQueue[MyUnboundedMailbox.MyMessageQueue] {

  import MyUnboundedMailbox._

  // This constructor signature must exist, it will be called by Akka
  def this(settings: ActorSystem.Settings, config: Config) = {
    // put your initialization code here
    this()
  }
```

```
  // The create method is called to create the MessageQueue
  final override def create(
    owner:  Option[ActorRef],
    system: Option[ActorSystem]): MessageQueue =
    new MyMessageQueue()
}
```

And then you just specify the FQCN of your MailboxType as the value of the "mailbox-type" in the dispatcher configuration, or the mailbox configuration.

Note: Make sure to include a constructor which takes `akka.actor.ActorSystem.Settings` and `com.typesafe.config.Config` arguments, as this constructor is invoked reflectively to construct your mailbox type. The config passed in as second argument is that section from the configuration which describes the dispatcher or mailbox setting using this mailbox type; the mailbox type will be instantiated once for each dispatcher or mailbox setting using it.

You can also use the mailbox as a requirement on the dispatcher like this:

```
custom-dispatcher {
  mailbox-requirement =
  "docs.dispatcher.MyUnboundedJMessageQueueSemantics"
}

akka.actor.mailbox.requirements {
  "docs.dispatcher.MyUnboundedJMessageQueueSemantics" =
  custom-dispatcher-mailbox
}

custom-dispatcher-mailbox {
  mailbox-type = "docs.dispatcher.MyUnboundedJMailbox"
}
```

Or by defining the requirement on your actor class like this:

```
class MySpecialActor extends Actor
  with RequiresMessageQueue[MyUnboundedMessageQueueSemantics] {
  // ...
}
```

3.5.5 Special Semantics of `system.actorOf`

In order to make `system.actorOf` both synchronous and non-blocking while keeping the return type `ActorRef` (and the semantics that the returned ref is fully functional), special handling takes place for this case. Behind the scenes, a hollow kind of actor reference is constructed, which is sent to the system's guardian actor who actually creates the actor and its context and puts those inside the reference. Until that has happened, messages sent to the `ActorRef` will be queued locally, and only upon swapping the real filling in will they be transferred into the real mailbox. Thus,

```
val props: Props = ...
// this actor uses MyCustomMailbox, which is assumed to be a singleton
system.actorOf(props.withDispatcher("myCustomMailbox")) ! "bang"
assert(MyCustomMailbox.instance.getLastEnqueuedMessage == "bang")
```

will probably fail; you will have to allow for some time to pass and retry the check à la `TestKit.awaitCond`.

3.6 Routing

Messages can be sent via a router to efficiently route them to destination actors, known as its *routees*. A `Router` can be used inside or outside of an actor, and you can manage the routees yourselves or use a self contained router actor with configuration capabilities.

Different routing strategies can be used, according to your application's needs. Akka comes with several useful routing strategies right out of the box. But, as you will see in this chapter, it is also possible to *create your own*.

3.6.1 A Simple Router

The following example illustrates how to use a `Router` and manage the routees from within an actor.

```
import akka.routing.{ ActorRefRoutee, RoundRobinRoutingLogic, Router }

class Master extends Actor {
  var router = {
    val routees = Vector.fill(5) {
      val r = context.actorOf(Props[Worker])
      context watch r
      ActorRefRoutee(r)
    }
    Router(RoundRobinRoutingLogic(), routees)
  }

  def receive = {
    case w: Work =>
      router.route(w, sender())
    case Terminated(a) =>
      router = router.removeRoutee(a)
      val r = context.actorOf(Props[Worker])
      context watch r
      router = router.addRoutee(r)
  }
}
```

We create a `Router` and specify that it should use `RoundRobinRoutingLogic` when routing the messages to the routees.

The routing logic shipped with Akka are:

- `akka.routing.RoundRobinRoutingLogic`
- `akka.routing.RandomRoutingLogic`
- `akka.routing.SmallestMailboxRoutingLogic`
- `akka.routing.BroadcastRoutingLogic`
- `akka.routing.ScatterGatherFirstCompletedRoutingLogic`
- `akka.routing.TailChoppingRoutingLogic`
- `akka.routing.ConsistentHashingRoutingLogic`

We create the routees as ordinary child actors wrapped in `ActorRefRoutee`. We watch the routees to be able to replace them if they are terminated.

Sending messages via the router is done with the `route` method, as is done for the `Work` messages in the example above.

The `Router` is immutable and the `RoutingLogic` is thread safe; meaning that they can also be used outside of actors.

Note: In general, any message sent to a router will be sent onwards to its routees, but there is one exception. The special *Broadcast Messages* will send to *all* of a router's routees. However, do not use *Broadcast Messages* when you use *BalancingPool* for routees as described in *Specially Handled Messages*.

3.6.2 A Router Actor

A router can also be created as a self contained actor that manages the routees itself and loads routing logic and other settings from configuration.

This type of router actor comes in two distinct flavors:

- Pool - The router creates routees as child actors and removes them from the router if they terminate.
- Group - The routee actors are created externally to the router and the router sends messages to the specified path using actor selection, without watching for termination.

The settings for a router actor can be defined in configuration or programmatically. In order to make an actor to make use of an externally configurable router the `FromConfig` props wrapper must be used to denote that the actor accepts routing settings from configuration. This is in contrast with Remote Deployment where such marker props is not necessary. If the props of an actor is NOT wrapped in FromConfig it will ignore the router section of the deployment configuration.

You send messages to the routees via the router actor in the same way as for ordinary actors, i.e. via its `ActorRef`. The router actor forwards messages onto its routees without changing the original sender. When a routee replies to a routed message, the reply will be sent to the original sender, not to the router actor.

Note: In general, any message sent to a router will be sent onwards to its routees, but there are a few exceptions. These are documented in the *Specially Handled Messages* section below.

Pool

The following code and configuration snippets show how to create a *round-robin* router that forwards messages to five `Worker` routees. The routees will be created as the router's children.

```
akka.actor.deployment {
  /parent/router1 {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

```
val router1: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

Here is the same example, but with the router configuration provided programmatically instead of from configuration.

```
val router2: ActorRef =
  context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

Remote Deployed Routees

In addition to being able to create local actors as routees, you can instruct the router to deploy its created children on a set of remote hosts. Routees will be deployed in round-robin fashion. In order to deploy routees remotely,

wrap the router configuration in a `RemoteRouterConfig`, attaching the remote addresses of the nodes to deploy to. Remote deployment requires the `akka-remote` module to be included in the classpath.

```
import akka.actor.{ Address, AddressFromURIString }
import akka.remote.routing.RemoteRouterConfig
val addresses = Seq(
  Address("akka.tcp", "remotesys", "otherhost", 1234),
  AddressFromURIString("akka.tcp://othersys@anotherhost:1234"))
val routerRemote = system.actorOf(
  RemoteRouterConfig(RoundRobinPool(5), addresses).props(Props[Echo]))
```

Senders

By default, when a routee sends a message, it will *implicitly set itself as the sender*.

```
sender() ! x // replies will go to this actor
```

However, it is often useful for routees to set the *router* as a sender. For example, you might want to set the router as the sender if you want to hide the details of the routees behind the router. The following code snippet shows how to set the parent router as sender.

```
sender().tell("reply", context.parent) // replies will go back to parent
sender().!("reply")(context.parent) // alternative syntax (beware of the parens!)
```

Supervision

Routees that are created by a pool router will be created as the router's children. The router is therefore also the children's supervisor.

The supervision strategy of the router actor can be configured with the `supervisorStrategy` property of the Pool. If no configuration is provided, routers default to a strategy of "always escalate". This means that errors are passed up to the router's supervisor for handling. The router's supervisor will decide what to do about any errors.

Note the router's supervisor will treat the error as an error with the router itself. Therefore a directive to stop or restart will cause the router *itself* to stop or restart. The router, in turn, will cause its children to stop and restart.

It should be mentioned that the router's restart behavior has been overridden so that a restart, while still re-creating the children, will still preserve the same number of actors in the pool.

This means that if you have not specified `supervisorStrategy` of the router or its parent a failure in a routee will escalate to the parent of the router, which will by default restart the router, which will restart all routees (it uses Escalate and does not stop routees during restart). The reason is to make the default behave such that adding `withRouter` to a child's definition does not change the supervision strategy applied to the child. This might be an inefficiency that you can avoid by specifying the strategy when defining the router.

Setting the strategy is easily done:

```
val escalator = OneForOneStrategy() {
  case e ⇒ testActor ! e; SupervisorStrategy.Escalate
}
val router = system.actorOf(RoundRobinPool(1, supervisorStrategy = escalator).
↪props(
  routeeProps = Props[TestActor]))
```

Note: If the child of a pool router terminates, the pool router will not automatically spawn a new child. In the event that all children of a pool router have terminated the router will terminate itself unless it is a dynamic router, e.g. using a resizer.

Group

Sometimes, rather than having the router actor create its routees, it is desirable to create routees separately and provide them to the router for its use. You can do this by passing an paths of the routees to the router's configuration. Messages will be sent with `ActorSelection` to these paths.

The example below shows how to create a router by providing it with the path strings of three routee actors.

```
akka.actor.deployment {
  /parent/router3 {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router3: ActorRef =
  context.actorOf(FromConfig.props(), "router3")
```

Here is the same example, but with the router configuration provided programmatically instead of from configuration.

```
val router4: ActorRef =
  context.actorOf(RoundRobinGroup(paths).props(), "router4")
```

The routee actors are created externally from the router:

```
system.actorOf(Props[Workers], "workers")
```

```
class Workers extends Actor {
  context.actorOf(Props[Worker], name = "w1")
  context.actorOf(Props[Worker], name = "w2")
  context.actorOf(Props[Worker], name = "w3")
  // ...
```

The paths may contain protocol and address information for actors running on remote hosts. Remoting requires the `akka-remote` module to be included in the classpath.

```
akka.actor.deployment {
  /parent/remoteGroup {
    router = round-robin-group
    routees.paths = [
      "akka.tcp://app@10.0.0.1:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.2:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.3:2552/user/workers/w1"]
  }
}
```

3.6.3 Router usage

In this section we will describe how to create the different types of router actors.

The router actors in this section are created from within a top level actor named `parent`. Note that deployment paths in the configuration starts with `/parent/` followed by the name of the router actor.

```
system.actorOf(Props[Parent], "parent")
```

RoundRobinPool and RoundRobinGroup

Routes in a round-robin fashion to its routees.

RoundRobinPool defined in configuration:

```
akka.actor.deployment {
  /parent/router1 {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

```
val router1: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

RoundRobinPool defined in code:

```
val router2: ActorRef =
  context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

RoundRobinGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router3 {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router3: ActorRef =
  context.actorOf(FromConfig.props(), "router3")
```

RoundRobinGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router4: ActorRef =
  context.actorOf(RoundRobinGroup(paths).props(), "router4")
```

RandomPool and RandomGroup

This router type selects one of its routees randomly for each message.

RandomPool defined in configuration:

```
akka.actor.deployment {
  /parent/router5 {
    router = random-pool
    nr-of-instances = 5
  }
}
```

```
val router5: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router5")
```

RandomPool defined in code:

```
val router6: ActorRef =
  context.actorOf(RandomPool(5).props(Props[Worker]), "router6")
```

RandomGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router7 {
```

```
    router = random-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router7: ActorRef =
  context.actorOf(FromConfig.props(), "router7")
```

RandomGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router8: ActorRef =
  context.actorOf(RandomGroup(paths).props(), "router8")
```

BalancingPool

A Router that will try to redistribute work from busy routees to idle routees. All routees share the same mailbox.

Note: The BalancingPool has the property that its routees do not have truly distinct identity: they have different names, but talking to them will not end up at the right actor in most cases. Therefore you cannot use it for workflows that require state to be kept within the routee, you would in this case have to include the whole state in the messages.

With a *SmallestMailboxPool* you can have a vertically scaling service that can interact in a stateful fashion with other services in the back-end before replying to the original client. The other advantage is that it does not place a restriction on the message queue implementation as BalancingPool does.

Note: Do not use *Broadcast Messages* when you use *BalancingPool* for routers. as described in *Specially Handled Messages*,

---

BalancingPool defined in configuration:

```
akka.actor.deployment {
  /parent/router9 {
    router = balancing-pool
    nr-of-instances = 5
  }
}
```

```
val router9: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router9")
```

BalancingPool defined in code:

```
val router10: ActorRef =
  context.actorOf(BalancingPool(5).props(Props[Worker]), "router10")
```

Addition configuration for the balancing dispatcher, which is used by the pool, can be configured in the `pool-dispatcher` section of the router deployment configuration.

```
akka.actor.deployment {
  /parent/router9b {
    router = balancing-pool
    nr-of-instances = 5
    pool-dispatcher {
      attempt-teamwork = off
```

```
    }
  }
}
```

The `BalancingPool` automatically uses a special `BalancingDispatcher` for its routees - disregarding any dispatcher that is set on the routee Props object. This is needed in order to implement the balancing semantics via sharing the same mailbox by all the routees.

While it is not possible to change the dispatcher used by the routees, it is possible to fine tune the used *executor*. By default the `fork-join-dispatcher` is used and can be configured as explained in *Dispatchers*. In situations where the routees are expected to perform blocking operations it may be useful to replace it with a `thread-pool-executor` hinting the number of allocated threads explicitly:

```
akka.actor.deployment {
  /parent/router10b {
    router = balancing-pool
    nr-of-instances = 5
    pool-dispatcher {
      executor = "thread-pool-executor"

      # allocate exactly 5 threads for this pool
      thread-pool-executor {
        core-pool-size-min = 5
        core-pool-size-max = 5
      }
    }
  }
}
```

It is also possible to change the `mailbox` used by the balancing dispatcher for scenarios where the default unbounded mailbox is not well suited. An example of such a scenario could arise whether there exists the need to manage priority for each message. You can then implement a priority mailbox and configure your dispatcher:

```
akka.actor.deployment {
  /parent/router10c {
    router = balancing-pool
    nr-of-instances = 5
    pool-dispatcher {
      mailbox = myapp.myprioritymailbox
    }
  }
}
```

Note: Bear in mind that `BalancingDispatcher` requires a message queue that must be thread-safe for multiple concurrent consumers. So it is mandatory for the message queue backing a custom mailbox for this kind of dispatcher to implement akka.dispatch.MultipleConsumerSemantics. See details on how to implement your custom mailbox in *Mailboxes*.

There is no Group variant of the BalancingPool.

SmallestMailboxPool

A Router that tries to send to the non-suspended child routee with fewest messages in mailbox. The selection is done in this order:

- pick any idle routee (not processing message) with empty mailbox
- pick any routee with empty mailbox
- pick routee with fewest pending messages in mailbox

- pick any remote routee, remote actors are consider lowest priority, since their mailbox size is unknown

SmallestMailboxPool defined in configuration:

```
akka.actor.deployment {
  /parent/router11 {
    router = smallest-mailbox-pool
    nr-of-instances = 5
  }
}
```

```
val router11: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router11")
```

SmallestMailboxPool defined in code:

```
val router12: ActorRef =
  context.actorOf(SmallestMailboxPool(5).props(Props[Worker]), "router12")
```

There is no Group variant of the SmallestMailboxPool because the size of the mailbox and the internal dispatching state of the actor is not practically available from the paths of the routees.

BroadcastPool and BroadcastGroup

A broadcast router forwards the message it receives to *all* its routees.

BroadcastPool defined in configuration:

```
akka.actor.deployment {
  /parent/router13 {
    router = broadcast-pool
    nr-of-instances = 5
  }
}
```

```
val router13: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router13")
```

BroadcastPool defined in code:

```
val router14: ActorRef =
  context.actorOf(BroadcastPool(5).props(Props[Worker]), "router14")
```

BroadcastGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router15 {
    router = broadcast-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router15: ActorRef =
  context.actorOf(FromConfig.props(), "router15")
```

BroadcastGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router16: ActorRef =
  context.actorOf(BroadcastGroup(paths).props(), "router16")
```

Note: Broadcast routers always broadcast *every* message to their routees. If you do not want to broadcast every message, then you can use a non-broadcasting router and use *Broadcast Messages* as needed.

ScatterGatherFirstCompletedPool and ScatterGatherFirstCompletedGroup

The ScatterGatherFirstCompletedRouter will send the message on to all its routees. It then waits for first reply it gets back. This result will be sent back to original sender. Other replies are discarded.

It is expecting at least one reply within a configured duration, otherwise it will reply with `akka.pattern.AskTimeoutException` in a `akka.actor.Status.Failure`.

ScatterGatherFirstCompletedPool defined in configuration:

```
akka.actor.deployment {
  /parent/router17 {
    router = scatter-gather-pool
    nr-of-instances = 5
    within = 10 seconds
  }
}
```

```
val router17: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router17")
```

ScatterGatherFirstCompletedPool defined in code:

```
val router18: ActorRef =
  context.actorOf(ScatterGatherFirstCompletedPool(5, within = 10.seconds).
    props(Props[Worker]), "router18")
```

ScatterGatherFirstCompletedGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router19 {
    router = scatter-gather-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    within = 10 seconds
  }
}
```

```
val router19: ActorRef =
  context.actorOf(FromConfig.props(), "router19")
```

ScatterGatherFirstCompletedGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router20: ActorRef =
  context.actorOf(ScatterGatherFirstCompletedGroup(
    paths,
    within = 10.seconds).props(), "router20")
```

TailChoppingPool and TailChoppingGroup

The TailChoppingRouter will first send the message to one, randomly picked, routee and then after a small delay to a second routee (picked randomly from the remaining routees) and so on. It waits for first reply it gets back and forwards it back to original sender. Other replies are discarded.

The goal of this router is to decrease latency by performing redundant queries to multiple routees, assuming that one of the other actors may still be faster to respond than the initial one.

This optimisation was described nicely in a blog post by Peter Bailis: Doing redundant work to speed up distributed queries.

TailChoppingPool defined in configuration:

```
akka.actor.deployment {
  /parent/router21 {
    router = tail-chopping-pool
    nr-of-instances = 5
    within = 10 seconds
    tail-chopping-router.interval = 20 milliseconds
  }
}
```

```
val router21: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router21")
```

TailChoppingPool defined in code:

```
val router22: ActorRef =
  context.actorOf(TailChoppingPool(5, within = 10.seconds, interval = 20.millis).
    props(Props[Worker]), "router22")
```

TailChoppingGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router23 {
    router = tail-chopping-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    within = 10 seconds
    tail-chopping-router.interval = 20 milliseconds
  }
}
```

```
val router23: ActorRef =
  context.actorOf(FromConfig.props(), "router23")
```

TailChoppingGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router24: ActorRef =
  context.actorOf(TailChoppingGroup(
    paths,
    within = 10.seconds, interval = 20.millis).props(), "router24")
```

ConsistentHashingPool and ConsistentHashingGroup

The ConsistentHashingPool uses consistent hashing to select a routee based on the sent message. This article gives good insight into how consistent hashing is implemented.

There is 3 ways to define what data to use for the consistent hash key.

- You can define `hashMapping` of the router to map incoming messages to their consistent hash key. This makes the decision transparent for the sender.
- The messages may implement `akka.routing.ConsistentHashingRouter.ConsistentHashable`. The key is part of the message and it's convenient to define it together with the message definition.
- The messages can be wrapped in a `akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelo` to define what data to use for the consistent hash key. The sender knows the key to use.

These ways to define the consistent hash key can be use together and at the same time for one router. The `hashMapping` is tried first.

Code example:

```
import akka.actor.Actor
import akka.routing.ConsistentHashingRouter.ConsistentHashable

class Cache extends Actor {
  var cache = Map.empty[String, String]

  def receive = {
    case Entry(key, value) => cache += (key -> value)
    case Get(key)          => sender() ! cache.get(key)
    case Evict(key)        => cache -= key
  }
}

final case class Evict(key: String)

final case class Get(key: String) extends ConsistentHashable {
  override def consistentHashKey: Any = key
}

final case class Entry(key: String, value: String)
```

```
import akka.actor.Props
import akka.routing.ConsistentHashingPool
import akka.routing.ConsistentHashingRouter.ConsistentHashMapping
import akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelope

def hashMapping: ConsistentHashMapping = {
  case Evict(key) => key
}

val cache: ActorRef =
  context.actorOf(ConsistentHashingPool(10, hashMapping = hashMapping).
    props(Props[Cache]), name = "cache")

cache ! ConsistentHashableEnvelope(
  message = Entry("hello", "HELLO"), hashKey = "hello")
cache ! ConsistentHashableEnvelope(
  message = Entry("hi", "HI"), hashKey = "hi")

cache ! Get("hello")
expectMsg(Some("HELLO"))

cache ! Get("hi")
expectMsg(Some("HI"))

cache ! Evict("hi")
cache ! Get("hi")
expectMsg(None)
```

In the above example you see that the `Get` message implements `ConsistentHashable` itself, while the `Entry` message is wrapped in a `ConsistentHashableEnvelope`. The `Evict` message is handled by the `hashMapping` partial function.

ConsistentHashingPool defined in configuration:

```
akka.actor.deployment {
  /parent/router25 {
    router = consistent-hashing-pool
```

```
    nr-of-instances = 5
    virtual-nodes-factor = 10
  }
}
```

```
val router25: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router25")
```

ConsistentHashingPool defined in code:

```
val router26: ActorRef =
  context.actorOf(
    ConsistentHashingPool(5).props(Props[Worker]),
    "router26")
```

ConsistentHashingGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router27 {
    router = consistent-hashing-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    virtual-nodes-factor = 10
  }
}
```

```
val router27: ActorRef =
  context.actorOf(FromConfig.props(), "router27")
```

ConsistentHashingGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router28: ActorRef =
  context.actorOf(ConsistentHashingGroup(paths).props(), "router28")
```

`virtual-nodes-factor` is the number of virtual nodes per routee that is used in the consistent hash node ring to make the distribution more uniform.

3.6.4 Specially Handled Messages

Most messages sent to router actors will be forwarded according to the routers' routing logic. However there are a few types of messages that have special behavior.

Note that these special messages, except for the `Broadcast` message, are only handled by self contained router actors and not by the `akka.routing.Router` component described in *A Simple Router*.

Broadcast Messages

A `Broadcast` message can be used to send a message to *all* of a router's routees. When a router receives a `Broadcast` message, it will broadcast that message's *payload* to all routees, no matter how that router would normally route its messages.

The example below shows how you would use a `Broadcast` message to send a very important message to every routee of a router.

```
import akka.routing.Broadcast
router ! Broadcast("Watch out for Davy Jones' locker")
```

In this example the router receives the `Broadcast` message, extracts its payload (`"Watch out for Davy Jones' locker"`), and then sends the payload on to all of the router's routees. It is up to each routee actor to handle the received payload message.

Note: Do not use *Broadcast Messages* when you use *BalancingPool* for routers. Routees on *BalancingPool* shares the same mailbox instance, thus some routees can possibly get the broadcast message multiple times, while other routees get no broadcast message.

PoisonPill Messages

A `PoisonPill` message has special handling for all actors, including for routers. When any actor receives a `PoisonPill` message, that actor will be stopped. See the *PoisonPill* documentation for details.

```
import akka.actor.PoisonPill
router ! PoisonPill
```

For a router, which normally passes on messages to routees, it is important to realise that `PoisonPill` messages are processed by the router only. `PoisonPill` messages sent to a router will *not* be sent on to routees.

However, a `PoisonPill` message sent to a router may still affect its routees, because it will stop the router and when the router stops it also stops its children. Stopping children is normal actor behavior. The router will stop routees that it has created as children. Each child will process its current message and then stop. This may lead to some messages being unprocessed. See the documentation on *Stopping actors* for more information.

If you wish to stop a router and its routees, but you would like the routees to first process all the messages currently in their mailboxes, then you should not send a `PoisonPill` message to the router. Instead you should wrap a `PoisonPill` message inside a `Broadcast` message so that each routee will receive the `PoisonPill` message. Note that this will stop all routees, even if the routees aren't children of the router, i.e. even routees programmatically provided to the router.

```
import akka.actor.PoisonPill
import akka.routing.Broadcast
router ! Broadcast(PoisonPill)
```

With the code shown above, each routee will receive a `PoisonPill` message. Each routee will continue to process its messages as normal, eventually processing the `PoisonPill`. This will cause the routee to stop. After all routees have stopped the router will itself be *stopped automatically* unless it is a dynamic router, e.g. using a resizer.

Note: Brendan W McAdams' excellent blog post Distributing Akka Workloads - And Shutting Down Afterwards discusses in more detail how `PoisonPill` messages can be used to shut down routers and routees.

Kill Messages

`Kill` messages are another type of message that has special handling. See *Killing an Actor* for general information about how actors handle `Kill` messages.

When a `Kill` message is sent to a router the router processes the message internally, and does *not* send it on to its routees. The router will throw an `ActorKilledException` and fail. It will then be either resumed, restarted or terminated, depending how it is supervised.

Routees that are children of the router will also be suspended, and will be affected by the supervision directive that is applied to the router. Routees that are not the routers children, i.e. those that were created externally to the router, will not be affected.

```
import akka.actor.Kill
router ! Kill
```

As with the `PoisonPill` message, there is a distinction between killing a router, which indirectly kills its children (who happen to be routees), and killing routees directly (some of whom may not be children.) To kill routees directly the router should be sent a `Kill` message wrapped in a `Broadcast` message.

```
import akka.actor.Kill
import akka.routing.Broadcast
router ! Broadcast(Kill)
```

Management Messages

- Sending `akka.routing.GetRoutees` to a router actor will make it send back its currently used routees in a `akka.routing.Routees` message.
- Sending `akka.routing.AddRoutee` to a router actor will add that routee to its collection of routees.
- Sending `akka.routing.RemoveRoutee` to a router actor will remove that routee to its collection of routees.
- Sending `akka.routing.AdjustPoolSize` to a pool router actor will add or remove that number of routees to its collection of routees.

These management messages may be handled after other messages, so if you send `AddRoutee` immediately followed by an ordinary message you are not guaranteed that the routees have been changed when the ordinary message is routed. If you need to know when the change has been applied you can send `AddRoutee` followed by `GetRoutees` and when you receive the `Routees` reply you know that the preceding change has been applied.

3.6.5 Dynamically Resizable Pool

Most pools can be used with a fixed number of routees or with a resize strategy to adjust the number of routees dynamically.

There are two types of resizers: the default `Resizer` and the `OptimalSizeExploringResizer`.

Default Resizer

The default resizer ramps up and down pool size based on pressure, measured by the percentage of busy routees in the pool. It ramps up pool size if the pressure is higher than a certain threshold and backs off if the pressure is lower than certain threshold. Both thresholds are configurable.

Pool with default resizer defined in configuration:

```
akka.actor.deployment {
  /parent/router29 {
    router = round-robin-pool
    resizer {
      lower-bound = 2
      upper-bound = 15
      messages-per-resize = 100
    }
  }
}
```

```
val router29: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router29")
```

Several more configuration options are available and described in `akka.actor.deployment.default.resizer` section of the reference *Configuration*.

Pool with resizer defined in code:

```
val resizer = DefaultResizer(lowerBound = 2, upperBound = 15)
val router30: ActorRef =
  context.actorOf(
```

```
    RoundRobinPool(5, Some(resizer)).props(Props[Worker]),
    "router30")
```

It is also worth pointing out that if you define the ‘‘router‘‘ in the configuration file then this value will be used instead of any programmatically sent parameters.

Optimal Size Exploring Resizer

The `OptimalSizeExploringResizer` resizes the pool to an optimal size that provides the most message throughput.

This resizer works best when you expect the pool size to performance function to be a convex function. For example, when you have a CPU bound tasks, the optimal size is bound to the number of CPU cores. When your task is IO bound, the optimal size is bound to optimal number of concurrent connections to that IO service - e.g. a 4 node elastic search cluster may handle 4-8 concurrent requests at optimal speed.

It achieves this by keeping track of message throughput at each pool size and performing the following three resizing operations (one at a time) periodically:

- Downsize if it hasn't seen all routees ever fully utilized for a period of time.
- Explore to a random nearby pool size to try and collect throughput metrics.
- Optimize to a nearby pool size with a better (than any other nearby sizes) throughput metrics.

When the pool is fully-utilized (i.e. all routees are busy), it randomly choose between exploring and optimizing. When the pool has not been fully-utilized for a period of time, it will downsize the pool to the last seen max utilization multiplied by a configurable ratio.

By constantly exploring and optimizing, the resizer will eventually walk to the optimal size and remain nearby. When the optimal size changes it will start walking towards the new one.

It keeps a performance log so it's stateful as well as having a larger memory footprint than the default `Resizer`. The memory usage is O(n) where n is the number of sizes you allow, i.e. upperBound - lowerBound.

Pool with `OptimalSizeExploringResizer` defined in configuration:

```
akka.actor.deployment {
  /parent/router31 {
    router = round-robin-pool
    optimal-size-exploring-resizer {
      enabled = on
      action-interval = 5s
      downsize-after-underutilized-for = 72h
    }
  }
}
```

```
val router31: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router31")
```

Several more configuration options are available and described in `akka.actor.deployment.default.optimal-size-ex` section of the reference *Configuration*.

Note: Resizing is triggered by sending messages to the actor pool, but it is not completed synchronously; instead a message is sent to the "head" `RouterActor` to perform the size change. Thus you cannot rely on resizing to instantaneously create new workers when all others are busy, because the message just sent will be queued to the mailbox of a busy actor. To remedy this, configure the pool to use a balancing dispatcher, see *Configuring Dispatchers* for more information.

3.6.6 How Routing is Designed within Akka

On the surface routers look like normal actors, but they are actually implemented differently. Routers are designed to be extremely efficient at receiving messages and passing them quickly on to routees.

A normal actor can be used for routing messages, but an actor's single-threaded processing can become a bottleneck. Routers can achieve much higher throughput with an optimization to the usual message-processing pipeline that allows concurrent routing. This is achieved by embedding routers' routing logic directly in their `ActorRef` rather than in the router actor. Messages sent to a router's `ActorRef` can be immediately routed to the routee, bypassing the single-threaded router actor entirely.

The cost to this is, of course, that the internals of routing code are more complicated than if routers were implemented with normal actors. Fortunately all of this complexity is invisible to consumers of the routing API. However, it is something to be aware of when implementing your own routers.

3.6.7 Custom Router

You can create your own router should you not find any of the ones provided by Akka sufficient for your needs. In order to roll your own router you have to fulfill certain criteria which are explained in this section.

Before creating your own router you should consider whether a normal actor with router-like behavior might do the job just as well as a full-blown router. As explained *above*, the primary benefit of routers over normal actors is their higher performance. But they are somewhat more complicated to write than normal actors. Therefore if lower maximum throughput is acceptable in your application you may wish to stick with traditional actors. This section, however, assumes that you wish to get maximum performance and so demonstrates how you can create your own router.

The router created in this example is replicating each message to a few destinations.

Start with the routing logic:

```
import scala.collection.immutable
import java.util.concurrent.ThreadLocalRandom
import akka.routing.RoundRobinRoutingLogic
import akka.routing.RoutingLogic
import akka.routing.Routee
import akka.routing.SeveralRoutees

class RedundancyRoutingLogic(nbrCopies: Int) extends RoutingLogic {
  val roundRobin = RoundRobinRoutingLogic()
  def select(message: Any, routees: immutable.IndexedSeq[Routee]): Routee = {
    val targets = (1 to nbrCopies).map(_ => roundRobin.select(message, routees))
    SeveralRoutees(targets)
  }
}
```

`select` will be called for each message and in this example pick a few destinations by round-robin, by reusing the existing `RoundRobinRoutingLogic` and wrap the result in a `SeveralRoutees` instance. `SeveralRoutees` will send the message to all of the supplied routes.

The implementation of the routing logic must be thread safe, since it might be used outside of actors.

A unit test of the routing logic:

```
final case class TestRoutee(n: Int) extends Routee {
  override def send(message: Any, sender: ActorRef): Unit = ()
}

  val logic = new RedundancyRoutingLogic(nbrCopies = 3)

  val routees = for (n <- 1 to 7) yield TestRoutee(n)

  val r1 = logic.select("msg", routees)
```

```
  r1.asInstanceOf[SeveralRoutees].routees should be(
    Vector(TestRoutee(1), TestRoutee(2), TestRoutee(3)))

  val r2 = logic.select("msg", routees)
  r2.asInstanceOf[SeveralRoutees].routees should be(
    Vector(TestRoutee(4), TestRoutee(5), TestRoutee(6)))

  val r3 = logic.select("msg", routees)
  r3.asInstanceOf[SeveralRoutees].routees should be(
    Vector(TestRoutee(7), TestRoutee(1), TestRoutee(2)))
```

You could stop here and use the `RedundancyRoutingLogic` with a `akka.routing.Router` as described in *A Simple Router*.

Let us continue and make this into a self contained, configurable, router actor.

Create a class that extends `Pool`, `Group` or `CustomRouterConfig`. That class is a factory for the routing logic and holds the configuration for the router. Here we make it a `Group`.

```
import akka.dispatch.Dispatchers
import akka.routing.Group
import akka.routing.Router
import akka.japi.Util.immutableSeq
import com.typesafe.config.Config

final case class RedundancyGroup(routeePaths: immutable.Iterable[String],␣
↪nbrCopies: Int) extends Group {

  def this(config: Config) = this(
    routeePaths = immutableSeq(config.getStringList("routees.paths")),
    nbrCopies = config.getInt("nbr-copies"))

  override def paths(system: ActorSystem): immutable.Iterable[String] = routeePaths

  override def createRouter(system: ActorSystem): Router =
    new Router(new RedundancyRoutingLogic(nbrCopies))

  override val routerDispatcher: String = Dispatchers.DefaultDispatcherId
}
```

This can be used exactly as the router actors provided by Akka.

```
for (n <- 1 to 10) system.actorOf(Props[Storage], "s" + n)

val paths = for (n <- 1 to 10) yield ("/user/s" + n)
val redundancy1: ActorRef =
  system.actorOf(
    RedundancyGroup(paths, nbrCopies = 3).props(),
    name = "redundancy1")
redundancy1 ! "important"
```

Note that we added a constructor in `RedundancyGroup` that takes a `Config` parameter. That makes it possible to define it in configuration.

```
akka.actor.deployment {
  /redundancy2 {
    router = "docs.routing.RedundancyGroup"
    routees.paths = ["/user/s1", "/user/s2", "/user/s3"]
    nbr-copies = 5
  }
}
```

Note the fully qualified class name in the `router` property. The router class must extend

`akka.routing.RouterConfig` (`Pool`, `Group` or `CustomRouterConfig`) and have constructor with one `com.typesafe.config.Config` parameter. The deployment section of the configuration is passed to the constructor.

```
val redundancy2: ActorRef = system.actorOf(
  FromConfig.props(),
  name = "redundancy2")
redundancy2 ! "very important"
```

3.6.8 Configuring Dispatchers

The dispatcher for created children of the pool will be taken from `Props` as described in *Dispatchers*.

To make it easy to define the dispatcher of the routees of the pool you can define the dispatcher inline in the deployment section of the config.

```
akka.actor.deployment {
  /poolWithDispatcher {
    router = random-pool
    nr-of-instances = 5
    pool-dispatcher {
      fork-join-executor.parallelism-min = 5
      fork-join-executor.parallelism-max = 5
    }
  }
}
```

That is the only thing you need to do enable a dedicated dispatcher for a pool.

Note: If you use a group of actors and route to their paths, then they will still use the same dispatcher that was configured for them in their `Props`, it is not possible to change an actors dispatcher after it has been created.

The "head" router cannot always run on the same dispatcher, because it does not process the same type of messages, hence this special actor does not use the dispatcher configured in `Props`, but takes the `routerDispatcher` from the `RouterConfig` instead, which defaults to the actor system's default dispatcher. All standard routers allow setting this property in their constructor or factory method, custom routers have to implement the method in a suitable way.

```
val router: ActorRef = system.actorOf(
  // "head" router actor will run on "router-dispatcher" dispatcher
  // Worker routees will run on "pool-dispatcher" dispatcher
  RandomPool(5, routerDispatcher = "router-dispatcher").props(Props[Worker]),
  name = "poolWithDispatcher")
```

Note: It is not allowed to configure the `routerDispatcher` to be a `akka.dispatch.BalancingDispatcherConfigurator` since the messages meant for the special router actor cannot be processed by any other actor.

3.7 FSM

3.7.1 Overview

The FSM (Finite State Machine) is available as a mixin for the Akka Actor and is best described in the Erlang design principles

A FSM can be described as a set of relations of the form:

> **State(S) x Event(E) -> Actions (A), State(S')**

These relations are interpreted as meaning:

> *If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.*

3.7.2 A Simple Example

To demonstrate most of the features of the `FSM` trait, consider an actor which shall receive and queue messages while they arrive in a burst and send them on after the burst ended or a flush request is received.

First, consider all of the below to use these import statements:

```
import akka.actor.{ ActorRef, FSM }
import scala.concurrent.duration._
```

The contract of our "Buncher" actor is that it accepts or produces the following messages:

```
// received events
final case class SetTarget(ref: ActorRef)
final case class Queue(obj: Any)
case object Flush

// sent events
final case class Batch(obj: immutable.Seq[Any])
```

`SetTarget` is needed for starting it up, setting the destination for the `Batches` to be passed on; `Queue` will add to the internal queue while `Flush` will mark the end of a burst.

```
// states
sealed trait State
case object Idle extends State
case object Active extends State

sealed trait Data
case object Uninitialized extends Data
final case class Todo(target: ActorRef, queue: immutable.Seq[Any]) extends Data
```

The actor can be in two states: no message queued (aka `Idle`) or some message queued (aka `Active`). It will stay in the active state as long as messages keep arriving and no flush is requested. The internal state data of the actor is made up of the target actor reference to send the batches to and the actual queue of messages.

Now let's take a look at the skeleton for our FSM actor:

```
class Buncher extends FSM[State, Data] {

  startWith(Idle, Uninitialized)

  when(Idle) {
    case Event(SetTarget(ref), Uninitialized) =>
      stay using Todo(ref, Vector.empty)
  }

  // transition elided ...

  when(Active, stateTimeout = 1 second) {
    case Event(Flush | StateTimeout, t: Todo) =>
      goto(Idle) using t.copy(queue = Vector.empty)
  }
```

```
  // unhandled elided ...

  initialize()
}
```

The basic strategy is to declare the actor, mixing in the `FSM` trait and specifying the possible states and data values as type parameters. Within the body of the actor a DSL is used for declaring the state machine:

- `startWith` defines the initial state and initial data
- then there is one `when(<state>) { ... }` declaration per state to be handled (could potentially be multiple ones, the passed `PartialFunction` will be concatenated using `orElse`)
- finally starting it up using `initialize`, which performs the transition into the initial state and sets up timers (if required).

In this case, we start out in the `Idle` and `Uninitialized` state, where only the `SetTarget()` message is handled; `stay` prepares to end this event's processing for not leaving the current state, while the `using` modifier makes the FSM replace the internal state (which is `Uninitialized` at this point) with a fresh `Todo()` object containing the target actor reference. The `Active` state has a state timeout declared, which means that if no message is received for 1 second, a `FSM.StateTimeout` message will be generated. This has the same effect as receiving the `Flush` command in this case, namely to transition back into the `Idle` state and resetting the internal queue to the empty vector. But how do messages get queued? Since this shall work identically in both states, we make use of the fact that any event which is not handled by the `when()` block is passed to the `whenUnhandled()` block:

```
whenUnhandled {
  // common code for both states
  case Event(Queue(obj), t @ Todo(_, v)) =>
    goto(Active) using t.copy(queue = v :+ obj)

  case Event(e, s) =>
    log.warning("received unhandled request {} in state {}/{}", e, stateName, s)
    stay
}
```

The first case handled here is adding `Queue()` requests to the internal queue and going to the `Active` state (this does the obvious thing of staying in the `Active` state if already there), but only if the FSM data are not `Uninitialized` when the `Queue()` event is received. Otherwise—and in all other non-handled cases—the second case just logs a warning and does not change the internal state.

The only missing piece is where the `Batches` are actually sent to the target, for which we use the `onTransition` mechanism: you can declare multiple such blocks and all of them will be tried for matching behavior in case a state transition occurs (i.e. only when the state actually changes).

```
onTransition {
  case Active -> Idle =>
    stateData match {
      case Todo(ref, queue) => ref ! Batch(queue)
      case _                => // nothing to do
    }
}
```

The transition callback is a partial function which takes as input a pair of states—the current and the next state. The FSM trait includes a convenience extractor for these in form of an arrow operator, which conveniently reminds you of the direction of the state change which is being matched. During the state change, the old state data is available via `stateData` as shown, and the new state data would be available as `nextStateData`.

Note: Same-state transitions can be implemented (when currently in state `S`) using `goto(S)` or `stay()`. The difference between those being that `goto(S)` will emit an event `S->S` event that can be handled by `onTransition`, whereas `stay()` will *not*.

To verify that this buncher actually works, it is quite easy to write a test using the *Testing Actor Systems*, which is conveniently bundled with ScalaTest traits into `AkkaSpec`:

```
import akka.actor.Props
import scala.collection.immutable

object FSMDocSpec {
  // messages and data types
}

class FSMDocSpec extends MyFavoriteTestFrameWorkPlusAkkaTestKit {
  import FSMDocSpec._

  // fsm code elided ...

  "simple finite state machine" must {

    "demonstrate NullFunction" in {
      class A extends FSM[Int, Null] {
        val SomeState = 0
        when(SomeState)(FSM.NullFunction)
      }
    }

    "batch correctly" in {
      val buncher = system.actorOf(Props(classOf[Buncher], this))
      buncher ! SetTarget(testActor)
      buncher ! Queue(42)
      buncher ! Queue(43)
      expectMsg(Batch(immutable.Seq(42, 43)))
      buncher ! Queue(44)
      buncher ! Flush
      buncher ! Queue(45)
      expectMsg(Batch(immutable.Seq(44)))
      expectMsg(Batch(immutable.Seq(45)))
    }

    "not batch if uninitialized" in {
      val buncher = system.actorOf(Props(classOf[Buncher], this))
      buncher ! Queue(42)
      expectNoMsg
    }
  }
}
```

3.7.3 Reference

The FSM Trait and Object

The `FSM` trait inherits directly from `Actor`, when you extend `FSM` you must be aware that an actor is actually created:

```
class Buncher extends FSM[State, Data] {

  // fsm body ...

  initialize()
}
```

Note: The FSM trait defines a `receive` method which handles internal messages and passes everything else

through to the FSM logic (according to the current state). When overriding the `receive` method, keep in mind that e.g. state timeout handling depends on actually passing the messages through the FSM logic.

The `FSM` trait takes two type parameters:

1. the supertype of all state names, usually a sealed trait with case objects extending it,
2. the type of the state data which are tracked by the `FSM` module itself.

Note: The state data together with the state name describe the internal state of the state machine; if you stick to this scheme and do not add mutable fields to the FSM class you have the advantage of making all changes of the internal state explicit in a few well-known places.

Defining States

A state is defined by one or more invocations of the method

```
when(<name>[, stateTimeout = <timeout>])(stateFunction).
```

The given name must be an object which is type-compatible with the first type parameter given to the `FSM` trait. This object is used as a hash key, so you must ensure that it properly implements `equals` and `hashCode`; in particular it must not be mutable. The easiest fit for these requirements are case objects.

If the `stateTimeout` parameter is given, then all transitions into this state, including staying, receive this timeout by default. Initiating the transition with an explicit timeout may be used to override this default, see *Initiating Transitions* for more information. The state timeout of any state may be changed during action processing with `setStateTimeout(state, duration)`. This enables runtime configuration e.g. via external message.

The `stateFunction` argument is a `PartialFunction[Event, State]`, which is conveniently given using the partial function literal syntax as demonstrated below:

```
when(Idle) {
  case Event(SetTarget(ref), Uninitialized) =>
    stay using Todo(ref, Vector.empty)
}

when(Active, stateTimeout = 1 second) {
  case Event(Flush | StateTimeout, t: Todo) =>
    goto(Idle) using t.copy(queue = Vector.empty)
}
```

The `Event(msg: Any, data: D)` case class is parameterized with the data type held by the FSM for convenient pattern matching.

Warning: It is required that you define handlers for each of the possible FSM states, otherwise there will be failures when trying to switch to undeclared states.

It is recommended practice to declare the states as objects extending a sealed trait and then verify that there is a `when` clause for each of the states. If you want to leave the handling of a state "unhandled" (more below), it still needs to be declared like this:

```
when(SomeState)(FSM.NullFunction)
```

Defining the Initial State

Each FSM needs a starting point, which is declared using

```
startWith(state, data[, timeout])
```

The optionally given timeout argument overrides any specification given for the desired initial state. If you want to cancel a default timeout, use `None`.

Unhandled Events

If a state doesn't handle a received event a warning is logged. If you want to do something else in this case you can specify that with `whenUnhandled(stateFunction)`:

```
whenUnhandled {
  case Event(x: X, data) =>
    log.info("Received unhandled event: " + x)
    stay
  case Event(msg, _) =>
    log.warning("Received unknown event: " + msg)
    goto(Error)
}
```

Within this handler the state of the FSM may be queried using the `stateName` method.

IMPORTANT: This handler is not stacked, meaning that each invocation of `whenUnhandled` replaces the previously installed handler.

Initiating Transitions

The result of any `stateFunction` must be a definition of the next state unless terminating the FSM, which is described in *Termination from Inside*. The state definition can either be the current state, as described by the `stay` directive, or it is a different state as given by `goto(state)`. The resulting object allows further qualification by way of the modifiers described in the following:

- `forMax(duration)`

  This modifier sets a state timeout on the next state. This means that a timer is started which upon expiry sends a `StateTimeout` message to the FSM. This timer is canceled upon reception of any other message in the meantime; you can rely on the fact that the `StateTimeout` message will not be processed after an intervening message.

  This modifier can also be used to override any default timeout which is specified for the target state. If you want to cancel the default timeout, use `Duration.Inf`.

- `using(data)`

  This modifier replaces the old state data with the new data given. If you follow the advice *above*, this is the only place where internal state data are ever modified.

- `replying(msg)`

  This modifier sends a reply to the currently processed message and otherwise does not modify the state transition.

All modifiers can be chained to achieve a nice and concise description:

```
when(SomeState) {
  case Event(msg, _) =>
    goto(Processing) using (newData) forMax (5 seconds) replying (WillDo)
}
```

The parentheses are not actually needed in all cases, but they visually distinguish between modifiers and their arguments and therefore make the code even more pleasant to read for foreigners.

Note: Please note that the `return` statement may not be used in `when` blocks or similar; this is a Scala restriction. Either refactor your code using `if () ... else ...` or move it into a method definition.

Monitoring Transitions

Transitions occur "between states" conceptually, which means after any actions you have put into the event handling block; this is obvious since the next state is only defined by the value returned by the event handling logic. You do not need to worry about the exact order with respect to setting the internal state variable, as everything within the FSM actor is running single-threaded anyway.

Internal Monitoring

Up to this point, the FSM DSL has been centered on states and events. The dual view is to describe it as a series of transitions. This is enabled by the method

```
onTransition(handler)
```

which associates actions with a transition instead of with a state and event. The handler is a partial function which takes a pair of states as input; no resulting state is needed as it is not possible to modify the transition in progress.

```
onTransition {
  case Idle -> Active => setTimer("timeout", Tick, 1 second, repeat = true)
  case Active -> _    => cancelTimer("timeout")
  case x -> Idle      => log.info("entering Idle from " + x)
}
```

The convenience extractor `->` enables decomposition of the pair of states with a clear visual reminder of the transition's direction. As usual in pattern matches, an underscore may be used for irrelevant parts; alternatively you could bind the unconstrained state to a variable, e.g. for logging as shown in the last case.

It is also possible to pass a function object accepting two states to `onTransition`, in case your transition handling logic is implemented as a method:

```
onTransition(handler _)

def handler(from: StateType, to: StateType) {
  // handle it here ...
}
```

The handlers registered with this method are stacked, so you can intersperse `onTransition` blocks with `when` blocks as suits your design. It should be noted, however, that *all handlers will be invoked for each transition*, not only the first matching one. This is designed specifically so you can put all transition handling for a certain aspect into one place without having to worry about earlier declarations shadowing later ones; the actions are still executed in declaration order, though.

Note: This kind of internal monitoring may be used to structure your FSM according to transitions, so that for example the cancellation of a timer upon leaving a certain state cannot be forgot when adding new target states.

External Monitoring

External actors may be registered to be notified of state transitions by sending a message `SubscribeTransitionCallBack(actorRef)`. The named actor will be sent a `CurrentState(self, stateName)` message immediately and will receive `Transition(actorRef, oldState, newState)` messages whenever a state change is triggered.

Please note that a state change includes the action of performing an `goto(S)`, while already being state `S`. In that case the monitoring actor will be notified with an `Transition(ref,S,S)` message. This may be useful if your `FSM` should react on all (also same-state) transitions. In case you'd rather not emit events for same-state transitions use `stay()` instead of `goto(S)`.

External monitors may be unregistered by sending `UnsubscribeTransitionCallBack(actorRef)` to the `FSM` actor.

Stopping a listener without unregistering will not remove the listener from the subscription list; use `UnsubscribeTransitionCallback` before stopping the listener.

Transforming State

The partial functions supplied as argument to the `when()` blocks can be transformed using Scala's full supplement of functional programming tools. In order to retain type inference, there is a helper function which may be used in case some common handling logic shall be applied to different clauses:

```
when(SomeState)(transform {
  case Event(bytes: ByteString, read) => stay using (read + bytes.length)
} using {
  case s @ FSM.State(state, read, timeout, stopReason, replies) if read > 1000 =>
    goto(Processing)
})
```

It goes without saying that the arguments to this method may also be stored, to be used several times, e.g. when applying the same transformation to several `when()` blocks:

```
val processingTrigger: PartialFunction[State, State] = {
  case s @ FSM.State(state, read, timeout, stopReason, replies) if read > 1000 =>
    goto(Processing)
}

when(SomeState)(transform {
  case Event(bytes: ByteString, read) => stay using (read + bytes.length)
} using processingTrigger)
```

Timers

Besides state timeouts, FSM manages timers identified by `String` names. You may set a timer using

```
setTimer(name, msg, interval, repeat)
```

where `msg` is the message object which will be sent after the duration `interval` has elapsed. If `repeat` is `true`, then the timer is scheduled at fixed rate given by the `interval` parameter. Any existing timer with the same name will automatically be canceled before adding the new timer.

Timers may be canceled using

```
cancelTimer(name)
```

which is guaranteed to work immediately, meaning that the scheduled message will not be processed after this call even if the timer already fired and queued it. The status of any timer may be inquired with

```
isTimerActive(name)
```

These named timers complement state timeouts because they are not affected by intervening reception of other messages.

Termination from Inside

The FSM is stopped by specifying the result state as

```
stop([reason[, data]])
```

The reason must be one of `Normal` (which is the default), `Shutdown` or `Failure(reason)`, and the second argument may be given to change the state data which is available during termination handling.

Note: It should be noted that `stop` does not abort the actions and stop the FSM immediately. The stop action must be returned from the event handler in the same way as a state transition (but note that the `return` statement may not be used within a `when` block).

```
when(Error) {
  case Event("stop", _) =>
    // do cleanup ...
    stop()
}
```

You can use `onTermination(handler)` to specify custom code that is executed when the FSM is stopped. The handler is a partial function which takes a `StopEvent(reason, stateName, stateData)` as argument:

```
onTermination {
  case StopEvent(FSM.Normal, state, data)         => // ...
  case StopEvent(FSM.Shutdown, state, data)       => // ...
  case StopEvent(FSM.Failure(cause), state, data) => // ...
}
```

As for the `whenUnhandled` case, this handler is not stacked, so each invocation of `onTermination` replaces the previously installed handler.

Termination from Outside

When an `ActorRef` associated to a FSM is stopped using the `stop` method, its `postStop` hook will be executed. The default implementation by the `FSM` trait is to execute the `onTermination` handler if that is prepared to handle a `StopEvent(Shutdown, ...)`.

Warning: In case you override `postStop` and want to have your `onTermination` handler called, do not forget to call `super.postStop`.

3.7.4 Testing and Debugging Finite State Machines

During development and for trouble shooting FSMs need care just as any other actor. There are specialized tools available as described in *Testing Finite State Machines* and in the following.

Event Tracing

The setting `akka.actor.debug.fsm` in *Configuration* enables logging of an event trace by `LoggingFSM` instances:

```
import akka.actor.LoggingFSM
class MyFSM extends LoggingFSM[StateType, Data] {
  // body elided ...
}
```

This FSM will log at DEBUG level:

- all processed events, including `StateTimeout` and scheduled timer messages
- every setting and cancellation of named timers
- all state transitions

Life cycle changes and special messages can be logged as described for *Actors*.

Rolling Event Log

The `LoggingFSM` trait adds one more feature to the FSM: a rolling event log which may be used during debugging (for tracing how the FSM entered a certain failure state) or for other creative uses:

```
import akka.actor.LoggingFSM
class MyFSM extends LoggingFSM[StateType, Data] {
  override def logDepth = 12
  onTermination {
    case StopEvent(FSM.Failure(_), state, data) =>
      val lastEvents = getLog.mkString("\n\t")
      log.warning("Failure in state " + state + " with data " + data + "\n" +
        "Events leading up to this point:\n\t" + lastEvents)
  }
  // ...
}
```

The `logDepth` defaults to zero, which turns off the event log.

> **Warning:** The log buffer is allocated during actor creation, which is why the configuration is done using a virtual method call. If you want to override with a `val`, make sure that its initialization happens before the initializer of `LoggingFSM` runs, and do not change the value returned by `logDepth` after the buffer has been allocated.

The contents of the event log are available using method `getLog`, which returns an `IndexedSeq[LogEntry]` where the oldest entry is at index zero.

3.7.5 Examples

A bigger FSM example contrasted with Actor's `become`/`unbecome` can be found in the Lightbend Activator template named Akka FSM in Scala

3.8 Persistence

Akka persistence enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). These changes are only ever appended to storage, nothing is ever mutated, which allows for very high transaction rates and efficient replication. Stateful actors are recovered by replaying stored changes to these actors from which they can rebuild internal state. This can be either the full history of changes or starting from a snapshot which can dramatically reduce recovery times. Akka persistence also provides point-to-point communication with at-least-once message delivery semantics.

Akka persistence is inspired by and the official replacement of the eventsourced library. It follows the same concepts and architecture of eventsourced but significantly differs on API and implementation level. See also *Migration Guide Eventsourced to Akka Persistence 2.3.x*

3.8.1 Dependencies

Akka persistence is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-persistence" % "2.4.14"
```

The Akka persistence extension comes with few built-in persistence plugins, including in-memory heap based journal, local file-system based snapshot-store and LevelDB based journal.

LevelDB based plugins will require the following additional dependency declaration:

```
"org.iq80.leveldb"            % "leveldb"          % "0.7"
"org.fusesource.leveldbjni"   % "leveldbjni-all"   % "1.8"
```

3.8.2 Architecture

- *PersistentActor*: Is a persistent, stateful actor. It is able to persist events to a journal and can react to them in a thread-safe manner. It can be used to implement both *command* as well as *event sourced* actors. When a persistent actor is started or restarted, journaled messages are replayed to that actor so that it can recover internal state from these messages.
- *PersistentView*: A view is a persistent, stateful actor that receives journaled messages that have been written by another persistent actor. A view itself does not journal new messages, instead, it updates internal state only from a persistent actor's replicated message stream.
- *AtLeastOnceDelivery*: To send messages with at-least-once delivery semantics to destinations, also in case of sender and receiver JVM crashes.
- *AsyncWriteJournal*: A journal stores the sequence of messages sent to a persistent actor. An application can control which messages are journaled and which are received by the persistent actor without being journaled. Journal maintains *highestSequenceNr* that is increased on each message. The storage backend of a journal is pluggable. The persistence extension comes with a "leveldb" journal plugin, which writes to the local filesystem. Replicated journals are available as Community plugins.
- *Snapshot store*: A snapshot store persists snapshots of a persistent actor's or a view's internal state. Snapshots are used for optimizing recovery times. The storage backend of a snapshot store is pluggable. The persistence extension comes with a "local" snapshot storage plugin, which writes to the local filesystem. Replicated snapshot stores are available as Community plugins.

3.8.3 Event sourcing

The basic idea behind Event Sourcing is quite simple. A persistent actor receives a (non-persistent) command which is first validated if it can be applied to the current state. Here validation can mean anything, from simple inspection of a command message's fields up to a conversation with several external services, for example. If validation succeeds, events are generated from the command, representing the effect of the command. These events are then persisted and, after successful persistence, used to change the actor's state. When the persistent actor needs to be recovered, only the persisted events are replayed of which we know that they can be successfully applied. In other words, events cannot fail when being replayed to a persistent actor, in contrast to commands. Event sourced actors may of course also process commands that do not change application state such as query commands for example.

Akka persistence supports event sourcing with the `PersistentActor` trait. An actor that extends this trait uses the `persist` method to persist and handle events. The behavior of a `PersistentActor` is defined by implementing `receiveRecover` and `receiveCommand`. This is demonstrated in the following example.

```
import akka.actor._
import akka.persistence._

case class Cmd(data: String)
case class Evt(data: String)

case class ExampleState(events: List[String] = Nil) {
  def updated(evt: Evt): ExampleState = copy(evt.data :: events)
  def size: Int = events.length
  override def toString: String = events.reverse.toString
}

class ExamplePersistentActor extends PersistentActor {
  override def persistenceId = "sample-id-1"
```

```
  var state = ExampleState()

  def updateState(event: Evt): Unit =
    state = state.updated(event)

  def numEvents =
    state.size

  val receiveRecover: Receive = {
    case evt: Evt                                 => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  val receiveCommand: Receive = {
    case Cmd(data) =>
      persist(Evt(s"${data}-${numEvents}"))(updateState)
      persist(Evt(s"${data}-${numEvents + 1}")) { event =>
        updateState(event)
        context.system.eventStream.publish(event)
      }
    case "snap"  => saveSnapshot(state)
    case "print" => println(state)
  }

}
```

The example defines two data types, `Cmd` and `Evt` to represent commands and events, respectively. The `state` of the `ExamplePersistentActor` is a list of persisted event data contained in `ExampleState`.

The persistent actor's `receiveRecover` method defines how `state` is updated during recovery by handling `Evt` and `SnapshotOffer` messages. The persistent actor's `receiveCommand` method is a command handler. In this example, a command is handled by generating two events which are then persisted and handled. Events are persisted by calling `persist` with an event (or a sequence of events) as first argument and an event handler as second argument.

The `persist` method persists events asynchronously and the event handler is executed for successfully persisted events. Successfully persisted events are internally sent back to the persistent actor as individual messages that trigger event handler executions. An event handler may close over persistent actor state and mutate it. The sender of a persisted event is the sender of the corresponding command. This allows event handlers to reply to the sender of a command (not shown).

The main responsibility of an event handler is changing persistent actor state using event data and notifying others about successful state changes by publishing events.

When persisting events with `persist` it is guaranteed that the persistent actor will not receive further commands between the `persist` call and the execution(s) of the associated event handler. This also holds for multiple `persist` calls in context of a single command. Incoming messages are *stashed* until the `persist` is completed.

If persistence of an event fails, `onPersistFailure` will be invoked (logging the error by default), and the actor will unconditionally be stopped. If persistence of an event is rejected before it is stored, e.g. due to serialization error, `onPersistRejected` will be invoked (logging a warning by default) and the actor continues with the next message.

The easiest way to run this example yourself is to download Lightbend Activator and open the tutorial named Akka Persistence Samples with Scala. It contains instructions on how to run the `PersistentActorExample`.

Note: It's also possible to switch between different command handlers during normal processing and recovery with `context.become()` and `context.unbecome()`. To get the actor into the same state after recovery you need to take special care to perform the same state transitions with `become` and `unbecome` in the `receiveRecover` method as you would have done in the command handler. Note that when using `become` from `receiveRecover` it will still only use the `receiveRecover` behavior when replaying the events. When

replay is completed it will use the new behavior.

Identifiers

A persistent actor must have an identifier that doesn't change across different actor incarnations. The identifier must be defined with the `persistenceId` method.

```
override def persistenceId = "my-stable-persistence-id"
```

Note: `persistenceId` must be unique to a given entity in the journal (database table/keyspace). When replaying messages persisted to the journal, you query messages with a `persistenceId`. So, if two different entities share the same `persistenceId`, message-replaying behavior is corrupted.

Recovery

By default, a persistent actor is automatically recovered on start and on restart by replaying journaled messages. New messages sent to a persistent actor during recovery do not interfere with replayed messages. They are cached and received by a persistent actor after recovery phase completes.

Note: Accessing the `sender()` for replayed messages will always result in a `deadLetters` reference, as the original sender is presumed to be long gone. If you indeed have to notify an actor during recovery in the future, store its `ActorPath` explicitly in your persisted events.

Recovery customization

Applications may also customise how recovery is performed by returning a customised `Recovery` object in the `recovery` method of a `PersistentActor`,

To skip loading snapshots and replay all events you can use `SnapshotSelectionCriteria.None`. This can be useful if snapshot serialization format has changed in an incompatible way. It should typically not be used when events have been deleted.

```
override def recovery =
  Recovery(fromSnapshot = SnapshotSelectionCriteria.None)
```

Another example, which can be fun for experiments but probably not in a real application, is setting an upper bound to the replay which allows the actor to be replayed to a certain point "in the past" instead to its most up to date state. Note that after that it is a bad idea to persist new events because a later recovery will probably be confused by the new events that follow the events that were previously skipped.

```
override def recovery = Recovery(toSequenceNr = 457L)
```

Recovery can be disabled by returning `Recovery.none()` in the `recovery` method of a `PersistentActor`:

```
override def recovery = Recovery.none
```

Recovery status

A persistent actor can query its own recovery status via the methods

```
def recoveryRunning: Boolean
def recoveryFinished: Boolean
```

Sometimes there is a need for performing additional initialization when the recovery has completed before processing any other message sent to the persistent actor. The persistent actor will receive a special `RecoveryCompleted` message right after recovery and before any other received messages.

```
override def receiveRecover: Receive = {
  case RecoveryCompleted =>
  // perform init after recovery, before any other messages
  //...
  case evt               => //...
}

override def receiveCommand: Receive = {
  case msg => //...
}
```

The actor will always receive a `RecoveryCompleted` message, even if there are no events in the journal and the snapshot store is empty, or if it's a new persistent actor with a previously unused `persistenceId`.

If there is a problem with recovering the state of the actor from the journal, `onRecoveryFailure` is called (logging the error by default) and the actor will be stopped.

Internal stash

The persistent actor has a private *stash* for internally caching incoming messages during *recovery* or the `persist\persistAll` method persisting events. You can still use/inherit from the `Stash` interface. The internal stash cooperates with the normal stash by hooking into `unstashAll` method and making sure messages are unstashed properly to the internal stash to maintain ordering guarantees.

You should be careful to not send more messages to a persistent actor than it can keep up with, otherwise the number of stashed messages will grow without bounds. It can be wise to protect against `OutOfMemoryError` by defining a maximum stash capacity in the mailbox configuration:

```
akka.actor.default-mailbox.stash-capacity=10000
```

Note that the stash capacity is per actor. If you have many persistent actors, e.g. when using cluster sharding, you may need to define a small stash capacity to ensure that the total number of stashed messages in the system don't consume too much memory. Additionally, The persistent actor defines three strategies to handle failure when the internal stash capacity is exceeded. The default overflow strategy is the `ThrowOverflowExceptionStrategy`, which discards the current received message and throws a `StashOverflowException`, causing actor restart if default supervision strategy is used. you can override the `internalStashOverflowStrategy` method to return `DiscardToDeadLetterStrategy` or `ReplyToStrategy` for any "individual" persistent actor, or define the "default" for all persistent actors by providing FQCN, which must be a subclass of `StashOverflowStrategyConfigurator`, in the persistence configuration:

```
akka.persistence.internal-stash-overflow-strategy=
  "akka.persistence.ThrowExceptionConfigurator"
```

The `DiscardToDeadLetterStrategy` strategy also has a pre-packaged companion configurator `akka.persistence.DiscardConfigurator`.

You can also query default strategy via the Akka persistence extension singleton:

```
Persistence(context.system).defaultInternalStashOverflowStrategy
```

Note: The bounded mailbox should be avoided in the persistent actor, by which the messages come from storage backends may be discarded. You can use bounded stash instead of it.

Relaxed local consistency requirements and high throughput use-cases

If faced with relaxed local consistency requirements and high throughput demands sometimes `PersistentActor` and its `persist` may not be enough in terms of consuming incoming Commands at a high rate, because it has to wait until all Events related to a given Command are processed in order to start processing the next Command. While this abstraction is very useful for most cases, sometimes you may be faced with relaxed requirements about consistency – for example you may want to process commands as fast as you can, assuming that the Event will eventually be persisted and handled properly in the background, retroactively reacting to persistence failures if needed.

The `persistAsync` method provides a tool for implementing high-throughput persistent actors. It will *not* stash incoming Commands while the Journal is still working on persisting and/or user code is executing event callbacks.

In the below example, the event callbacks may be called "at any time", even after the next Command has been processed. The ordering between events is still guaranteed ("evt-b-1" will be sent after "evt-a-2", which will be sent after "evt-a-1" etc.).

```
class MyPersistentActor extends PersistentActor {

  override def persistenceId = "my-stable-persistence-id"

  override def receiveRecover: Receive = {
    case _ => // handle recovery here
  }

  override def receiveCommand: Receive = {
    case c: String => {
      sender() ! c
      persistAsync(s"evt-$c-1") { e => sender() ! e }
      persistAsync(s"evt-$c-2") { e => sender() ! e }
    }
  }
}

// usage
persistentActor ! "a"
persistentActor ! "b"

// possible order of received messages:
// a
// b
// evt-a-1
// evt-a-2
// evt-b-1
// evt-b-2
```

Note: In order to implement the pattern known as "*command sourcing*" simply call `persistAsync(cmd)(...)` right away on all incoming messages and handle them in the callback.

Warning: The callback will not be invoked if the actor is restarted (or stopped) in between the call to `persistAsync` and the journal has confirmed the write.

Deferring actions until preceding persist handlers have executed

Sometimes when working with `persistAsync` you may find that it would be nice to define some actions in terms of ''happens-after the previous `persistAsync` handlers have been invoked''. `PersistentActor` provides an utility method called `deferAsync`, which works similarly to `persistAsync` yet does not persist the passed in event. It is recommended to use it for *read* operations, and actions which do not have corresponding events in your domain model.

Using this method is very similar to the persist family of methods, yet it does **not** persist the passed in event. It will be kept in memory and used when invoking the handler.

```
class MyPersistentActor extends PersistentActor {

  override def persistenceId = "my-stable-persistence-id"

  override def receiveRecover: Receive = {
    case _ => // handle recovery here
  }

  override def receiveCommand: Receive = {
    case c: String => {
      sender() ! c
      persistAsync(s"evt-$c-1") { e => sender() ! e }
      persistAsync(s"evt-$c-2") { e => sender() ! e }
      deferAsync(s"evt-$c-3") { e => sender() ! e }
    }
  }
}
```

Notice that the `sender()` is **safe** to access in the handler callback, and will be pointing to the original sender of the command for which this `deferAsync` handler was called.

The calling side will get the responses in this (guaranteed) order:

```
persistentActor ! "a"
persistentActor ! "b"

// order of received messages:
// a
// b
// evt-a-1
// evt-a-2
// evt-a-3
// evt-b-1
// evt-b-2
// evt-b-3
```

> **Warning:** The callback will not be invoked if the actor is restarted (or stopped) in between the call to `deferAsync` and the journal has processed and confirmed all preceding writes.

Nested persist calls

It is possible to call `persist` and `persistAsync` inside their respective callback blocks and they will properly retain both the thread safety (including the right value of `sender()`) as well as stashing guarantees.

In general it is encouraged to create command handlers which do not need to resort to nested event persisting, however there are situations where it may be useful. It is important to understand the ordering of callback execution in those situations, as well as their implication on the stashing behaviour (that `persist()` enforces). In the following example two persist calls are issued, and each of them issues another persist inside its callback:

```
override def receiveCommand: Receive = {
  case c: String =>
    sender() ! c

    persist(s"$c-1-outer") { outer1 =>
      sender() ! outer1
      persist(s"$c-1-inner") { inner1 =>
        sender() ! inner1
      }
    }

    persist(s"$c-2-outer") { outer2 =>
      sender() ! outer2
      persist(s"$c-2-inner") { inner2 =>
        sender() ! inner2
      }
    }
}
```

When sending two commands to this `PersistentActor`, the persist handlers will be executed in the following order:

```
persistentActor ! "a"
persistentActor ! "b"

// order of received messages:
// a
// a-outer-1
// a-outer-2
// a-inner-1
// a-inner-2
// and only then process "b"
// b
// b-outer-1
// b-outer-2
// b-inner-1
// b-inner-2
```

First the "outer layer" of persist calls is issued and their callbacks are applied. After these have successfully completed, the inner callbacks will be invoked (once the events they are persisting have been confirmed to be persisted by the journal). Only after all these handlers have been successfully invoked will the next command be delivered to the persistent Actor. In other words, the stashing of incoming commands that is guaranteed by initially calling `persist()` on the outer layer is extended until all nested `persist` callbacks have been handled.

It is also possible to nest `persistAsync` calls, using the same pattern:

```
override def receiveCommand: Receive = {
  case c: String =>
    sender() ! c
    persistAsync(c + "-outer-1") { outer =>
      sender() ! outer
      persistAsync(c + "-inner-1") { inner => sender() ! inner }
    }
    persistAsync(c + "-outer-2") { outer =>
      sender() ! outer
      persistAsync(c + "-inner-2") { inner => sender() ! inner }
    }
}
```

In this case no stashing is happening, yet events are still persisted and callbacks are executed in the expected order:

```
persistentActor ! "a"
persistentActor ! "b"

// order of received messages:
// a
// b
// a-outer-1
// a-outer-2
// b-outer-1
// b-outer-2
// a-inner-1
// a-inner-2
// b-inner-1
// b-inner-2

// which can be seen as the following causal relationship:
// a -> a-outer-1 -> a-outer-2 -> a-inner-1 -> a-inner-2
// b -> b-outer-1 -> b-outer-2 -> b-inner-1 -> b-inner-2
```

While it is possible to nest mixed `persist` and `persistAsync` with keeping their respective semantics it is not a recommended practice, as it may lead to overly complex nesting.

Warning: While it is possible to nest `persist` calls within one another, it is *not* legal call `persist` from any other Thread than the Actors message processing Thread. For example, it is not legal to call `persist` from Futures! Doing so will break the guarantees that the persist methods aim to provide. Always call `persist` and `persistAsync` from within the Actor's receive block (or methods synchronously invoked from there).

Failures

If persistence of an event fails, `onPersistFailure` will be invoked (logging the error by default), and the actor will unconditionally be stopped.

The reason that it cannot resume when persist fails is that it is unknown if the event was actually persisted or not, and therefore it is in an inconsistent state. Restarting on persistent failures will most likely fail anyway since the journal is probably unavailable. It is better to stop the actor and after a back-off timeout start it again. The `akka.pattern.BackoffSupervisor` actor is provided to support such restarts.

```
val childProps = Props[MyPersistentActor]
val props = BackoffSupervisor.props(
  Backoff.onStop(
    childProps,
    childName = "myActor",
    minBackoff = 3.seconds,
    maxBackoff = 30.seconds,
    randomFactor = 0.2))
context.actorOf(props, name = "mySupervisor")
```

If persistence of an event is rejected before it is stored, e.g. due to serialization error, `onPersistRejected` will be invoked (logging a warning by default), and the actor continues with next message.

If there is a problem with recovering the state of the actor from the journal when the actor is started, `onRecoveryFailure` is called (logging the error by default), and the actor will be stopped. Note that failure to load snapshot is also treated like this, but you can disable loading of snapshots if you for example know that serialization format has changed in an incompatible way, see *Recovery customization*.

Atomic writes

Each event is of course stored atomically, but it is also possible to store several events atomically by using the `persistAll` or `persistAllAsync` method. That means that all events passed to that method are stored or none of them are stored if there is an error.

The recovery of a persistent actor will therefore never be done partially with only a subset of events persisted by *persistAll*.

Some journals may not support atomic writes of several events and they will then reject the `persistAll` command, i.e. `onPersistRejected` is called with an exception (typically `UnsupportedOperationException`).

Batch writes

In order to optimize throughput when using `persistAsync`, a persistent actor internally batches events to be stored under high load before writing them to the journal (as a single batch). The batch size is dynamically determined by how many events are emitted during the time of a journal round-trip: after sending a batch to the journal no further batch can be sent before confirmation has been received that the previous batch has been written. Batch writes are never timer-based which keeps latencies at a minimum.

Message deletion

It is possible to delete all messages (journaled by a single persistent actor) up to a specified sequence number; Persistent actors may call the `deleteMessages` method to this end.

Deleting messages in event sourcing based applications is typically either not used at all, or used in conjunction with *snapshotting*, i.e. after a snapshot has been successfully stored, a `deleteMessages(toSequenceNr)` up until the sequence number of the data held by that snapshot can be issued to safely delete the previous events while still having access to the accumulated state during replays - by loading the snapshot.

> **Warning:** If you are using *Persistence Query*, query results may be missing deleted messages in a journal, depending on how deletions are implemented in the journal plugin. Unless you use a plugin which still shows deleted messages in persistence query results, you have to design your application so that it is not affected by missing messages.

The result of the `deleteMessages` request is signaled to the persistent actor with a `DeleteMessagesSuccess` message if the delete was successful or a `DeleteMessagesFailure` message if it failed.

Message deletion doesn't affect the highest sequence number of the journal, even if all messages were deleted from it after `deleteMessages` invocation.

Persistence status handling

Persisting, deleting, and replaying messages can either succeed or fail.

Method	Success	Failure / Rejection	After failure handler invoked
`persist` / `persistAsync`	persist handler invoked	`onPersistFailure`	Actor is stopped.
		`onPersistRejected`	No automatic actions.
`recovery`	`RecoveryCompleted`	`onRecoveryFailure`	Actor is stopped.
`deleteMessages`	`DeleteMessagesSuccess`	`DeleteMessagesFailure`	No automatic actions.

The most important operations (`persist` and `recovery`) have failure handlers modelled as explicit callbacks which the user can override in the `PersistentActor`. The default implementations of these handlers emit a log message (`error` for persist/recovery failures, and `warning` for others), logging the failure cause and information about which message caused the failure.

For critical failures, such as recovery or persisting events failing, the persistent actor will be stopped after the failure handler is invoked. This is because if the underlying journal implementation is signalling persistence failures it is most likely either failing completely or overloaded and restarting right-away and trying to persist the event again will most likely not help the journal recover – as it would likely cause a Thundering herd problem, as many persistent actors would restart and try to persist their events again. Instead, using a `BackoffSupervisor` (as described in *Failures*) which implements an exponential-backoff strategy which allows for more breathing room for the journal to recover between restarts of the persistent actor.

Note: Journal implementations may choose to implement a retry mechanism, e.g. such that only after a write fails N number of times a persistence failure is signalled back to the user. In other words, once a journal returns a failure, it is considered *fatal* by Akka Persistence, and the persistent actor which caused the failure will be stopped.

Check the documentation of the journal implementation you are using for details if/how it is using this technique.

Safely shutting down persistent actors

Special care should be given when shutting down persistent actors from the outside. With normal Actors it is often acceptable to use the special *PoisonPill* message to signal to an Actor that it should stop itself once it receives this message – in fact this message is handled automatically by Akka, leaving the target actor no way to refuse stopping itself when given a poison pill.

This can be dangerous when used with `PersistentActor` due to the fact that incoming commands are *stashed* while the persistent actor is awaiting confirmation from the Journal that events have been written when `persist()` was used. Since the incoming commands will be drained from the Actor's mailbox and put into its internal stash while awaiting the confirmation (thus, before calling the persist handlers) the Actor **may receive and (auto)handle the PoisonPill before it processes the other messages which have been put into its stash**, causing a pre-mature shutdown of the Actor.

> **Warning:** Consider using explicit shut-down messages instead of `PoisonPill` when working with persistent actors.

The example below highlights how messages arrive in the Actor's mailbox and how they interact with its internal stashing mechanism when `persist()` is used. Notice the early stop behaviour that occurs when `PoisonPill` is used:

```
/** Explicit shutdown message */
case object Shutdown

class SafePersistentActor extends PersistentActor {
  override def persistenceId = "safe-actor"

  override def receiveCommand: Receive = {
    case c: String =>
      println(c)
      persist(s"handle-$c") { println(_) }
    case Shutdown =>
      context.stop(self)
  }

  override def receiveRecover: Receive = {
    case _ => // handle recovery here
  }
}
```

```
// UN-SAFE, due to PersistentActor's command stashing:
persistentActor ! "a"
persistentActor ! "b"
```

```
persistentActor ! PoisonPill
// order of received messages:
// a
//   # b arrives at mailbox, stashing;        internal-stash = [b]
// PoisonPill is an AutoReceivedMessage, is handled automatically
// !! stop !!
// Actor is stopped without handling `b` nor the `a` handler!
```

```
// SAFE:
persistentActor ! "a"
persistentActor ! "b"
persistentActor ! Shutdown
// order of received messages:
// a
//   # b arrives at mailbox, stashing;        internal-stash = [b]
//   # Shutdown arrives at mailbox, stashing; internal-stash = [b, Shutdown]
// handle-a
//   # unstashing;                            internal-stash = [Shutdown]
// b
// handle-b
//   # unstashing;                            internal-stash = []
// Shutdown
// -- stop --
```

Replay Filter

There could be cases where event streams are corrupted and multiple writers (i.e. multiple persistent actor instances) journaled different messages with the same sequence number. In such a case, you can configure how you filter replayed messages from multiple writers, upon recovery.

In your configuration, under the `akka.persistence.journal.xxx.replay-filter` section (where `xxx` is your journal plugin id), you can select the replay filter `mode` from one of the following values:

- repair-by-discard-old
- fail
- warn
- off

For example, if you configure the replay filter for leveldb plugin, it looks like this:

```
# The replay filter can detect a corrupt event stream by inspecting
# sequence numbers and writerUuid when replaying events.
akka.persistence.journal.leveldb.replay-filter {
  # What the filter should do when detecting invalid events.
  # Supported values:
  # `repair-by-discard-old` : discard events from old writers,
  #                           warning is logged
  # `fail` : fail the replay, error is logged
  # `warn` : log warning but emit events untouched
  # `off` : disable this feature completely
  mode = repair-by-discard-old
}
```

3.8.4 Persistent Views

Warning: `PersistentView` is deprecated. Use *Persistence Query* instead. The corresponding query type is `EventsByPersistenceId`. There are several alternatives for connecting the `Source` to an actor corresponding to a previous `PersistentView` actor:

- Sink.actorRef is simple, but has the disadvantage that there is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow
- mapAsync combined with *Ask: Send-And-Receive-Future* is almost as simple with the advantage of back-pressure being propagated all the way
- ActorSubscriber in case you need more fine grained control

The consuming actor may be a plain `Actor` or a `PersistentActor` if it needs to store its own state (e.g. fromSequenceNr offset).

Persistent views can be implemented by extending the `PersistentView` trait and implementing the `receive` and the `persistenceId` methods.

```
class MyView extends PersistentView {
  override def persistenceId: String = "some-persistence-id"
  override def viewId: String = "some-persistence-id-view"

  def receive: Receive = {
    case payload if isPersistent =>
    // handle message from journal...
    case payload                 =>
    // handle message from user-land...
  }
}
```

The `persistenceId` identifies the persistent actor from which the view receives journaled messages. It is not necessary that the referenced persistent actor is actually running. Views read messages from a persistent actor's journal directly. When a persistent actor is started later and begins to write new messages, by default the corresponding view is updated automatically.

It is possible to determine if a message was sent from the Journal or from another actor in user-land by calling the `isPersistent` method. Having that said, very often you don't need this information at all and can simply apply the same logic to both cases (skip the `if isPersistent` check).

Updates

The default update interval of all views of an actor system is configurable:

```
akka.persistence.view.auto-update-interval = 5s
```

`PersistentView` implementation classes may also override the `autoUpdateInterval` method to return a custom update interval for a specific view class or view instance. Applications may also trigger additional updates at any time by sending a view an `Update` message.

```
val view = system.actorOf(Props[MyView])
view ! Update(await = true)
```

If the `await` parameter is set to `true`, messages that follow the `Update` request are processed when the incremental message replay, triggered by that update request, completed. If set to `false` (default), messages following the update request may interleave with the replayed message stream. Automated updates always run with `await = false`.

Automated updates of all persistent views of an actor system can be turned off by configuration:

```
akka.persistence.view.auto-update = off
```

Implementation classes may override the configured default value by overriding the `autoUpdate` method. To limit the number of replayed messages per update request, applications can configure a custom `akka.persistence.view.auto-update-replay-max` value or override the `autoUpdateReplayMax` method. The number of replayed messages for manual updates can be limited with the `replayMax` parameter of the `Update` message.

Recovery

Initial recovery of persistent views works the very same way as for persistent actors (i.e. by sending a `Recover` message to self). The maximum number of replayed messages during initial recovery is determined by `autoUpdateReplayMax`. Further possibilities to customize initial recovery are explained in section *Recovery*.

Identifiers

A persistent view must have an identifier that doesn't change across different actor incarnations. The identifier must be defined with the `viewId` method.

The `viewId` must differ from the referenced `persistenceId`, unless *Snapshots* of a view and its persistent actor should be shared (which is what applications usually do not want).

3.8.5 Snapshots

Snapshots can dramatically reduce recovery times of persistent actors and views. The following discusses snapshots in context of persistent actors but this is also applicable to persistent views.

Persistent actors can save snapshots of internal state by calling the `saveSnapshot` method. If saving of a snapshot succeeds, the persistent actor receives a `SaveSnapshotSuccess` message, otherwise a `SaveSnapshotFailure` message

```
var state: Any = _

override def receiveCommand: Receive = {
  case "snap"                                 => saveSnapshot(state)
  case SaveSnapshotSuccess(metadata)          => // ...
  case SaveSnapshotFailure(metadata, reason) => // ...
}
```

where `metadata` is of type `SnapshotMetadata`:

```
final case class SnapshotMetadata(persistenceId: String, sequenceNr: Long,␣
↪timestamp: Long = 0L)
```

During recovery, the persistent actor is offered a previously saved snapshot via a `SnapshotOffer` message from which it can initialize internal state.

```
var state: Any = _

override def receiveRecover: Receive = {
  case SnapshotOffer(metadata, offeredSnapshot) => state = offeredSnapshot
  case RecoveryCompleted                        =>
  case event                                    => // ...
}
```

The replayed messages that follow the `SnapshotOffer` message, if any, are younger than the offered snapshot. They finally recover the persistent actor to its current (i.e. latest) state.

In general, a persistent actor is only offered a snapshot if that persistent actor has previously saved one or more snapshots and at least one of these snapshots matches the `SnapshotSelectionCriteria` that can be specified for recovery.

```
override def recovery = Recovery(fromSnapshot = SnapshotSelectionCriteria(
  maxSequenceNr = 457L,
  maxTimestamp = System.currentTimeMillis))
```

If not specified, they default to `SnapshotSelectionCriteria.Latest` which selects the latest (= youngest) snapshot. To disable snapshot-based recovery, applications should use `SnapshotSelectionCriteria.None`. A recovery where no saved snapshot matches the specified `SnapshotSelectionCriteria` will replay all journaled messages.

Note: In order to use snapshots, a default snapshot-store (`akka.persistence.snapshot-store.plugin`) must be configured, or the `PersistentActor` can pick a snapshot store explicitly by overriding `def snapshotPluginId: String`.

Since it is acceptable for some applications to not use any snapshotting, it is legal to not configure a snapshot store. However, Akka will log a warning message when this situation is detected and then continue to operate until an actor tries to store a snapshot, at which point the operation will fail (by replying with an `SaveSnapshotFailure` for example).

Note that *Cluster Sharding* is using snapshots, so if you use Cluster Sharding you need to define a snapshot store plugin.

Snapshot deletion

A persistent actor can delete individual snapshots by calling the `deleteSnapshot` method with the sequence number of when the snapshot was taken.

To bulk-delete a range of snapshots matching `SnapshotSelectionCriteria`, persistent actors should use the `deleteSnapshots` method.

Snapshot status handling

Saving or deleting snapshots can either succeed or fail – this information is reported back to the persistent actor via status messages as illustrated in the following table.

Method	Success	Failure message
`saveSnapshot(Any)`	`SaveSnapshotSuccess`	`SaveSnapshotFailure`
`deleteSnapshot(Long)`	`DeleteSnapshotSuccess`	`DeleteSnapshotFailure`
`deleteSnapshots(SnapshotSelectionCriteria)`	`DeleteSnapshotsSuccess`	`DeleteSnapshotsFailure`

If failure messages are left unhandled by the actor, a default warning log message will be logged for each incoming failure message. No default action is performed on the success messages, however you're free to handle them e.g. in order to delete an in memory representation of the snapshot, or in the case of failure to attempt save the snapshot again.

3.8.6 At-Least-Once Delivery

To send messages with at-least-once delivery semantics to destinations you can mix-in `AtLeastOnceDelivery` trait to your `PersistentActor` on the sending side. It takes care of re-sending messages when they have not been confirmed within a configurable timeout.

The state of the sending actor, including which messages have been sent that have not been confirmed by the recipient must be persistent so that it can survive a crash of the sending actor or JVM. The `AtLeastOnceDelivery` trait does not persist anything by itself. It is your responsibility to persist the intent that a message is sent and that a confirmation has been received.

Note: At-least-once delivery implies that original message sending order is not always preserved, and the destination may receive duplicate messages. Semantics do not match those of a normal `ActorRef` send operation:

- it is not at-most-once delivery
- message order for the same sender–receiver pair is not preserved due to possible resends
- after a crash and restart of the destination messages are still delivered to the new actor incarnation

These semantics are similar to what an `ActorPath` represents (see *Actor Lifecycle*), therefore you need to supply a path and not a reference when delivering messages. The messages are sent to the path with an actor selection.

Use the `deliver` method to send a message to a destination. Call the `confirmDelivery` method when the destination has replied with a confirmation message.

Relationship between deliver and confirmDelivery

To send messages to the destination path, use the `deliver` method after you have persisted the intent to send the message.

The destination actor must send back a confirmation message. When the sending actor receives this confirmation message you should persist the fact that the message was delivered successfully and then call the `confirmDelivery` method.

If the persistent actor is not currently recovering, the `deliver` method will send the message to the destination actor. When recovering, messages will be buffered until they have been confirmed using `confirmDelivery`. Once recovery has completed, if there are outstanding messages that have not been confirmed (during the message replay), the persistent actor will resend these before sending any other messages.

Deliver requires a `deliveryIdToMessage` function to pass the provided `deliveryId` into the message so that the correlation between `deliver` and `confirmDelivery` is possible. The `deliveryId` must do the round trip. Upon receipt of the message, the destination actor will send the same‘‘deliveryId‘‘ wrapped in a confirmation message back to the sender. The sender will then use it to call `confirmDelivery` method to complete the delivery routine.

```
import akka.actor.{ Actor, ActorSelection }
import akka.persistence.AtLeastOnceDelivery

case class Msg(deliveryId: Long, s: String)
case class Confirm(deliveryId: Long)

sealed trait Evt
case class MsgSent(s: String) extends Evt
case class MsgConfirmed(deliveryId: Long) extends Evt

class MyPersistentActor(destination: ActorSelection)
  extends PersistentActor with AtLeastOnceDelivery {

  override def persistenceId: String = "persistence-id"

  override def receiveCommand: Receive = {
    case s: String           => persist(MsgSent(s))(updateState)
    case Confirm(deliveryId) => persist(MsgConfirmed(deliveryId))(updateState)
  }

  override def receiveRecover: Receive = {
    case evt: Evt => updateState(evt)
  }

  def updateState(evt: Evt): Unit = evt match {
    case MsgSent(s) =>
      deliver(destination)(deliveryId => Msg(deliveryId, s))
```

```
    case MsgConfirmed(deliveryId) => confirmDelivery(deliveryId)
  }
}

class MyDestination extends Actor {
  def receive = {
    case Msg(deliveryId, s) =>
      // ...
      sender() ! Confirm(deliveryId)
  }
}
```

The `deliveryId` generated by the persistence module is a strictly monotonically increasing sequence number without gaps. The same sequence is used for all destinations of the actor, i.e. when sending to multiple destinations the destinations will see gaps in the sequence. It is not possible to use custom `deliveryId`. However, you can send a custom correlation identifier in the message to the destination. You must then retain a mapping between the internal `deliveryId` (passed into the `deliveryIdToMessage` function) and your custom correlation id (passed into the message). You can do this by storing such mapping in a `Map(correlationId -> deliveryId)` from which you can retrieve the `deliveryId` to be passed into the `confirmDelivery` method once the receiver of your message has replied with your custom correlation id.

The `AtLeastOnceDelivery` trait has a state consisting of unconfirmed messages and a sequence number. It does not store this state itself. You must persist events corresponding to the `deliver` and `confirmDelivery` invocations from your `PersistentActor` so that the state can be restored by calling the same methods during the recovery phase of the `PersistentActor`. Sometimes these events can be derived from other business level events, and sometimes you must create separate events. During recovery, calls to `deliver` will not send out messages, those will be sent later if no matching `confirmDelivery` will have been performed.

Support for snapshots is provided by `getDeliverySnapshot` and `setDeliverySnapshot`. The `AtLeastOnceDeliverySnapshot` contains the full delivery state, including unconfirmed messages. If you need a custom snapshot for other parts of the actor state you must also include the `AtLeastOnceDeliverySnapshot`. It is serialized using protobuf with the ordinary Akka serialization mechanism. It is easiest to include the bytes of the `AtLeastOnceDeliverySnapshot` as a blob in your custom snapshot.

The interval between redelivery attempts is defined by the `redeliverInterval` method. The default value can be configured with the `akka.persistence.at-least-once-delivery.redeliver-interval` configuration key. The method can be overridden by implementation classes to return non-default values.

The maximum number of messages that will be sent at each redelivery burst is defined by the `redeliveryBurstLimit` method (burst frequency is half of the redelivery interval). If there's a lot of unconfirmed messages (e.g. if the destination is not available for a long time), this helps to prevent an overwhelming amount of messages to be sent at once. The default value can be configured with the `akka.persistence.at-least-once-delivery.redelivery-burst-limit` configuration key. The method can be overridden by implementation classes to return non-default values.

After a number of delivery attempts a `AtLeastOnceDelivery.UnconfirmedWarning` message will be sent to `self`. The re-sending will still continue, but you can choose to call `confirmDelivery` to cancel the re-sending. The number of delivery attempts before emitting the warning is defined by the `warnAfterNumberOfUnconfirmedAttempts` method. The default value can be configured with the `akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts` configuration key. The method can be overridden by implementation classes to return non-default values.

The `AtLeastOnceDelivery` trait holds messages in memory until their successful delivery has been confirmed. The maximum number of unconfirmed messages that the actor is allowed to hold in memory is defined by the `maxUnconfirmedMessages` method. If this limit is exceed the `deliver` method will not accept more messages and it will throw `AtLeastOnceDelivery.MaxUnconfirmedMessagesExceededException`. The default value can be configured with the `akka.persistence.at-least-once-delivery.max-unconfirmed-messages` configuration key. The method can be overridden by implementation classes to return non-default values.

3.8.7 Event Adapters

In long running projects using event sourcing sometimes the need arises to detach the data model from the domain model completely.

Event Adapters help in situations where:

- **Version Migrations** – existing events stored in *Version 1* should be "upcasted" to a new *Version 2* representation, and the process of doing so involves actual code, not just changes on the serialization layer. For these scenarios the `toJournal` function is usually an identity function, however the `fromJournal` is implemented as `v1.Event=>v2.Event`, performing the neccessary mapping inside the fromJournal method. This technique is sometimes refered to as "upcasting" in other CQRS libraries.
- **Separating Domain and Data models** – thanks to EventAdapters it is possible to completely separate the domain model from the model used to persist data in the Journals. For example one may want to use case classes in the domain model, however persist their protocol-buffer (or any other binary serialization format) counter-parts to the Journal. A simple `toJournal:MyModel=>MyDataModel` and `fromJournal:MyDataModel=>MyModel` adapter can be used to implement this feature.
- **Journal Specialized Data Types** – exposing data types understood by the underlying Journal, for example for data stores which understand JSON it is possible to write an EventAdapter `toJournal:Any=>JSON` such that the Journal can *directly* store the json instead of serializing the object to its binary representation.

Implementing an EventAdapter is rather stright forward:

```
class MyEventAdapter(system: ExtendedActorSystem) extends EventAdapter {
  override def manifest(event: Any): String =
    "" // when no manifest needed, return ""

  override def toJournal(event: Any): Any =
    event // identity

  override def fromJournal(event: Any, manifest: String): EventSeq =
    EventSeq.single(event) // identity
}
```

Then in order for it to be used on events coming to and from the journal you must bind it using the below configuration syntax:

```
akka.persistence.journal {
  inmem {
    event-adapters {
      tagging        = "docs.persistence.MyTaggingEventAdapter"
      user-upcasting = "docs.persistence.UserUpcastingEventAdapter"
      item-upcasting = "docs.persistence.ItemUpcastingEventAdapter"
    }

    event-adapter-bindings {
      "docs.persistence.Item"        = tagging
      "docs.persistence.TaggedEvent" = tagging
      "docs.persistence.v1.Event"    = [user-upcasting, item-upcasting]
    }
  }
}
```

It is possible to bind multiple adapters to one class *for recovery*, in which case the `fromJournal` methods of all bound adapters will be applied to a given matching event (in order of definition in the configuration). Since each adapter may return from `0` to `n` adapted events (called as `EventSeq`), each adapter can investigate the event and if it should indeed adapt it return the adapted event(s) for it. Other adapters which do not have anything to contribute during this adaptation simply return `EventSeq.empty`. The adapted events are then delivered in-order to the `PersistentActor` during replay.

Note: For more advanced schema evolution techniques refer to the *Persistence - Schema Evolution* documentation.

3.8.8 Persistent FSM

`PersistentFSM` handles the incoming messages in an FSM like fashion. Its internal state is persisted as a sequence of changes, later referred to as domain events. Relationship between incoming messages, FSM's states and transitions, persistence of domain events is defined by a DSL.

Warning: `PersistentFSM` is marked as **"experimental"** as of its introduction in Akka 2.4.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply to the contents of the *classes related to ''PersistentFSM'*.

A Simple Example

To demonstrate the features of the `PersistentFSM` trait, consider an actor which represents a Web store customer. The contract of our "WebStoreCustomerFSMActor" is that it accepts the following commands:

```
sealed trait Command
case class AddItem(item: Item) extends Command
case object Buy extends Command
case object Leave extends Command
case object GetCurrentCart extends Command
```

`AddItem` sent when the customer adds an item to a shopping cart `Buy` - when the customer finishes the purchase `Leave` - when the customer leaves the store without purchasing anything `GetCurrentCart` allows to query the current state of customer's shopping cart

The customer can be in one of the following states:

```
sealed trait UserState extends FSMState
case object LookingAround extends UserState {
  override def identifier: String = "Looking Around"
}
case object Shopping extends UserState {
  override def identifier: String = "Shopping"
}
case object Inactive extends UserState {
  override def identifier: String = "Inactive"
}
case object Paid extends UserState {
  override def identifier: String = "Paid"
}
```

`LookingAround` customer is browsing the site, but hasn't added anything to the shopping cart `Shopping` customer has recently added items to the shopping cart `Inactive` customer has items in the shopping cart, but hasn't added anything recently `Paid` customer has purchased the items

Note: `PersistentFSM` states must inherit from trait `PersistentFSM.FSMState` and implement the `def identifier: String` method. This is required in order to simplify the serialization of FSM states. String identifiers should be unique!

Customer's actions are "recorded" as a sequence of "domain events" which are persisted. Those events are replayed on an actor's start in order to restore the latest customer's state:

```
sealed trait DomainEvent
case class ItemAdded(item: Item) extends DomainEvent
case object OrderExecuted extends DomainEvent
case object OrderDiscarded extends DomainEvent
```

Customer state data represents the items in a customer's shopping cart:

```
case class Item(id: String, name: String, price: Float)

sealed trait ShoppingCart {
  def addItem(item: Item): ShoppingCart
  def empty(): ShoppingCart
}
case object EmptyShoppingCart extends ShoppingCart {
  def addItem(item: Item) = NonEmptyShoppingCart(item :: Nil)
  def empty() = this
}
case class NonEmptyShoppingCart(items: Seq[Item]) extends ShoppingCart {
  def addItem(item: Item) = NonEmptyShoppingCart(items :+ item)
  def empty() = EmptyShoppingCart
}
```

Here is how everything is wired together:

```
startWith(LookingAround, EmptyShoppingCart)

when(LookingAround) {
  case Event(AddItem(item), _) ⇒
    goto(Shopping) applying ItemAdded(item) forMax (1 seconds)
  case Event(GetCurrentCart, data) ⇒
    stay replying data
}

when(Shopping) {
  case Event(AddItem(item), _) ⇒
    stay applying ItemAdded(item) forMax (1 seconds)
  case Event(Buy, _) ⇒
    goto(Paid) applying OrderExecuted andThen {
      case NonEmptyShoppingCart(items) ⇒
        reportActor ! PurchaseWasMade(items)
        saveStateSnapshot()
      case EmptyShoppingCart ⇒ saveStateSnapshot()
    }
  case Event(Leave, _) ⇒
    stop applying OrderDiscarded andThen {
      case _ ⇒
        reportActor ! ShoppingCardDiscarded
        saveStateSnapshot()
    }
  case Event(GetCurrentCart, data) ⇒
    stay replying data
  case Event(StateTimeout, _) ⇒
    goto(Inactive) forMax (2 seconds)
}

when(Inactive) {
  case Event(AddItem(item), _) ⇒
    goto(Shopping) applying ItemAdded(item) forMax (1 seconds)
  case Event(StateTimeout, _) ⇒
    stop applying OrderDiscarded andThen {
      case _ ⇒ reportActor ! ShoppingCardDiscarded
    }
}
```

```
when(Paid) {
  case Event(Leave, _) ⇒ stop()
  case Event(GetCurrentCart, data) ⇒
    stay replying data
}
```

Note: State data can only be modified directly on initialization. Later it's modified only as a result of applying domain events. Override the `applyEvent` method to define how state data is affected by domain events, see the example below

```
override def applyEvent(event: DomainEvent, cartBeforeEvent: ShoppingCart):␣
↪ShoppingCart = {
  event match {
    case ItemAdded(item) ⇒ cartBeforeEvent.addItem(item)
    case OrderExecuted   ⇒ cartBeforeEvent
    case OrderDiscarded  ⇒ cartBeforeEvent.empty()
  }
}
```

`andThen` can be used to define actions which will be executed following event's persistence - convenient for "side effects" like sending a message or logging. Notice that actions defined in `andThen` block are not executed on recovery:

```
goto(Paid) applying OrderExecuted andThen {
  case NonEmptyShoppingCart(items) ⇒
    reportActor ! PurchaseWasMade(items)
}
```

A snapshot of state data can be persisted by calling the `saveStateSnapshot()` method:

```
stop applying OrderDiscarded andThen {
  case _ ⇒
    reportActor ! ShoppingCardDiscarded
    saveStateSnapshot()
}
```

On recovery state data is initialized according to the latest available snapshot, then the remaining domain events are replayed, triggering the `applyEvent` method.

3.8.9 Storage plugins

Storage backends for journals and snapshot stores are pluggable in the Akka persistence extension.

A directory of persistence journal and snapshot store plugins is available at the Akka Community Projects page, see Community plugins

Plugins can be selected either by "default" for all persistent actors and views, or "individually", when a persistent actor or view defines its own set of plugins.

When a persistent actor or view does NOT override the `journalPluginId` and `snapshotPluginId` methods, the persistence extension will use the "default" journal and snapshot-store plugins configured in `reference.conf`:

```
akka.persistence.journal.plugin = ""
akka.persistence.snapshot-store.plugin = ""
```

However, these entries are provided as empty "", and require explicit user configuration via override in the user `application.conf`. For an example of a journal plugin which writes messages to LevelDB see *Local LevelDB*

journal. For an example of a snapshot store plugin which writes snapshots as individual files to the local filesystem see *Local snapshot store*.

Applications can provide their own plugins by implementing a plugin API and activating them by configuration. Plugin development requires the following imports:

```
import akka.persistence._
import akka.persistence.journal._
import akka.persistence.snapshot._
```

Eager initialization of persistence plugin

By default, persistence plugins are started on-demand, as they are used. In some case, however, it might be beneficial to start a certain plugin eagerly. In order to do that, you should first add the `akka.persistence.Persistence` under the `akka.extensions` key. Then, specify the IDs of plugins you wish to start automatically under `akka.persistence.journal.auto-start-journals` and `akka.persistence.snapshot-store.auto-start-snapshot-stores`.

Journal plugin API

A journal plugin extends `AsyncWriteJournal`.

`AsyncWriteJournal` is an actor and the methods to be implemented are:

```
/**
 * Plugin API: asynchronously writes a batch (`Seq`) of persistent messages to the
 * journal.
 *
 * The batch is only for performance reasons, i.e. all messages don't have to be␣
↪written
 * atomically. Higher throughput can typically be achieved by using batch inserts␣
↪of many
 * records compared to inserting records one-by-one, but this aspect depends on the
 * underlying data store and a journal implementation can implement it as␣
↪efficient as
 * possible. Journals should aim to persist events in-order for a given␣
↪`persistenceId`
 * as otherwise in case of a failure, the persistent state may be end up being␣
↪inconsistent.
 *
 * Each `AtomicWrite` message contains the single `PersistentRepr` that␣
↪corresponds to
 * the event that was passed to the `persist` method of the `PersistentActor`, or␣
↪it
 * contains several `PersistentRepr` that corresponds to the events that were␣
↪passed
 * to the `persistAll` method of the `PersistentActor`. All `PersistentRepr` of the
 * `AtomicWrite` must be written to the data store atomically, i.e. all or none␣
↪must
 * be stored. If the journal (data store) cannot support atomic writes of multiple
 * events it should reject such writes with a `Try` `Failure` with an
 * `UnsupportedOperationException` describing the issue. This limitation should
 * also be documented by the journal plugin.
 *
 * If there are failures when storing any of the messages in the batch the returned
 * `Future` must be completed with failure. The `Future` must only be completed␣
↪with
 * success when all messages in the batch have been confirmed to be stored␣
↪successfully,
 * i.e. they will be readable, and visible, in a subsequent replay. If there is
 * uncertainty about if the messages were stored or not the `Future` must be␣
↪completed
```

```
 * with failure.
 *
 * Data store connection problems must be signaled by completing the `Future` with
 * failure.
 *
 * The journal can also signal that it rejects individual messages (`AtomicWrite`)␣
↪by
 * the returned `immutable.Seq[Try[Unit]]`. It is possible but not mandatory to␣
↪reduce
 * number of allocations by returning `Future.successful(Nil)` for the happy path,
 * i.e. when no messages are rejected. Otherwise the returned `Seq` must have as␣
↪many elements
 * as the input `messages` `Seq`. Each `Try` element signals if the corresponding
 * `AtomicWrite` is rejected or not, with an exception describing the problem.␣
↪Rejecting
 * a message means it was not stored, i.e. it must not be included in a later␣
↪replay.
 * Rejecting a message is typically done before attempting to store it, e.g.␣
↪because of
 * serialization error.
 *
 * Data store connection problems must not be signaled as rejections.
 *
 * It is possible but not mandatory to reduce number of allocations by returning
 * `Future.successful(Nil)` for the happy path, i.e. when no messages are rejected.
 *
 * Calls to this method are serialized by the enclosing journal actor. If you spawn
 * work in asynchronous tasks it is alright that they complete the futures in any␣
↪order,
 * but the actual writes for a specific persistenceId should be serialized to avoid
 * issues such as events of a later write are visible to consumers (query side, or␣
↪replay)
 * before the events of an earlier write are visible.
 * A PersistentActor will not send a new WriteMessages request before the previous␣
↪one
 * has been completed.
 *
 * Please note that the `sender` field of the contained PersistentRepr objects has␣
↪been
 * nulled out (i.e. set to `ActorRef.noSender`) in order to not use space in the␣
↪journal
 * for a sender reference that will likely be obsolete during replay.
 *
 * Please also note that requests for the highest sequence number may be made␣
↪concurrently
 * to this call executing for the same `persistenceId`, in particular it is␣
↪possible that
 * a restarting actor tries to recover before its outstanding writes have␣
↪completed. In
 * the latter case it is highly desirable to defer reading the highest sequence␣
↪number
 * until all outstanding writes have completed, otherwise the PersistentActor may␣
↪reuse
 * sequence numbers.
 *
 * This call is protected with a circuit-breaker.
 */
def asyncWriteMessages(messages: immutable.Seq[AtomicWrite]): Future[immutable.
↪Seq[Try[Unit]]]

/**
 * Plugin API: asynchronously deletes all persistent messages up to `toSequenceNr`
 * (inclusive).
```

```
 *
 * This call is protected with a circuit-breaker.
 * Message deletion doesn't affect the highest sequence number of messages,␣
↪journal must maintain the highest sequence number and never decrease it.
 */
def asyncDeleteMessagesTo(persistenceId: String, toSequenceNr: Long): Future[Unit]

/**
 * Plugin API
 *
 * Allows plugin implementers to use `f pipeTo self` and
 * handle additional messages for implementing advanced features
 *
 */
def receivePluginInternal: Actor.Receive = Actor.emptyBehavior
```

If the storage backend API only supports synchronous, blocking writes, the methods should be implemented as:

```
def asyncWriteMessages(messages: immutable.Seq[AtomicWrite]): Future[immutable.
↪Seq[Try[Unit]]] =
  Future.fromTry(Try {
    // blocking call here
    ???
  })
```

A journal plugin must also implement the methods defined in `AsyncRecovery` for replays and sequence number recovery:

```
/**
 * Plugin API: asynchronously replays persistent messages. Implementations replay
 * a message by calling `replayCallback`. The returned future must be completed
 * when all messages (matching the sequence number bounds) have been replayed.
 * The future must be completed with a failure if any of the persistent messages
 * could not be replayed.
 *
 * The `replayCallback` must also be called with messages that have been marked
 * as deleted. In this case a replayed message's `deleted` method must return
 * `true`.
 *
 * The `toSequenceNr` is the lowest of what was returned by [[
↪#asyncReadHighestSequenceNr]]
 * and what the user specified as recovery [[akka.persistence.Recovery]] parameter.
 * This does imply that this call is always preceded by reading the highest␣
↪sequence
 * number for the given `persistenceId`.
 *
 * This call is NOT protected with a circuit-breaker because it may take long time
 * to replay all events. The plugin implementation itself must protect against
 * an unresponsive backend store and make sure that the returned Future is
 * completed with success or failure within reasonable time. It is not allowed
 * to ignore completing the future.
 *
 * @param persistenceId persistent actor id.
 * @param fromSequenceNr sequence number where replay should start (inclusive).
 * @param toSequenceNr sequence number where replay should end (inclusive).
 * @param max maximum number of messages to be replayed.
 * @param recoveryCallback called to replay a single message. Can be called from␣
↪any
 *                         thread.
 *
 * @see [[AsyncWriteJournal]]
 */
```

```
def asyncReplayMessages(persistenceId: String, fromSequenceNr: Long, toSequenceNr:␣
↪Long,
                        max: Long)(recoveryCallback: PersistentRepr ⇒ Unit):␣
↪Future[Unit]

/**
 * Plugin API: asynchronously reads the highest stored sequence number for the
 * given `persistenceId`. The persistent actor will use the highest sequence
 * number after recovery as the starting point when persisting new events.
 * This sequence number is also used as `toSequenceNr` in subsequent call
 * to [[#asyncReplayMessages]] unless the user has specified a lower␣
↪`toSequenceNr`.
 * Journal must maintain the highest sequence number and never decrease it.
 *
 * This call is protected with a circuit-breaker.
 *
 * Please also note that requests for the highest sequence number may be made␣
↪concurrently
 * to writes executing for the same `persistenceId`, in particular it is possible␣
↪that
 * a restarting actor tries to recover before its outstanding writes have␣
↪completed.
 *
 * @param persistenceId persistent actor id.
 * @param fromSequenceNr hint where to start searching for the highest sequence
 *                       number. When a persistent actor is recovering this
 *                       `fromSequenceNr` will be the sequence number of the used
 *                       snapshot or `0L` if no snapshot is used.
 */
def asyncReadHighestSequenceNr(persistenceId: String, fromSequenceNr: Long):␣
↪Future[Long]
```

A journal plugin can be activated with the following minimal configuration:

```
# Path to the journal plugin to be used
akka.persistence.journal.plugin = "my-journal"

# My custom journal plugin
my-journal {
  # Class name of the plugin.
  class = "docs.persistence.MyJournal"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.actor.default-dispatcher"
}
```

The specified plugin `class` must have a no-arg constructor. The `plugin-dispatcher` is the dispatcher used for the plugin actor. If not specified, it defaults to `akka.persistence.dispatchers.default-plugin-dispatcher`.

The journal plugin instance is an actor so the methods corresponding to requests from persistent actors are executed sequentially. It may delegate to asynchronous libraries, spawn futures, or delegate to other actors to achive parallelism.

The journal plugin class must have a constructor without parameters or a constructor with one `com.typesafe.config.Config` parameter. The plugin section of the actor system's config will be passed in the config constructor parameter.

Don't run journal tasks/futures on the system default dispatcher, since that might starve other tasks.

Snapshot store plugin API

A snapshot store plugin must extend the `SnapshotStore` actor and implement the following methods:

```
/**
 * Plugin API: asynchronously loads a snapshot.
 *
 * If the future `Option` is `None` then all events will be replayed,
 * i.e. there was no snapshot. If snapshot could not be loaded the `Future`
 * should be completed with failure. That is important because events may
 * have been deleted and just replaying the events might not result in a valid
 * state.
 *
 * This call is protected with a circuit-breaker.
 *
 * @param persistenceId id of the persistent actor.
 * @param criteria selection criteria for loading.
 */
def loadAsync(persistenceId: String, criteria: SnapshotSelectionCriteria):␣
↪Future[Option[SelectedSnapshot]]

/**
 * Plugin API: asynchronously saves a snapshot.
 *
 * This call is protected with a circuit-breaker.
 *
 * @param metadata snapshot metadata.
 * @param snapshot snapshot.
 */
def saveAsync(metadata: SnapshotMetadata, snapshot: Any): Future[Unit]

/**
 * Plugin API: deletes the snapshot identified by `metadata`.
 *
 * This call is protected with a circuit-breaker.
 *
 * @param metadata snapshot metadata.
 */
def deleteAsync(metadata: SnapshotMetadata): Future[Unit]

/**
 * Plugin API: deletes all snapshots matching `criteria`.
 *
 * This call is protected with a circuit-breaker.
 *
 * @param persistenceId id of the persistent actor.
 * @param criteria selection criteria for deleting.
 */
def deleteAsync(persistenceId: String, criteria: SnapshotSelectionCriteria):␣
↪Future[Unit]

/**
 * Plugin API
 * Allows plugin implementers to use `f pipeTo self` and
 * handle additional messages for implementing advanced features
 */
def receivePluginInternal: Actor.Receive = Actor.emptyBehavior
```

A snapshot store plugin can be activated with the following minimal configuration:

```
# Path to the snapshot store plugin to be used
akka.persistence.snapshot-store.plugin = "my-snapshot-store"

# My custom snapshot store plugin
my-snapshot-store {
  # Class name of the plugin.
  class = "docs.persistence.MySnapshotStore"
```

```
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
}
```

The specified plugin `class` must have a no-arg constructor. The `plugin-dispatcher` is the dispatcher used for the plugin actor. If not specified, it defaults to `akka.persistence.dispatchers.default-plugin-dispatcher`.

The snapshot store instance is an actor so the methods corresponding to requests from persistent actors are executed sequentially. It may delegate to asynchronous libraries, spawn futures, or delegate to other actors to achive parallelism.

The snapshot store plugin class must have a constructor without parameters or a constructor with one `com.typesafe.config.Config` parameter. The plugin section of the actor system's config will be passed in the config constructor parameter.

Don't run snapshot store tasks/futures on the system default dispatcher, since that might starve other tasks.

Plugin TCK

In order to help developers build correct and high quality storage plugins, we provide a Technology Compatibility Kit (TCK for short).

The TCK is usable from Java as well as Scala projects. For Scala you need to include the akka-persistence-tck dependency:

```
"com.typesafe.akka" %% "akka-persistence-tck" % "2.4.14" % "test"
```

To include the Journal TCK tests in your test suite simply extend the provided `JournalSpec`:

```
class MyJournalSpec extends JournalSpec(
  config = ConfigFactory.parseString(
    """akka.persistence.journal.plugin = "my.journal.plugin"""")) {

  override def supportsRejectingNonSerializableObjects: CapabilityFlag =
    false // or CapabilityFlag.off
}
```

Please note that some of the tests are optional, and by overriding the `supports...` methods you give the TCK the needed information about which tests to run. You can implement these methods using boolean falues or the provided `CapabilityFlag.on` / `CapabilityFlag.off` values.

We also provide a simple benchmarking class `JournalPerfSpec` which includes all the tests that `JournalSpec` has, and also performs some longer operations on the Journal while printing its performance stats. While it is NOT aimed to provide a proper benchmarking environment it can be used to get a rough feel about your journal's performance in the most typical scenarios.

In order to include the `SnapshotStore` TCK tests in your test suite simply extend the `SnapshotStoreSpec`:

```
class MySnapshotStoreSpec extends SnapshotStoreSpec(
  config = ConfigFactory.parseString(
    """
    akka.persistence.snapshot-store.plugin = "my.snapshot-store.plugin"
    """))
```

In case your plugin requires some setting up (starting a mock database, removing temporary files etc.) you can override the `beforeAll` and `afterAll` methods to hook into the tests lifecycle:

```
class MyJournalSpec extends JournalSpec(
  config = ConfigFactory.parseString(
    """
    akka.persistence.journal.plugin = "my.journal.plugin"
```

```
    """)) {

  override def supportsRejectingNonSerializableObjects: CapabilityFlag =
    true // or CapabilityFlag.on

  val storageLocations = List(
    new File(system.settings.config.getString("akka.persistence.journal.leveldb.dir
↪")),
    new File(config.getString("akka.persistence.snapshot-store.local.dir")))

  override def beforeAll() {
    super.beforeAll()
    storageLocations foreach FileUtils.deleteRecursively
  }

  override def afterAll() {
    storageLocations foreach FileUtils.deleteRecursively
    super.afterAll()
  }

}
```

We *highly recommend* including these specifications in your test suite, as they cover a broad range of cases you might have otherwise forgotten to test for when writing a plugin from scratch.

3.8.10 Pre-packaged plugins

Local LevelDB journal

The LevelDB journal plugin config entry is `akka.persistence.journal.leveldb`. It writes messages to a local LevelDB instance. Enable this plugin by defining config property:

```
# Path to the journal plugin to be used
akka.persistence.journal.plugin = "akka.persistence.journal.leveldb"
```

LevelDB based plugins will also require the following additional dependency declaration:

```
"org.iq80.leveldb"            % "leveldb"          % "0.7"
"org.fusesource.leveldbjni"   % "leveldbjni-all"   % "1.8"
```

The default location of LevelDB files is a directory named `journal` in the current working directory. This location can be changed by configuration where the specified path can be relative or absolute:

```
akka.persistence.journal.leveldb.dir = "target/journal"
```

With this plugin, each actor system runs its own private LevelDB instance.

Shared LevelDB journal

A LevelDB instance can also be shared by multiple actor systems (on the same or on different nodes). This, for example, allows persistent actors to failover to a backup node and continue using the shared journal instance from the backup node.

> **Warning:** A shared LevelDB instance is a single point of failure and should therefore only be used for testing purposes. Highly-available, replicated journals are available as Community plugins.

Note: This plugin has been supplanted by *Persistence Plugin Proxy*.

A shared LevelDB instance is started by instantiating the `SharedLeveldbStore` actor.

```
import akka.persistence.journal.leveldb.SharedLeveldbStore

val store = system.actorOf(Props[SharedLeveldbStore], "store")
```

By default, the shared instance writes journaled messages to a local directory named `journal` in the current working directory. The storage location can be changed by configuration:

```
akka.persistence.journal.leveldb-shared.store.dir = "target/shared"
```

Actor systems that use a shared LevelDB store must activate the `akka.persistence.journal.leveldb-shared` plugin.

```
akka.persistence.journal.plugin = "akka.persistence.journal.leveldb-shared"
```

This plugin must be initialized by injecting the (remote) `SharedLeveldbStore` actor reference. Injection is done by calling the `SharedLeveldbJournal.setStore` method with the actor reference as argument.

```
trait SharedStoreUsage extends Actor {
  override def preStart(): Unit = {
    context.actorSelection("akka.tcp://example@127.0.0.1:2552/user/store") !␣
↪Identify(1)
  }

  def receive = {
    case ActorIdentity(1, Some(store)) =>
      SharedLeveldbJournal.setStore(store, context.system)
  }
}
```

Internal journal commands (sent by persistent actors) are buffered until injection completes. Injection is idempotent i.e. only the first injection is used.

Local snapshot store

The local snapshot store plugin config entry is `akka.persistence.snapshot-store.local`. It writes snapshot files to the local filesystem. Enable this plugin by defining config property:

```
# Path to the snapshot store plugin to be used
akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"
```

The default storage location is a directory named `snapshots` in the current working directory. This can be changed by configuration where the specified path can be relative or absolute:

```
akka.persistence.snapshot-store.local.dir = "target/snapshots"
```

Note that it is not mandatory to specify a snapshot store plugin. If you don't use snapshots you don't have to configure it.

Persistence Plugin Proxy

A persistence plugin proxy allows sharing of journals and snapshot stores across multiple actor systems (on the same or on different nodes). This, for example, allows persistent actors to failover to a backup node and continue using the shared journal instance from the backup node. The proxy works by forwarding all the journal/snapshot

store messages to a single, shared, persistence plugin instance, and therefor supports any use case supported by the proxied plugin.

> **Warning:** A shared journal/snapshot store is a single point of failure and should therefore only be used for testing purposes. Highly-available, replicated persistence plugins are available as Community plugins.

The journal and snapshot store proxies are controlled via the `akka.persistence.journal.proxy` and `akka.persistence.snapshot-store.proxy` configuration entries, respectively. Set the `target-journal-plugin` or `target-snapshot-store-plugin` keys to the underlying plugin you wish to use (for example: `akka.persistence.journal.leveldb`). The `start-target-journal` and `start-target-snapshot-store` keys should be set to `on` in exactly one actor system - this is the system that will instantiate the shared persistence plugin. Next, the proxy needs to be told how to find the shared plugin. This can be done by setting the `target-journal-address` and `target-snapshot-store-address` configuration keys, or programmatically by calling the `PersistencePluginProxy.setTargetLocation` method.

Note: Akka starts extensions lazily when they are required, and this includes the proxy. This means that in order for the proxy to work, the persistence plugin on the target node must be instantiated. This can be done by instantiating the `PersistencePluginProxyExtension` *extension*, or by calling the `PersistencePluginProxy.start` method.

Note: The proxied persistence plugin can (and should) be configured using its original configuration keys.

3.8.11 Custom serialization

Serialization of snapshots and payloads of `Persistent` messages is configurable with Akka's *Serialization* infrastructure. For example, if an application wants to serialize

- payloads of type `MyPayload` with a custom `MyPayloadSerializer` and
- snapshots of type `MySnapshot` with a custom `MySnapshotSerializer`

it must add

```
akka.actor {
  serializers {
    my-payload = "docs.persistence.MyPayloadSerializer"
    my-snapshot = "docs.persistence.MySnapshotSerializer"
  }
  serialization-bindings {
    "docs.persistence.MyPayload" = my-payload
    "docs.persistence.MySnapshot" = my-snapshot
  }
}
```

to the application configuration. If not specified, a default serializer is used.

For more advanced schema evolution techniques refer to the *Persistence - Schema Evolution* documentation.

3.8.12 Testing

When running tests with LevelDB default settings in `sbt`, make sure to set `fork := true` in your sbt project. Otherwise, you'll see an `UnsatisfiedLinkError`. Alternatively, you can switch to a LevelDB Java port by setting

```
akka.persistence.journal.leveldb.native = off
```

or

```
akka.persistence.journal.leveldb-shared.store.native = off
```

in your Akka configuration. The LevelDB Java port is for testing purposes only.

Warning: It is not possible to test persistence provided classes (i.e. *PersistentActor* and *AtLeastOnceDelivery*) using `TestActorRef` due to its *synchronous* nature. These traits need to be able to perform asynchronous tasks in the background in order to handle internal persistence related events.

When testing Persistence based projects always rely on *asynchronous messaging using the TestKit*.

3.8.13 Configuration

There are several configuration properties for the persistence module, please refer to the *reference configuration*.

3.8.14 Multiple persistence plugin configurations

By default, a persistent actor or view will use the "default" journal and snapshot store plugins configured in the following sections of the `reference.conf` configuration resource:

```
# Absolute path to the default journal plugin configuration entry.
akka.persistence.journal.plugin = "akka.persistence.journal.inmem"
# Absolute path to the default snapshot store plugin configuration entry.
akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"
```

Note that in this case the actor or view overrides only the `persistenceId` method:

```
trait ActorWithDefaultPlugins extends PersistentActor {
  override def persistenceId = "123"
}
```

When the persistent actor or view overrides the `journalPluginId` and `snapshotPluginId` methods, the actor or view will be serviced by these specific persistence plugins instead of the defaults:

```
trait ActorWithOverridePlugins extends PersistentActor {
  override def persistenceId = "123"
  // Absolute path to the journal plugin configuration entry in the `reference.
↪conf`.
  override def journalPluginId = "akka.persistence.chronicle.journal"
  // Absolute path to the snapshot store plugin configuration entry in the␣
↪`reference.conf`.
  override def snapshotPluginId = "akka.persistence.chronicle.snapshot-store"
}
```

Note that `journalPluginId` and `snapshotPluginId` must refer to properly configured `reference.conf` plugin entries with a standard `class` property as well as settings which are specific for those plugins, i.e.:

```
# Configuration entry for the custom journal plugin, see `journalPluginId`.
akka.persistence.chronicle.journal {
  # Standard persistence extension property: provider FQCN.
  class = "akka.persistence.chronicle.ChronicleSyncJournal"
  # Custom setting specific for the journal `ChronicleSyncJournal`.
  folder = $${user.dir}/store/journal
}
```

```
# Configuration entry for the custom snapshot store plugin, see `snapshotPluginId`.
akka.persistence.chronicle.snapshot-store {
  # Standard persistence extension property: provider FQCN.
  class = "akka.persistence.chronicle.ChronicleSnapshotStore"
  # Custom setting specific for the snapshot store `ChronicleSnapshotStore`.
  folder = ${user.dir}/store/snapshot
}
```

3.9 Persistence - Schema Evolution

When working on long running projects using *Persistence*, or any kind of Event Sourcing architectures, schema evolution becomes one of the more important technical aspects of developing your application. The requirements as well as our own understanding of the business domain may (and will) change in time.

In fact, if a project matures to the point where you need to evolve its schema to adapt to changing business requirements you can view this as first signs of its success – if you wouldn't need to adapt anything over an apps lifecycle that could mean that no-one is really using it actively.

In this chapter we will investigate various schema evolution strategies and techniques from which you can pick and choose the ones that match your domain and challenge at hand.

Note: This page proposes a number of possible solutions to the schema evolution problem and explains how some of the utilities Akka provides can be used to achieve this, it is by no means a complete (closed) set of solutions.

Sometimes, based on the capabilities of your serialization formats, you may be able to evolve your schema in different ways than outlined in the sections below. If you discover useful patterns or techniques for schema evolution feel free to submit Pull Requests to this page to extend it.

3.9.1 Schema evolution in event-sourced systems

In recent years we have observed a tremendous move towards immutable append-only datastores, with event-sourcing being the prime technique successfully being used in these settings. For an excellent overview why and how immutable data makes scalability and systems design much simpler you may want to read Pat Helland's excellent Immutability Changes Everything whitepaper.

Since with Event Sourcing the **events are immutable** and usually never deleted – the way schema evolution is handled differs from how one would go about it in a mutable database setting (e.g. in typical CRUD database applications).

The system needs to be able to continue to work in the presence of "old" events which were stored under the "old" schema. We also want to limit complexity in the business logic layer, exposing a consistent view over all of the events of a given type to `PersistentActor` s and *persistence queries*. This allows the business logic layer to focus on solving business problems instead of having to explicitly deal with different schemas.

In summary, schema evolution in event sourced systems exposes the following characteristics:

- Allow the system to continue operating without large scale migrations to be applied,
- Allow the system to read "old" events from the underlying storage, however present them in a "new" view to the application logic,
- Transparently promote events to the latest versions during recovery (or queries) such that the business logic need not consider multiple versions of events

Types of schema evolution

Before we explain the various techniques that can be used to safely evolve the schema of your persistent events over time, we first need to define what the actual problem is, and what the typical styles of changes are.

Since events are never deleted, we need to have a way to be able to replay (read) old events, in such way that does not force the `PersistentActor` to be aware of all possible versions of an event that it may have persisted in the past. Instead, we want the Actors to work on some form of "latest" version of the event and provide some means of either converting old "versions" of stored events into this "latest" event type, or constantly evolve the event definition - in a backwards compatible way - such that the new deserialization code can still read old events.

The most common schema changes you will likely are:

- *adding a field to an event type*,
- *remove or rename field in event type*,
- *remove event type*,
- *split event into multiple smaller events*.

The following sections will explain some patterns which can be used to safely evolve your schema when facing those changes.

3.9.2 Picking the right serialization format

Picking the serialization format is a very important decision you will have to make while building your application. It affects which kind of evolutions are simple (or hard) to do, how much work is required to add a new datatype, and, last but not least, serialization performance.

If you find yourself realising you have picked "the wrong" serialization format, it is always possible to change the format used for storing new events, however you would have to keep the old deserialization code in order to be able to replay events that were persisted using the old serialization scheme. It is possible to "rebuild" an event-log from one serialization format to another one, however it may be a more involved process if you need to perform this on a live system.

Binary serialization formats that we have seen work well for long-lived applications include the very flexible IDL based: Google Protobuf, Apache Thrift or Apache Avro. Avro schema evolution is more "entire schema" based, instead of single fields focused like in protobuf or thrift, and usually requires using some kind of schema registry.

Users who want their data to be human-readable directly in the write-side datastore may opt to use plain-old JSON as the storage format, though that comes at a cost of lacking support for schema evolution and relatively large marshalling latency.

There are plenty excellent blog posts explaining the various trade-offs between popular serialization formats, one post we would like to highlight is the very well illustrated Schema evolution in Avro, Protocol Buffers and Thrift by Martin Kleppmann.

Provided default serializers

Akka Persistence provides Google Protocol Buffers based serializers (using *Akka Serialization*) for it's own message types such as `PersistentRepr`, `AtomicWrite` and snapshots. Journal plugin implementations *may* choose to use those provided serializers, or pick a serializer which suits the underlying database better.

Note: Serialization is **NOT** handled automatically by Akka Persistence itself. Instead, it only provides the above described serializers, and in case a `AsyncWriteJournal` plugin implementation chooses to use them directly, the above serialization scheme will be used.

Please refer to your write journal's documentation to learn more about how it handles serialization!

For example, some journals may choose to not use Akka Serialization *at all* and instead store the data in a format that is more "native" for the underlying datastore, e.g. using JSON or some other kind of format that the target datastore understands directly.

The below figure explains how the default serialization scheme works, and how it fits together with serializing the user provided message itself, which we will from here on refer to as the `payload` (highlighted in yellow):

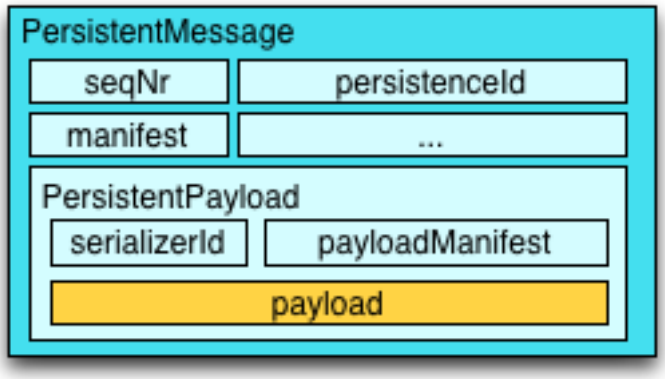


Fig. 3.1: Akka Persistence provided serializers wrap the user payload in an envelope containing all persistence-relevant information. **If the Journal uses provided Protobuf serializers for the wrapper types (e.g. PersistentRepr), then the payload will be serialized using the user configured serializer, and if none is provided explicitly, Java serialization will be used for it.**

The blue colored regions of the `PersistentMessage` indicate what is serialized using the generated protocol buffers serializers, and the yellow payload indicates the user provided event (by calling `persist(payload)(...)`). As you can see, the `PersistentMessage` acts as an envelope around the payload, adding various fields related to the origin of the event (`persistenceId`, `sequenceNr` and more).

More advanced techniques (e.g. *Remove event class and ignore events*) will dive into using the manifests for increasing the flexibility of the persisted vs. exposed types even more. However for now we will focus on the simpler evolution techniques, concerning simply configuring the payload serializers.

By default the `payload` will be serialized using Java Serialization. This is fine for testing and initial phases of your development (while you're still figuring out things and the data will not need to stay persisted forever). However, once you move to production you should really *pick a different serializer for your payloads*.

> **Warning:** Do not rely on Java serialization (which will be picked by Akka by default if you don't specify any serializers) for *serious* application development! It does not lean itself well to evolving schemas over long periods of time, and its performance is also not very high (it never was designed for high-throughput scenarios).

Configuring payload serializers

This section aims to highlight the complete basics on how to define custom serializers using *Akka Serialization*. Many journal plugin implementations use Akka Serialization, thus it is tremendously important to understand how to configure it to work with your event classes.

Note: Read the *Akka Serialization* docs to learn more about defining custom serializers, to improve performance and maintainability of your system. Do not depend on Java serialization for production deployments.

The below snippet explains in the minimal amount of lines how a custom serializer can be registered. For more in-depth explanations on how serialization picks the serializer to use etc, please refer to its documentation.

First we start by defining our domain model class, here representing a person:

```
final case class Person(name: String, surname: String)
```

Next we implement a serializer (or extend an existing one to be able to handle the new `Person` class):

```
/**
 * Simplest possible serializer, uses a string representation of the Person class.
 *
 * Usually a serializer like this would use a library like:
 * protobuf, kryo, avro, cap'n proto, flatbuffers, SBE or some other dedicated␣
↪serializer backend
 * to perform the actual to/from bytes marshalling.
 */
class SimplestPossiblePersonSerializer extends SerializerWithStringManifest {
  val Utf8 = Charset.forName("UTF-8")

  val PersonManifest = classOf[Person].getName

  // unique identifier of the serializer
  def identifier = 1234567

  // extract manifest to be stored together with serialized object
  override def manifest(o: AnyRef): String = o.getClass.getName

  // serialize the object
  override def toBinary(obj: AnyRef): Array[Byte] = obj match {
    case p: Person => s"""${p.name}|${p.surname}""".getBytes(Utf8)
    case _ => throw new IllegalArgumentException(
      s"Unable to serialize to bytes, clazz was: ${obj.getClass}!")
  }

  // deserialize the object, using the manifest to indicate which logic to apply
  override def fromBinary(bytes: Array[Byte], manifest: String): AnyRef =
    manifest match {
      case PersonManifest =>
        val nameAndSurname = new String(bytes, Utf8)
        val Array(name, surname) = nameAndSurname.split("[|]")
        Person(name, surname)
      case _ => throw new IllegalArgumentException(
        s"Unable to deserialize from bytes, manifest was: $manifest! Bytes length:
↪" +
          bytes.length)
    }

}
```

And finally we register the serializer and bind it to handle the `docs.persistence.Person` class:

```
# application.conf
akka {
  actor {
    serializers {
      person = "docs.persistence.SimplestPossiblePersonSerializer"
    }

    serialization-bindings {
      "docs.persistence.Person" = person
    }
  }
}
```

Deserialization will be performed by the same serializer which serialized the message initially because of the `identifier` being stored together with the message.

Please refer to the *Akka Serialization* documentation for more advanced use of serializers, especially the *Serializer with String Manifest* section since it is very useful for Persistence based applications dealing with schema evolutions, as we will see in some of the examples below.

3.9.3 Schema evolution in action

In this section we will discuss various schema evolution techniques using concrete examples and explaining some of the various options one might go about handling the described situation. The list below is by no means a complete guide, so feel free to adapt these techniques depending on your serializer's capabilities and/or other domain specific limitations.

Add fields

Situation: You need to add a field to an existing message type. For example, a `SeatReservation(letter:String,row:Int)` now needs to have an associated code which indicates if it is a window or aisle seat.

Solution: Adding fields is the most common change you'll need to apply to your messages so make sure the serialization format you picked for your payloads can handle it apropriately, i.e. such changes should be *binary compatible*. This is easily achieved using the right serializer toolkit – we recommend something like Google Protocol Buffers or Apache Thrift however other tools may fit your needs just as well – picking a serializer backend is something you should research before picking one to run with. In the following examples we will be using protobuf, mostly because we are familiar with it, it does its job well and Akka is using it internally as well.

While being able to read messages with missing fields is half of the solution, you also need to deal with the missing values somehow. This is usually modeled as some kind of default value, or by representing the field as an `Option[T]` See below for an example how reading an optional field from a serialized protocol buffers message might look like.

```
sealed abstract class SeatType { def code: String }
object SeatType {
  def fromString(s: String) = s match {
    case Window.code => Window
    case Aisle.code  => Aisle
    case Other.code  => Other
    case _           => Unknown
  }
  case object Window extends SeatType { override val code = "W" }
  case object Aisle extends SeatType { override val code = "A" }
  case object Other extends SeatType { override val code = "O" }
  case object Unknown extends SeatType { override val code = "" }

}

case class SeatReserved(letter: String, row: Int, seatType: SeatType)
```

Next we prepare an protocol definition using the protobuf Interface Description Language, which we'll use to generate the serializer code to be used on the Akka Serialization layer (notice that the schema aproach allows us to easily rename fields, as long as the numeric identifiers of the fields do not change):

```
// FlightAppModels.proto
option java_package = "docs.persistence.proto";
option optimize_for = SPEED;

message SeatReserved {
  required string letter   = 1;
  required uint32 row      = 2;
  optional string seatType = 3; // the new field
}
```

The serializer implementation uses the protobuf generated classes to marshall the payloads. Optional fields can be handled explicitly or missing values by calling the `has...` methods on the protobuf object, which we do for `seatType` in order to use a `Unknown` type in case the event was stored before we had introduced the field to this event type:

```
/**
 * Example serializer impl which uses protocol buffers generated classes (proto.*)
 * to perform the to/from binary marshalling.
 */
class AddedFieldsSerializerWithProtobuf extends SerializerWithStringManifest {
  override def identifier = 67876

  final val SeatReservedManifest = classOf[SeatReserved].getName

  override def manifest(o: AnyRef): String = o.getClass.getName

  override def fromBinary(bytes: Array[Byte], manifest: String): AnyRef =
    manifest match {
      case SeatReservedManifest =>
        // use generated protobuf serializer
        seatReserved(FlightAppModels.SeatReserved.parseFrom(bytes))
      case _ =>
        throw new IllegalArgumentException("Unable to handle manifest: " +␣
↪manifest)
    }

  override def toBinary(o: AnyRef): Array[Byte] = o match {
    case s: SeatReserved =>
      FlightAppModels.SeatReserved.newBuilder
        .setRow(s.row)
        .setLetter(s.letter)
        .setSeatType(s.seatType.code)
        .build().toByteArray
  }

  // -- fromBinary helpers --

  private def seatReserved(p: FlightAppModels.SeatReserved): SeatReserved =
    SeatReserved(p.getLetter, p.getRow, seatType(p))

  // handle missing field by assigning "Unknown" value
  private def seatType(p: FlightAppModels.SeatReserved): SeatType =
    if (p.hasSeatType) SeatType.fromString(p.getSeatType) else SeatType.Unknown

}
```

Rename fields

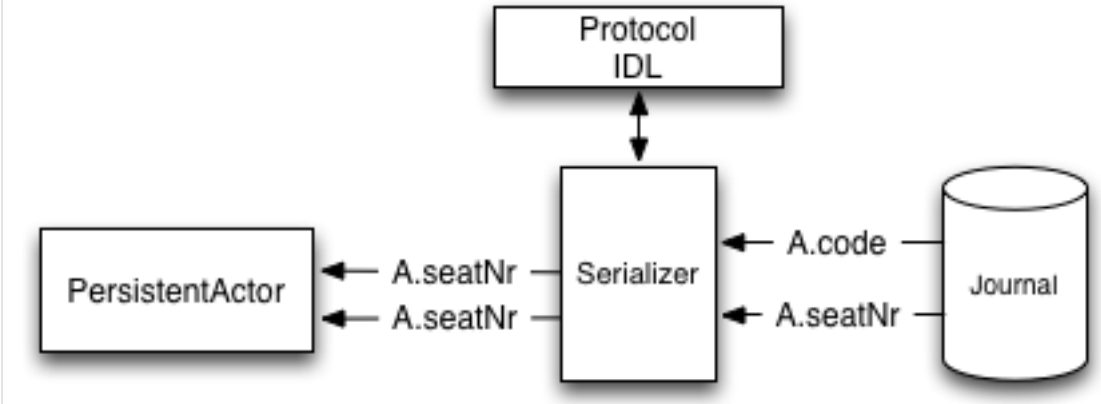
Situation: When first designing the system the `SeatReverved` event featured an `code` field. After some time you discover that what was originally called `code` actually means `seatNr`, thus the model should be changed to reflect this concept more accurately.

Solution 1 - using IDL based serializers: First, we will discuss the most efficient way of dealing with such kinds of schema changes – IDL based serializers.

IDL stands for Interface Description Language, and means that the schema of the messages that will be stored is based on this description. Most IDL based serializers also generate the serializer / deserializer code so that using them is not too hard. Examples of such serializers are protobuf or thrift.

Using these libraries rename operations are "free", because the field name is never actually stored in the binary representation of the message. This is one of the advantages of schema based serializers, even though that they add the overhead of having to maintain the schema. When using serializers like this, no additional code change (except renaming the field and method used during serialization) is needed to perform such evolution:

This is how such a rename would look in protobuf:




```
// protobuf message definition, BEFORE:
message SeatReserved {
  required string code = 1;
}

// protobuf message definition, AFTER:
message SeatReserved {
  required string seatNr = 1; // field renamed, id remains the same
}
```

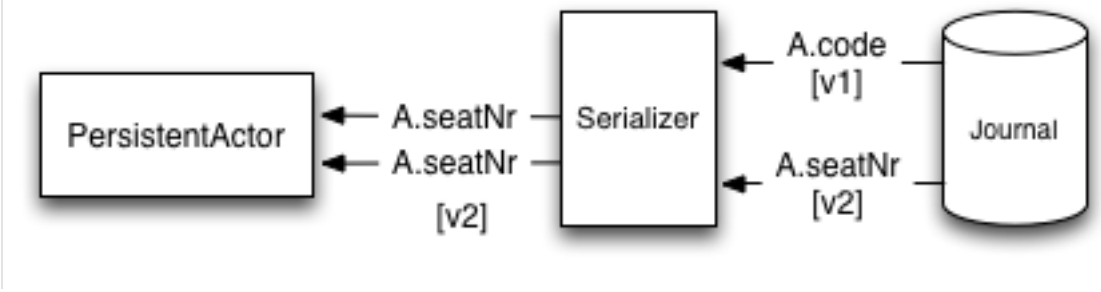
It is important to learn about the strengths and limitations of your serializers, in order to be able to move swiftly and refactor your models fearlessly as you go on with the project.

Note: Learn in-depth about the serialization engine you're using as it will impact how you can aproach schema evolution.

Some operations are "free" in certain serialization formats (more often than not: removing/adding optional fields, sometimes renaming fields etc.), while some other operations are strictly not possible.

Solution 2 - by manually handling the event versions: Another solution, in case your serialization format does not support renames as easily as the above mentioned formats, is versioning your schema. For example, you could have made your events carry an additional field called `_version` which was set to `1` (because it was the initial schema), and once you change the schema you bump this number to `2`, and write an adapter which can perform the rename.

This approach is popular when your serialization format is something like JSON, where renames can not be performed automatically by the serializer. You can do these kinds of "promotions" either manually (as shown in the example below) or using a library like Stamina which helps to create those `V1->V2->V3->...->Vn` promotion chains without much boilerplate.



The following snippet showcases how one could apply renames if working with plain JSON (using `spray.json.JsObject`):

```
class JsonRenamedFieldAdapter extends EventAdapter {
  val marshaller = new ExampleJsonMarshaller

  val V1 = "v1"
  val V2 = "v2"

  // this could be done independently for each event type
  override def manifest(event: Any): String = V2

  override def toJournal(event: Any): JsObject =
    marshaller.toJson(event)

  override def fromJournal(event: Any, manifest: String): EventSeq = event match {
    case json: JsObject => EventSeq(marshaller.fromJson(manifest match {
      case V1      => rename(json, "code", "seatNr")
      case V2      => json // pass-through
      case unknown => throw new IllegalArgumentException(s"Unknown manifest:
↪$unknown")
    }))
    case _ =>
      val c = event.getClass
      throw new IllegalArgumentException("Can only work with JSON, was: %s".
↪format(c))
  }

  def rename(json: JsObject, from: String, to: String): JsObject = {
    val value = json.fields(from)
    val withoutOld = json.fields - from
    JsObject(withoutOld + (to -> value))
  }

}
```

As you can see, manually handling renames induces some boilerplate onto the EventAdapter, however much of it you will find is common infrastructure code that can be either provided by an external library (for promotion management) or put together in a simple helper trait.

Note: The technique of versioning events and then promoting them to the latest version using JSON transformations can of course be applied to more than just field renames – it also applies to adding fields and all kinds of changes in the message format.

Remove event class and ignore events

Situation: While investigating app performance you notice that insane amounts of `CustomerBlinked` events are being stored for every customer each time he/she blinks. Upon investigation you decide that the event does not add any value and should be deleted. You still have to be able to replay from a journal which contains those old CustomerBlinked events though.

Naive solution - drop events in EventAdapter:

The problem of removing an event type from the domain model is not as much its removal, as the implications for the recovery mechanisms that this entails. For example, a naive way of filtering out certain kinds of events from being delivered to a recovering `PersistentActor` is pretty simple, as one can simply filter them out in an *EventAdapter*:

This however does not address the underlying cost of having to deserialize all the events during recovery, even those which will be filtered out by the adapter. In the next section we will improve the above explained mechanism

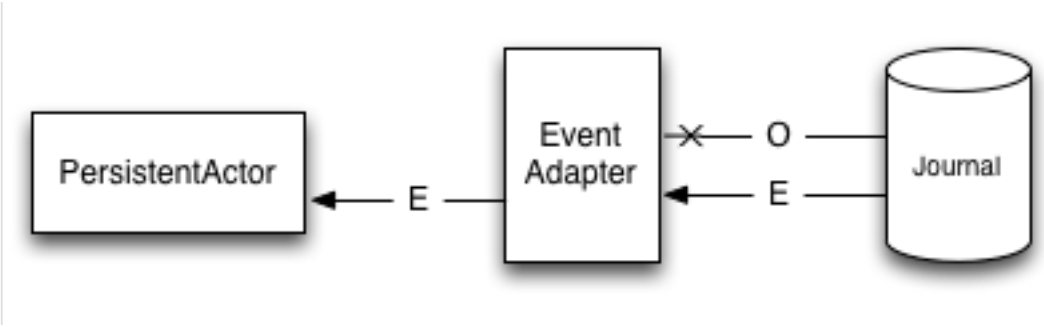


Fig. 3.2: The `EventAdapter` can drop old events (**O**) by emitting an empty `EventSeq`. Other events can simply be passed through (**E**).

to avoid deserializing events which would be filtered out by the adapter anyway, thus allowing to save precious time during a recovery containing lots of such events (without actually having to delete them).

Improved solution - deserialize into tombstone:

In the just described technique we have saved the PersistentActor from receiving un-wanted events by filtering them out in the `EventAdapter`, however the event itself still was deserialized and loaded into memory. This has two notable *downsides*:

- first, that the deserialization was actually performed, so we spent some of out time budget on the deserialization, even though the event does not contribute anything to the persistent actors state.
- second, that we are *unable to remove the event class* from the system – since the serializer still needs to create the actuall instance of it, as it does not know it will not be used.

The solution to these problems is to use a serializer that is aware of that event being no longer needed, and can notice this before starting to deserialize the object.

This aproach allows us to *remove the original class from our classpath*, which makes for less "old" classes lying around in the project. This can for example be implemented by using an `SerializerWithStringManifest` (documented in depth in *Serializer with String Manifest*). By looking at the string manifest, the serializer can notice that the type is no longer needed, and skip the deserialization all-together:

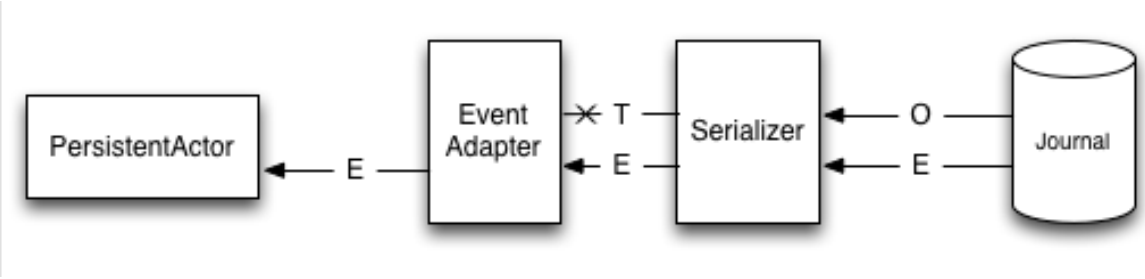


Fig. 3.3: The serializer is aware of the old event types that need to be skipped (**O**), and can skip deserializing them alltogether by simply returning a "tombstone" (**T**), which the EventAdapter converts into an empty EventSeq. Other events (**E**) can simply be passed through.

The serializer detects that the string manifest points to a removed event type and skips attempting to deserialize it:

```
case object EventDeserializationSkipped

class RemovedEventsAwareSerializer extends SerializerWithStringManifest {
  val utf8 = Charset.forName("UTF-8")
  override def identifier: Int = 8337

  val SkipEventManifestsEvents = Set(
    "docs.persistence.CustomerBlinked" // ...
```

```
  )

  override def manifest(o: AnyRef): String = o.getClass.getName

  override def toBinary(o: AnyRef): Array[Byte] = o match {
    case _ => o.toString.getBytes(utf8) // example serialization
  }

  override def fromBinary(bytes: Array[Byte], manifest: String): AnyRef =
    manifest match {
      case m if SkipEventManifestsEvents.contains(m) =>
        EventDeserializationSkipped

      case other => new String(bytes, utf8)
    }
}
```

The EventAdapter we implemented is aware of `EventDeserializationSkipped` events (our "Tombstones"), and emits and empty `EventSeq` whenever such object is encoutered:

```
class SkippedEventsAwareAdapter extends EventAdapter {
  override def manifest(event: Any) = ""
  override def toJournal(event: Any) = event

  override def fromJournal(event: Any, manifest: String) = event match {
    case EventDeserializationSkipped => EventSeq.empty
    case _                           => EventSeq(event)
  }
}
```

Detach domain model from data model

Situation: You want to separate the application model (often called the "*domain model*") completely from the models used to persist the corresponding events (the "*data model*"). For example because the data representation may change independently of the domain model.

Another situation where this technique may be useful is when your serialization tool of choice requires generated classes to be used for serialization and deserialization of objects, like for example Google Protocol Buffers do, yet you do not want to leak this implementation detail into the domain model itself, which you'd like to model as plain Scala case classes.

Solution: In order to detach the domain model, which is often represented using pure scala (case) classes, from the data model classes which very often may be less user-friendly yet highly optimised for throughput and schema evolution (like the classes generated by protobuf for example), it is possible to use a simple EventAdapter which maps between these types in a 1:1 style as illustrated below:

We will use the following domain and data models to showcase how the separation can be implemented by the adapter:

```
/** Domain model - highly optimised for domain language and maybe "fluent" usage */
object DomainModel {
  final case class Customer(name: String)
  final case class Seat(code: String) {
    def bookFor(customer: Customer): SeatBooked = SeatBooked(code, customer)
  }

  final case class SeatBooked(code: String, customer: Customer)
}

/** Data model - highly optimised for schema evolution and persistence */
object DataModel {
```

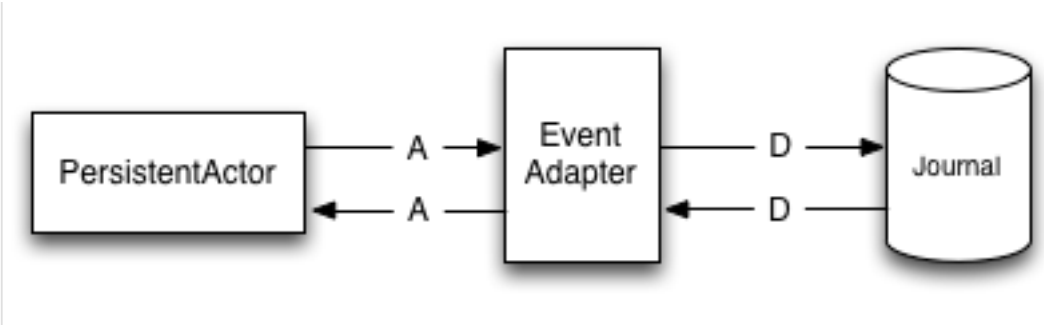


Fig. 3.4: Domain events (**A**) are adapted to the data model events (**D**) by the `EventAdapter`. The data model can be a format natively understood by the journal, such that it can store it more efficiently or include additional data for the event (e.g. tags), for ease of later querying.

```
  final case class SeatBooked(code: String, customerName: String)
}
```

The `EventAdapter` takes care of converting from one model to the other one (in both directions), alowing the models to be completely detached from each other, such that they can be optimised independently as long as the mapping logic is able to convert between them:

```
class DetachedModelsAdapter extends EventAdapter {
  override def manifest(event: Any): String = ""

  override def toJournal(event: Any): Any = event match {
    case DomainModel.SeatBooked(code, customer) =>
      DataModel.SeatBooked(code, customer.name)
  }
  override def fromJournal(event: Any, manifest: String): EventSeq = event match {
    case DataModel.SeatBooked(code, customerName) =>
      EventSeq(DomainModel.SeatBooked(code, DomainModel.Customer(customerName)))
  }
}
```

The same technique could also be used directly in the Serializer if the end result of marshalling is bytes. Then the serializer can simply convert the bytes do the domain object by using the generated protobuf builders.

Store events as human-readable data model

Situation: You want to keep your persisted events in a human-readable format, for example JSON.

Solution: This is a special case of the *Detach domain model from data model* pattern, and thus requires some co-operation from the Journal implementation to achieve this.

An example of a Journal which may implement this pattern is MongoDB, however other databases such as PostgreSQL and Cassandra could also do it because of their built-in JSON capabilities.

In this aproach, the `EventAdapter` is used as the marshalling layer: it serializes the events to/from JSON. The journal plugin notices that the incoming event type is JSON (for example by performing a `match` on the incoming event) and stores the incoming object directly.

```
class JsonDataModelAdapter extends EventAdapter {
  override def manifest(event: Any): String = ""

  val marshaller = new ExampleJsonMarshaller

  override def toJournal(event: Any): JsObject =
    marshaller.toJson(event)
```

```
  override def fromJournal(event: Any, manifest: String): EventSeq = event match {
    case json: JsObject =>
      EventSeq(marshaller.fromJson(json))
    case _ =>
      throw new IllegalArgumentException(
        "Unable to fromJournal a non-JSON object! Was: " + event.getClass)
  }
}
```

Note: This technique only applies if the Akka Persistence plugin you are using provides this capability. Check the documentation of your favourite plugin to see if it supports this style of persistence.

If it doesn't, you may want to skim the list of existing journal plugins, just in case some other plugin for your favourite datastore *does* provide this capability.

Alternative solution:

In fact, an AsyncWriteJournal implementation could natively decide to not use binary serialization at all, and *always* serialize the incoming messages as JSON - in which case the `toJournal` implementation of the `EventAdapter` would be an identity function, and the `fromJournal` would need to de-serialize messages from JSON.

Note: If in need of human-readable events on the *write-side* of your application reconsider whether preparing materialized views using *Persistence Query* would not be an efficient way to go about this, without compromising the write-side's throughput characteristics.

If indeed you want to use a human-readable representation on the write-side, pick a Persistence plugin that provides that functionality, or – implement one yourself.

Split large event into fine-grained events

Situation: While refactoring your domain events, you find that one of the events has become too large (coarse-grained) and needs to be split up into multiple fine-grained events.

Solution: Let us consider a situation where an event represents "user details changed". After some time we discover that this event is too coarse, and needs to be split into "user name changed" and "user address changed", because somehow users keep changing their usernames a lot and we'd like to keep this as a separate event.

The write side change is very simple, we simply persist `UserNameChanged` or `UserAddressChanged` depending on what the user actually intended to change (instead of the composite `UserDetailsChanged` that we had in version 1 of our model).

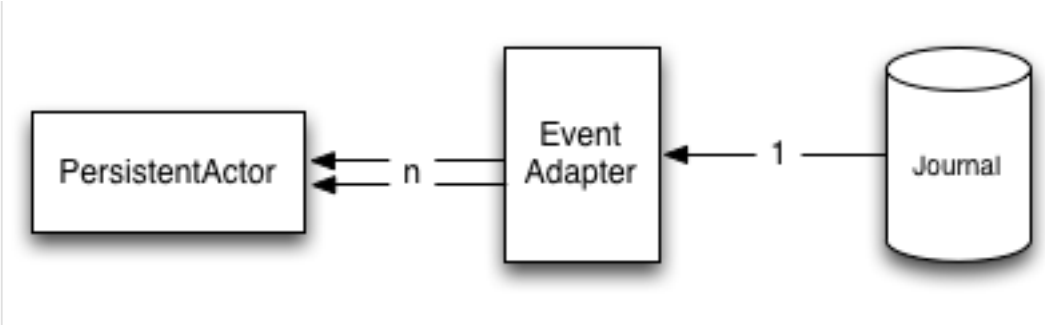


Fig. 3.5: The `EventAdapter` splits the incoming event into smaller more fine grained events during recovery.

During recovery however, we now need to convert the old `V1` model into the `V2` representation of the change.

Depending if the old event contains a name change, we either emit the `UserNameChanged` or we don't, and the address change is handled similarily:

```
trait V1
trait V2

// V1 event:
final case class UserDetailsChanged(name: String, address: String) extends V1

// corresponding V2 events:
final case class UserNameChanged(name: String) extends V2
final case class UserAddressChanged(address: String) extends V2

// event splitting adapter:
class UserEventsAdapter extends EventAdapter {
  override def manifest(event: Any): String = ""

  override def fromJournal(event: Any, manifest: String): EventSeq = event match {
    case UserDetailsChanged(null, address) => EventSeq(UserAddressChanged(address))
    case UserDetailsChanged(name, null)    => EventSeq(UserNameChanged(name))
    case UserDetailsChanged(name, address) =>
      EventSeq(
        UserNameChanged(name),
        UserAddressChanged(address))
    case event: V2 => EventSeq(event)
  }

  override def toJournal(event: Any): Any = event
}
```

By returning an `EventSeq` from the event adapter, the recovered event can be converted to multiple events before being delivered to the persistent actor.

3.10 Persistence Query

Akka persistence query complements *Persistence* by providing a universal asynchronous stream based query interface that various journal plugins can implement in order to expose their query capabilities.

The most typical use case of persistence query is implementing the so-called query side (also known as "read side") in the popular CQRS architecture pattern - in which the writing side of the application (e.g. implemented using akka persistence) is completely separated from the "query side". Akka Persistence Query itself is *not* directly the query side of an application, however it can help to migrate data from the write side to the query side database. In very simple scenarios Persistence Query may be powerful enough to fulfill the query needs of your app, however we highly recommend (in the spirit of CQRS) of splitting up the write/read sides into separate datastores as the need arises.

Warning: This module is marked as **"experimental"** as of its introduction in Akka 2.4.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply to the contents of the `akka.persistence.query` package.

3.10.1 Dependencies

Akka persistence query is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-persistence-query-experimental" % "2.4.14"
```

3.10.2 Design overview

Akka persistence query is purposely designed to be a very loosely specified API. This is in order to keep the provided APIs general enough for each journal implementation to be able to expose its best features, e.g. a SQL journal can use complex SQL queries or if a journal is able to subscribe to a live event stream this should also be possible to expose the same API - a typed stream of events.

Each read journal must explicitly document which types of queries it supports. Refer to your journal's plugins documentation for details on which queries and semantics it supports.

While Akka Persistence Query does not provide actual implementations of ReadJournals, it defines a number of pre-defined query types for the most common query scenarios, that most journals are likely to implement (however they are not required to).

3.10.3 Read Journals

In order to issue queries one has to first obtain an instance of a `ReadJournal`. Read journals are implemented as Community plugins, each targeting a specific datastore (for example Cassandra or JDBC databases). For example, given a library that provides a `akka.persistence.query.my-read-journal` obtaining the related journal is as simple as:

```
// obtain read journal by plugin id
val readJournal =
  PersistenceQuery(system).readJournalFor[MyScaladslReadJournal](
    "akka.persistence.query.my-read-journal")

// issue query to journal
val source: Source[EventEnvelope, NotUsed] =
  readJournal.eventsByPersistenceId("user-1337", 0, Long.MaxValue)

// materialize stream, consuming events
implicit val mat = ActorMaterializer()
source.runForeach { event => println("Event: " + event) }
```

Journal implementers are encouraged to put this identifier in a variable known to the user, such that one can access it via `readJournalFor[NoopJournal](NoopJournal.identifier)`, however this is not enforced.

Read journal implementations are available as Community plugins.

Predefined queries

Akka persistence query comes with a number of query interfaces built in and suggests Journal implementors to implement them according to the semantics described below. It is important to notice that while these query types are very common a journal is not obliged to implement all of them - for example because in a given journal such query would be significantly inefficient.

Note: Refer to the documentation of the `ReadJournal` plugin you are using for a specific list of supported query types. For example, Journal plugins should document their stream completion strategies.

The predefined queries are:

AllPersistenceIdsQuery and CurrentPersistenceIdsQuery

`allPersistenceIds` which is designed to allow users to subscribe to a stream of all persistent ids in the system. By default this stream should be assumed to be a "live" stream, which means that the journal should keep emitting new persistence ids as they come into the system:

```
readJournal.allPersistenceIds()
```

If your usage does not require a live stream, you can use the `currentPersistenceIds` query:

```
readJournal.currentPersistenceIds()
```

EventsByPersistenceIdQuery and CurrentEventsByPersistenceIdQuery

`eventsByPersistenceId` is a query equivalent to replaying a *PersistentActor*, however, since it is a stream it is possible to keep it alive and watch for additional incoming events persisted by the persistent actor identified by the given `persistenceId`.

```
readJournal.eventsByPersistenceId("user-us-1337")
```

Most journals will have to revert to polling in order to achieve this, which can typically be configured with a `refresh-interval` configuration property.

If your usage does not require a live stream, you can use the `currentEventsByPersistenceId` query.

EventsByTag and CurrentEventsByTag

`eventsByTag` allows querying events regardless of which `persistenceId` they are associated with. This query is hard to implement in some journals or may need some additional preparation of the used data store to be executed efficiently. The goal of this query is to allow querying for all events which are "tagged" with a specific tag. That includes the use case to query all domain events of an Aggregate Root type. Please refer to your read journal plugin's documentation to find out if and how it is supported.

Some journals may support tagging of events via an *Event Adapters* that wraps the events in a `akka.persistence.journal.Tagged` with the given `tags`. The journal may support other ways of doing tagging - again, how exactly this is implemented depends on the used journal. Here is an example of such a tagging event adapter:

```
import akka.persistence.journal.WriteEventAdapter
import akka.persistence.journal.Tagged

class MyTaggingEventAdapter extends WriteEventAdapter {
  val colors = Set("green", "black", "blue")
  override def toJournal(event: Any): Any = event match {
    case s: String =>
      var tags = colors.foldLeft(Set.empty[String]) { (acc, c) =>
        if (s.contains(c)) acc + c else acc
      }
      if (tags.isEmpty) event
      else Tagged(event, tags)
    case _ => event
  }

  override def manifest(event: Any): String = ""
}
```

Note: A very important thing to keep in mind when using queries spanning multiple persistenceIds, such as `EventsByTag` is that the order of events at which the events appear in the stream rarely is guaranteed (or stable between materializations).

Journals *may* choose to opt for strict ordering of the events, and should then document explicitly what kind of ordering guarantee they provide - for example "*ordered by timestamp ascending, independently of persistenceId*" is easy to achieve on relational databases, yet may be hard to implement efficiently on plain key-value datastores.

In the example below we query all events which have been tagged (we assume this was performed by the write-side using an *EventAdapter*, or that the journal is smart enough that it can figure out what we mean by this tag - for example if the journal stored the events as json it may try to find those with the field `tag` set to this value etc.).

```
// assuming journal is able to work with numeric offsets we can:

val blueThings: Source[EventEnvelope2, NotUsed] =
  readJournal.eventsByTag("blue")

// find top 10 blue things:
val top10BlueThings: Future[Vector[Any]] =
  blueThings
    .map(_.event)
    .take(10) // cancels the query stream after pulling 10 elements
    .runFold(Vector.empty[Any])(_ :+ _)

// start another query, from the known offset
val furtherBlueThings = readJournal.eventsByTag("blue", offset = Sequence(10))
```

As you can see, we can use all the usual stream combinators available from Akka Streams on the resulting query stream, including for example taking the first 10 and cancelling the stream. It is worth pointing out that the built-in `EventsByTag` query has an optionally supported offset parameter (of type `Long`) which the journals can use to implement resumable-streams. For example a journal may be able to use a WHERE clause to begin the read starting from a specific row, or in a datastore that is able to order events by insertion time it could treat the Long as a timestamp and select only older events.

If your usage does not require a live stream, you can use the `currentEventsByTag` query.

Materialized values of queries

Journals are able to provide additional information related to a query by exposing materialized values, which are a feature of Akka Streams that allows to expose additional values at stream materialization time.

More advanced query journals may use this technique to expose information about the character of the materialized stream, for example if it's finite or infinite, strictly ordered or not ordered at all. The materialized value type is defined as the second type parameter of the returned `Source`, which allows journals to provide users with their specialised query object, as demonstrated in the sample below:

```
final case class RichEvent(tags: Set[String], payload: Any)

// a plugin can provide:
case class QueryMetadata(deterministicOrder: Boolean, infinite: Boolean)
```

```
def byTagsWithMeta(tags: Set[String]): Source[RichEvent, QueryMetadata] = {
```

```
val query: Source[RichEvent, QueryMetadata] =
  readJournal.byTagsWithMeta(Set("red", "blue"))

query
  .mapMaterializedValue { meta =>
    println(s"The query is: " +
      s"ordered deterministically: ${meta.deterministicOrder}, " +
      s"infinite: ${meta.infinite}")
  }
  .map { event => println(s"Event payload: ${event.payload}") }
  .runWith(Sink.ignore)
```

3.10.4 Performance and denormalization

When building systems using *Event sourcing* and CQRS (Command & Query Responsibility Segregation) techniques it is tremendously important to realise that the write-side has completely different needs from the read-side, and separating those concerns into datastores that are optimised for either side makes it possible to offer the best experience for the write and read sides independently.

For example, in a bidding system it is important to "take the write" and respond to the bidder that we have accepted the bid as soon as possible, which means that write-throughput is of highest importance for the write-side – often this means that data stores which are able to scale to accommodate these requirements have a less expressive query side.

On the other hand the same application may have some complex statistics view or we may have analysts working with the data to figure out best bidding strategies and trends – this often requires some kind of expressive query capabilities like for example SQL or writing Spark jobs to analyse the data. Therefore the data stored in the write-side needs to be projected into the other read-optimised datastore.

Note: When referring to **Materialized Views** in Akka Persistence think of it as "some persistent storage of the result of a Query". In other words, it means that the view is created once, in order to be afterwards queried multiple times, as in this format it may be more efficient or interesting to query it (instead of the source events directly).

Materialize view to Reactive Streams compatible datastore

If the read datastore exposes a Reactive Streams interface then implementing a simple projection is as simple as, using the read-journal and feeding it into the databases driver interface, for example like so:

```
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val readJournal =
  PersistenceQuery(system).readJournalFor[MyScaladslReadJournal](JournalId)
val dbBatchWriter: Subscriber[immutable.Seq[Any]] =
  ReactiveStreamsCompatibleDBDriver.batchWriter

// Using an example (Reactive Streams) Database driver
readJournal
  .eventsByPersistenceId("user-1337")
  .map(envelope => envelope.event)
  .map(convertToReadSideTypes) // convert to datatype
  .grouped(20) // batch inserts into groups of 20
  .runWith(Sink.fromSubscriber(dbBatchWriter)) // write batches to read-side␣
↪database
```

Materialize view using mapAsync

If the target database does not provide a reactive streams `Subscriber` that can perform writes, you may have to implement the write logic using plain functions or Actors instead.

In case your write logic is state-less and you just need to convert the events from one data type to another before writing into the alternative datastore, then the projection is as simple as:

```
trait ExampleStore {
  def save(event: Any): Future[Unit]
}
val store: ExampleStore = ???

readJournal
  .eventsByTag("bid")
```

```
  .mapAsync(1) { e => store.save(e) }
  .runWith(Sink.ignore)
```

Resumable projections

Sometimes you may need to implement "resumable" projections, that will not start from the beginning of time each time when run. In this case you will need to store the sequence number (or `offset`) of the processed event and use it the next time this projection is started. This pattern is not built-in, however is rather simple to implement yourself.

The example below additionally highlights how you would use Actors to implement the write side, in case you need to do some complex logic that would be best handled inside an Actor before persisting the event into the other datastore:

```
import akka.pattern.ask
import system.dispatcher
implicit val timeout = Timeout(3.seconds)

val bidProjection = new MyResumableProjection("bid")

val writerProps = Props(classOf[TheOneWhoWritesToQueryJournal], "bid")
val writer = system.actorOf(writerProps, "bid-projection-writer")

bidProjection.latestOffset.foreach { startFromOffset =>
  readJournal
    .eventsByTag("bid", Sequence(startFromOffset))
    .mapAsync(8) { envelope => (writer ? envelope.event).map(_ => envelope.offset)␣
↪}
    .mapAsync(1) { offset => bidProjection.saveProgress(offset) }
    .runWith(Sink.ignore)
}
```

```
class TheOneWhoWritesToQueryJournal(id: String) extends Actor {
  val store = new DummyStore()

  var state: ComplexState = ComplexState()

  def receive = {
    case m =>
      state = updateState(state, m)
      if (state.readyToSave) store.save(Record(state))
  }

  def updateState(state: ComplexState, msg: Any): ComplexState = {
    // some complicated aggregation logic here ...
    state
  }
}
```

3.10.5 Query plugins

Query plugins are various (mostly community driven) `ReadJournal` implementations for all kinds of available datastores. The complete list of available plugins is maintained on the Akka Persistence Query Community Plugins page.

The plugin for LevelDB is described in *Persistence Query for LevelDB*.

This section aims to provide tips and guide plugin developers through implementing a custom query plugin. Most users will not need to implement journals themselves, except if targeting a not yet supported datastore.

Note: Since different data stores provide different query capabilities journal plugins **must extensively document** their exposed semantics as well as handled query scenarios.

ReadJournal plugin API

A read journal plugin must implement `akka.persistence.query.ReadJournalProvider` which creates instances of `akka.persistence.query.scaladsl.ReadJournal` and `akka.persistence.query.javaadsl.ReadJournal`. The plugin must implement both the `scaladsl` and the `javadsl` traits because the `akka.stream.scaladsl.Source` and `akka.stream.javadsl.Source` are different types and even though those types can easily be converted to each other it is most convenient for the end user to get access to the Java or Scala directly. As illustrated below one of the implementations can delegate to the other.

Below is a simple journal implementation:

```
class MyReadJournalProvider(system: ExtendedActorSystem, config: Config)
  extends ReadJournalProvider {

  override val scaladslReadJournal: MyScaladslReadJournal =
    new MyScaladslReadJournal(system, config)

  override val javadslReadJournal: MyJavadslReadJournal =
    new MyJavadslReadJournal(scaladslReadJournal)
}

class MyScaladslReadJournal(system: ExtendedActorSystem, config: Config)
  extends akka.persistence.query.scaladsl.ReadJournal
  with akka.persistence.query.scaladsl.EventsByTagQuery2
  with akka.persistence.query.scaladsl.EventsByPersistenceIdQuery
  with akka.persistence.query.scaladsl.AllPersistenceIdsQuery
  with akka.persistence.query.scaladsl.CurrentPersistenceIdsQuery {

  private val refreshInterval: FiniteDuration =
    config.getDuration("refresh-interval", MILLISECONDS).millis

  override def eventsByTag(
    tag: String, offset: Offset = Sequence(0L)): Source[EventEnvelope2, NotUsed] =␣
↪offset match {
    case Sequence(offsetValue) ⇒
      val props = MyEventsByTagPublisher.props(tag, offsetValue, refreshInterval)
      Source.actorPublisher[EventEnvelope](props)
        .mapMaterializedValue(_ => NotUsed)
        .map {
          case EventEnvelope(offset, id, seqNr, event) =>
            EventEnvelope2(Sequence(offset), id, seqNr, event)
        }
    case _ ⇒
      throw new IllegalArgumentException("LevelDB does not support " + offset.
↪getClass.getName + " offsets")
  }

  override def eventsByPersistenceId(
    persistenceId: String, fromSequenceNr: Long = 0L,
    toSequenceNr: Long = Long.MaxValue): Source[EventEnvelope, NotUsed] = {
    // implement in a similar way as eventsByTag
    ???
  }

  override def allPersistenceIds(): Source[String, NotUsed] = {
    // implement in a similar way as eventsByTag
```

```
    ???
  }

  override def currentPersistenceIds(): Source[String, NotUsed] = {
    // implement in a similar way as eventsByTag
    ???
  }

  // possibility to add more plugin specific queries

  def byTagsWithMeta(tags: Set[String]): Source[RichEvent, QueryMetadata] = {
    // implement in a similar way as eventsByTag
    ???
  }

}

class MyJavadslReadJournal(scaladslReadJournal: MyScaladslReadJournal)
  extends akka.persistence.query.javadsl.ReadJournal
  with akka.persistence.query.javadsl.EventsByTagQuery2
  with akka.persistence.query.javadsl.EventsByPersistenceIdQuery
  with akka.persistence.query.javadsl.AllPersistenceIdsQuery
  with akka.persistence.query.javadsl.CurrentPersistenceIdsQuery {

  override def eventsByTag(
    tag: String, offset: Offset = Sequence(0L)): javadsl.Source[EventEnvelope2,␣
↪NotUsed] =
    scaladslReadJournal.eventsByTag(tag, offset).asJava

  override def eventsByPersistenceId(
    persistenceId: String, fromSequenceNr: Long = 0L,
    toSequenceNr: Long = Long.MaxValue): javadsl.Source[EventEnvelope, NotUsed] =
    scaladslReadJournal.eventsByPersistenceId(
      persistenceId, fromSequenceNr, toSequenceNr).asJava

  override def allPersistenceIds(): javadsl.Source[String, NotUsed] =
    scaladslReadJournal.allPersistenceIds().asJava

  override def currentPersistenceIds(): javadsl.Source[String, NotUsed] =
    scaladslReadJournal.currentPersistenceIds().asJava

  // possibility to add more plugin specific queries

  def byTagsWithMeta(
    tags: java.util.Set[String]): javadsl.Source[RichEvent, QueryMetadata] = {
    import scala.collection.JavaConverters._
    scaladslReadJournal.byTagsWithMeta(tags.asScala.toSet).asJava
  }
}
```

And the `eventsByTag` could be backed by such an Actor for example:

```
class MyEventsByTagPublisher(tag: String, offset: Long, refreshInterval:␣
↪FiniteDuration)
  extends ActorPublisher[EventEnvelope2] {

  private case object Continue

  private val connection: java.sql.Connection = ???

  private val Limit = 1000
  private var currentOffset = offset
  var buf = Vector.empty[EventEnvelope2]
```

```
  import context.dispatcher
  val continueTask = context.system.scheduler.schedule(
    refreshInterval, refreshInterval, self, Continue)

  override def postStop(): Unit = {
    continueTask.cancel()
  }

  def receive = {
    case _: Request | Continue =>
      query()
      deliverBuf()

    case Cancel =>
      context.stop(self)
  }

  object Select {
    private def statement() = connection.prepareStatement(
      """
        SELECT id, persistent_repr FROM journal
        WHERE tag = ? AND id >= ?
        ORDER BY id LIMIT ?
      """)

    def run(tag: String, from: Long, limit: Int): Vector[(Long, Array[Byte])] = {
      val s = statement()
      try {
        s.setString(1, tag)
        s.setLong(2, from)
        s.setLong(3, limit)
        val rs = s.executeQuery()

        val b = Vector.newBuilder[(Long, Array[Byte])]
        while (rs.next())
          b += (rs.getLong(1) -> rs.getBytes(2))
        b.result()
      } finally s.close()
    }
  }

  def query(): Unit =
    if (buf.isEmpty) {
      try {
        val result = Select.run(tag, currentOffset, Limit)
        currentOffset = if (result.nonEmpty) result.last._1 else currentOffset
        val serialization = SerializationExtension(context.system)

        buf = result.map {
          case (id, bytes) =>
            val p = serialization.deserialize(bytes, classOf[PersistentRepr]).get
            EventEnvelope2(offset = Sequence(id), p.persistenceId, p.sequenceNr, p.
↪payload)
        }
      } catch {
        case e: Exception =>
          onErrorThenStop(e)
      }
    }

  final def deliverBuf(): Unit =
    if (totalDemand > 0 && buf.nonEmpty) {
```

```
      if (totalDemand <= Int.MaxValue) {
        val (use, keep) = buf.splitAt(totalDemand.toInt)
        buf = keep
        use foreach onNext
      } else {
        buf foreach onNext
        buf = Vector.empty
      }
    }
}
```

If the underlying datastore only supports queries that are completed when they reach the end of the "result set", the journal has to submit new queries after a while in order to support "infinite" event streams that include events stored after the initial query has completed. It is recommended that the plugin use a configuration property named `refresh-interval` for defining such a refresh interval.

Plugin TCK

TODO, not available yet.

3.11 Persistence Query for LevelDB

This is documentation for the LevelDB implementation of the *Persistence Query* API. Note that implementations for other journals may have different semantics.

> **Warning:** This module is marked as **"experimental"** as of its introduction in Akka 2.4.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply to the contents of the `akka.persistence.query` package.

3.11.1 Dependencies

Akka persistence LevelDB query implementation is bundled in the `akka-persistence-query-experimental` artifact. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-persistence-query-experimental" % "2.4.14"
```

3.11.2 How to get the ReadJournal

The `ReadJournal` is retrieved via the `akka.persistence.query.PersistenceQuery` extension:

```
import akka.persistence.query.PersistenceQuery
import akka.persistence.query.journal.leveldb.scaladsl.LeveldbReadJournal

val queries = PersistenceQuery(system).readJournalFor[LeveldbReadJournal](
  LeveldbReadJournal.Identifier)
```

3.11.3 Supported Queries

EventsByPersistenceIdQuery and CurrentEventsByPersistenceIdQuery

`eventsByPersistenceId` is used for retrieving events for a specific `PersistentActor` identified by `persistenceId`.

```
implicit val mat = ActorMaterializer()(system)
val queries = PersistenceQuery(system).readJournalFor[LeveldbReadJournal](
  LeveldbReadJournal.Identifier)

val src: Source[EventEnvelope, NotUsed] =
  queries.eventsByPersistenceId("some-persistence-id", 0L, Long.MaxValue)

val events: Source[Any, NotUsed] = src.map(_.event)
```

You can retrieve a subset of all events by specifying `fromSequenceNr` and `toSequenceNr` or use `0L` and `Long.MaxValue` respectively to retrieve all events. Note that the corresponding sequence number of each event is provided in the `EventEnvelope`, which makes it possible to resume the stream at a later point from a given sequence number.

The returned event stream is ordered by sequence number, i.e. the same order as the `PersistentActor` persisted the events. The same prefix of stream elements (in same order) are returned for multiple executions of the query, except for when events have been deleted.

The stream is not completed when it reaches the end of the currently stored events, but it continues to push new events when new events are persisted. Corresponding query that is completed when it reaches the end of the currently stored events is provided by `currentEventsByPersistenceId`.

The LevelDB write journal is notifying the query side as soon as events are persisted, but for efficiency reasons the query side retrieves the events in batches that sometimes can be delayed up to the configured `refresh-interval` or given `RefreshInterval` hint.

The stream is completed with failure if there is a failure in executing the query in the backend journal.

AllPersistenceIdsQuery and CurrentPersistenceIdsQuery

`allPersistenceIds` is used for retrieving all `persistenceIds` of all persistent actors.

```
implicit val mat = ActorMaterializer()(system)
val queries = PersistenceQuery(system).readJournalFor[LeveldbReadJournal](
  LeveldbReadJournal.Identifier)

val src: Source[String, NotUsed] = queries.allPersistenceIds()
```

The returned event stream is unordered and you can expect different order for multiple executions of the query.

The stream is not completed when it reaches the end of the currently used *persistenceIds*, but it continues to push new *persistenceIds* when new persistent actors are created. Corresponding query that is completed when it reaches the end of the currently used *persistenceIds* is provided by `currentPersistenceIds`.

The LevelDB write journal is notifying the query side as soon as new `persistenceIds` are created and there is no periodic polling or batching involved in this query.

The stream is completed with failure if there is a failure in executing the query in the backend journal.

EventsByTag and CurrentEventsByTag

`eventsByTag` is used for retrieving events that were marked with a given tag, e.g. all domain events of an Aggregate Root type.

```
implicit val mat = ActorMaterializer()(system)
val queries = PersistenceQuery(system).readJournalFor[LeveldbReadJournal](
  LeveldbReadJournal.Identifier)
```

```
val src: Source[EventEnvelope2, NotUsed] =
  queries.eventsByTag(tag = "green", offset = Sequence(0L))
```

To tag events you create an *Event Adapters* that wraps the events in a `akka.persistence.journal.Tagged` with the given `tags`.

```
import akka.persistence.journal.WriteEventAdapter
import akka.persistence.journal.Tagged

class MyTaggingEventAdapter extends WriteEventAdapter {
  val colors = Set("green", "black", "blue")
  override def toJournal(event: Any): Any = event match {
    case s: String =>
      var tags = colors.foldLeft(Set.empty[String]) { (acc, c) =>
        if (s.contains(c)) acc + c else acc
      }
      if (tags.isEmpty) event
      else Tagged(event, tags)
    case _ => event
  }

  override def manifest(event: Any): String = ""
}
```

You can retrieve a subset of all events by specifying `offset`, or use `0L` to retrieve all events with a given tag. The `offset` corresponds to an ordered sequence number for the specific tag. Note that the corresponding offset of each event is provided in the `EventEnvelope`, which makes it possible to resume the stream at a later point from a given offset.

In addition to the `offset` the `EventEnvelope` also provides `persistenceId` and `sequenceNr` for each event. The `sequenceNr` is the sequence number for the persistent actor with the `persistenceId` that persisted the event. The `persistenceId` + `sequenceNr` is an unique identifier for the event.

The returned event stream is ordered by the offset (tag sequence number), which corresponds to the same order as the write journal stored the events. The same stream elements (in same order) are returned for multiple executions of the query. Deleted events are not deleted from the tagged event stream.

Note: Events deleted using `deleteMessages(toSequenceNr)` are not deleted from the "tagged stream".

The stream is not completed when it reaches the end of the currently stored events, but it continues to push new events when new events are persisted. Corresponding query that is completed when it reaches the end of the currently stored events is provided by `currentEventsByTag`.

The LevelDB write journal is notifying the query side as soon as tagged events are persisted, but for efficiency reasons the query side retrieves the events in batches that sometimes can be delayed up to the configured `refresh-interval` or given `RefreshInterval` hint.

The stream is completed with failure if there is a failure in executing the query in the backend journal.

3.11.4 Configuration

Configuration settings can be defined in the configuration section with the absolute path corresponding to the identifier, which is `"akka.persistence.query.journal.leveldb"` for the default `LeveldbReadJournal.Identifier`.

It can be configured with the following properties:

```
# Configuration for the LeveldbReadJournal
akka.persistence.query.journal.leveldb {
```

```
  # Implementation class of the LevelDB ReadJournalProvider
  class = "akka.persistence.query.journal.leveldb.LeveldbReadJournalProvider"

  # Absolute path to the write journal plugin configuration entry that this
  # query journal will connect to. That must be a LeveldbJournal or␣
↪SharedLeveldbJournal.
  # If undefined (or "") it will connect to the default journal as specified by the
  # akka.persistence.journal.plugin property.
  write-plugin = ""

  # The LevelDB write journal is notifying the query side as soon as things
  # are persisted, but for efficiency reasons the query side retrieves the events
  # in batches that sometimes can be delayed up to the configured `refresh-
↪interval`.
  refresh-interval = 3s

  # How many events to fetch in one query (replay) and keep buffered until they
  # are delivered downstreams.
  max-buffer-size = 100
}
```

3.12 Testing Actor Systems

3.12.1 TestKit Example

Ray Roestenburg's example code from his blog adapted to work with Akka 2.x.

```
import scala.util.Random

import org.scalatest.BeforeAndAfterAll
import org.scalatest.WordSpecLike
import org.scalatest.Matchers

import com.typesafe.config.ConfigFactory

import akka.actor.Actor
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.actor.Props
import akka.testkit.{ TestActors, DefaultTimeout, ImplicitSender, TestKit }
import scala.concurrent.duration._
import scala.collection.immutable

/**
 * a Test to show some TestKit examples
 */
class TestKitUsageSpec
  extends TestKit(ActorSystem(
    "TestKitUsageSpec",
    ConfigFactory.parseString(TestKitUsageSpec.config)))
  with DefaultTimeout with ImplicitSender
  with WordSpecLike with Matchers with BeforeAndAfterAll {
  import TestKitUsageSpec._

  val echoRef = system.actorOf(TestActors.echoActorProps)
  val forwardRef = system.actorOf(Props(classOf[ForwardingActor], testActor))
  val filterRef = system.actorOf(Props(classOf[FilteringActor], testActor))
  val randomHead = Random.nextInt(6)
  val randomTail = Random.nextInt(10)
```

```
  val headList = immutable.Seq().padTo(randomHead, "0")
  val tailList = immutable.Seq().padTo(randomTail, "1")
  val seqRef =
    system.actorOf(Props(classOf[SequencingActor], testActor, headList, tailList))

  override def afterAll {
    shutdown()
  }

  "An EchoActor" should {
    "Respond with the same message it receives" in {
      within(500 millis) {
        echoRef ! "test"
        expectMsg("test")
      }
    }
  }
  "A ForwardingActor" should {
    "Forward a message it receives" in {
      within(500 millis) {
        forwardRef ! "test"
        expectMsg("test")
      }
    }
  }
  "A FilteringActor" should {
    "Filter all messages, except expected messagetypes it receives" in {
      var messages = Seq[String]()
      within(500 millis) {
        filterRef ! "test"
        expectMsg("test")
        filterRef ! 1
        expectNoMsg
        filterRef ! "some"
        filterRef ! "more"
        filterRef ! 1
        filterRef ! "text"
        filterRef ! 1

        receiveWhile(500 millis) {
          case msg: String => messages = msg +: messages
        }
      }
      messages.length should be(3)
      messages.reverse should be(Seq("some", "more", "text"))
    }
  }
  "A SequencingActor" should {
    "receive an interesting message at some point " in {
      within(500 millis) {
        ignoreMsg {
          case msg: String => msg != "something"
        }
        seqRef ! "something"
        expectMsg("something")
        ignoreMsg {
          case msg: String => msg == "1"
        }
        expectNoMsg
        ignoreNoMsg
      }
    }
  }
```

```
}

object TestKitUsageSpec {
  // Define your test specific configuration here
  val config = """
    akka {
      loglevel = "WARNING"
    }
    """

  /**
   * An Actor that forwards every message to a next Actor
   */
  class ForwardingActor(next: ActorRef) extends Actor {
    def receive = {
      case msg => next ! msg
    }
  }

  /**
   * An Actor that only forwards certain messages to a next Actor
   */
  class FilteringActor(next: ActorRef) extends Actor {
    def receive = {
      case msg: String => next ! msg
      case _           => None
    }
  }

  /**
   * An actor that sends a sequence of messages with a random head list, an
   * interesting value and a random tail list. The idea is that you would
   * like to test that the interesting value is received and that you cant
   * be bothered with the rest
   */
  class SequencingActor(next: ActorRef, head: immutable.Seq[String],
                        tail: immutable.Seq[String]) extends Actor {
    def receive = {
      case msg => {
        head foreach { next ! _ }
        next ! msg
        tail foreach { next ! _ }
      }
    }
  }
}
```

As with any piece of software, automated tests are a very important part of the development cycle. The actor model presents a different view on how units of code are delimited and how they interact, which has an influence on how to perform tests.

Akka comes with a dedicated module `akka-testkit` for supporting tests at different levels, which fall into two clearly distinct categories:

- Testing isolated pieces of code without involving the actor model, meaning without multiple threads; this implies completely deterministic behavior concerning the ordering of events and no concurrency concerns and will be called **Unit Testing** in the following.
- Testing (multiple) encapsulated actors including multi-threaded scheduling; this implies non-deterministic order of events but shielding from concurrency concerns by the actor model and will be called **Integration Testing** in the following.

There are of course variations on the granularity of tests in both categories, where unit testing reaches down to

white-box tests and integration testing can encompass functional tests of complete actor networks. The important distinction lies in whether concurrency concerns are part of the test or not. The tools offered are described in detail in the following sections.

Note: Be sure to add the module `akka-testkit` to your dependencies.

3.12.2 Synchronous Unit Testing with `TestActorRef`

Testing the business logic inside `Actor` classes can be divided into two parts: first, each atomic operation must work in isolation, then sequences of incoming events must be processed correctly, even in the presence of some possible variability in the ordering of events. The former is the primary use case for single-threaded unit testing, while the latter can only be verified in integration tests.

Normally, the `ActorRef` shields the underlying `Actor` instance from the outside, the only communications channel is the actor's mailbox. This restriction is an impediment to unit testing, which led to the inception of the `TestActorRef`. This special type of reference is designed specifically for test purposes and allows access to the actor in two ways: either by obtaining a reference to the underlying actor instance, or by invoking or querying the actor's behaviour (`receive`). Each one warrants its own section below.

Note: It is highly recommended to stick to traditional behavioural testing (using messaging to ask the Actor to reply with the state you want to run assertions against), instead of using `TestActorRef` whenever possible.

Warning: Due to the synchronous nature of `TestActorRef` it will **not** work with some support traits that Akka provides as they require asynchronous behaviours to function properly. Examples of traits that do not mix well with test actor refs are *PersistentActor* and *AtLeastOnceDelivery* provided by *Akka Persistence*.

Obtaining a Reference to an `Actor`

Having access to the actual `Actor` object allows application of all traditional unit testing techniques on the contained methods. Obtaining a reference is done like this:

```
import akka.testkit.TestActorRef

val actorRef = TestActorRef[MyActor]
val actor = actorRef.underlyingActor
```

Since `TestActorRef` is generic in the actor type it returns the underlying actor with its proper static type. From this point on you may bring any unit testing tool to bear on your actor as usual.

Testing Finite State Machines

If your actor under test is a `FSM`, you may use the special `TestFSMRef` which offers all features of a normal `TestActorRef` and in addition allows access to the internal state:

```
import akka.testkit.TestFSMRef
import akka.actor.FSM
import scala.concurrent.duration._

val fsm = TestFSMRef(new TestFsmActor)

val mustBeTypedProperly: TestActorRef[TestFsmActor] = fsm

assert(fsm.stateName == 1)
```

```
assert(fsm.stateData == "")
fsm ! "go" // being a TestActorRef, this runs also on the CallingThreadDispatcher
assert(fsm.stateName == 2)
assert(fsm.stateData == "go")

fsm.setState(stateName = 1)
assert(fsm.stateName == 1)

assert(fsm.isTimerActive("test") == false)
fsm.setTimer("test", 12, 10 millis, true)
assert(fsm.isTimerActive("test") == true)
fsm.cancelTimer("test")
assert(fsm.isTimerActive("test") == false)
```

Due to a limitation in Scala's type inference, there is only the factory method shown above, so you will probably write code like `TestFSMRef(new MyFSM)` instead of the hypothetical `ActorRef`-inspired `TestFSMRef[MyFSM]`. All methods shown above directly access the FSM state without any synchronization; this is perfectly alright if the `CallingThreadDispatcher` is used and no other threads are involved, but it may lead to surprises if you were to actually exercise timer events, because those are executed on the `Scheduler` thread.

Testing the Actor's Behavior

When the dispatcher invokes the processing behavior of an actor on a message, it actually calls `apply` on the current behavior registered for the actor. This starts out with the return value of the declared `receive` method, but it may also be changed using `become` and `unbecome` in response to external messages. All of this contributes to the overall actor behavior and it does not lend itself to easy testing on the `Actor` itself. Therefore the `TestActorRef` offers a different mode of operation to complement the `Actor` testing: it supports all operations also valid on normal `ActorRef`. Messages sent to the actor are processed synchronously on the current thread and answers may be sent back as usual. This trick is made possible by the `CallingThreadDispatcher` described below (see *CallingThreadDispatcher*); this dispatcher is set implicitly for any actor instantiated into a `TestActorRef`.

```
import akka.testkit.TestActorRef
import scala.concurrent.duration._
import scala.concurrent.Await
import akka.pattern.ask

val actorRef = TestActorRef(new MyActor)
// hypothetical message stimulating a '42' answer
val future = actorRef ? Say42
val Success(result: Int) = future.value.get
result should be(42)
```

As the `TestActorRef` is a subclass of `LocalActorRef` with a few special extras, also aspects like supervision and restarting work properly, but beware that execution is only strictly synchronous as long as all actors involved use the `CallingThreadDispatcher`. As soon as you add elements which include more sophisticated scheduling you leave the realm of unit testing as you then need to think about asynchronicity again (in most cases the problem will be to wait until the desired effect had a chance to happen).

One more special aspect which is overridden for single-threaded tests is the `receiveTimeout`, as including that would entail asynchronous queuing of `ReceiveTimeout` messages, violating the synchronous contract.

Note: To summarize: `TestActorRef` overwrites two fields: it sets the dispatcher to `CallingThreadDispatcher.global` and it sets the `receiveTimeout` to None.

The Way In-Between: Expecting Exceptions

If you want to test the actor behavior, including hotswapping, but without involving a dispatcher and without having the `TestActorRef` swallow any thrown exceptions, then there is another mode available for you: just use the `receive` method on `TestActorRef`, which will be forwarded to the underlying actor:

```
import akka.testkit.TestActorRef

val actorRef = TestActorRef(new Actor {
  def receive = {
    case "hello" => throw new IllegalArgumentException("boom")
  }
})
intercept[IllegalArgumentException] { actorRef.receive("hello") }
```

Use Cases

You may of course mix and match both modi operandi of `TestActorRef` as suits your test needs:

- one common use case is setting up the actor into a specific internal state before sending the test message
- another is to verify correct internal state transitions after having sent the test message

Feel free to experiment with the possibilities, and if you find useful patterns, don't hesitate to let the Akka forums know about them! Who knows, common operations might even be worked into nice DSLs.

3.12.3 Asynchronous Integration Testing with `TestKit`

When you are reasonably sure that your actor's business logic is correct, the next step is verifying that it works correctly within its intended environment (if the individual actors are simple enough, possibly because they use the `FSM` module, this might also be the first step). The definition of the environment depends of course very much on the problem at hand and the level at which you intend to test, ranging for functional/integration tests to full system tests. The minimal setup consists of the test procedure, which provides the desired stimuli, the actor under test, and an actor receiving replies. Bigger systems replace the actor under test with a network of actors, apply stimuli at varying injection points and arrange results to be sent from different emission points, but the basic principle stays the same in that a single procedure drives the test.

The `TestKit` class contains a collection of tools which makes this common task easy.

```
import akka.actor.ActorSystem
import akka.actor.Actor
import akka.actor.Props
import akka.testkit.{ TestActors, TestKit, ImplicitSender }
import org.scalatest.WordSpecLike
import org.scalatest.Matchers
import org.scalatest.BeforeAndAfterAll

class MySpec() extends TestKit(ActorSystem("MySpec")) with ImplicitSender
  with WordSpecLike with Matchers with BeforeAndAfterAll {

  override def afterAll {
    TestKit.shutdownActorSystem(system)
  }

  "An Echo actor" must {

    "send back messages unchanged" in {
      val echo = system.actorOf(TestActors.echoActorProps)
      echo ! "hello world"
      expectMsg("hello world")
    }
```

```
  }
}
```

The `TestKit` contains an actor named `testActor` which is the entry point for messages to be examined with the various `expectMsg...` assertions detailed below. When mixing in the trait `ImplicitSender` this test actor is implicitly used as sender reference when dispatching messages from the test procedure. The `testActor` may also be passed to other actors as usual, usually subscribing it as notification listener. There is a whole set of examination methods, e.g. receiving all consecutive messages matching certain criteria, receiving a whole sequence of fixed messages or classes, receiving nothing for some time, etc.

The ActorSystem passed in to the constructor of TestKit is accessible via the `system` member. Remember to shut down the actor system after the test is finished (also in case of failure) so that all actors—including the test actor—are stopped.

Built-In Assertions

The above mentioned `expectMsg` is not the only method for formulating assertions concerning received messages. Here is the full list:

- `expectMsg[T](d: Duration, msg: T): T`

  The given message object must be received within the specified time; the object will be returned.

- `expectMsgPF[T](d: Duration)(pf: PartialFunction[Any, T]): T`

  Within the given time period, a message must be received and the given partial function must be defined for that message; the result from applying the partial function to the received message is returned. The duration may be left unspecified (empty parentheses are required in this case) to use the deadline from the innermost enclosing *within* block instead.

- `expectMsgClass[T](d: Duration, c: Class[T]): T`

  An object which is an instance of the given `Class` must be received within the allotted time frame; the object will be returned. Note that this does a conformance check; if you need the class to be equal, have a look at `expectMsgAllClassOf` with a single given class argument.

- `expectMsgType[T: Manifest](d: Duration)`

  An object which is an instance of the given type (after erasure) must be received within the allotted time frame; the object will be returned. This method is approximately equivalent to `expectMsgClass(implicitly[ClassTag[T]].runtimeClass)`.

- `expectMsgAnyOf[T](d: Duration, obj: T*): T`

  An object must be received within the given time, and it must be equal (compared with ==) to at least one of the passed reference objects; the received object will be returned.

- `expectMsgAnyClassOf[T](d: Duration, obj: Class[_ <: T]*): T`

  An object must be received within the given time, and it must be an instance of at least one of the supplied `Class` objects; the received object will be returned.

- `expectMsgAllOf[T](d: Duration, obj: T*): Seq[T]`

  A number of objects matching the size of the supplied object array must be received within the given time, and for each of the given objects there must exist at least one among the received ones which equals (compared with ==) it. The full sequence of received objects is returned.

- `expectMsgAllClassOf[T](d: Duration, c: Class[_ <: T]*): Seq[T]`

  A number of objects matching the size of the supplied `Class` array must be received within the given time, and for each of the given classes there must exist at least one among the received objects whose class equals (compared with ==) it (this is *not* a conformance check). The full sequence of received objects is returned.

- `expectMsgAllConformingOf[T](d: Duration, c: Class[_ <: T]*): Seq[T]`

  A number of objects matching the size of the supplied `Class` array must be received within the given time, and for each of the given classes there must exist at least one among the received objects which is an instance of this class. The full sequence of received objects is returned.

- `expectNoMsg(d: Duration)`

  No message must be received within the given time. This also fails if a message has been received before calling this method which has not been removed from the queue using one of the other methods.

- `receiveN(n: Int, d: Duration): Seq[AnyRef]`

  `n` messages must be received within the given time; the received messages are returned.

- `fishForMessage(max: Duration, hint: String)(pf: PartialFunction[Any, Boolean]): Any`

  Keep receiving messages as long as the time is not used up and the partial function matches and returns `false`. Returns the message received for which it returned `true` or throws an exception, which will include the provided hint for easier debugging.

In addition to message reception assertions there are also methods which help with message flows:

- `receiveOne(d: Duration): AnyRef`

  Tries to receive one message for at most the given time interval and returns `null` in case of failure. If the given Duration is zero, the call is non-blocking (polling mode).

- `receiveWhile[T](max: Duration, idle: Duration, messages: Int)(pf: PartialFunction[An`

  Collect messages as long as

  - they are matching the given partial function
  - the given time interval is not used up
  - the next message is received within the idle timeout
  - the number of messages has not yet reached the maximum

  All collected messages are returned. The maximum duration defaults to the time remaining in the innermost enclosing *within* block and the idle duration defaults to infinity (thereby disabling the idle timeout feature). The number of expected messages defaults to `Int.MaxValue`, which effectively disables this limit.

- `awaitCond(p: => Boolean, max: Duration, interval: Duration)`

  Poll the given condition every `interval` until it returns `true` or the `max` duration is used up. The interval defaults to 100 ms and the maximum defaults to the time remaining in the innermost enclosing *within* block.

- `awaitAssert(a: => Any, max: Duration, interval: Duration)`

  Poll the given assert function every `interval` until it does not throw an exception or the `max` duration is used up. If the timeout expires the last exception is thrown. The interval defaults to 100 ms and the maximum defaults to the time remaining in the innermost enclosing *within* block.The interval defaults to 100 ms and the maximum defaults to the time remaining in the innermost enclosing *within* block.

- `ignoreMsg(pf: PartialFunction[AnyRef, Boolean])`

  `ignoreNoMsg`

  The internal `testActor` contains a partial function for ignoring messages: it will only enqueue messages which do not match the function or for which the function returns `false`. This function can be set and reset using the methods given above; each invocation replaces the previous function, they are not composed.

  This feature is useful e.g. when testing a logging system, where you want to ignore regular messages and are only interested in your specific ones.

Expecting Log Messages

Since an integration test does not allow to the internal processing of the participating actors, verifying expected exceptions cannot be done directly. Instead, use the logging system for this purpose: replacing the normal event handler with the `TestEventListener` and using an `EventFilter` allows assertions on log messages, including those which are generated by exceptions:

```
import akka.testkit.EventFilter
import com.typesafe.config.ConfigFactory

implicit val system = ActorSystem("testsystem", ConfigFactory.parseString("""
  akka.loggers = ["akka.testkit.TestEventListener"]
  """))
try {
  val actor = system.actorOf(Props.empty)
  EventFilter[ActorKilledException](occurrences = 1) intercept {
    actor ! Kill
  }
} finally {
  shutdown(system)
}
```

If a number of occurrences is specific—as demonstrated above—then `intercept` will block until that number of matching messages have been received or the timeout configured in `akka.test.filter-leeway` is used up (time starts counting after the passed-in block of code returns). In case of a timeout the test fails.

Note: Be sure to exchange the default logger with the `TestEventListener` in your `application.conf` to enable this function:

```
akka.loggers = [akka.testkit.TestEventListener]
```

Timing Assertions

Another important part of functional testing concerns timing: certain events must not happen immediately (like a timer), others need to happen before a deadline. Therefore, all examination methods accept an upper time limit within the positive or negative result must be obtained. Lower time limits need to be checked external to the examination, which is facilitated by a new construct for managing time constraints:

```
within([min, ]max) {
  ...
}
```

The block given to `within` must complete after a *Duration* which is between `min` and `max`, where the former defaults to zero. The deadline calculated by adding the `max` parameter to the block's start time is implicitly available within the block to all examination methods, if you do not specify it, it is inherited from the innermost enclosing `within` block.

It should be noted that if the last message-receiving assertion of the block is `expectNoMsg` or `receiveWhile`, the final check of the `within` is skipped in order to avoid false positives due to wake-up latencies. This means that while individual contained assertions still use the maximum time bound, the overall block may take arbitrarily longer in this case.

```
import akka.actor.Props
import scala.concurrent.duration._

val worker = system.actorOf(Props[Worker])
within(200 millis) {
  worker ! "some work"
  expectMsg("some result")
```

```
  expectNoMsg // will block for the rest of the 200ms
  Thread.sleep(300) // will NOT make this block fail
}
```

Note: All times are measured using `System.nanoTime`, meaning that they describe wall time, not CPU time.

Ray Roestenburg has written a great article on using the TestKit: http://roestenburg.agilesquad.com/2011/02/unit-testing-akka-actors-with-testkit_12.html. His full example is also available *here*.

Accounting for Slow Test Systems

The tight timeouts you use during testing on your lightning-fast notebook will invariably lead to spurious test failures on the heavily loaded Jenkins server (or similar). To account for this situation, all maximum durations are internally scaled by a factor taken from the *Configuration*, `akka.test.timefactor`, which defaults to 1.

You can scale other durations with the same factor by using the implicit conversion in `akka.testkit` package object to add dilated function to `Duration`.

```
import scala.concurrent.duration._
import akka.testkit._
10.milliseconds.dilated
```

Resolving Conflicts with Implicit ActorRef

If you want the sender of messages inside your TestKit-based tests to be the `testActor` simply mix in `ImplicitSender` into your test.

```
class MySpec() extends TestKit(ActorSystem("MySpec")) with ImplicitSender
  with WordSpecLike with Matchers with BeforeAndAfterAll {
```

Using Multiple Probe Actors

When the actors under test are supposed to send various messages to different destinations, it may be difficult distinguishing the message streams arriving at the `testActor` when using the `TestKit` as a mixin. Another approach is to use it for creation of simple probe actors to be inserted in the message flows. To make this more powerful and convenient, there is a concrete implementation called `TestProbe`. The functionality is best explained using a small example:

```
import scala.concurrent.duration._
import scala.concurrent.Future
import akka.actor._
import akka.testkit.TestProbe
```

```
class MyDoubleEcho extends Actor {
  var dest1: ActorRef = _
  var dest2: ActorRef = _
  def receive = {
    case (d1: ActorRef, d2: ActorRef) =>
      dest1 = d1
      dest2 = d2
    case x =>
      dest1 ! x
      dest2 ! x
  }
}
```

```
val probe1 = TestProbe()
val probe2 = TestProbe()
val actor = system.actorOf(Props[MyDoubleEcho])
actor ! ((probe1.ref, probe2.ref))
actor ! "hello"
probe1.expectMsg(500 millis, "hello")
probe2.expectMsg(500 millis, "hello")
```

Here a the system under test is simulated by `MyDoubleEcho`, which is supposed to mirror its input to two outputs. Attaching two test probes enables verification of the (simplistic) behavior. Another example would be two actors A and B which collaborate by A sending messages to B. In order to verify this message flow, a `TestProbe` could be inserted as target of A, using the forwarding capabilities or auto-pilot described below to include a real B in the test setup.

If you have many test probes, you can name them to get meaningful actor names in test logs and assertions:

```
val worker = TestProbe("worker")
val aggregator = TestProbe("aggregator")

worker.ref.path.name should startWith("worker")
aggregator.ref.path.name should startWith("aggregator")
```

Probes may also be equipped with custom assertions to make your test code even more concise and clear:

```
final case class Update(id: Int, value: String)

val probe = new TestProbe(system) {
  def expectUpdate(x: Int) = {
    expectMsgPF() {
      case Update(id, _) if id == x => true
    }
    sender() ! "ACK"
  }
}
```

You have complete flexibility here in mixing and matching the `TestKit` facilities with your own checks and choosing an intuitive name for it. In real life your code will probably be a bit more complicated than the example given above; just use the power!

Warning: Any message send from a `TestProbe` to another actor which runs on the CallingThreadDispatcher runs the risk of dead-lock, if that other actor might also send to this probe. The implementation of `TestProbe.watch` and `TestProbe.unwatch` will also send a message to the watchee, which means that it is dangerous to try watching e.g. `TestActorRef` from a `TestProbe`.

Watching Other Actors from Probes

A `TestProbe` can register itself for DeathWatch of any other actor:

```
val probe = TestProbe()
probe watch target
target ! PoisonPill
probe.expectTerminated(target)
```

Replying to Messages Received by Probes

The probes keep track of the communications channel for replies, if possible, so they can also reply:

```
val probe = TestProbe()
val future = probe.ref ? "hello"
probe.expectMsg(0 millis, "hello") // TestActor runs on CallingThreadDispatcher
probe.reply("world")
assert(future.isCompleted && future.value == Some(Success("world")))
```

Forwarding Messages Received by Probes

Given a destination actor `dest` which in the nominal actor network would receive a message from actor `source`. If you arrange for the message to be sent to a `TestProbe probe` instead, you can make assertions concerning volume and timing of the message flow while still keeping the network functioning:

```
class Source(target: ActorRef) extends Actor {
  def receive = {
    case "start" => target ! "work"
  }
}

class Destination extends Actor {
  def receive = {
    case x => // Do something..
  }
}
```

```
val probe = TestProbe()
val source = system.actorOf(Props(classOf[Source], probe.ref))
val dest = system.actorOf(Props[Destination])
source ! "start"
probe.expectMsg("work")
probe.forward(dest)
```

The `dest` actor will receive the same message invocation as if no test probe had intervened.

Auto-Pilot

Receiving messages in a queue for later inspection is nice, but in order to keep a test running and verify traces later you can also install an `AutoPilot` in the participating test probes (actually in any `TestKit`) which is invoked before enqueueing to the inspection queue. This code can be used to forward messages, e.g. in a chain `A --> Probe --> B`, as long as a certain protocol is obeyed.

```
val probe = TestProbe()
probe.setAutoPilot(new TestActor.AutoPilot {
  def run(sender: ActorRef, msg: Any): TestActor.AutoPilot =
    msg match {
      case "stop" ⇒ TestActor.NoAutoPilot
      case x      ⇒ testActor.tell(x, sender); TestActor.KeepRunning
    }
})
```

The `run` method must return the auto-pilot for the next message, which may be `KeepRunning` to retain the current one or `NoAutoPilot` to switch it off.

Caution about Timing Assertions

The behavior of `within` blocks when using test probes might be perceived as counter-intuitive: you need to remember that the nicely scoped deadline as described *above* is local to each probe. Hence, probes do not react to each other's deadlines or to the deadline set in an enclosing `TestKit` instance:

```
val probe = TestProbe()
within(1 second) {
  probe.expectMsg("hello")
}
```

Here, the `expectMsg` call will use the default timeout.

Testing parent-child relationships

The parent of an actor is always the actor that created it. At times this leads to a coupling between the two that may not be straightforward to test. There are several approaches to improve testability of a child actor that needs to refer to its parent:

1. when creating a child, pass an explicit reference to its parent
2. create the child with a `TestProbe` as parent
3. create a fabricated parent when testing

Conversely, a parent's binding to its child can be lessened as follows:

4. when creating a parent, tell the parent how to create its child

For example, the structure of the code you want to test may follow this pattern:

```
class Parent extends Actor {
  val child = context.actorOf(Props[Child], "child")
  var ponged = false

  def receive = {
    case "pingit" => child ! "ping"
    case "pong"   => ponged = true
  }
}

class Child extends Actor {
  def receive = {
    case "ping" => context.parent ! "pong"
  }
}
```

Introduce child to its parent

The first option is to avoid use of the `context.parent` function and create a child with a custom parent by passing an explicit reference to its parent instead.

```
class DependentChild(parent: ActorRef) extends Actor {
  def receive = {
    case "ping" => parent ! "pong"
  }
}
```

Create the child using TestProbe

The `TestProbe` class can in fact create actors that will run with the test probe as parent. This will cause any messages the the child actor sends to *context.parent* to end up in the test probe.

```
"A TestProbe serving as parent" should {
  "test its child responses" in {
    val parent = TestProbe()
    val child = parent.childActorOf(Props[Child])
    parent.send(child, "ping")
    parent.expectMsg("pong")
  }
}
```

Using a fabricated parent

If you prefer to avoid modifying the parent or child constructor you can create a fabricated parent in your test. This, however, does not enable you to test the parent actor in isolation.

```
"A fabricated parent" should {
  "test its child responses" in {
    val proxy = TestProbe()
    val parent = system.actorOf(Props(new Actor {
      val child = context.actorOf(Props[Child], "child")
      def receive = {
        case x if sender == child => proxy.ref forward x
        case x                    => child forward x
      }
    }))

    proxy.send(parent, "ping")
    proxy.expectMsg("pong")
  }
}
```

Externalize child making from the parent

Alternatively, you can tell the parent how to create its child. There are two ways to do this: by giving it a `Props` object or by giving it a function which takes care of creating the child actor:

```
class DependentParent(childProps: Props) extends Actor {
  val child = context.actorOf(childProps, "child")
  var ponged = false

  def receive = {
    case "pingit" => child ! "ping"
    case "pong"   => ponged = true
  }
}

class GenericDependentParent(childMaker: ActorRefFactory => ActorRef) extends␣
↪Actor {
  val child = childMaker(context)
  var ponged = false

  def receive = {
    case "pingit" => child ! "ping"
    case "pong"   => ponged = true
  }
}
```

Creating the `Props` is straightforward and the function may look like this in your test code:

```
val maker = (_: ActorRefFactory) => probe.ref
val parent = system.actorOf(Props(classOf[GenericDependentParent], maker))
```

And like this in your application code:

```
val maker = (f: ActorRefFactory) => f.actorOf(Props[Child])
val parent = system.actorOf(Props(classOf[GenericDependentParent], maker))
```

Which of these methods is the best depends on what is most important to test. The most generic option is to create the parent actor by passing it a function that is responsible for the Actor creation, but the fabricated parent is often sufficient.

3.12.4 CallingThreadDispatcher

The `CallingThreadDispatcher` serves good purposes in unit testing, as described above, but originally it was conceived in order to allow contiguous stack traces to be generated in case of an error. As this special dispatcher runs everything which would normally be queued directly on the current thread, the full history of a message's processing chain is recorded on the call stack, so long as all intervening actors run on this dispatcher.

How to use it

Just set the dispatcher as you normally would:

```
import akka.testkit.CallingThreadDispatcher
val ref = system.actorOf(Props[MyActor].withDispatcher(CallingThreadDispatcher.Id))
```

How it works

When receiving an invocation, the `CallingThreadDispatcher` checks whether the receiving actor is already active on the current thread. The simplest example for this situation is an actor which sends a message to itself. In this case, processing cannot continue immediately as that would violate the actor model, so the invocation is queued and will be processed when the active invocation on that actor finishes its processing; thus, it will be processed on the calling thread, but simply after the actor finishes its previous work. In the other case, the invocation is simply processed immediately on the current thread. Futures scheduled via this dispatcher are also executed immediately.

This scheme makes the `CallingThreadDispatcher` work like a general purpose dispatcher for any actors which never block on external events.

In the presence of multiple threads it may happen that two invocations of an actor running on this dispatcher happen on two different threads at the same time. In this case, both will be processed directly on their respective threads, where both compete for the actor's lock and the loser has to wait. Thus, the actor model is left intact, but the price is loss of concurrency due to limited scheduling. In a sense this is equivalent to traditional mutex style concurrency.

The other remaining difficulty is correct handling of suspend and resume: when an actor is suspended, subsequent invocations will be queued in thread-local queues (the same ones used for queuing in the normal case). The call to `resume`, however, is done by one specific thread, and all other threads in the system will probably not be executing this specific actor, which leads to the problem that the thread-local queues cannot be emptied by their native threads. Hence, the thread calling `resume` will collect all currently queued invocations from all threads into its own queue and process them.

Limitations

Warning: In case the CallingThreadDispatcher is used for top-level actors, but without going through TestActorRef, then there is a time window during which the actor is awaiting construction by the user guardian actor. Sending messages to the actor during this time period will result in them being enqueued and then executed on the guardian's thread instead of the caller's thread. To avoid this, use TestActorRef.

If an actor's behavior blocks on a something which would normally be affected by the calling actor after having sent the message, this will obviously dead-lock when using this dispatcher. This is a common scenario in actor tests based on `CountDownLatch` for synchronization:

```
val latch = new CountDownLatch(1)
actor ! startWorkAfter(latch)   // actor will call latch.await() before proceeding
doSomeSetupStuff()
latch.countDown()
```

The example would hang indefinitely within the message processing initiated on the second line and never reach the fourth line, which would unblock it on a normal dispatcher.

Thus, keep in mind that the `CallingThreadDispatcher` is not a general-purpose replacement for the normal dispatchers. On the other hand it may be quite useful to run your actor network on it for testing, because if it runs without dead-locking chances are very high that it will not dead-lock in production.

Warning: The above sentence is unfortunately not a strong guarantee, because your code might directly or indirectly change its behavior when running on a different dispatcher. If you are looking for a tool to help you debug dead-locks, the `CallingThreadDispatcher` may help with certain error scenarios, but keep in mind that it has may give false negatives as well as false positives.

Thread Interruptions

If the CallingThreadDispatcher sees that the current thread has its `isInterrupted()` flag set when message processing returns, it will throw an `InterruptedException` after finishing all its processing (i.e. all messages which need processing as described above are processed before this happens). As `tell` cannot throw exceptions due to its contract, this exception will then be caught and logged, and the thread's interrupted status will be set again.

If during message processing an `InterruptedException` is thrown then it will be caught inside the CallingThreadDispatcher's message handling loop, the thread's interrupted flag will be set and processing continues normally.

Note: The summary of these two paragraphs is that if the current thread is interrupted while doing work under the CallingThreadDispatcher, then that will result in the `isInterrupted` flag to be `true` when the message send returns and no `InterruptedException` will be thrown.

Benefits

To summarize, these are the features with the `CallingThreadDispatcher` has to offer:

- Deterministic execution of single-threaded tests while retaining nearly full actor semantics
- Full message processing history leading up to the point of failure in exception stack traces
- Exclusion of certain classes of dead-lock scenarios

3.12.5 Tracing Actor Invocations

The testing facilities described up to this point were aiming at formulating assertions about a system's behavior. If a test fails, it is usually your job to find the cause, fix it and verify the test again. This process is supported by debuggers as well as logging, where the Akka toolkit offers the following options:

- *Logging of exceptions thrown within Actor instances*

  This is always on; in contrast to the other logging mechanisms, this logs at `ERROR` level.

- *Logging of message invocations on certain actors*

  This is enabled by a setting in the *Configuration* — namely `akka.actor.debug.receive` — which enables the `loggable` statement to be applied to an actor's `receive` function:

```
import akka.event.LoggingReceive
def receive = LoggingReceive {
  case msg => // Do something ...
}
def otherState: Receive = LoggingReceive.withLabel("other") {
  case msg => // Do something else ...
}
```

If the aforementioned setting is not given in the *Configuration*, this method will pass through the given `Receive` function unmodified, meaning that there is no runtime cost unless actually enabled.

The logging feature is coupled to this specific local mark-up because enabling it uniformly on all actors is not usually what you need, and it would lead to endless loops if it were applied to event bus logger listeners.

- *Logging of special messages*

  Actors handle certain special messages automatically, e.g. `Kill`, `PoisonPill`, etc. Tracing of these message invocations is enabled by the setting `akka.actor.debug.autoreceive`, which enables this on all actors.

- *Logging of the actor lifecycle*

  Actor creation, start, restart, monitor start, monitor stop and stop may be traced by enabling the setting `akka.actor.debug.lifecycle`; this, too, is enabled uniformly on all actors.

All these messages are logged at `DEBUG` level. To summarize, you can enable full logging of actor activities using this configuration fragment:

```
akka {
  loglevel = "DEBUG"
  actor {
    debug {
      receive = on
      autoreceive = on
      lifecycle = on
    }
  }
}
```

3.12.6 Different Testing Frameworks

Akka's own test suite is written using ScalaTest, which also shines through in documentation examples. However, the TestKit and its facilities do not depend on that framework, you can essentially use whichever suits your development style best.

This section contains a collection of known gotchas with some other frameworks, which is by no means exhaustive and does not imply endorsement or special support.

When you need it to be a trait

If for some reason it is a problem to inherit from `TestKit` due to it being a concrete class instead of a trait, there's `TestKitBase`:

```
import akka.testkit.TestKitBase

class MyTest extends TestKitBase {
  implicit lazy val system = ActorSystem()

  // put your test code here ...

  shutdown(system)
}
```

The `implicit lazy val system` must be declared exactly like that (you can of course pass arguments to the actor system factory as needed) because trait `TestKitBase` needs the system during its construction.

Warning: Use of the trait is discouraged because of potential issues with binary backwards compatibility in the future, use at own risk.

Specs2

Some Specs2 users have contributed examples of how to work around some clashes which may arise:

- Mixing TestKit into `org.specs2.mutable.Specification` results in a name clash involving the `end` method (which is a private variable in TestKit and an abstract method in Specification); if mixing in TestKit first, the code may compile but might then fail at runtime. The workaround—which is actually beneficial also for the third point—is to apply the TestKit together with `org.specs2.specification.Scope`.
- The Specification traits provide a `Duration` DSL which uses partly the same method names as `scala.concurrent.duration.Duration`, resulting in ambiguous implicits if `scala.concurrent.duration._` is imported. There are two workarounds:
  - either use the Specification variant of Duration and supply an implicit conversion to the Akka Duration. This conversion is not supplied with the Akka distribution because that would mean that our JAR files would depend on Specs2, which is not justified by this little feature.
  - or mix `org.specs2.time.NoTimeConversions` into the Specification.
- Specifications are by default executed concurrently, which requires some care when writing the tests or alternatively the `sequential` keyword.

3.12.7 Configuration

There are several configuration properties for the TestKit module, please refer to the *reference configuration*.

3.13 Actor DSL

3.13.1 The Actor DSL

Simple actors—for example one-off workers or even when trying things out in the REPL—can be created more concisely using the `Act` trait. The supporting infrastructure is bundled in the following import:

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

implicit val system = ActorSystem("demo")
```

This import is assumed for all code samples throughout this section. The implicit actor system serves as `ActorRefFactory` for all examples below. To define a simple actor, the following is sufficient:

```
val a = actor(new Act {
  become {
    case "hello" ⇒ sender() ! "hi"
  }
})
```

Here, `actor` takes the role of either `system.actorOf` or `context.actorOf`, depending on which context it is called in: it takes an implicit `ActorRefFactory`, which within an actor is available in the form of the `implicit val context: ActorContext`. Outside of an actor, you'll have to either declare an implicit `ActorSystem`, or you can give the factory explicitly (see further below).

The two possible ways of issuing a `context.become` (replacing or adding the new behavior) are offered separately to enable a clutter-free notation of nested receives:

```
val a = actor(new Act {
  become { // this will replace the initial (empty) behavior
    case "info" ⇒ sender() ! "A"
    case "switch" ⇒
      becomeStacked { // this will stack upon the "A" behavior
        case "info"   ⇒ sender() ! "B"
        case "switch" ⇒ unbecome() // return to the "A" behavior
      }
    case "lobotomize" ⇒ unbecome() // OH NOES: Actor.emptyBehavior
  }
})
```

Please note that calling `unbecome` more often than `becomeStacked` results in the original behavior being installed, which in case of the `Act` trait is the empty behavior (the outer `become` just replaces it during construction).

Life-cycle management

Life-cycle hooks are also exposed as DSL elements (see *Start Hook* and *Stop Hook*), where later invocations of the methods shown below will replace the contents of the respective hooks:

```
val a = actor(new Act {
  whenStarting { testActor ! "started" }
  whenStopping { testActor ! "stopped" }
})
```

The above is enough if the logical life-cycle of the actor matches the restart cycles (i.e. `whenStopping` is executed before a restart and `whenStarting` afterwards). If that is not desired, use the following two hooks (see *Restart Hooks*):

```
val a = actor(new Act {
  become {
    case "die" ⇒ throw new Exception
  }
  whenFailing { case m @ (cause, msg) ⇒ testActor ! m }
  whenRestarted { cause ⇒ testActor ! cause }
})
```

It is also possible to create nested actors, i.e. grand-children, like this:

```
// here we pass in the ActorRefFactory explicitly as an example
val a = actor(system, "fred")(new Act {
  val b = actor("barney")(new Act {
    whenStarting { context.parent ! ("hello from " + self.path) }
  })
  become {
    case x ⇒ testActor ! x
  }
})
```

Note: In some cases it will be necessary to explicitly pass the `ActorRefFactory` to the `actor` method (you will notice when the compiler tells you about ambiguous implicits).

The grand-child will be supervised by the child; the supervisor strategy for this relationship can also be configured using a DSL element (supervision directives are part of the `Act` trait):

```
superviseWith(OneForOneStrategy() {
  case e: Exception if e.getMessage == "hello" ⇒ Stop
  case _: Exception                            ⇒ Resume
})
```

Actor with `Stash`

Last but not least there is a little bit of convenience magic built-in, which detects if the runtime class of the statically given actor subtype extends the `RequiresMessageQueue` trait via the `Stash` trait (this is a complicated way of saying that `new Act with Stash` would not work because its runtime erased type is just an anonymous subtype of `Act`). The purpose is to automatically use the appropriate deque-based mailbox type required by `Stash`. If you want to use this magic, simply extend `ActWithStash`:

```
val a = actor(new ActWithStash {
  become {
    case 1 ⇒ stash()
    case 2 ⇒
      testActor ! 2; unstashAll(); becomeStacked {
        case 1 ⇒ testActor ! 1; unbecome()
      }
  }
})
```

3.14 Typed Actors

Note: This module will be deprecated as it will be superseded by the *Akka Typed* project which is currently being developed in open preview mode.

Akka Typed Actors is an implementation of the Active Objects pattern. Essentially turning method invocations into asynchronous dispatch instead of synchronous that has been the default way since Smalltalk came out.

Typed Actors consist of 2 "parts", a public interface and an implementation, and if you've done any work in "enterprise" Java, this will be very familiar to you. As with normal Actors you have an external API (the public interface instance) that will delegate method calls asynchronously to a private instance of the implementation.

The advantage of Typed Actors vs. Actors is that with TypedActors you have a static contract, and don't need to define your own messages, the downside is that it places some limitations on what you can do and what you can't, i.e. you cannot use `become`/`unbecome`.

Typed Actors are implemented using JDK Proxies which provide a pretty easy-worked API to intercept method calls.

Note: Just as with regular Akka Actors, Typed Actors process one call at a time.

3.14.1 When to use Typed Actors

Typed actors are nice for bridging between actor systems (the "inside") and non-actor code (the "outside"), because they allow you to write normal OO-looking code on the outside. Think of them like doors: their practicality lies in interfacing between private sphere and the public, but you don't want that many doors inside your house, do you? For a longer discussion see this blog post.

A bit more background: TypedActors can easily be abused as RPC, and that is an abstraction which is well-known to be leaky. Hence TypedActors are not what we think of first when we talk about making highly scalable concurrent software easier to write correctly. They have their niche, use them sparingly.

3.14.2 The tools of the trade

Before we create our first Typed Actor we should first go through the tools that we have at our disposal, it's located in `akka.actor.TypedActor`.

```
import akka.actor.TypedActor

//Returns the Typed Actor Extension
val extension = TypedActor(system) //system is an instance of ActorSystem

//Returns whether the reference is a Typed Actor Proxy or not
TypedActor(system).isTypedActor(someReference)

//Returns the backing Akka Actor behind an external Typed Actor Proxy
TypedActor(system).getActorRefFor(someReference)

//Returns the current ActorContext,
// method only valid within methods of a TypedActor implementation
val c: ActorContext = TypedActor.context

//Returns the external proxy of the current Typed Actor,
// method only valid within methods of a TypedActor implementation
val s: Squarer = TypedActor.self[Squarer]

//Returns a contextual instance of the Typed Actor Extension
//this means that if you create other Typed Actors with this,
//they will become children to the current Typed Actor.
TypedActor(TypedActor.context)
```

Warning: Same as not exposing `this` of an Akka Actor, it's important not to expose `this` of a Typed Actor, instead you should pass the external proxy reference, which is obtained from within your Typed Actor as `TypedActor.self`, this is your external identity, as the `ActorRef` is the external identity of an Akka Actor.

3.14.3 Creating Typed Actors

To create a Typed Actor you need to have one or more interfaces, and one implementation.

Our example interface:

```
trait Squarer {
  def squareDontCare(i: Int): Unit //fire-forget

  def square(i: Int): Future[Int] //non-blocking send-request-reply

  def squareNowPlease(i: Int): Option[Int] //blocking send-request-reply

  def squareNow(i: Int): Int //blocking send-request-reply

  @throws(classOf[Exception]) //declare it or you will get an␣
↪UndeclaredThrowableException
  def squareTry(i: Int): Int //blocking send-request-reply with possible exception
}
```

Alright, now we've got some methods we can call, but we need to implement those in SquarerImpl.

```
class SquarerImpl(val name: String) extends Squarer {

  def this() = this("default")
  def squareDontCare(i: Int): Unit = i * i //Nobody cares :(

  def square(i: Int): Future[Int] = Future.successful(i * i)

  def squareNowPlease(i: Int): Option[Int] = Some(i * i)

  def squareNow(i: Int): Int = i * i

  def squareTry(i: Int): Int = throw new Exception("Catch me!")
}
```

Excellent, now we have an interface and an implementation of that interface, and we know how to create a Typed Actor from that, so let's look at calling these methods.

The most trivial way of creating a Typed Actor instance of our Squarer:

```
val mySquarer: Squarer =
  TypedActor(system).typedActorOf(TypedProps[SquarerImpl]())
```

First type is the type of the proxy, the second type is the type of the implementation. If you need to call a specific constructor you do it like this:

```
val otherSquarer: Squarer =
  TypedActor(system).typedActorOf(TypedProps(
    classOf[Squarer],
    new SquarerImpl("foo")), "name")
```

Since you supply a Props, you can specify which dispatcher to use, what the default timeout should be used and more.

3.14.4 Method dispatch semantics

Methods returning:

- `Unit` will be dispatched with `fire-and-forget` semantics, exactly like `ActorRef.tell`
- `scala.concurrent.Future[_]` will use `send-request-reply` semantics, exactly like `ActorRef.ask`
- `scala.Option[_]` will use `send-request-reply` semantics, but *will* block to wait for an answer, and return `scala.None` if no answer was produced within the timeout, or `scala.Some[_]` containing the result otherwise. Any exception that was thrown during this call will be rethrown.

- Any other type of value will use `send-request-reply` semantics, but *will* block to wait for an answer, throwing `java.util.concurrent.TimeoutException` if there was a timeout or rethrow any exception that was thrown during this call.

3.14.5 Messages and immutability

While Akka cannot enforce that the parameters to the methods of your Typed Actors are immutable, we *strongly* recommend that parameters passed are immutable.

One-way message send

```
mySquarer.squareDontCare(10)
```

As simple as that! The method will be executed on another thread; asynchronously.

Request-reply message send

```
val oSquare = mySquarer.squareNowPlease(10) //Option[Int]
```

This will block for as long as the timeout that was set in the Props of the Typed Actor, if needed. It will return `None` if a timeout occurs.

```
val iSquare = mySquarer.squareNow(10) //Int
```

This will block for as long as the timeout that was set in the Props of the Typed Actor, if needed. It will throw a `java.util.concurrent.TimeoutException` if a timeout occurs.

Request-reply-with-future message send

```
val fSquare = mySquarer.square(10) //A Future[Int]
```

This call is asynchronous, and the Future returned can be used for asynchronous composition.

3.14.6 Stopping Typed Actors

Since Akka's Typed Actors are backed by Akka Actors they must be stopped when they aren't needed anymore.

```
TypedActor(system).stop(mySquarer)
```

This asynchronously stops the Typed Actor associated with the specified proxy ASAP.

```
TypedActor(system).poisonPill(otherSquarer)
```

This asynchronously stops the Typed Actor associated with the specified proxy after it's done with all calls that were made prior to this call.

3.14.7 Typed Actor Hierarchies

Since you can obtain a contextual Typed Actor Extension by passing in an `ActorContext` you can create child Typed Actors by invoking `typedActorOf(..)` on that:

```
//Inside your Typed Actor
val childSquarer: Squarer =
  TypedActor(TypedActor.context).typedActorOf(TypedProps[SquarerImpl]())
//Use "childSquarer" as a Squarer
```

You can also create a child Typed Actor in regular Akka Actors by giving the `ActorContext` as an input parameter to TypedActor.get(...).

3.14.8 Supervisor Strategy

By having your Typed Actor implementation class implement `TypedActor.Supervisor` you can define the strategy to use for supervising child actors, as described in *Supervision and Monitoring* and *Fault Tolerance*.

3.14.9 Lifecycle callbacks

By having your Typed Actor implementation class implement any and all of the following:

- `TypedActor.PreStart`
- `TypedActor.PostStop`
- `TypedActor.PreRestart`
- `TypedActor.PostRestart`

You can hook into the lifecycle of your Typed Actor.

3.14.10 Receive arbitrary messages

If your implementation class of your TypedActor extends `akka.actor.TypedActor.Receiver`, all messages that are not `MethodCall` instances will be passed into the `onReceive`-method.

This allows you to react to DeathWatch `Terminated`-messages and other types of messages, e.g. when interfacing with untyped actors.

3.14.11 Proxying

You can use the `typedActorOf` that takes a TypedProps and an ActorRef to proxy the given ActorRef as a TypedActor. This is usable if you want to communicate remotely with TypedActors on other machines, just pass the `ActorRef` to `typedActorOf`.

Note: The ActorRef needs to accept `MethodCall` messages.

3.14.12 Lookup & Remoting

Since `TypedActors` are backed by `Akka Actors`, you can use `typedActorOf` to proxy `ActorRefs` potentially residing on remote nodes.

```
val typedActor: Foo with Bar =
  TypedActor(system).
    typedActorOf(
      TypedProps[FooBar],
      actorRefToRemoteActor)
//Use "typedActor" as a FooBar
```

3.14.13 Supercharging

Here's an example on how you can use traits to mix in behavior in your Typed Actors.

```
trait Foo {
  def doFoo(times: Int): Unit = println("doFoo(" + times + ")")
}

trait Bar {
  def doBar(str: String): Future[String] =
    Future.successful(str.toUpperCase)
}

class FooBar extends Foo with Bar
```

```
val awesomeFooBar: Foo with Bar =
  TypedActor(system).typedActorOf(TypedProps[FooBar]())

awesomeFooBar.doFoo(10)
val f = awesomeFooBar.doBar("yes")

TypedActor(system).poisonPill(awesomeFooBar)
```

3.14.14 Typed Router pattern

Sometimes you want to spread messages between multiple actors. The easiest way to achieve this in Akka is to use a *Router*, which can implement a specific routing logic, such as `smallest-mailbox` or `consistent-hashing` etc.

Routers are not provided directly for typed actors, but it is really easy to leverage an untyped router and use a typed proxy in front of it. To showcase this let's create typed actors that assign themselves some random `id`, so we know that in fact, the router has sent the message to different actors:

```
trait HasName {
  def name(): String
}

class Named extends HasName {
  import scala.util.Random
  private val id = Random.nextInt(1024)

  def name(): String = "name-" + id
}
```

In order to round robin among a few instances of such actors, you can simply create a plain untyped router, and then facade it with a `TypedActor` like shown in the example below. This works because typed actors of course communicate using the same mechanisms as normal actors, and methods calls on them get transformed into message sends of `MethodCall` messages.

```
def namedActor(): HasName = TypedActor(system).typedActorOf(TypedProps[Named]())

// prepare routees
val routees: List[HasName] = List.fill(5) { namedActor() }
val routeePaths = routees map { r =>
  TypedActor(system).getActorRefFor(r).path.toStringWithoutAddress
}

// prepare untyped router
val router: ActorRef = system.actorOf(RoundRobinGroup(routeePaths).props())

// prepare typed proxy, forwarding MethodCall messages to `router`
```

```
val typedRouter: HasName =
  TypedActor(system).typedActorOf(TypedProps[Named](), actorRef = router)

println("actor was: " + typedRouter.name()) // name-184
println("actor was: " + typedRouter.name()) // name-753
println("actor was: " + typedRouter.name()) // name-320
println("actor was: " + typedRouter.name()) // name-164
```

CHAPTER

FOUR

FUTURES AND AGENTS

4.1 Futures

4.1.1 Introduction

In the Scala Standard Library, a Future is a data structure used to retrieve the result of some concurrent operation. This result can be accessed synchronously (blocking) or asynchronously (non-blocking).

4.1.2 Execution Contexts

In order to execute callbacks and operations, Futures need something called an `ExecutionContext`, which is very similar to a `java.util.concurrent.Executor`. if you have an `ActorSystem` in scope, it will use its default dispatcher as the `ExecutionContext`, or you can use the factory methods provided by the `ExecutionContext` companion object to wrap `Executors` and `ExecutorServices`, or even create your own.

```
import scala.concurrent.{ ExecutionContext, Promise }

implicit val ec = ExecutionContext.fromExecutorService(yourExecutorServiceGoesHere)

// Do stuff with your brand new shiny ExecutionContext
val f = Promise.successful("foo")

// Then shut your ExecutionContext down at some
// appropriate place in your program/application
ec.shutdown()
```

Within Actors

Each actor is configured to be run on a `MessageDispatcher`, and that dispatcher doubles as an `ExecutionContext`. If the nature of the Future calls invoked by the actor matches or is compatible with the activities of that actor (e.g. all CPU bound and no latency requirements), then it may be easiest to reuse the dispatcher for running the Futures by importing `context.dispatcher`.

```
class A extends Actor {
  import context.dispatcher
  val f = Future("hello")
  def receive = {
    // receive omitted ...
  }
}
```

4.1.3 Use With Actors

There are generally two ways of getting a reply from an `Actor`: the first is by a sent message (`actor ! msg`), which only works if the original sender was an `Actor`) and the second is through a `Future`.

Using an `Actor`'s `?` method to send a message will return a `Future`:

```
import scala.concurrent.Await
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.duration._

implicit val timeout = Timeout(5 seconds)
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

This will cause the current thread to block and wait for the `Actor` to 'complete' the `Future` with it's reply. Blocking is discouraged though as it will cause performance problems. The blocking operations are located in `Await.result` and `Await.ready` to make it easy to spot where blocking occurs. Alternatives to blocking are discussed further within this documentation. Also note that the `Future` returned by an `Actor` is a `Future[Any]` since an `Actor` is dynamic. That is why the `asInstanceOf` is used in the above sample. When using non-blocking it is better to use the `mapTo` method to safely try to cast a `Future` to an expected type:

```
import scala.concurrent.Future
import akka.pattern.ask

val future: Future[String] = ask(actor, msg).mapTo[String]
```

The `mapTo` method will return a new `Future` that contains the result if the cast was successful, or a `ClassCastException` if not. Handling `Exception`s will be discussed further within this documentation.

To send the result of a `Future` to an `Actor`, you can use the `pipe` construct:

```
import akka.pattern.pipe
future pipeTo actor
```

4.1.4 Use Directly

A common use case within Akka is to have some computation performed concurrently without needing the extra utility of an `Actor`. If you find yourself creating a pool of `Actor`s for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
import scala.concurrent.Await
import scala.concurrent.Future
import scala.concurrent.duration._

val future = Future {
  "Hello" + "World"
}
future foreach println
```

In the above code the block passed to `Future` will be executed by the default `Dispatcher`, with the return value of the block used to complete the `Future` (in this case, the result would be the string: "HelloWorld"). Unlike a `Future` that is returned from an `Actor`, this `Future` is properly typed, and we also avoid the overhead of managing an `Actor`.

You can also create already completed Futures using the `Future` companion, which can be either successes:

```
val future = Future.successful("Yay!")
```

Or failures:

```
val otherFuture = Future.failed[String](new IllegalArgumentException("Bang!"))
```

It is also possible to create an empty `Promise`, to be filled later, and obtain the corresponding `Future`:

```
val promise = Promise[String]()
val theFuture = promise.future
promise.success("hello")
```

4.1.5 Functional Futures

Scala's `Future` has several monadic methods that are very similar to the ones used by Scala's collections. These allow you to create 'pipelines' or 'streams' that the result will travel through.

Future is a Monad

The first method for working with `Future` functionally is `map`. This method takes a `Function` which performs some operation on the result of the `Future`, and returning a new result. The return value of the `map` method is another `Future` that will contain the new result:

```
val f1 = Future {
  "Hello" + "World"
}
val f2 = f1 map { x =>
  x.length
}
f2 foreach println
```

In this example we are joining two strings together within a `Future`. Instead of waiting for this to complete, we apply our function that calculates the length of the string using the `map` method. Now we have a second `Future` that will eventually contain an `Int`. When our original `Future` completes, it will also apply our function and complete the second `Future` with its result. When we finally get the result, it will contain the number 10. Our original `Future` still contains the string "HelloWorld" and is unaffected by the `map`.

The `map` method is fine if we are modifying a single `Future`, but if 2 or more `Futures` are involved `map` will not allow you to combine them together:

```
val f1 = Future {
  "Hello" + "World"
}
val f2 = Future.successful(3)
val f3 = f1 map { x =>
  f2 map { y =>
    x.length * y
  }
}
f3 foreach println
```

`f3` is a `Future[Future[Int]]` instead of the desired `Future[Int]`. Instead, the `flatMap` method should be used:

```
val f1 = Future {
  "Hello" + "World"
}
val f2 = Future.successful(3)
val f3 = f1 flatMap { x =>
  f2 map { y =>
    x.length * y
  }
```

```
}
f3 foreach println
```

Composing futures using nested combinators it can sometimes become quite complicated and hard to read, in these cases using Scala's 'for comprehensions' usually yields more readable code. See next section for examples.

If you need to do conditional propagation, you can use `filter`:

```
val future1 = Future.successful(4)
val future2 = future1.filter(_ % 2 == 0)

future2 foreach println

val failedFilter = future1.filter(_ % 2 == 1).recover {
  // When filter fails, it will have a java.util.NoSuchElementException
  case m: NoSuchElementException => 0
}

failedFilter foreach println
```

For Comprehensions

Since `Future` has a `map`, `filter` and `flatMap` method it can be easily used in a 'for comprehension':

```
val f = for {
  a <- Future(10 / 2) // 10 / 2 = 5
  b <- Future(a + 1) //  5 + 1 = 6
  c <- Future(a - 1) //  5 - 1 = 4
  if c > 3 // Future.filter
} yield b * c //  6 * 4 = 24

// Note that the execution of futures a, b, and c
// are not done in parallel.

f foreach println
```

Something to keep in mind when doing this is even though it looks like parts of the above example can run in parallel, each step of the for comprehension is run sequentially. This will happen on separate threads for each step but there isn't much benefit over running the calculations all within a single `Future`. The real benefit comes when the `Future`s are created first, and then combining them together.

Composing Futures

The example for comprehension above is an example of composing `Future`s. A common use case for this is combining the replies of several `Actor`s into a single calculation without resorting to calling `Await.result` or `Await.ready` to block for each result. First an example of using `Await.result`:

```
val f1 = ask(actor1, msg1)
val f2 = ask(actor2, msg2)

val a = Await.result(f1, 3 seconds).asInstanceOf[Int]
val b = Await.result(f2, 3 seconds).asInstanceOf[Int]

val f3 = ask(actor3, (a + b))

val result = Await.result(f3, 3 seconds).asInstanceOf[Int]
```

Warning: `Await.result` and `Await.ready` are provided for exceptional situations where you **must** block, a good rule of thumb is to only use them if you know why you **must** block. For all other cases, use asynchronous composition as described below.

Here we wait for the results from the first 2 `Actors` before sending that result to the third `Actor`. We called `Await.result` 3 times, which caused our little program to block 3 times before getting our final result. Now compare that to this example:

```
val f1 = ask(actor1, msg1)
val f2 = ask(actor2, msg2)

val f3 = for {
  a <- f1.mapTo[Int]
  b <- f2.mapTo[Int]
  c <- ask(actor3, (a + b)).mapTo[Int]
} yield c

f3 foreach println
```

Here we have 2 actors processing a single message each. Once the 2 results are available (note that we don't block to get these results!), they are being added together and sent to a third `Actor`, which replies with a string, which we assign to 'result'.

This is fine when dealing with a known amount of Actors, but can grow unwieldy if we have more than a handful. The `sequence` and `traverse` helper methods can make it easier to handle more complex use cases. Both of these methods are ways of turning, for a subclass `T` of `Traversable`, `T[Future[A]]` into a `Future[T[A]]`. For example:

```
// oddActor returns odd numbers sequentially from 1 as a List[Future[Int]]
val listOfFutures = List.fill(100)(akka.pattern.ask(oddActor, GetNext).mapTo[Int])

// now we have a Future[List[Int]]
val futureList = Future.sequence(listOfFutures)

// Find the sum of the odd numbers
val oddSum = futureList.map(_.sum)
oddSum foreach println
```

To better explain what happened in the example, `Future.sequence` is taking the `List[Future[Int]]` and turning it into a `Future[List[Int]]`. We can then use `map` to work with the `List[Int]` directly, and we find the sum of the `List`.

The `traverse` method is similar to `sequence`, but it takes a `T[A]` and a function `A => Future[B]` to return a `Future[T[B]]`, where `T` is again a subclass of Traversable. For example, to use `traverse` to sum the first 100 odd numbers:

```
val futureList = Future.traverse((1 to 100).toList)(x => Future(x * 2 - 1))
val oddSum = futureList.map(_.sum)
oddSum foreach println
```

This is the same result as this example:

```
val futureList = Future.sequence((1 to 100).toList.map(x => Future(x * 2 - 1)))
val oddSum = futureList.map(_.sum)
oddSum foreach println
```

But it may be faster to use `traverse` as it doesn't have to create an intermediate `List[Future[Int]]`.

Then there's a method that's called `fold` that takes a start-value, a sequence of `Futures` and a function from the type of the start-value and the type of the futures and returns something with the same type as the start-value, and

then applies the function to all elements in the sequence of futures, asynchronously, the execution will start when the last of the Futures is completed.

```
// Create a sequence of Futures
val futures = for (i <- 1 to 1000) yield Future(i * 2)
val futureSum = Future.fold(futures)(0)(_ + _)
futureSum foreach println
```

That's all it takes!

If the sequence passed to `fold` is empty, it will return the start-value, in the case above, that will be 0. In some cases you don't have a start-value and you're able to use the value of the first completing `Future` in the sequence as the start-value, you can use `reduce`, it works like this:

```
// Create a sequence of Futures
val futures = for (i <- 1 to 1000) yield Future(i * 2)
val futureSum = Future.reduce(futures)(_ + _)
futureSum foreach println
```

Same as with `fold`, the execution will be done asynchronously when the last of the `Future` is completed, you can also parallelize it by chunking your futures into sub-sequences and reduce them, and then reduce the reduced results again.

4.1.6 Callbacks

Sometimes you just want to listen to a `Future` being completed, and react to that not by creating a new `Future`, but by side-effecting. For this Scala supports `onComplete`, `onSuccess` and `onFailure`, of which the last two are specializations of the first.

```
future onSuccess {
  case "bar"      => println("Got my bar alright!")
  case x: String => println("Got some random string: " + x)
}
```

```
future onFailure {
  case ise: IllegalStateException if ise.getMessage == "OHNOES" =>
  //OHNOES! We are in deep trouble, do something!
  case e: Exception =>
  //Do something else
}
```

```
future onComplete {
  case Success(result)  => doSomethingOnSuccess(result)
  case Failure(failure) => doSomethingOnFailure(failure)
}
```

4.1.7 Define Ordering

Since callbacks are executed in any order and potentially in parallel, it can be tricky at the times when you need sequential ordering of operations. But there's a solution and it's name is `andThen`. It creates a new `Future` with the specified callback, a `Future` that will have the same result as the `Future` it's called on, which allows for ordering like in the following sample:

```
val result = Future { loadPage(url) } andThen {
  case Failure(exception) => log(exception)
} andThen {
  case _ => watchSomeTV()
}
result foreach println
```

4.1.8 Auxiliary Methods

`Future fallbackTo` combines 2 Futures into a new `Future`, and will hold the successful value of the second `Future` if the first `Future` fails.

```
val future4 = future1 fallbackTo future2 fallbackTo future3
future4 foreach println
```

You can also combine two Futures into a new `Future` that will hold a tuple of the two Futures successful results, using the `zip` operation.

```
val future3 = future1 zip future2 map { case (a, b) => a + " " + b }
future3 foreach println
```

4.1.9 Exceptions

Since the result of a `Future` is created concurrently to the rest of the program, exceptions must be handled differently. It doesn't matter if an `Actor` or the dispatcher is completing the `Future`, if an `Exception` is caught the `Future` will contain it instead of a valid result. If a `Future` does contain an `Exception`, calling `Await.result` will cause it to be thrown again so it can be handled properly.

It is also possible to handle an `Exception` by returning a different result. This is done with the `recover` method. For example:

```
val future = akka.pattern.ask(actor, msg1) recover {
  case e: ArithmeticException => 0
}
future foreach println
```

In this example, if the actor replied with a `akka.actor.Status.Failure` containing the `ArithmeticException`, our `Future` would have a result of 0. The `recover` method works very similarly to the standard try/catch blocks, so multiple `Exceptions` can be handled in this manner, and if an `Exception` is not handled this way it will behave as if we hadn't used the `recover` method.

You can also use the `recoverWith` method, which has the same relationship to `recover` as `flatMap` has to `map`, and is use like this:

```
val future = akka.pattern.ask(actor, msg1) recoverWith {
  case e: ArithmeticException => Future.successful(0)
  case foo: IllegalArgumentException =>
    Future.failed[Int](new IllegalStateException("All br0ken!"))
}
future foreach println
```

4.1.10 After

`akka.pattern.after` makes it easy to complete a `Future` with a value or exception after a timeout.

```
// TODO after is unfortunately shadowed by ScalaTest, fix as part of #3759
// import akka.pattern.after

val delayed = akka.pattern.after(200 millis, using = system.scheduler)(Future.
↪failed(
  new IllegalStateException("OHNOES")))
val future = Future { Thread.sleep(1000); "foo" }
val result = Future firstCompletedOf Seq(future, delayed)
```

4.2 Agents

Agents in Akka are inspired by agents in Clojure.

Agents provide asynchronous change of individual locations. Agents are bound to a single storage location for their lifetime, and only allow mutation of that location (to a new state) to occur as a result of an action. Update actions are functions that are asynchronously applied to the Agent's state and whose return value becomes the Agent's new state. The state of an Agent should be immutable.

While updates to Agents are asynchronous, the state of an Agent is always immediately available for reading by any thread (using `get` or `apply`) without any messages.

Agents are reactive. The update actions of all Agents get interleaved amongst threads in an `ExecutionContext`. At any point in time, at most one `send` action for each Agent is being executed. Actions dispatched to an agent from another thread will occur in the order they were sent, potentially interleaved with actions dispatched to the same agent from other threads.

Note: Agents are local to the node on which they are created. This implies that you should generally not include them in messages that may be passed to remote Actors or as constructor parameters for remote Actors; those remote Actors will not be able to read or update the Agent.

4.2.1 Creating Agents

Agents are created by invoking `Agent(value)` passing in the Agent's initial value and providing an implicit `ExecutionContext` to be used for it, for these examples we're going to use the default global one, but YMMV:

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent
val agent = Agent(5)
```

4.2.2 Reading an Agent's value

Agents can be dereferenced (you can get an Agent's value) by invoking the Agent with parentheses like this:

```
val result = agent()
```

Or by using the get method:

```
val result = agent.get
```

Reading an Agent's current value does not involve any message passing and happens immediately. So while updates to an Agent are asynchronous, reading the state of an Agent is synchronous.

4.2.3 Updating Agents (send & alter)

You update an Agent by sending a function that transforms the current value or by sending just a new value. The Agent will apply the new value or function atomically and asynchronously. The update is done in a fire-forget manner and you are only guaranteed that it will be applied. There is no guarantee of when the update will be applied but dispatches to an Agent from a single thread will occur in order. You apply a value or a function by invoking the `send` function.

```
// send a value, enqueues this change
// of the value of the Agent
agent send 7

// send a function, enqueues this change
```

```
// to the value of the Agent
agent send (_ + 1)
agent send (_ * 2)
```

You can also dispatch a function to update the internal state but on its own thread. This does not use the reactive thread pool and can be used for long-running or blocking operations. You do this with the `sendOff` method. Dispatches using either `sendOff` or `send` will still be executed in order.

```
// the ExecutionContext you want to run the function on
implicit val ec = someExecutionContext()
// sendOff a function
agent sendOff longRunningOrBlockingFunction
```

All `send` methods also have a corresponding `alter` method that returns a `Future`. See *Futures* for more information on `Futures`.

```
// alter a value
val f1: Future[Int] = agent alter 7

// alter a function
val f2: Future[Int] = agent alter (_ + 1)
val f3: Future[Int] = agent alter (_ * 2)
```

```
// the ExecutionContext you want to run the function on
implicit val ec = someExecutionContext()
// alterOff a function
val f4: Future[Int] = agent alterOff longRunningOrBlockingFunction
```

4.2.4 Awaiting an Agent's value

You can also get a `Future` to the Agents value, that will be completed after the currently queued updates have completed:

```
val future = agent.future
```

See *Futures* for more information on `Futures`.

4.2.5 Monadic usage

Agents are also monadic, allowing you to compose operations using for-comprehensions. In monadic usage, new Agents are created leaving the original Agents untouched. So the old values (Agents) are still available as-is. They are so-called 'persistent'.

Example of monadic usage:

```
import scala.concurrent.ExecutionContext.Implicits.global
val agent1 = Agent(3)
val agent2 = Agent(5)

// uses foreach
for (value <- agent1)
  println(value)

// uses map
val agent3 = for (value <- agent1) yield value + 1

// or using map directly
val agent4 = agent1 map (_ + 1)
```

```
// uses flatMap
val agent5 = for {
  value1 <- agent1
  value2 <- agent2
} yield value1 + value2
```

4.2.6 Configuration

There are several configuration properties for the agents module, please refer to the *reference configuration*.

4.2.7 Deprecated Transactional Agents

Agents participating in enclosing STM transaction is a deprecated feature in 2.3.

If an Agent is used within an enclosing transaction, then it will participate in that transaction. If you send to an Agent within a transaction then the dispatch to the Agent will be held until that transaction commits, and discarded if the transaction is aborted. Here's an example:

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent
import scala.concurrent.duration._
import scala.concurrent.stm._

def transfer(from: Agent[Int], to: Agent[Int], amount: Int): Boolean = {
  atomic { txn =>
    if (from.get < amount) false
    else {
      from send (_ - amount)
      to send (_ + amount)
      true
    }
  }
}

val from = Agent(100)
val to = Agent(20)
val ok = transfer(from, to, 50)

val fromValue = from.future // -> 50
val toValue = to.future // -> 70
```

CHAPTER

FIVE

NETWORKING

5.1 Cluster Specification

Note: This document describes the design concepts of the clustering.

5.1.1 Intro

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster *membership* service with no single point of failure or single point of bottleneck. It does this using *gossip* protocols and an automatic *failure detector*.

5.1.2 Terms

node A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a *hostname:port:uid* tuple.

cluster A set of nodes joined together through the *membership* service.

leader A single node in the cluster that acts as the leader. Managing cluster convergence and membership state transitions.

5.1.3 Membership

A cluster is made up of a set of member nodes. The identifier for each node is a `hostname:port:uid` tuple. An Akka application can be distributed over a cluster with each node hosting some part of the application. Cluster membership and the actors running on that node of the application are decoupled. A node could be a member of a cluster without hosting any actors. Joining a cluster is initiated by issuing a `Join` command to one of the nodes in the cluster to join.

The node identifier internally also contains a UID that uniquely identifies this actor system instance at that `hostname:port`. Akka uses the UID to be able to reliably trigger remote death watch. This means that the same actor system can never join a cluster again once it's been removed from that cluster. To re-join an actor system with the same `hostname:port` to a cluster you have to stop the actor system and start a new one with the same `hostname:port` which will then receive a different UID.

The cluster membership state is a specialized CRDT, which means that it has a monotonic merge function. When concurrent changes occur on different nodes the updates can always be merged and converge to the same end result.

Gossip

The cluster membership used in Akka is based on Amazon's Dynamo system and particularly the approach taken in Basho's' Riak distributed database. Cluster membership is communicated using a Gossip Protocol, where the

current state of the cluster is gossiped randomly through the cluster, with preference to members that have not seen the latest version.

Vector Clocks

Vector clocks are a type of data structure and algorithm for generating a partial ordering of events in a distributed system and detecting causality violations.

We use vector clocks to reconcile and merge differences in cluster state during gossiping. A vector clock is a set of (node, counter) pairs. Each update to the cluster state has an accompanying update to the vector clock.

Gossip Convergence

Information about the cluster converges locally at a node at certain points in time. This is when a node can prove that the cluster state he is observing has been observed by all other nodes in the cluster. Convergence is implemented by passing a set of nodes that have seen current state version during gossip. This information is referred to as the seen set in the gossip overview. When all nodes are included in the seen set there is convergence.

Gossip convergence cannot occur while any nodes are `unreachable`. The nodes need to become `reachable` again, or moved to the `down` and `removed` states (see the *Membership Lifecycle* section below). This only blocks the leader from performing its cluster membership management and does not influence the application running on top of the cluster. For example this means that during a network partition it is not possible to add more nodes to the cluster. The nodes can join, but they will not be moved to the `up` state until the partition has healed or the unreachable nodes have been downed.

Failure Detector

The failure detector is responsible for trying to detect if a node is `unreachable` from the rest of the cluster. For this we are using an implementation of The Phi Accrual Failure Detector by Hayashibara et al.

An accrual failure detector decouple monitoring and interpretation. That makes them applicable to a wider area of scenarios and more adequate to build generic failure detection services. The idea is that it is keeping a history of failure statistics, calculated from heartbeats received from other nodes, and is trying to do educated guesses by taking multiple factors, and how they accumulate over time, into account in order to come up with a better guess if a specific node is up or down. Rather than just answering "yes" or "no" to the question "is the node down?" it returns a `phi` value representing the likelihood that the node is down.

The `threshold` that is the basis for the calculation is configurable by the user. A low `threshold` is prone to generate many wrong suspicions but ensures a quick detection in the event of a real crash. Conversely, a high `threshold` generates fewer mistakes but needs more time to detect actual crashes. The default `threshold` is 8 and is appropriate for most situations. However in cloud environments, such as Amazon EC2, the value could be increased to 12 in order to account for network issues that sometimes occur on such platforms.

In a cluster each node is monitored by a few (default maximum 5) other nodes, and when any of these detects the node as `unreachable` that information will spread to the rest of the cluster through the gossip. In other words, only one node needs to mark a node `unreachable` to have the rest of the cluster mark that node `unreachable`.

The nodes to monitor are picked out of neighbors in a hashed ordered node ring. This is to increase the likelihood to monitor across racks and data centers, but the order is the same on all nodes, which ensures full coverage.

Heartbeats are sent out every second and every heartbeat is performed in a request/reply handshake with the replies used as input to the failure detector.

The failure detector will also detect if the node becomes `reachable` again. When all nodes that monitored the `unreachable` node detects it as `reachable` again the cluster, after gossip dissemination, will consider it as `reachable`.

If system messages cannot be delivered to a node it will be quarantined and then it cannot come back from `unreachable`. This can happen if the there are too many unacknowledged system messages (e.g. watch, Terminated, remote actor deployment, failures of actors supervised by remote parent). Then the node needs to be

moved to the `down` or `removed` states (see the *Membership Lifecycle* section below) and the actor system must be restarted before it can join the cluster again.

Leader

After gossip convergence a `leader` for the cluster can be determined. There is no `leader` election process, the `leader` can always be recognised deterministically by any node whenever there is gossip convergence. The leader is just a role, any node can be the leader and it can change between convergence rounds. The `leader` is simply the first node in sorted order that is able to take the leadership role, where the preferred member states for a `leader` are `up` and `leaving` (see the *Membership Lifecycle* section below for more information about member states).

The role of the `leader` is to shift members in and out of the cluster, changing `joining` members to the `up` state or `exiting` members to the `removed` state. Currently `leader` actions are only triggered by receiving a new cluster state with gossip convergence.

The `leader` also has the power, if configured so, to "auto-down" a node that according to the *Failure Detector* is considered `unreachable`. This means setting the `unreachable` node status to `down` automatically after a configured time of unreachability.

Seed Nodes

The seed nodes are configured contact points for new nodes joining the cluster. When a new node is started it sends a message to all seed nodes and then sends a join command to the seed node that answers first.

The seed nodes configuration value does not have any influence on the running cluster itself, it is only relevant for new nodes joining the cluster as it helps them to find contact points to send the join command to; a new member can send this command to any current member of the cluster, not only to the seed nodes.

Gossip Protocol

A variation of *push-pull gossip* is used to reduce the amount of gossip information sent around the cluster. In push-pull gossip a digest is sent representing current versions but not actual values; the recipient of the gossip can then send back any values for which it has newer versions and also request values for which it has outdated versions. Akka uses a single shared state with a vector clock for versioning, so the variant of push-pull gossip used in Akka makes use of this version to only push the actual state as needed.

Periodically, the default is every 1 second, each node chooses another random node to initiate a round of gossip with. If less than ½ of the nodes resides in the seen set (have seen the new state) then the cluster gossips 3 times instead of once every second. This adjusted gossip interval is a way to speed up the convergence process in the early dissemination phase after a state change.

The choice of node to gossip with is random but it is biased to towards nodes that might not have seen the current state version. During each round of gossip exchange when no convergence it uses a probability of 0.8 (configurable) to gossip to a node not part of the seen set, i.e. that probably has an older version of the state. Otherwise gossip to any random live node.

This biased selection is a way to speed up the convergence process in the late dissemination phase after a state change.

For clusters larger than 400 nodes (configurable, and suggested by empirical evidence) the 0.8 probability is gradually reduced to avoid overwhelming single stragglers with too many concurrent gossip requests. The gossip receiver also has a mechanism to protect itself from too many simultaneous gossip messages by dropping messages that have been enqueued in the mailbox for too long time.

While the cluster is in a converged state the gossiper only sends a small gossip status message containing the gossip version to the chosen node. As soon as there is a change to the cluster (meaning non-convergence) then it goes back to biased gossip again.

The recipient of the gossip state or the gossip status can use the gossip version (vector clock) to determine whether:

1. it has a newer version of the gossip state, in which case it sends that back to the gossiper
2. it has an outdated version of the state, in which case the recipient requests the current state from the gossiper by sending back its version of the gossip state
3. it has conflicting gossip versions, in which case the different versions are merged and sent back

If the recipient and the gossip have the same version then the gossip state is not sent or requested.

The periodic nature of the gossip has a nice batching effect of state changes, e.g. joining several nodes quickly after each other to one node will result in only one state change to be spread to other members in the cluster.

The gossip messages are serialized with protobuf and also gzipped to reduce payload size.

Membership Lifecycle

A node begins in the `joining` state. Once all nodes have seen that the new node is joining (through gossip convergence) the `leader` will set the member state to `up`.

If a node is leaving the cluster in a safe, expected manner then it switches to the `leaving` state. Once the leader sees the convergence on the node in the `leaving` state, the leader will then move it to `exiting`. Once all nodes have seen the exiting state (convergence) the `leader` will remove the node from the cluster, marking it as `removed`.

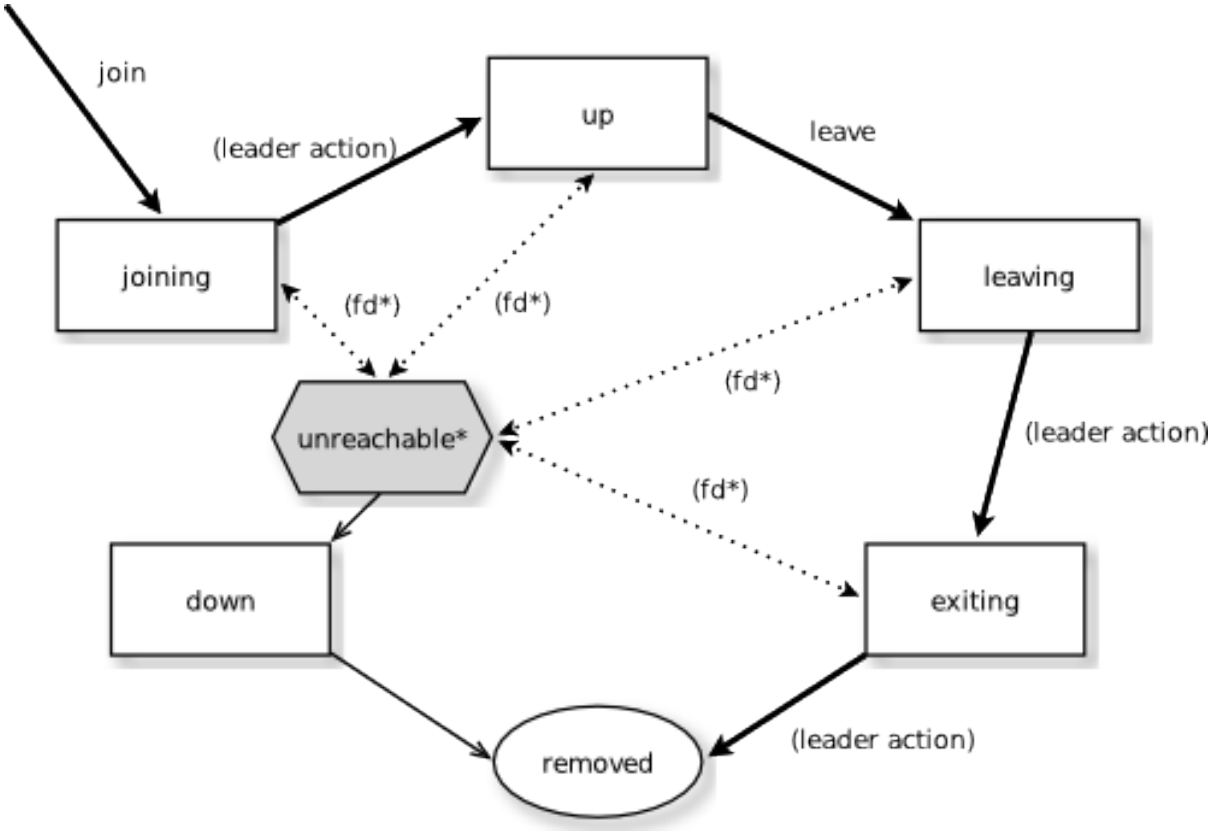
If a node is `unreachable` then gossip convergence is not possible and therefore any `leader` actions are also not possible (for instance, allowing a node to become a part of the cluster). To be able to move forward the state of the `unreachable` nodes must be changed. It must become `reachable` again or marked as `down`. If the node is to join the cluster again the actor system must be restarted and go through the joining process again. The cluster can, through the leader, also *auto-down* a node after a configured time of unreachability. If new incarnation of unreachable node tries to rejoin the cluster old incarnation will be marked as `down` and new incarnation can rejoin the cluster without manual intervention.

Note: If you have *auto-down* enabled and the failure detector triggers, you can over time end up with a lot of single node clusters if you don't put measures in place to shut down nodes that have become `unreachable`. This follows from the fact that the `unreachable` node will likely see the rest of the cluster as `unreachable`, become its own leader and form its own cluster.

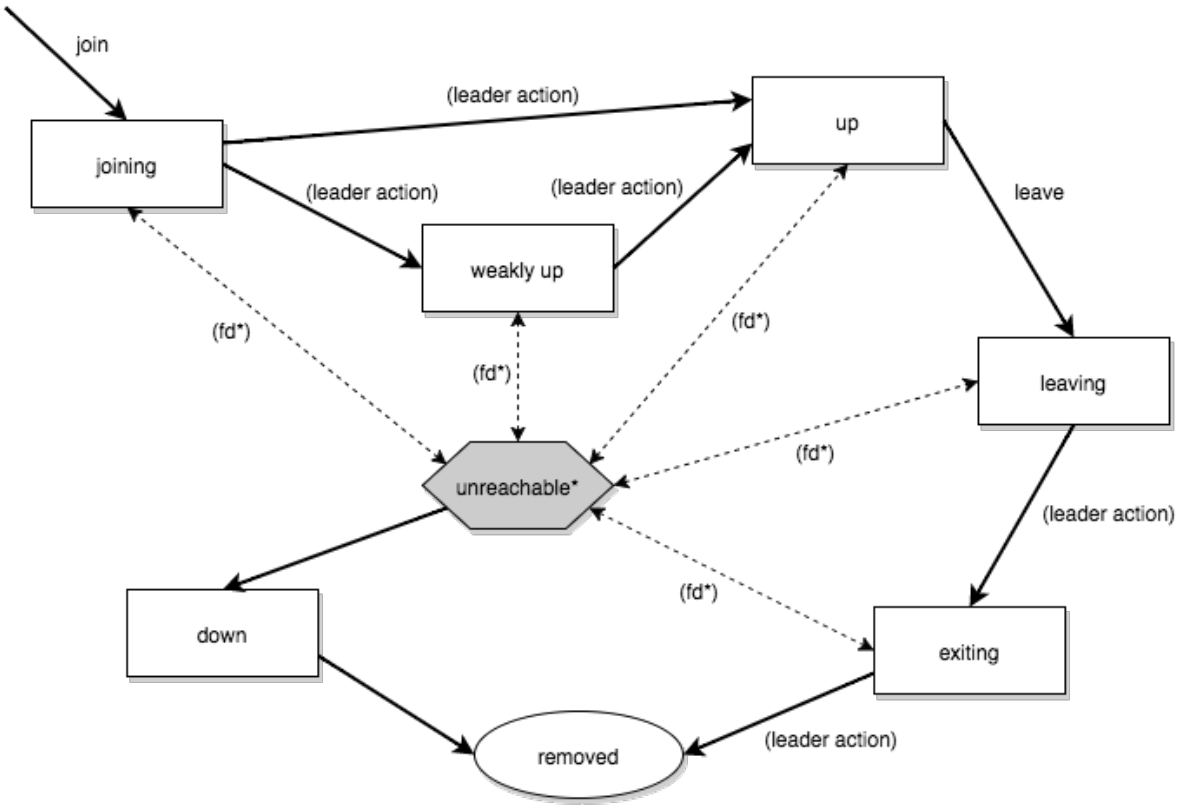
As mentioned before, if a node is `unreachable` then gossip convergence is not possible and therefore any `leader` actions are also not possible. By enabling `akka.cluster.allow-weakly-up-members` it is possible to let new joining nodes be promoted while convergence is not yet reached. These `Joining` nodes will be promoted as `WeaklyUp`. Once gossip convergence is reached, the leader will move `WeaklyUp` members to `Up`.

Note that members on the other side of a network partition have no knowledge about the existence of the new members. You should for example not count `WeaklyUp` members in quorum decisions.

State Diagram for the Member States (`akka.cluster.allow-weakly-up-members=off`)



State Diagram for the Member States (`akka.cluster.allow-weakly-up-members=on`)



Member States

- **joining** transient state when joining a cluster
- **weakly up** transient state while network split (only if `akka.cluster.allow-weakly-up-members=on`)
- **up** normal operating state
- **leaving / exiting** states during graceful removal
- **down** marked as down (no longer part of cluster decisions)
- **removed** tombstone state (no longer a member)

User Actions

- **join** join a single node to a cluster - can be explicit or automatic on startup if a node to join have been specified in the configuration
- **leave** tell a node to leave the cluster gracefully
- **down** mark a node as down

Leader Actions

The `leader` has the following duties:

- shifting members in and out of the cluster
  - joining -> up
  - exiting -> removed

Failure Detection and Unreachability

- **fd*** the failure detector of one of the monitoring nodes has triggered causing the monitored node to be marked as unreachable
- **unreachable*** unreachable is not a real member states but more of a flag in addition to the state signaling that the cluster is unable to talk to this node, after being unreachable the failure detector may detect it as reachable again and thereby remove the flag

5.2 Cluster Usage

For introduction to the Akka Cluster concepts please see *Cluster Specification*.

5.2.1 Preparing Your Project for Clustering

The Akka cluster is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-cluster" % "2.4.14"
```

5.2.2 A Simple Cluster Example

The following configuration enables the `Cluster` extension to be used. It joins the cluster and an actor subscribes to cluster membership events and logs them.

The `application.conf` configuration looks like this:

```
akka {
  actor {
    provider = cluster
  }
  remote {
    log-remote-lifecycle-events = off
    netty.tcp {
      hostname = "127.0.0.1"
      port = 0
    }
  }

  cluster {
    seed-nodes = [
      "akka.tcp://ClusterSystem@127.0.0.1:2551",
      "akka.tcp://ClusterSystem@127.0.0.1:2552"]

    # auto downing is NOT safe for production deployments.
    # you may want to use it during development, read more about it in the docs.
    #
    # auto-down-unreachable-after = 10s
  }
}

# Disable legacy metrics in akka-cluster.
akka.cluster.metrics.enabled=off

# Enable metrics extension in akka-cluster-metrics.
akka.extensions=["akka.cluster.metrics.ClusterMetricsExtension"]

# Sigar native library extract location during tests.
# Note: use per-jvm-instance folder when running multiple jvm on one host.
akka.cluster.metrics.native-library-extract-folder=${user.dir}/target/native
```

To enable cluster capabilities in your Akka project you should, at a minimum, add the *Remoting* settings, but with `cluster`. The `akka.cluster.seed-nodes` should normally also be added to your `application.conf` file.

Note: If you are running Akka in a Docker container or the nodes for some other reason have separate internal and external ip addresses you must configure remoting according to *Akka behind NAT or in a Docker container*

The seed nodes are configured contact points for initial, automatic, join of the cluster.

Note that if you are going to start the nodes on different machines you need to specify the ip-addresses or host names of the machines in `application.conf` instead of `127.0.0.1`

An actor that uses the cluster extension may look like this:

```
package sample.cluster.simple

import akka.cluster.Cluster
import akka.cluster.ClusterEvent._
import akka.actor.ActorLogging
import akka.actor.Actor
```

```
class SimpleClusterListener extends Actor with ActorLogging {

  val cluster = Cluster(context.system)

  // subscribe to cluster changes, re-subscribe when restart
  override def preStart(): Unit = {
    //#subscribe
    cluster.subscribe(self, initialStateMode = InitialStateAsEvents,
      classOf[MemberEvent], classOf[UnreachableMember])
    //#subscribe
  }
  override def postStop(): Unit = cluster.unsubscribe(self)

  def receive = {
    case MemberUp(member) =>
      log.info("Member is Up: {}", member.address)
    case UnreachableMember(member) =>
      log.info("Member detected as unreachable: {}", member)
    case MemberRemoved(member, previousStatus) =>
      log.info("Member is Removed: {} after {}",
        member.address, previousStatus)
    case _: MemberEvent => // ignore
  }
}
```

The actor registers itself as subscriber of certain cluster events. It receives events corresponding to the current state of the cluster when the subscription starts and then it receives events for changes that happen in the cluster.

The easiest way to run this example yourself is to download Lightbend Activator and open the tutorial named Akka Cluster Samples with Scala. It contains instructions of how to run the `SimpleClusterApp`.

5.2.3 Joining to Seed Nodes

You may decide if joining to the cluster should be done manually or automatically to configured initial contact points, so-called seed nodes. When a new node is started it sends a message to all seed nodes and then sends join command to the one that answers first. If no one of the seed nodes replied (might not be started yet) it retries this procedure until successful or shutdown.

You define the seed nodes in the *Configuration* file (application.conf):

```
akka.cluster.seed-nodes = [
  "akka.tcp://ClusterSystem@host1:2552",
  "akka.tcp://ClusterSystem@host2:2552"]
```

This can also be defined as Java system properties when starting the JVM using the following syntax:

```
-Dakka.cluster.seed-nodes.0=akka.tcp://ClusterSystem@host1:2552
-Dakka.cluster.seed-nodes.1=akka.tcp://ClusterSystem@host2:2552
```

The seed nodes can be started in any order and it is not necessary to have all seed nodes running, but the node configured as the first element in the `seed-nodes` configuration list must be started when initially starting a cluster, otherwise the other seed-nodes will not become initialized and no other node can join the cluster. The reason for the special first seed node is to avoid forming separated islands when starting from an empty cluster. It is quickest to start all configured seed nodes at the same time (order doesn't matter), otherwise it can take up to the configured `seed-node-timeout` until the nodes can join.

Once more than two seed nodes have been started it is no problem to shut down the first seed node. If the first seed node is restarted, it will first try to join the other seed nodes in the existing cluster.

If you don't configure seed nodes you need to join the cluster programmatically or manually.

Manual joining can be performed by using ref:*cluster_jmx_scala* or *Command Line Management*. Joining programmatically can be performed with `Cluster(system).join`. Unsuccessful join attempts are automatically retried after the time period defined in configuration property `retry-unsuccessful-join-after`. Retries can be disabled by setting the property to `off`.

You can join to any node in the cluster. It does not have to be configured as a seed node. Note that you can only join to an existing cluster member, which means that for bootstrapping some node must join itself,and then the following nodes could join them to make up a cluster.

You may also use `Cluster(system).joinSeedNodes` to join programmatically, which is attractive when dynamically discovering other nodes at startup by using some external tool or API. When using `joinSeedNodes` you should not include the node itself except for the node that is supposed to be the first seed node, and that should be placed first in parameter to `joinSeedNodes`.

Unsuccessful attempts to contact seed nodes are automatically retried after the time period defined in configuration property `seed-node-timeout`. Unsuccessful attempt to join a specific seed node is automatically retried after the configured `retry-unsuccessful-join-after`. Retrying means that it tries to contact all seed nodes and then joins the node that answers first. The first node in the list of seed nodes will join itself if it cannot contact any of the other seed nodes within the configured `seed-node-timeout`.

An actor system can only join a cluster once. Additional attempts will be ignored. When it has successfully joined it must be restarted to be able to join another cluster or to join the same cluster again.It can use the same host name and port after the restart, when it come up as new incarnation of existing member in the cluster, trying to join in, then the existing one will be removed from the cluster and then it will be allowed to join.

Note: The name of the `ActorSystem` must be the same for all members of a cluster. The name is given when you start the `ActorSystem`.

5.2.4 Downing

When a member is considered by the failure detector to be unreachable the leader is not allowed to perform its duties, such as changing status of new joining members to 'Up'. The node must first become reachable again, or the status of the unreachable member must be changed to 'Down'. Changing status to 'Down' can be performed automatically or manually. By default it must be done manually, using *JMX* or *Command Line Management*.

It can also be performed programmatically with `Cluster(system).down(address)`.

A pre-packaged solution for the downing problem is provided by Split Brain Resolver, which is part of the Lightbend Reactive Platform. If you don't use RP, you should anyway carefully read the documentation of the Split Brain Resolver and make sure that the solution you are using handles the concerns described there.

Auto-downing (DO NOT USE)

There is an automatic downing feature that you should not use in production. For testing purpose you can enable it with configuration:

```
akka.cluster.auto-down-unreachable-after = 120s
```

This means that the cluster leader member will change the `unreachable` node status to `down` automatically after the configured time of unreachability.

This is a naïve approach to remove unreachable nodes from the cluster membership. It works great for crashes and short transient network partitions, but not for long network partitions. Both sides of the network partition will see the other side as unreachable and after a while remove it from its cluster membership. Since this happens on both sides the result is that two separate disconnected clusters have been created. This can also happen because of long GC pauses or system overload.

Warning: We recommend against using the auto-down feature of Akka Cluster in production. This is crucial for correct behavior if you use *Cluster Singleton* or *Cluster Sharding*, especially together with Akka *Persistence*. For Akka Persistence with Cluster Sharding it can result in corrupt data in case of network partitions.

5.2.5 Leaving

There are two ways to remove a member from the cluster.

You can just stop the actor system (or the JVM process). It will be detected as unreachable and removed after the automatic or manual downing as described above.

A more graceful exit can be performed if you tell the cluster that a node shall leave. This can be performed using *JMX* or *Command Line Management*. It can also be performed programmatically with:

```
val cluster = Cluster(system)
cluster.leave(cluster.selfAddress)
```

Note that this command can be issued to any member in the cluster, not necessarily the one that is leaving. The cluster extension, but not the actor system or JVM, of the leaving member will be shutdown after the leader has changed status of the member to *Exiting*. Thereafter the member will be removed from the cluster. Normally this is handled automatically, but in case of network failures during this process it might still be necessary to set the node's status to `Down` in order to complete the removal.

5.2.6 WeaklyUp Members

If a node is `unreachable` then gossip convergence is not possible and therefore any `leader` actions are also not possible. However, we still might want new nodes to join the cluster in this scenario.

Warning: The WeaklyUp feature is marked as **"experimental"** as of its introduction in Akka 2.4.0. We will continue to improve this feature based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply this feature.

This feature is disabled by default. With a configuration option you can allow this behavior:

```
akka.cluster.allow-weakly-up-members = on
```

When `allow-weakly-up-members` is enabled and there is no gossip convergence, `Joining` members will be promoted to `WeaklyUp` and they will become part of the cluster. Once gossip convergence is reached, the leader will move `WeaklyUp` members to `Up`.

You can subscribe to the `WeaklyUp` membership event to make use of the members that are in this state, but you should be aware of that members on the other side of a network partition have no knowledge about the existence of the new members. You should for example not count `WeaklyUp` members in quorum decisions.

Warning: This feature is only available from Akka 2.4.0 and cannot be used if some of your cluster members are running an older version of Akka.

5.2.7 Subscribe to Cluster Events

You can subscribe to change notifications of the cluster membership by using `Cluster(system).subscribe`.

```
cluster.subscribe(self, classOf[MemberEvent], classOf[UnreachableMember])
```

A snapshot of the full state, `akka.cluster.ClusterEvent.CurrentClusterState`, is sent to the subscriber as the first message, followed by events for incremental updates.

Note that you may receive an empty `CurrentClusterState`, containing no members, if you start the subscription before the initial join procedure has completed. This is expected behavior. When the node has been accepted in the cluster you will receive `MemberUp` for that node, and other nodes.

If you find it inconvenient to handle the `CurrentClusterState` you can use `ClusterEvent.InitialStateAsEvents` as parameter to `subscribe`. That means that instead of receiving `CurrentClusterState` as the first message you will receive the events corresponding to the current state to mimic what you would have seen if you were listening to the events when they occurred in the past. Note that those initial events only correspond to the current state and it is not the full history of all changes that actually has occurred in the cluster.

```
cluster.subscribe(self, initialStateMode = InitialStateAsEvents,
  classOf[MemberEvent], classOf[UnreachableMember])
```

The events to track the life-cycle of members are:

- `ClusterEvent.MemberJoined` - A new member has joined the cluster and its status has been changed to `Joining`.
- `ClusterEvent.MemberUp` - A new member has joined the cluster and its status has been changed to `Up`.
- `ClusterEvent.MemberExited` - A member is leaving the cluster and its status has been changed to `Exiting` Note that the node might already have been shutdown when this event is published on another node.
- `ClusterEvent.MemberRemoved` - Member completely removed from the cluster.
- `ClusterEvent.UnreachableMember` - A member is considered as unreachable, detected by the failure detector of at least one other node.
- `ClusterEvent.ReachableMember` - A member is considered as reachable again, after having been unreachable. All nodes that previously detected it as unreachable has detected it as reachable again.

There are more types of change events, consult the API documentation of classes that extends `akka.cluster.ClusterEvent.ClusterDomainEvent` for details about the events.

Instead of subscribing to cluster events it can sometimes be convenient to only get the full membership state with `Cluster(system).state`. Note that this state is not necessarily in sync with the events published to a cluster subscription.

Worker Dial-in Example

Let's take a look at an example that illustrates how workers, here named *backend*, can detect and register to new master nodes, here named *frontend*.

The example application provides a service to transform text. When some text is sent to one of the frontend services, it will be delegated to one of the backend workers, which performs the transformation job, and sends the result back to the original client. New backend nodes, as well as new frontend nodes, can be added or removed to the cluster dynamically.

Messages:

```
final case class TransformationJob(text: String)
final case class TransformationResult(text: String)
final case class JobFailed(reason: String, job: TransformationJob)
case object BackendRegistration
```

The backend worker that performs the transformation job:

```
class TransformationBackend extends Actor {

  val cluster = Cluster(context.system)

  // subscribe to cluster changes, MemberUp
  // re-subscribe when restart
  override def preStart(): Unit = cluster.subscribe(self, classOf[MemberUp])
  override def postStop(): Unit = cluster.unsubscribe(self)

  def receive = {
    case TransformationJob(text) => sender() ! TransformationResult(text.
↪toUpperCase)
    case state: CurrentClusterState =>
      state.members.filter(_.status == MemberStatus.Up) foreach register
    case MemberUp(m) => register(m)
  }

  def register(member: Member): Unit =
    if (member.hasRole("frontend"))
      context.actorSelection(RootActorPath(member.address) / "user" / "frontend") !
        BackendRegistration
}
```

Note that the `TransformationBackend` actor subscribes to cluster events to detect new, potential, frontend nodes, and send them a registration message so that they know that they can use the backend worker.

The frontend that receives user jobs and delegates to one of the registered backend workers:

```
class TransformationFrontend extends Actor {

  var backends = IndexedSeq.empty[ActorRef]
  var jobCounter = 0

  def receive = {
    case job: TransformationJob if backends.isEmpty =>
      sender() ! JobFailed("Service unavailable, try again later", job)

    case job: TransformationJob =>
      jobCounter += 1
      backends(jobCounter % backends.size) forward job

    case BackendRegistration if !backends.contains(sender()) =>
      context watch sender()
      backends = backends :+ sender()

    case Terminated(a) =>
      backends = backends.filterNot(_ == a)
  }
}
```

Note that the `TransformationFrontend` actor watch the registered backend to be able to remove it from its list of available backend workers. Death watch uses the cluster failure detector for nodes in the cluster, i.e. it detects network failures and JVM crashes, in addition to graceful termination of watched actor. Death watch generates the `Terminated` message to the watching actor when the unreachable cluster node has been downed and removed.

The Lightbend Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Worker Dial-in Example**.

5.2.8 Node Roles

Not all nodes of a cluster need to perform the same function: there might be one sub-set which runs the web front-end, one which runs the data access layer and one for the number-crunching. Deployment of actors—for example by cluster-aware routers—can take node roles into account to achieve this distribution of responsibilities.

The roles of a node is defined in the configuration property named `akka.cluster.roles` and it is typically defined in the start script as a system property or environment variable.

The roles of the nodes is part of the membership information in `MemberEvent` that you can subscribe to.

5.2.9 How To Startup when Cluster Size Reached

A common use case is to start actors after the cluster has been initialized, members have joined, and the cluster has reached a certain size.

With a configuration option you can define required number of members before the leader changes member status of 'Joining' members to 'Up'.

```
akka.cluster.min-nr-of-members = 3
```

In a similar way you can define required number of members of a certain role before the leader changes member status of 'Joining' members to 'Up'.

```
akka.cluster.role {
  frontend.min-nr-of-members = 1
  backend.min-nr-of-members = 2
}
```

You can start the actors in a `registerOnMemberUp` callback, which will be invoked when the current member status is changed to 'Up', i.e. the cluster has at least the defined number of members.

```
Cluster(system) registerOnMemberUp {
  system.actorOf(Props(classOf[FactorialFrontend], upToN, true),
    name = "factorialFrontend")
}
```

This callback can be used for other things than starting actors.

5.2.10 How To Cleanup when Member is Removed

You can do some clean up in a `registerOnMemberRemoved` callback, which will be invoked when the current member status is changed to 'Removed' or the cluster have been shutdown.

For example, this is how to shut down the `ActorSystem` and thereafter exit the JVM:

```
Cluster(system).registerOnMemberRemoved {
  // exit JVM when ActorSystem has been terminated
  system.registerOnTermination(System.exit(0))
  // shut down ActorSystem
  system.terminate()

  // In case ActorSystem shutdown takes longer than 10 seconds,
  // exit the JVM forcefully anyway.
  // We must spawn a separate thread to not block current thread,
  // since that would have blocked the shutdown of the ActorSystem.
  new Thread {
    override def run(): Unit = {
      if (Try(Await.ready(system.whenTerminated, 10.seconds)).isFailure)
        System.exit(-1)
    }
```

```
  }.start()
}
```

Note: Register a OnMemberRemoved callback on a cluster that have been shutdown, the callback will be invoked immediately on the caller thread, otherwise it will be invoked later when the current member status changed to 'Removed'. You may want to install some cleanup handling after the cluster was started up, but the cluster might already be shutting down when you installing, and depending on the race is not healthy.

5.2.11 Cluster Singleton

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

This can be implemented by subscribing to member events, but there are several corner cases to consider. Therefore, this specific use case is made easily accessible by the *Cluster Singleton*.

5.2.12 Cluster Sharding

Distributes actors across several nodes in the cluster and supports interaction with the actors using their logical identifier, but without having to care about their physical location in the cluster.

See *Cluster Sharding*

5.2.13 Distributed Publish Subscribe

Publish-subscribe messaging between actors in the cluster, and point-to-point messaging using the logical path of the actors, i.e. the sender does not have to know on which node the destination actor is running.

See *Distributed Publish Subscribe in Cluster*.

5.2.14 Cluster Client

Communication from an actor system that is not part of the cluster to actors running somewhere in the cluster. The client does not have to know on which node the destination actor is running.

See *Cluster Client*.

5.2.15 Distributed Data

Akka Distributed Data is useful when you need to share data between nodes in an Akka Cluster. The data is accessed with an actor providing a key-value store like API.

See *Distributed Data*.

5.2.16 Failure Detector

In a cluster each node is monitored by a few (default maximum 5) other nodes, and when any of these detects the node as `unreachable` that information will spread to the rest of the cluster through the gossip. In other words, only one node needs to mark a node `unreachable` to have the rest of the cluster mark that node `unreachable`.

The failure detector will also detect if the node becomes `reachable` again. When all nodes that monitored the `unreachable` node detects it as `reachable` again the cluster, after gossip dissemination, will consider it as `reachable`.

If system messages cannot be delivered to a node it will be quarantined and then it cannot come back from `unreachable`. This can happen if the there are too many unacknowledged system messages (e.g. watch, Terminated, remote actor deployment, failures of actors supervised by remote parent). Then the node needs to be moved to the `down` or `removed` states and the actor system of the quarantined node must be restarted before it can join the cluster again.

The nodes in the cluster monitor each other by sending heartbeats to detect if a node is unreachable from the rest of the cluster. The heartbeat arrival times is interpreted by an implementation of The Phi Accrual Failure Detector.

The suspicion level of failure is given by a value called *phi*. The basic idea of the phi failure detector is to express the value of *phi* on a scale that is dynamically adjusted to reflect current network conditions.

The value of *phi* is calculated as:

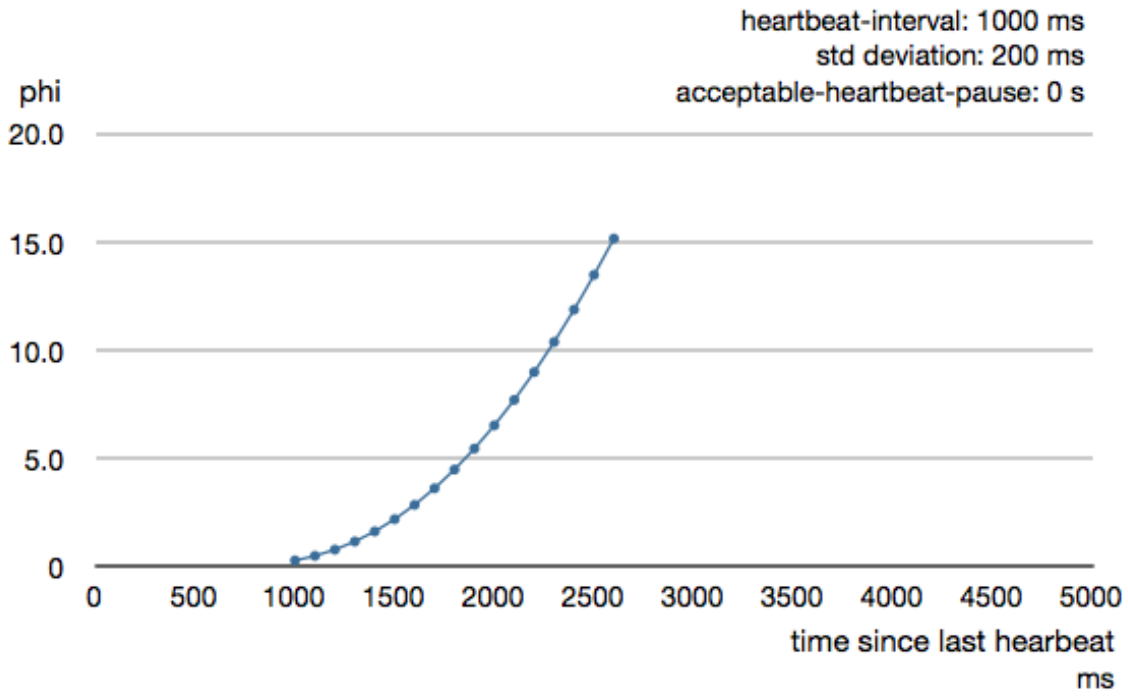
```
phi = -log10(1 - F(timeSinceLastHeartbeat))
```

where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times.

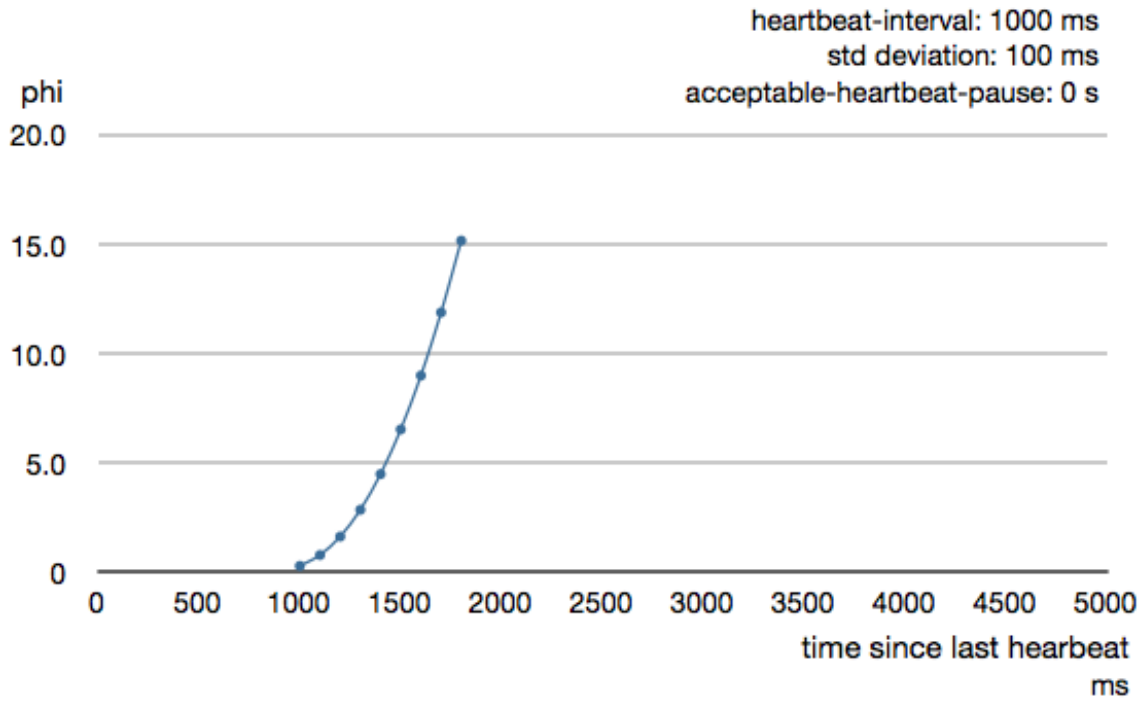
In the *Configuration* you can adjust the `akka.cluster.failure-detector.threshold` to define when a *phi* value is considered to be a failure.

A low `threshold` is prone to generate many false positives but ensures a quick detection in the event of a real crash. Conversely, a high `threshold` generates fewer mistakes but needs more time to detect actual crashes. The default `threshold` is 8 and is appropriate for most situations. However in cloud environments, such as Amazon EC2, the value could be increased to 12 in order to account for network issues that sometimes occur on such platforms.

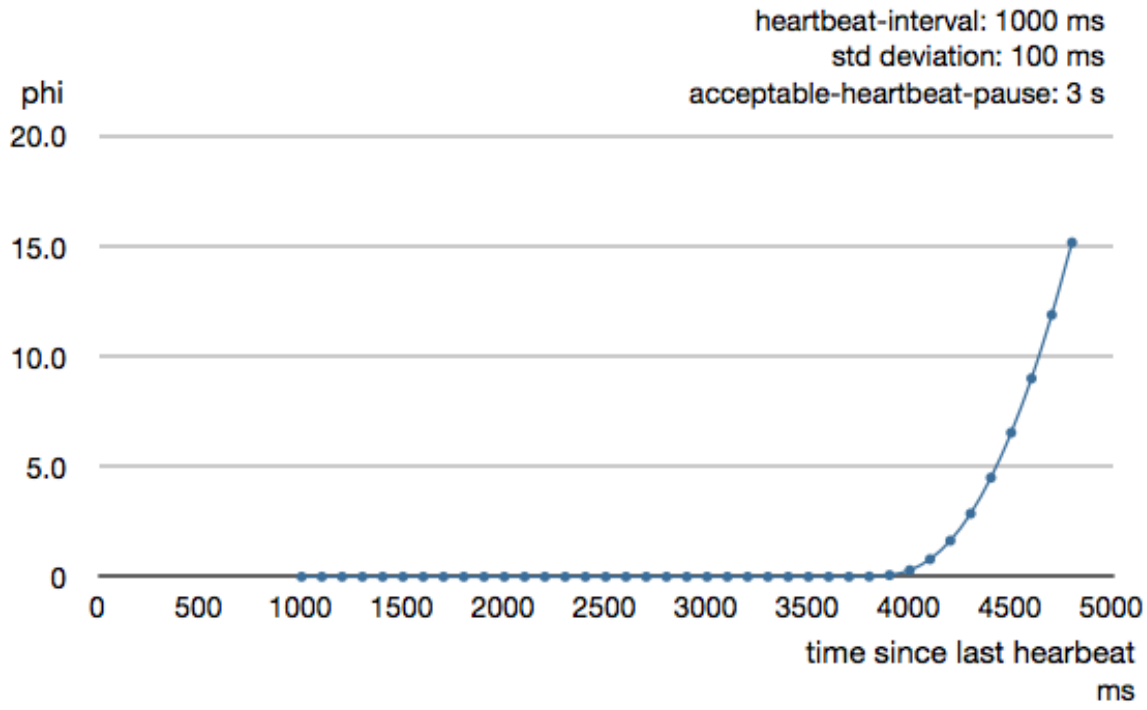
The following chart illustrates how *phi* increase with increasing time since the previous heartbeat.



Phi is calculated from the mean and standard deviation of historical inter arrival times. The previous chart is an example for standard deviation of 200 ms. If the heartbeats arrive with less deviation the curve becomes steeper, i.e. it is possible to determine failure more quickly. The curve looks like this for a standard deviation of 100 ms.



To be able to survive sudden abnormalities, such as garbage collection pauses and transient network failures the failure detector is configured with a margin, `akka.cluster.failure-detector.acceptable-heartbeat-pause`. You may want to adjust the *Configuration* of this depending on you environment. This is how the curve looks like for `acceptable-heartbeat-pause` configured to 3 seconds.



Death watch uses the cluster failure detector for nodes in the cluster, i.e. it detects network failures and JVM crashes, in addition to graceful termination of watched actor. Death watch generates the `Terminated` message to the watching actor when the unreachable cluster node has been downed and removed.

If you encounter suspicious false positives when the system is under load you should define a separate dispatcher for the cluster actors as described in *Cluster Dispatcher*.

5.2.17 Cluster Aware Routers

All *routers* can be made aware of member nodes in the cluster, i.e. deploying new routees or looking up routees on nodes in the cluster. When a node becomes unreachable or leaves the cluster the routees of that node are automatically unregistered from the router. When new nodes join the cluster, additional routees are added to the router, according to the configuration. Routees are also added when a node becomes reachable again, after having been unreachable.

Cluster aware routers make use of members with status *WeaklyUp* if that feature is enabled.

There are two distinct types of routers.

- **Group - router that sends messages to the specified path using actor selection** The routees can be shared among routers running on different nodes in the cluster. One example of a use case for this type of router is a service running on some backend nodes in the cluster and used by routers running on front-end nodes in the cluster.
- **Pool - router that creates routees as child actors and deploys them on remote nodes.** Each router will have its own routee instances. For example, if you start a router on 3 nodes in a 10-node cluster, you will have 30 routees in total if the router is configured to use one instance per node. The routees created by the different routers will not be shared among the routers. One example of a use case for this type of router is a single master that coordinates jobs and delegates the actual work to routees running on other nodes in the cluster.

Router with Group of Routees

When using a `Group` you must start the routee actors on the cluster member nodes. That is not done by the router. The configuration for a group looks like this:

```
akka.actor.deployment {
  /statsService/workerRouter {
      router = consistent-hashing-group
      routees.paths = ["/user/statsWorker"]
      cluster {
        enabled = on
        allow-local-routees = on
        use-role = compute
      }
    }
}
```

Note: The routee actors should be started as early as possible when starting the actor system, because the router will try to use them as soon as the member status is changed to 'Up'.

The actor paths without address information that are defined in `routees.paths` are used for selecting the actors to which the messages will be forwarded to by the router. Messages will be forwarded to the routees using *ActorSelection*, so the same delivery semantics should be expected. It is possible to limit the lookup of routees to member nodes tagged with a certain role by specifying `use-role`.

`max-total-nr-of-instances` defines total number of routees in the cluster. By default `max-total-nr-of-instances` is set to a high value (10000) that will result in new routees added to the router when nodes join the cluster. Set it to a lower value if you want to limit total number of routees.

The same type of router could also have been defined in code:

```
import akka.cluster.routing.ClusterRouterGroup
import akka.cluster.routing.ClusterRouterGroupSettings
import akka.routing.ConsistentHashingGroup

val workerRouter = context.actorOf(
  ClusterRouterGroup(ConsistentHashingGroup(Nil), ClusterRouterGroupSettings(
    totalInstances = 100, routeesPaths = List("/user/statsWorker"),
    allowLocalRoutees = true, useRole = Some("compute"))).props(),
  name = "workerRouter2")
```

See *Configuration* section for further descriptions of the settings.

Router Example with Group of Routees

Let's take a look at how to use a cluster aware router with a group of routees, i.e. router sending to the paths of the routees.

The example application provides a service to calculate statistics for a text. When some text is sent to the service it splits it into words, and delegates the task to count number of characters in each word to a separate worker, a routee of a router. The character count for each word is sent back to an aggregator that calculates the average number of characters per word when all results have been collected.

Messages:

```
final case class StatsJob(text: String)
final case class StatsResult(meanWordLength: Double)
final case class JobFailed(reason: String)
```

The worker that counts number of characters in each word:

```
class StatsWorker extends Actor {
  var cache = Map.empty[String, Int]
  def receive = {
    case word: String =>
      val length = cache.get(word) match {
        case Some(x) => x
        case None =>
          val x = word.length
          cache += (word -> x)
          x
      }

      sender() ! length
  }
}
```

The service that receives text from users and splits it up into words, delegates to workers and aggregates:

```
class StatsService extends Actor {
  // This router is used both with lookup and deploy of routees. If you
  // have a router with only lookup of routees you can use Props.empty
  // instead of Props[StatsWorker.class].
  val workerRouter = context.actorOf(FromConfig.props(Props[StatsWorker]),
    name = "workerRouter")

  def receive = {
    case StatsJob(text) if text != "" =>
      val words = text.split(" ")
      val replyTo = sender() // important to not close over sender()
      // create actor that collects replies from workers
      val aggregator = context.actorOf(Props(
        classOf[StatsAggregator], words.size, replyTo))
```

```
      words foreach { word =>
        workerRouter.tell(
          ConsistentHashableEnvelope(word, word), aggregator)
      }
  }
}

class StatsAggregator(expectedResults: Int, replyTo: ActorRef) extends Actor {
  var results = IndexedSeq.empty[Int]
  context.setReceiveTimeout(3.seconds)

  def receive = {
    case wordCount: Int =>
      results = results :+ wordCount
      if (results.size == expectedResults) {
        val meanWordLength = results.sum.toDouble / results.size
        replyTo ! StatsResult(meanWordLength)
        context.stop(self)
      }
    case ReceiveTimeout =>
      replyTo ! JobFailed("Service unavailable, try again later")
      context.stop(self)
  }
}
```

Note, nothing cluster specific so far, just plain actors.

All nodes start `StatsService` and `StatsWorker` actors. Remember, routees are the workers in this case. The router is configured with `routees.paths`:

```
akka.actor.deployment {
  /statsService/workerRouter {
    router = consistent-hashing-group
    routees.paths = ["/user/statsWorker"]
    cluster {
      enabled = on
      allow-local-routees = on
      use-role = compute
    }
  }
}
```

This means that user requests can be sent to `StatsService` on any node and it will use `StatsWorker` on all nodes.

The Lightbend Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Router Example with Group of Routees**.

Router with Pool of Remote Deployed Routees

When using a `Pool` with routees created and deployed on the cluster member nodes the configuration for a router looks like this:

```
akka.actor.deployment {
  /statsService/singleton/workerRouter {
      router = consistent-hashing-pool
      cluster {
        enabled = on
        max-nr-of-instances-per-node = 3
        allow-local-routees = on
        use-role = compute
      }
```

```
    }
}
```

It is possible to limit the deployment of routees to member nodes tagged with a certain role by specifying `use-role`.

`max-total-nr-of-instances` defines total number of routees in the cluster, but the number of routees per node, `max-nr-of-instances-per-node`, will not be exceeded. By default `max-total-nr-of-instances` is set to a high value (10000) that will result in new routees added to the router when nodes join the cluster. Set it to a lower value if you want to limit total number of routees.

The same type of router could also have been defined in code:

```
import akka.cluster.routing.ClusterRouterPool
import akka.cluster.routing.ClusterRouterPoolSettings
import akka.routing.ConsistentHashingPool

val workerRouter = context.actorOf(
  ClusterRouterPool(ConsistentHashingPool(0), ClusterRouterPoolSettings(
    totalInstances = 100, maxInstancesPerNode = 3,
    allowLocalRoutees = false, useRole = None)).props(Props[StatsWorker]),
  name = "workerRouter3")
```

See *Configuration* section for further descriptions of the settings.

Router Example with Pool of Remote Deployed Routees

Let's take a look at how to use a cluster aware router on single master node that creates and deploys workers. To keep track of a single master we use the *Cluster Singleton* in the contrib module. The `ClusterSingletonManager` is started on each node.

```
system.actorOf(ClusterSingletonManager.props(
  singletonProps = Props[StatsService],
  terminationMessage = PoisonPill,
  settings = ClusterSingletonManagerSettings(system).withRole("compute")),
  name = "statsService")
```

We also need an actor on each node that keeps track of where current single master exists and delegates jobs to the `StatsService`. That is provided by the `ClusterSingletonProxy`.

```
system.actorOf(ClusterSingletonProxy.props(singletonManagerPath = "/user/
↪statsService",
  settings = ClusterSingletonProxySettings(system).withRole("compute")),
  name = "statsServiceProxy")
```

The `ClusterSingletonProxy` receives text from users and delegates to the current `StatsService`, the single master. It listens to cluster events to lookup the `StatsService` on the oldest node.

All nodes start `ClusterSingletonProxy` and the `ClusterSingletonManager`. The router is now configured like this:

```
akka.actor.deployment {
  /statsService/singleton/workerRouter {
    router = consistent-hashing-pool
    cluster {
      enabled = on
      max-nr-of-instances-per-node = 3
      allow-local-routees = on
      use-role = compute
    }
  }
}
```

The Lightbend Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Router Example with Pool of Remote Deployed Routees**.

5.2.18 Cluster Metrics

The member nodes of the cluster can collect system health metrics and publish that to other cluster nodes and to the registered subscribers on the system event bus with the help of *Cluster Metrics Extension*.

5.2.19 How to Test

Multi Node Testing is useful for testing cluster applications.

Set up your project according to the instructions in *Multi Node Testing* and *Multi JVM Testing*, i.e. add the `sbt-multi-jvm` plugin and the dependency to `akka-multi-node-testkit`.

First, as described in *Multi Node Testing*, we need some scaffolding to configure the `MultiNodeSpec`. Define the participating roles and their *Configuration* in an object extending `MultiNodeConfig`:

```
import akka.remote.testkit.MultiNodeConfig
import com.typesafe.config.ConfigFactory

object StatsSampleSpecConfig extends MultiNodeConfig {
  // register the named roles (nodes) of the test
  val first = role("first")
  val second = role("second")
  val third = role("thrid")

  def nodeList = Seq(first, second, third)

  // Extract individual sigar library for every node.
  nodeList foreach { role =>
    nodeConfig(role) {
      ConfigFactory.parseString(s"""
      # Disable legacy metrics in akka-cluster.
      akka.cluster.metrics.enabled=off
      # Enable metrics extension in akka-cluster-metrics.
      akka.extensions=["akka.cluster.metrics.ClusterMetricsExtension"]
      # Sigar native library extract location during tests.
      akka.cluster.metrics.native-library-extract-folder=target/native/${role.name}
      """)
    }
  }

  // this configuration will be used for all nodes
  // note that no fixed host names and ports are used
  commonConfig(ConfigFactory.parseString("""
    akka.actor.provider = cluster
    akka.remote.log-remote-lifecycle-events = off
    akka.cluster.roles = [compute]
     // router lookup config ...
    """))

}
```

Define one concrete test class for each role/node. These will be instantiated on the different nodes (JVMs). They can be implemented differently, but often they are the same and extend an abstract test class, as illustrated here.

```
// need one concrete test class per node
class StatsSampleSpecMultiJvmNode1 extends StatsSampleSpec
```

```
class StatsSampleSpecMultiJvmNode2 extends StatsSampleSpec
class StatsSampleSpecMultiJvmNode3 extends StatsSampleSpec
```

Note the naming convention of these classes. The name of the classes must end with `MultiJvmNode1`, `MultiJvmNode2` and so on. It is possible to define another suffix to be used by the `sbt-multi-jvm`, but the default should be fine in most cases.

Then the abstract `MultiNodeSpec`, which takes the `MultiNodeConfig` as constructor parameter.

```
import org.scalatest.BeforeAndAfterAll
import org.scalatest.WordSpecLike
import org.scalatest.Matchers
import akka.remote.testkit.MultiNodeSpec
import akka.testkit.ImplicitSender

abstract class StatsSampleSpec extends MultiNodeSpec(StatsSampleSpecConfig)
  with WordSpecLike with Matchers with BeforeAndAfterAll
  with ImplicitSender {

  import StatsSampleSpecConfig._

  override def initialParticipants = roles.size

  override def beforeAll() = multiNodeSpecBeforeAll()

  override def afterAll() = multiNodeSpecAfterAll()
```

Most of this can of course be extracted to a separate trait to avoid repeating this in all your tests.

Typically you begin your test by starting up the cluster and let the members join, and create some actors. That can be done like this:

```
"illustrate how to startup cluster" in within(15 seconds) {
  Cluster(system).subscribe(testActor, classOf[MemberUp])
  expectMsgClass(classOf[CurrentClusterState])

  val firstAddress = node(first).address
  val secondAddress = node(second).address
  val thirdAddress = node(third).address

  Cluster(system) join firstAddress

  system.actorOf(Props[StatsWorker], "statsWorker")
  system.actorOf(Props[StatsService], "statsService")

  receiveN(3).collect { case MemberUp(m) => m.address }.toSet should be(
    Set(firstAddress, secondAddress, thirdAddress))

  Cluster(system).unsubscribe(testActor)

  testConductor.enter("all-up")
}
```

From the test you interact with the cluster using the `Cluster` extension, e.g. `join`.

```
Cluster(system) join firstAddress
```

Notice how the *testActor* from *testkit* is added as *subscriber* to cluster changes and then waiting for certain events, such as in this case all members becoming 'Up'.

The above code was running for all roles (JVMs). `runOn` is a convenient utility to declare that a certain block of code should only run for a specific role.

```
"show usage of the statsService from one node" in within(15 seconds) {
  runOn(second) {
    assertServiceOk()
  }

  testConductor.enter("done-2")
}

def assertServiceOk(): Unit = {
  val service = system.actorSelection(node(third) / "user" / "statsService")
  // eventually the service should be ok,
  // first attempts might fail because worker actors not started yet
  awaitAssert {
    service ! StatsJob("this is the text that will be analyzed")
    expectMsgType[StatsResult](1.second).meanWordLength should be(
      3.875 +- 0.001)
  }

}
```

Once again we take advantage of the facilities in *testkit* to verify expected behavior. Here using `testActor` as sender (via `ImplicitSender`) and verifying the reply with `expectMsgPF`.

In the above code you can see `node(third)`, which is useful facility to get the root actor reference of the actor system for a specific role. This can also be used to grab the `akka.actor.Address` of that node.

```
val firstAddress = node(first).address
val secondAddress = node(second).address
val thirdAddress = node(third).address
```

5.2.20 JMX

Information and management of the cluster is available as JMX MBeans with the root name `akka.Cluster`. The JMX information can be displayed with an ordinary JMX console such as JConsole or JVisualVM.

From JMX you can:

- see what members that are part of the cluster
- see status of this node
- see roles of each member
- join this node to another node in cluster
- mark any node in the cluster as down
- tell any node in the cluster to leave

Member nodes are identified by their address, in format *akka.<protocol>://<actor-system-name>@<hostname>:<port>*.

5.2.21 Command Line Management

The cluster can be managed with the script *bin/akka-cluster* provided in the Akka distribution.

Run it without parameters to see instructions about how to use the script:

```
Usage: bin/akka-cluster <node-hostname> <jmx-port> <command> ...

Supported commands are:
           join <node-url> - Sends request a JOIN node with the specified URL
          leave <node-url> - Sends a request for node with URL to LEAVE the cluster
```

```
           down <node-url> - Sends a request for marking node with URL as DOWN
            member-status - Asks the member node for its current status
                  members - Asks the cluster for addresses of current members
              unreachable - Asks the cluster for addresses of unreachable members
           cluster-status - Asks the cluster for its current status (member ring,
                            unavailable nodes, meta data etc.)
                   leader - Asks the cluster who the current leader is
             is-singleton - Checks if the cluster is a singleton cluster (single
                            node cluster)
            is-available - Checks if the member node is available
Where the <node-url> should be on the format of
  'akka.<protocol>://<actor-system-name>@<hostname>:<port>'

Examples: bin/akka-cluster localhost 9999 is-available
          bin/akka-cluster localhost 9999 join akka.tcp://MySystem@darkstar:2552
          bin/akka-cluster localhost 9999 cluster-status
```

To be able to use the script you must enable remote monitoring and management when starting the JVMs of the cluster nodes, as described in Monitoring and Management Using JMX Technology

Example of system properties to enable remote monitoring and management:

```
java -Dcom.sun.management.jmxremote.port=9999 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

5.2.22 Configuration

There are several configuration properties for the cluster. We refer to the *reference configuration* for more information.

Cluster Info Logging

You can silence the logging of cluster events at info level with configuration property:

```
akka.cluster.log-info = off
```

Cluster Dispatcher

Under the hood the cluster extension is implemented with actors and it can be necessary to create a bulkhead for those actors to avoid disturbance from other actors. Especially the heartbeating actors that is used for failure detection can generate false positives if they are not given a chance to run at regular intervals. For this purpose you can define a separate dispatcher to be used for the cluster actors:

```
akka.cluster.use-dispatcher = cluster-dispatcher

cluster-dispatcher {
  type = "Dispatcher"
  executor = "fork-join-executor"
  fork-join-executor {
    parallelism-min = 2
    parallelism-max = 4
  }
}
```

Note: Normally it should not be necessary to configure a separate dispatcher for the Cluster. The default-dispatcher should be sufficient for performing the Cluster tasks, i.e. `akka.cluster.use-dispatcher`

should not be changed. If you have Cluster related problems when using the default-dispatcher that is typically an indication that you are running blocking or CPU intensive actors/tasks on the default-dispatcher. Use dedicated dispatchers for such actors/tasks instead of running them on the default-dispatcher, because that may starve system internal tasks. Related config properties: `akka.cluster.use-dispatcher = akka.cluster.cluster-dispatcher`. Corresponding default values: `akka.cluster.use-dispatcher =`.

5.3 Cluster Singleton

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

Some examples:

- single point of responsibility for certain cluster-wide consistent decisions, or coordination of actions across the cluster system
- single entry point to an external system
- single master, many workers
- centralized naming service, or routing logic

Using a singleton should not be the first design choice. It has several drawbacks, such as single-point of bottleneck. Single-point of failure is also a relevant concern, but for some cases this feature takes care of that by making sure that another singleton instance will eventually be started.

The cluster singleton pattern is implemented by `akka.cluster.singleton.ClusterSingletonManager`. It manages one singleton actor instance among all cluster nodes or a group of nodes tagged with a specific role. `ClusterSingletonManager` is an actor that is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The actual singleton actor is started by the `ClusterSingletonManager` on the oldest node by creating a child actor from supplied `Props`. `ClusterSingletonManager` makes sure that at most one singleton instance is running at any point in time.

The singleton actor is always running on the oldest member with specified role. The oldest member is determined by `akka.cluster.Member#isOlderThan`. This can change when removing that member from the cluster. Be aware that there is a short time period when there is no active singleton during the hand-over process.

The cluster failure detector will notice when oldest node becomes unreachable due to things like JVM crash, hard shut down, or network failure. Then a new oldest node will take over and a new singleton actor is created. For these failure scenarios there will not be a graceful hand-over, but more than one active singletons is prevented by all reasonable means. Some corner cases are eventually resolved by configurable timeouts.

You can access the singleton actor by using the provided `akka.cluster.singleton.ClusterSingletonProxy`, which will route all messages to the current instance of the singleton. The proxy will keep track of the oldest node in the cluster and resolve the singleton's `ActorRef` by explicitly sending the singleton's `actorSelection` the `akka.actor.Identify` message and waiting for it to reply. This is performed periodically if the singleton doesn't reply within a certain (configurable) time. Given the implementation, there might be periods of time during which the `ActorRef` is unavailable, e.g., when a node leaves the cluster. In these cases, the proxy will buffer the messages sent to the singleton and then deliver them when the singleton is finally available. If the buffer is full the `ClusterSingletonProxy` will drop old messages when new messages are sent via the proxy. The size of the buffer is configurable and it can be disabled by using a buffer size of 0.

It's worth noting that messages can always be lost because of the distributed nature of these actors. As always, additional logic should be implemented in the singleton (acknowledgement) and in the client (retry) actors to ensure at-least-once message delivery.

The singleton instance will not run on members with status *WeaklyUp* if that feature is enabled.

5.3.1 Potential problems to be aware of

This pattern may seem to be very tempting to use at first, but it has several drawbacks, some of them are listed below:

- the cluster singleton may quickly become a *performance bottleneck*,
- you can not rely on the cluster singleton to be *non-stop* available — e.g. when the node on which the singleton has been running dies, it will take a few seconds for this to be noticed and the singleton be migrated to another node,
- in the case of a *network partition* appearing in a Cluster that is using Automatic Downing (see Auto Downing docs for *Downing*), it may happen that the isolated clusters each decide to spin up their own singleton, meaning that there might be multiple singletons running in the system, yet the Clusters have no way of finding out about them (because of the partition).

Especially the last point is something you should be aware of — in general when using the Cluster Singleton pattern you should take care of downing nodes yourself and not rely on the timing based auto-down feature.

Warning: **Don't use Cluster Singleton together with Automatic Downing**, since it allows the cluster to split up into two separate clusters, which in turn will result in *multiple Singletons* being started, one in each separate cluster!

5.3.2 An Example

Assume that we need one single entry point to an external system. An actor that receives messages from a JMS queue with the strict requirement that only one JMS consumer must exist to be make sure that the messages are processed in order. That is perhaps not how one would like to design things, but a typical real-world scenario when integrating with external systems.

On each node in the cluster you need to start the `ClusterSingletonManager` and supply the `Props` of the singleton actor, in this case the JMS queue consumer.

```
system.actorOf(
  ClusterSingletonManager.props(
    singletonProps = Props(classOf[Consumer], queue, testActor),
    terminationMessage = End,
    settings = ClusterSingletonManagerSettings(system).withRole("worker")),
  name = "consumer")
```

Here we limit the singleton to nodes tagged with the `"worker"` role, but all nodes, independent of role, can be used by not specifying `withRole`.

Here we use an application specific `terminationMessage` to be able to close the resources before actually stopping the singleton actor. Note that `PoisonPill` is a perfectly fine `terminationMessage` if you only need to stop the actor.

Here is how the singleton actor handles the `terminationMessage` in this example.

```
case End ⇒
  queue ! UnregisterConsumer
case UnregistrationOk ⇒
  stoppedBeforeUnregistration = false
  context stop self
case Ping ⇒
  sender() ! Pong
```

With the names given above, access to the singleton can be obtained from any cluster node using a properly configured proxy.

```
system.actorOf(
  ClusterSingletonProxy.props(
    singletonManagerPath = "/user/consumer",
    settings = ClusterSingletonProxySettings(system).withRole("worker")),
  name = "consumerProxy")
```

A more comprehensive sample is available in the Lightbend Activator tutorial named Distributed workers with Akka and Scala!.

5.3.3 Dependencies

To use the Cluster Singleton you must add the following dependency in your project.

sbt:

```
"com.typesafe.akka" %% "akka-cluster-tools" % "2.4.14"
```

maven:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-cluster-tools_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

5.3.4 Configuration

The following configuration properties are read by the `ClusterSingletonManagerSettings` when created with a `ActorSystem` parameter. It is also possible to amend the `ClusterSingletonManagerSettings` or create it from another config section with the same layout as below. `ClusterSingletonManagerSettings` is a parameter to the `ClusterSingletonManager.props` factory method, i.e. each singleton can be configured with different settings if needed.

```
akka.cluster.singleton {
  # The actor name of the child singleton actor.
  singleton-name = "singleton"

  # Singleton among the nodes tagged with specified role.
  # If the role is not specified it's a singleton among all nodes in the cluster.
  role = ""

  # When a node is becoming oldest it sends hand-over request to previous oldest,
  # that might be leaving the cluster. This is retried with this interval until
  # the previous oldest confirms that the hand over has started or the previous
  # oldest member is removed from the cluster (+ akka.cluster.down-removal-margin).
  hand-over-retry-interval = 1s

  # The number of retries are derived from hand-over-retry-interval and
  # akka.cluster.down-removal-margin (or ClusterSingletonManagerSettings.
↪removalMargin),
  # but it will never be less than this property.
  min-number-of-hand-over-retries = 10
}
```

The following configuration properties are read by the `ClusterSingletonProxySettings` when created with a `ActorSystem` parameter. It is also possible to amend the `ClusterSingletonProxySettings` or create it from another config section with the same layout as below. `ClusterSingletonProxySettings`

is a parameter to the `ClusterSingletonProxy.props` factory method, i.e. each singleton proxy can be configured with different settings if needed.

```
akka.cluster.singleton-proxy {
  # The actor name of the singleton actor that is started by the␣
↪ClusterSingletonManager
  singleton-name = ${akka.cluster.singleton.singleton-name}

  # The role of the cluster nodes where the singleton can be deployed.
  # If the role is not specified then any node will do.
  role = ""

  # Interval at which the proxy will try to resolve the singleton instance.
  singleton-identification-interval = 1s

  # If the location of the singleton is unknown the proxy will buffer this
  # number of messages and deliver them when the singleton is identified.
  # When the buffer is full old messages will be dropped when new messages are
  # sent via the proxy.
  # Use 0 to disable buffering, i.e. messages will be dropped immediately if
  # the location of the singleton is unknown.
  # Maximum allowed buffer size is 10000.
  buffer-size = 1000
}
```

5.4 Distributed Publish Subscribe in Cluster

How do I send a message to an actor without knowing which node it is running on?

How do I send messages to all actors in the cluster that have registered interest in a named topic?

This pattern provides a mediator actor, `akka.cluster.pubsub.DistributedPubSubMediator`, that manages a registry of actor references and replicates the entries to peer actors among all cluster nodes or a group of nodes tagged with a specific role.

The `DistributedPubSubMediator` actor is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The mediator can be started with the `DistributedPubSub` extension or as an ordinary actor.

The registry is eventually consistent, i.e. changes are not immediately visible at other nodes, but typically they will be fully replicated to all other nodes after a few seconds. Changes are only performed in the own part of the registry and those changes are versioned. Deltas are disseminated in a scalable way to other nodes with a gossip protocol.

Cluster members with status *WeaklyUp*, if that feature is enabled, will participate in Distributed Publish Subscribe, i.e. subscribers on nodes with `WeaklyUp` status will receive published messages if the publisher and subscriber are on same side of a network partition.

You can send messages via the mediator on any node to registered actors on any other node.

There a two different modes of message delivery, explained in the sections *Publish* and *Send* below.

A more comprehensive sample is available in the Lightbend Activator tutorial named Akka Clustered PubSub with Scala!.

5.4.1 Publish

This is the true pub/sub mode. A typical usage of this mode is a chat room in an instant messaging application.

Actors are registered to a named topic. This enables many subscribers on each node. The message will be delivered to all subscribers of the topic.

For efficiency the message is sent over the wire only once per node (that has a matching topic), and then delivered to all subscribers of the local topic representation. (See more in)

You register actors to the local mediator with `DistributedPubSubMediator.Subscribe`. Successful `Subscribe` and `Unsubscribe` is acknowledged with `DistributedPubSubMediator.SubscribeAck` and `DistributedPubSubMediator.UnsubscribeAck` replies. The acknowledgment means that the subscription is registered, but it can still take some time until it is replicated to other nodes.

You publish messages by sending `DistributedPubSubMediator.Publish` message to the local mediator.

Actors are automatically removed from the registry when they are terminated, or you can explicitly remove entries with `DistributedPubSubMediator.Unsubscribe`.

An example of a subscriber actor:

```
class Subscriber extends Actor with ActorLogging {
  import DistributedPubSubMediator.{ Subscribe, SubscribeAck }
  val mediator = DistributedPubSub(context.system).mediator
  // subscribe to the topic named "content"
  mediator ! Subscribe("content", self)

  def receive = {
    case s: String ⇒
      log.info("Got {}", s)
    case SubscribeAck(Subscribe("content", None, `self`)) ⇒
      log.info("subscribing");
  }
}
```

Subscriber actors can be started on several nodes in the cluster, and all will receive messages published to the "content" topic.

```
runOn(first) {
  system.actorOf(Props[Subscriber], "subscriber1")
}
runOn(second) {
  system.actorOf(Props[Subscriber], "subscriber2")
  system.actorOf(Props[Subscriber], "subscriber3")
}
```

A simple actor that publishes to this "content" topic:

```
class Publisher extends Actor {
  import DistributedPubSubMediator.Publish
  // activate the extension
  val mediator = DistributedPubSub(context.system).mediator

  def receive = {
    case in: String ⇒
      val out = in.toUpperCase
      mediator ! Publish("content", out)
  }
}
```

It can publish messages to the topic from anywhere in the cluster:

```
runOn(third) {
  val publisher = system.actorOf(Props[Publisher], "publisher")
  later()
  // after a while the subscriptions are replicated
  publisher ! "hello"
}
```

Topic Groups

Actors may also be subscribed to a named topic with a `group` id. If subscribing with a group id, each message published to a topic with the `sendOneMessageToEachGroup` flag set to `true` is delivered via the supplied `RoutingLogic` (default random) to one actor within each subscribing group.

If all the subscribed actors have the same group id, then this works just like `Send` and each message is only delivered to one subscriber.

If all the subscribed actors have different group names, then this works like normal `Publish` and each message is broadcasted to all subscribers.

Note: Note that if the group id is used it is part of the topic identifier. Messages published with `sendOneMessageToEachGroup=false` will not be delivered to subscribers that subscribed with a group id. Messages published with `sendOneMessageToEachGroup=true` will not be delivered to subscribers that subscribed without a group id.

5.4.2 Send

This is a point-to-point mode where each message is delivered to one destination, but you still does not have to know where the destination is located. A typical usage of this mode is private chat to one other user in an instant messaging application. It can also be used for distributing tasks to registered workers, like a cluster aware router where the routees dynamically can register themselves.

The message will be delivered to one recipient with a matching path, if any such exists in the registry. If several entries match the path because it has been registered on several nodes the message will be sent via the supplied `RoutingLogic` (default random) to one destination. The sender() of the message can specify that local affinity is preferred, i.e. the message is sent to an actor in the same local actor system as the used mediator actor, if any such exists, otherwise route to any other matching entry.

You register actors to the local mediator with `DistributedPubSubMediator.Put`. The `ActorRef` in `Put` must belong to the same local actor system as the mediator. The path without address information is the key to which you send messages. On each node there can only be one actor for a given path, since the path is unique within one local actor system.

You send messages by sending `DistributedPubSubMediator.Send` message to the local mediator with the path (without address information) of the destination actors.

Actors are automatically removed from the registry when they are terminated, or you can explicitly remove entries with `DistributedPubSubMediator.Remove`.

An example of a destination actor:

```
class Destination extends Actor with ActorLogging {
  import DistributedPubSubMediator.Put
  val mediator = DistributedPubSub(context.system).mediator
  // register to the path
  mediator ! Put(self)

  def receive = {
    case s: String ⇒
      log.info("Got {}", s)
  }
}
```

Destination actors can be started on several nodes in the cluster, and all will receive messages sent to the path (without address information).

```
runOn(first) {
  system.actorOf(Props[Destination], "destination")
}
```

```
runOn(second) {
  system.actorOf(Props[Destination], "destination")
}
```

A simple actor that sends to the path:

```
class Sender extends Actor {
  import DistributedPubSubMediator.Send
  // activate the extension
  val mediator = DistributedPubSub(context.system).mediator

  def receive = {
    case in: String ⇒
      val out = in.toUpperCase
      mediator ! Send(path = "/user/destination", msg = out, localAffinity = true)
  }
}
```

It can send messages to the path from anywhere in the cluster:

```
runOn(third) {
  val sender = system.actorOf(Props[Sender], "sender")
  later()
  // after a while the destinations are replicated
  sender ! "hello"
}
```

It is also possible to broadcast messages to the actors that have been registered with `Put`. Send `DistributedPubSubMediator.SendToAll` message to the local mediator and the wrapped message will then be delivered to all recipients with a matching path. Actors with the same path, without address information, can be registered on different nodes. On each node there can only be one such actor, since the path is unique within one local actor system.

Typical usage of this mode is to broadcast messages to all replicas with the same path, e.g. 3 actors on different nodes that all perform the same actions, for redundancy. You can also optionally specify a property (`allButSelf`) deciding if the message should be sent to a matching path on the self node or not.

5.4.3 DistributedPubSub Extension

In the example above the mediator is started and accessed with the `akka.cluster.pubsub.DistributedPubSub` extension. That is convenient and perfectly fine in most cases, but it can be good to know that it is possible to start the mediator actor as an ordinary actor and you can have several different mediators at the same time to be able to divide a large number of actors/topics to different mediators. For example you might want to use different cluster roles for different mediators.

The `DistributedPubSub` extension can be configured with the following properties:

```
# Settings for the DistributedPubSub extension
akka.cluster.pub-sub {
  # Actor name of the mediator actor, /system/distributedPubSubMediator
  name = distributedPubSubMediator

  # Start the mediator on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # The routing logic to use for 'Send'
  # Possible values: random, round-robin, broadcast
  routing-logic = random

  # How often the DistributedPubSubMediator should send out gossip information
```

```
  gossip-interval = 1s

  # Removed entries are pruned after this duration
  removed-time-to-live = 120s

  # Maximum number of elements to transfer in one message when synchronizing the␣
↪registries.
  # Next chunk will be transferred in next round of gossip.
  max-delta-elements = 3000

  # The id of the dispatcher to use for DistributedPubSubMediator actors.
  # If not specified default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  use-dispatcher = ""
}
```

It is recommended to load the extension when the actor system is started by defining it in `akka.extensions` configuration property. Otherwise it will be activated when first used and then it takes a while for it to be populated.

```
akka.extensions = ["akka.cluster.pubsub.DistributedPubSub"]
```

5.4.4 Delivery Guarantee

As in *Message Delivery Reliability* of Akka, message delivery guarantee in distributed pub sub modes is **at-most-once delivery**. In other words, messages can be lost over the wire.

If you are looking for at-least-once delivery guarantee, we recommend Kafka Akka Streams integration.

5.4.5 Dependencies

To use Distributed Publish Subscribe you must add the following dependency in your project.

sbt:

```
"com.typesafe.akka" %% "akka-cluster-tools" % "2.4.14"
```

maven:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-cluster-tools_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

5.5 Cluster Client

An actor system that is not part of the cluster can communicate with actors somewhere in the cluster via this `ClusterClient`. The client can of course be part of another cluster. It only needs to know the location of one (or more) nodes to use as initial contact points. It will establish a connection to a `ClusterReceptionist` somewhere in the cluster. It will monitor the connection to the receptionist and establish a new connection if the link goes down. When looking for a new receptionist it uses fresh contact points retrieved from previous establishment, or periodically refreshed contacts, i.e. not necessarily the initial contact points.

Note: `ClusterClient` should not be used when sending messages to actors that run within the same cluster. Similar functionality as the `ClusterClient` is provided in a more efficient way by *Distributed Publish*

Subscribe in Cluster for actors that belong to the same cluster.

Also, note it's necessary to change `akka.actor.provider` from `local` to `remote` or `cluster` when using the cluster client.

The receptionist is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The receptionist can be started with the `ClusterClientReceptionist` extension or as an ordinary actor.

You can send messages via the `ClusterClient` to any actor in the cluster that is registered in the `DistributedPubSubMediator` used by the `ClusterReceptionist`. The `ClusterClientReceptionist` provides methods for registration of actors that should be reachable from the client. Messages are wrapped in `ClusterClient.Send`, `ClusterClient.SendToAll` or `ClusterClient.Publish`.

Both the `ClusterClient` and the `ClusterClientReceptionist` emit events that can be subscribed to. The `ClusterClient` sends out notifications in relation to having received a list of contact points from the `ClusterClientReceptionist`. One use of this list might be for the client to record its contact points. A client that is restarted could then use this information to supersede any previously configured contact points.

The `ClusterClientReceptionist` sends out notifications in relation to having received contact from a `ClusterClient`. This notification enables the server containing the receptionist to become aware of what clients are connected.

1. ClusterClient.Send

The message will be delivered to one recipient with a matching path, if any such exists. If several entries match the path the message will be delivered to one random destination. The sender() of the message can specify that local affinity is preferred, i.e. the message is sent to an actor in the same local actor system as the used receptionist actor, if any such exists, otherwise random to any other matching entry.

2. ClusterClient.SendToAll

The message will be delivered to all recipients with a matching path.

3. ClusterClient.Publish

The message will be delivered to all recipients Actors that have been registered as subscribers to the named topic.

Response messages from the destination actor are tunneled via the receptionist to avoid inbound connections from other cluster nodes to the client, i.e. the `sender()`, as seen by the destination actor, is not the client itself. The `sender()` of the response messages, as seen by the client, is `deadLetters` since the client should normally send subsequent messages via the `ClusterClient`. It is possible to pass the original sender inside the reply messages if the client is supposed to communicate directly to the actor in the cluster.

While establishing a connection to a receptionist the `ClusterClient` will buffer messages and send them when the connection is established. If the buffer is full the `ClusterClient` will drop old messages when new messages are sent via the client. The size of the buffer is configurable and it can be disabled by using a buffer size of 0.

It's worth noting that messages can always be lost because of the distributed nature of these actors. As always, additional logic should be implemented in the destination (acknowledgement) and in the client (retry) actors to ensure at-least-once message delivery.

5.5.1 An Example

On the cluster nodes first start the receptionist. Note, it is recommended to load the extension when the actor system is started by defining it in the `akka.extensions` configuration property:

```
akka.extensions = ["akka.cluster.client.ClusterClientReceptionist"]
```

Next, register the actors that should be available for the client.

```
runOn(host1) {
  val serviceA = system.actorOf(Props[Service], "serviceA")
  ClusterClientReceptionist(system).registerService(serviceA)
}

runOn(host2, host3) {
  val serviceB = system.actorOf(Props[Service], "serviceB")
  ClusterClientReceptionist(system).registerService(serviceB)
}
```

On the client you create the `ClusterClient` actor and use it as a gateway for sending messages to the actors identified by their path (without address information) somewhere in the cluster.

```
runOn(client) {
  val c = system.actorOf(ClusterClient.props(
    ClusterClientSettings(system).withInitialContacts(initialContacts)), "client")
  c ! ClusterClient.Send("/user/serviceA", "hello", localAffinity = true)
  c ! ClusterClient.SendToAll("/user/serviceB", "hi")
}
```

The `initialContacts` parameter is a `Set[ActorPath]`, which can be created like this:

```
val initialContacts = Set(
  ActorPath.fromString("akka.tcp://OtherSys@host1:2552/system/receptionist"),
  ActorPath.fromString("akka.tcp://OtherSys@host2:2552/system/receptionist"))
val settings = ClusterClientSettings(system)
  .withInitialContacts(initialContacts)
```

You will probably define the address information of the initial contact points in configuration or system property. See also *Configuration*.

A more comprehensive sample is available in the Lightbend Activator tutorial named Distributed workers with Akka and Scala!.

5.5.2 ClusterClientReceptionist Extension

In the example above the receptionist is started and accessed with the `akka.cluster.client.ClusterClientReceptionist` extension. That is convenient and perfectly fine in most cases, but it can be good to know that it is possible to start the `akka.cluster.client.ClusterReceptionist` actor as an ordinary actor and you can have several different receptionists at the same time, serving different types of clients.

Note that the `ClusterClientReceptionist` uses the `DistributedPubSub` extension, which is described in *Distributed Publish Subscribe in Cluster*.

It is recommended to load the extension when the actor system is started by defining it in the `akka.extensions` configuration property:

```
akka.extensions = ["akka.cluster.client.ClusterClientReceptionist"]
```

5.5.3 Events

As mentioned earlier, both the `ClusterClient` and `ClusterClientReceptionist` emit events that can be subscribed to. The following code snippet declares an actor that will receive notifications on contact points (addresses to the available receptionists), as they become available. The code illustrates subscribing to the events and receiving the `ClusterClient` initial state.

```
class ClientListener(targetClient: ActorRef) extends Actor {
  override def preStart(): Unit =
```

```
    targetClient ! SubscribeContactPoints

  def receive: Receive =
    receiveWithContactPoints(Set.empty)

  def receiveWithContactPoints(contactPoints: Set[ActorPath]): Receive = {
    case ContactPoints(cps) ⇒
      context.become(receiveWithContactPoints(cps))
    // Now do something with the up-to-date "cps"
    case ContactPointAdded(cp) ⇒
      context.become(receiveWithContactPoints(contactPoints + cp))
    // Now do something with an up-to-date "contactPoints + cp"
    case ContactPointRemoved(cp) ⇒
      context.become(receiveWithContactPoints(contactPoints - cp))
    // Now do something with an up-to-date "contactPoints - cp"
  }
}
```

Similarly we can have an actor that behaves in a similar fashion for learning what cluster clients contact a `ClusterClientReceptionist`:

```
class ReceptionistListener(targetReceptionist: ActorRef) extends Actor {
  override def preStart(): Unit =
    targetReceptionist ! SubscribeClusterClients

  def receive: Receive =
    receiveWithClusterClients(Set.empty)

  def receiveWithClusterClients(clusterClients: Set[ActorRef]): Receive = {
    case ClusterClients(cs) ⇒
      context.become(receiveWithClusterClients(cs))
    // Now do something with the up-to-date "c"
    case ClusterClientUp(c) ⇒
      context.become(receiveWithClusterClients(clusterClients + c))
    // Now do something with an up-to-date "clusterClients + c"
    case ClusterClientUnreachable(c) ⇒
      context.become(receiveWithClusterClients(clusterClients - c))
    // Now do something with an up-to-date "clusterClients - c"
  }
}
```

5.5.4 Dependencies

To use the Cluster Client you must add the following dependency in your project.

sbt:

```
"com.typesafe.akka" %% "akka-cluster-tools" % "2.4.14"
```

maven:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-cluster-tools_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

5.5.5 Configuration

The `ClusterClientReceptionist` extension (or `ClusterReceptionistSettings`) can be configured with the following properties:

```
# Settings for the ClusterClientReceptionist extension
akka.cluster.client.receptionist {
  # Actor name of the ClusterReceptionist actor, /system/receptionist
  name = receptionist

  # Start the receptionist on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # The receptionist will send this number of contact points to the client
  number-of-contacts = 3

  # The actor that tunnel response messages to the client will be stopped
  # after this time of inactivity.
  response-tunnel-receive-timeout = 30s

  # The id of the dispatcher to use for ClusterReceptionist actors.
  # If not specified default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  use-dispatcher = ""

  # How often failure detection heartbeat messages should be received for
  # each ClusterClient
  heartbeat-interval = 2s

  # Number of potentially lost/delayed heartbeats that will be
  # accepted before considering it to be an anomaly.
  # The ClusterReceptionist is using the akka.remote.DeadlineFailureDetector, which
  # will trigger if there are no heartbeats within the duration
  # heartbeat-interval + acceptable-heartbeat-pause, i.e. 15 seconds with
  # the default settings.
  acceptable-heartbeat-pause = 13s

  # Failure detection checking interval for checking all ClusterClients
  failure-detection-interval = 2s
}
```

The following configuration properties are read by the `ClusterClientSettings` when created with a `ActorSystem` parameter. It is also possible to amend the `ClusterClientSettings` or create it from another config section with the same layout as below. `ClusterClientSettings` is a parameter to the `ClusterClient.props` factory method, i.e. each client can be configured with different settings if needed.

```
# Settings for the ClusterClient
akka.cluster.client {
  # Actor paths of the ClusterReceptionist actors on the servers (cluster nodes)
  # that the client will try to contact initially. It is mandatory to specify
  # at least one initial contact.
  # Comma separated full actor paths defined by a string on the form of
  # "akka.tcp://system@hostname:port/system/receptionist"
  initial-contacts = []

  # Interval at which the client retries to establish contact with one of
  # ClusterReceptionist on the servers (cluster nodes)
  establishing-get-contacts-interval = 3s

  # Interval at which the client will ask the ClusterReceptionist for
  # new contact points to be used for next reconnect.
  refresh-contacts-interval = 60s
```

```
  # How often failure detection heartbeat messages should be sent
  heartbeat-interval = 2s

  # Number of potentially lost/delayed heartbeats that will be
  # accepted before considering it to be an anomaly.
  # The ClusterClient is using the akka.remote.DeadlineFailureDetector, which
  # will trigger if there are no heartbeats within the duration
  # heartbeat-interval + acceptable-heartbeat-pause, i.e. 15 seconds with
  # the default settings.
  acceptable-heartbeat-pause = 13s

  # If connection to the receptionist is not established the client will buffer
  # this number of messages and deliver them the connection is established.
  # When the buffer is full old messages will be dropped when new messages are sent
  # via the client. Use 0 to disable buffering, i.e. messages will be dropped
  # immediately if the location of the singleton is unknown.
  # Maximum allowed buffer size is 10000.
  buffer-size = 1000

  # If connection to the receiptionist is lost and the client has not been
  # able to acquire a new connection for this long the client will stop itself.
  # This duration makes it possible to watch the cluster client and react on a␣
↪more permanent
  # loss of connection with the cluster, for example by accessing some kind of
  # service registry for an updated set of initial contacts to start a new cluster␣
↪client with.
  # If this is not wanted it can be set to "off" to disable the timeout and retry
  # forever.
  reconnect-timeout = off
}
```

5.5.6 Failure handling

When the cluster client is started it must be provided with a list of initial contacts which are cluster nodes where receptionists are running. It will then repeatedly (with an interval configurable by `establishing-get-contacts-interval`) try to contact those until it gets in contact with one of them. While running, the list of contacts are continuously updated with data from the receptionists (again, with an interval configurable with `refresh-contacts-interval`), so that if there are more receptionists in the cluster than the initial contacts provided to the client the client will learn about them.

While the client is running it will detect failures in its connection to the receptionist by heartbeats if more than a configurable amount of heartbeats are missed the client will try to reconnect to its known set of contacts to find a receptionist it can access.

5.5.7 When the cluster cannot be reached at all

It is possible to make the cluster client stop entirely if it cannot find a receptionist it can talk to within a configurable interval. This is configured with the `reconnect-timeout`, which defaults to `off`. This can be useful when initial contacts are provided from some kind of service registry, cluster node addresses are entirely dynamic and the entire cluster might shut down or crash, be restarted on new addresses. Since the client will be stopped in that case a monitoring actor can watch it and upon `Terminate` a new set of initial contacts can be fetched and a new cluster client started.

5.6 Cluster Sharding

Cluster sharding is useful when you need to distribute actors across several nodes in the cluster and want to be able to interact with them using their logical identifier, but without having to care about their physical location in the cluster, which might also change over time.

It could for example be actors representing Aggregate Roots in Domain-Driven Design terminology. Here we call these actors "entities". These actors typically have persistent (durable) state, but this feature is not limited to actors with persistent state.

Cluster sharding is typically used when you have many stateful actors that together consume more resources (e.g. memory) than fit on one machine. If you only have a few stateful actors it might be easier to run them on a *Cluster Singleton* node.

In this context sharding means that actors with an identifier, so called entities, can be automatically distributed across multiple nodes in the cluster. Each entity actor runs only at one place, and messages can be sent to the entity without requiring the sender to know the location of the destination actor. This is achieved by sending the messages via a `ShardRegion` actor provided by this extension, which knows how to route the message with the entity id to the final destination.

Cluster sharding will not be active on members with status *WeaklyUp* if that feature is enabled.

Warning: **Don't use Cluster Sharding together with Automatic Downing**, since it allows the cluster to split up into two separate clusters, which in turn will result in *multiple shards and entities* being started, one in each separate cluster! See automatic-vs-manual-downing-java.

5.6.1 An Example

This is how an entity actor may look like:

```
case object Increment
case object Decrement
final case class Get(counterId: Long)
final case class EntityEnvelope(id: Long, payload: Any)

case object Stop
final case class CounterChanged(delta: Int)

class Counter extends PersistentActor {
  import ShardRegion.Passivate

  context.setReceiveTimeout(120.seconds)

  // self.path.name is the entity identifier (utf-8 URL-encoded)
  override def persistenceId: String = "Counter-" + self.path.name

  var count = 0

  def updateState(event: CounterChanged): Unit =
    count += event.delta

  override def receiveRecover: Receive = {
    case evt: CounterChanged ⇒ updateState(evt)
  }

  override def receiveCommand: Receive = {
    case Increment      ⇒ persist(CounterChanged(+1))(updateState)
    case Decrement      ⇒ persist(CounterChanged(-1))(updateState)
    case Get(_)         ⇒ sender() ! count
    case ReceiveTimeout ⇒ context.parent ! Passivate(stopMessage = Stop)
```

```
    case Stop            ⇒ context.stop(self)
  }
}
```

The above actor uses event sourcing and the support provided in `PersistentActor` to store its state. It does not have to be a persistent actor, but in case of failure or migration of entities between nodes it must be able to recover its state if it is valuable.

Note how the `persistenceId` is defined. The name of the actor is the entity identifier (utf-8 URL-encoded). You may define it another way, but it must be unique.

When using the sharding extension you are first, typically at system startup on each node in the cluster, supposed to register the supported entity types with the `ClusterSharding.start` method. `ClusterSharding.start` gives you the reference which you can pass along.

```
val counterRegion: ActorRef = ClusterSharding(system).start(
  typeName = "Counter",
  entityProps = Props[Counter],
  settings = ClusterShardingSettings(system),
  extractEntityId = extractEntityId,
  extractShardId = extractShardId)
```

The `extractEntityId` and `extractShardId` are two application specific functions to extract the entity identifier and the shard identifier from incoming messages.

```
val extractEntityId: ShardRegion.ExtractEntityId = {
  case EntityEnvelope(id, payload) ⇒ (id.toString, payload)
  case msg @ Get(id)               ⇒ (id.toString, msg)
}

val numberOfShards = 100

val extractShardId: ShardRegion.ExtractShardId = {
  case EntityEnvelope(id, _) ⇒ (id % numberOfShards).toString
  case Get(id)               ⇒ (id % numberOfShards).toString
}
```

This example illustrates two different ways to define the entity identifier in the messages:

- The `Get` message includes the identifier itself.
- The `EntityEnvelope` holds the identifier, and the actual message that is sent to the entity actor is wrapped in the envelope.

Note how these two messages types are handled in the `extractEntityId` function shown above. The message sent to the entity actor is the second part of the tuple return by the `extractEntityId` and that makes it possible to unwrap envelopes if needed.

A shard is a group of entities that will be managed together. The grouping is defined by the `extractShardId` function shown above. For a specific entity identifier the shard identifier must always be the same.

Creating a good sharding algorithm is an interesting challenge in itself. Try to produce a uniform distribution, i.e. same amount of entities in each shard. As a rule of thumb, the number of shards should be a factor ten greater than the planned maximum number of cluster nodes. Less shards than number of nodes will result in that some nodes will not host any shards. Too many shards will result in less efficient management of the shards, e.g. rebalancing overhead, and increased latency because the coordinator is involved in the routing of the first message for each shard. The sharding algorithm must be the same on all nodes in a running cluster. It can be changed after stopping all nodes in the cluster.

A simple sharding algorithm that works fine in most cases is to take the absolute value of the `hashCode` of the entity identifier modulo number of shards. As a convenience this is provided by the `ShardRegion.HashCodeMessageExtractor`.

Messages to the entities are always sent via the local `ShardRegion`. The `ShardRegion` actor reference for a named entity type is returned by `ClusterSharding.start` and it can also be retrieved with `ClusterSharding.shardRegion`. The `ShardRegion` will lookup the location of the shard for the entity if it does not already know its location. It will delegate the message to the right node and it will create the entity actor on demand, i.e. when the first message for a specific entity is delivered.

```
val counterRegion: ActorRef = ClusterSharding(system).shardRegion("Counter")
counterRegion ! Get(123)
expectMsg(0)

counterRegion ! EntityEnvelope(123, Increment)
counterRegion ! Get(123)
expectMsg(1)
```

A more comprehensive sample is available in the Lightbend Activator tutorial named Akka Cluster Sharding with Scala!.

5.6.2 How it works

The `ShardRegion` actor is started on each node in the cluster, or group of nodes tagged with a specific role. The `ShardRegion` is created with two application specific functions to extract the entity identifier and the shard identifier from incoming messages. A shard is a group of entities that will be managed together. For the first message in a specific shard the `ShardRegion` request the location of the shard from a central coordinator, the `ShardCoordinator`.

The `ShardCoordinator` decides which `ShardRegion` shall own the `Shard` and informs that `ShardRegion`. The region will confirm this request and create the `Shard` supervisor as a child actor. The individual `Entities` will then be created when needed by the `Shard` actor. Incoming messages thus travel via the `ShardRegion` and the `Shard` to the target `Entity`.

If the shard home is another `ShardRegion` instance messages will be forwarded to that `ShardRegion` instance instead. While resolving the location of a shard incoming messages for that shard are buffered and later delivered when the shard home is known. Subsequent messages to the resolved shard can be delivered to the target destination immediately without involving the `ShardCoordinator`.

Scenario 1:

1. Incoming message M1 to `ShardRegion` instance R1.
2. M1 is mapped to shard S1. R1 doesn't know about S1, so it asks the coordinator C for the location of S1.
3. C answers that the home of S1 is R1.
4. R1 creates child actor for the entity E1 and sends buffered messages for S1 to E1 child
5. All incoming messages for S1 which arrive at R1 can be handled by R1 without C. It creates entity children as needed, and forwards messages to them.

Scenario 2:

1. Incoming message M2 to R1.
2. M2 is mapped to S2. R1 doesn't know about S2, so it asks C for the location of S2.
3. C answers that the home of S2 is R2.
4. R1 sends buffered messages for S2 to R2
5. All incoming messages for S2 which arrive at R1 can be handled by R1 without C. It forwards messages to R2.
6. R2 receives message for S2, ask C, which answers that the home of S2 is R2, and we are in Scenario 1 (but for R2).

To make sure that at most one instance of a specific entity actor is running somewhere in the cluster it is important that all nodes have the same view of where the shards are located. Therefore the shard allocation decisions are

taken by the central `ShardCoordinator`, which is running as a cluster singleton, i.e. one instance on the oldest member among all cluster nodes or a group of nodes tagged with a specific role.

The logic that decides where a shard is to be located is defined in a pluggable shard allocation strategy. The default implementation `ShardCoordinator.LeastShardAllocationStrategy` allocates new shards to the `ShardRegion` with least number of previously allocated shards. This strategy can be replaced by an application specific implementation.

To be able to use newly added members in the cluster the coordinator facilitates rebalancing of shards, i.e. migrate entities from one node to another. In the rebalance process the coordinator first notifies all `ShardRegion` actors that a handoff for a shard has started. That means they will start buffering incoming messages for that shard, in the same way as if the shard location is unknown. During the rebalance process the coordinator will not answer any requests for the location of shards that are being rebalanced, i.e. local buffering will continue until the handoff is completed. The `ShardRegion` responsible for the rebalanced shard will stop all entities in that shard by sending the specified `handOffStopMessage` (default `PoisonPill`) to them. When all entities have been terminated the `ShardRegion` owning the entities will acknowledge the handoff as completed to the coordinator. Thereafter the coordinator will reply to requests for the location of the shard and thereby allocate a new home for the shard and then buffered messages in the `ShardRegion` actors are delivered to the new location. This means that the state of the entities are not transferred or migrated. If the state of the entities are of importance it should be persistent (durable), e.g. with *Persistence*, so that it can be recovered at the new location.

The logic that decides which shards to rebalance is defined in a pluggable shard allocation strategy. The default implementation `ShardCoordinator.LeastShardAllocationStrategy` picks shards for handoff from the `ShardRegion` with most number of previously allocated shards. They will then be allocated to the `ShardRegion` with least number of previously allocated shards, i.e. new members in the cluster. There is a configurable threshold of how large the difference must be to begin the rebalancing. This strategy can be replaced by an application specific implementation.

The state of shard locations in the `ShardCoordinator` is persistent (durable) with *Persistence* to survive failures. Since it is running in a cluster *Persistence* must be configured with a distributed journal. When a crashed or unreachable coordinator node has been removed (via down) from the cluster a new `ShardCoordinator` singleton actor will take over and the state is recovered. During such a failure period shards with known location are still available, while messages for new (unknown) shards are buffered until the new `ShardCoordinator` becomes available.

As long as a sender uses the same `ShardRegion` actor to deliver messages to an entity actor the order of the messages is preserved. As long as the buffer limit is not reached messages are delivered on a best effort basis, with at-most once delivery semantics, in the same way as ordinary message sending. Reliable end-to-end messaging, with at-least-once semantics can be added by using `AtLeastOnceDelivery` in *Persistence*.

Some additional latency is introduced for messages targeted to new or previously unused shards due to the round-trip to the coordinator. Rebalancing of shards may also add latency. This should be considered when designing the application specific shard resolution, e.g. to avoid too fine grained shards.

5.6.3 Distributed Data Mode

Instead of using *Persistence* it is possible to use the *Distributed Data* module as storage for the state of the sharding coordinator. In such case the state of the `ShardCoordinator` will be replicated inside a cluster by the *Distributed Data* module with `WriteMajority`/`ReadMajority` consistency.

This mode can be enabled by setting configuration property:

```
akka.cluster.sharding.state-store-mode = ddata
```

It is using the Distributed Data extension that must be running on all nodes in the cluster. Therefore you should add that extension to the configuration to make sure that it is started on all nodes:

```
akka.extensions += "akka.cluster.ddata.DistributedData"
```

You must explicitly add the `akka-distributed-data-experimental` dependency to your build if you use this mode. It is possible to remove `akka-persistence` dependency from a project if it is not used in

user code and `remember-entities` is `off`. Using it together with `Remember Entities` shards will be recreated after rebalancing, however will not be recreated after a clean cluster start as the Sharding Coordinator state is empty after a clean cluster start when using ddata mode. When `Remember Entities` is `on` Sharding Region always keeps data usig persistence, no matter how `State Store Mode` is set.

Warning: The `ddata` mode is considered as **"experimental"** as of its introduction in Akka 2.4.0, since it depends on the experimental Distributed Data module.

5.6.4 Startup after minimum number of members

It's good to use Cluster Sharding with the Cluster setting `akka.cluster.min-nr-of-members` or `akka.cluster.role.<role-name>.min-nr-of-members`. That will defer the allocation of the shards until at least that number of regions have been started and registered to the coordinator. This avoids that many shards are allocated to the first region that registers and only later are rebalanced to other nodes.

See *How To Startup when Cluster Size Reached* for more information about `min-nr-of-members`.

5.6.5 Proxy Only Mode

The `ShardRegion` actor can also be started in proxy only mode, i.e. it will not host any entities itself, but knows how to delegate messages to the right location. A `ShardRegion` is started in proxy only mode with the method `ClusterSharding.startProxy` method.

5.6.6 Passivation

If the state of the entities are persistent you may stop entities that are not used to reduce memory consumption. This is done by the application specific implementation of the entity actors for example by defining receive timeout (`context.setReceiveTimeout`). If a message is already enqueued to the entity when it stops itself the enqueued message in the mailbox will be dropped. To support graceful passivation without losing such messages the entity actor can send `ShardRegion.Passivate` to its parent `Shard`. The specified wrapped message in `Passivate` will be sent back to the entity, which is then supposed to stop itself. Incoming messages will be buffered by the `Shard` between reception of `Passivate` and termination of the entity. Such buffered messages are thereafter delivered to a new incarnation of the entity.

5.6.7 Remembering Entities

The list of entities in each `Shard` can be made persistent (durable) by setting the `rememberEntities` flag to true in `ClusterShardingSettings` when calling `ClusterSharding.start`. When configured to remember entities, whenever a `Shard` is rebalanced onto another node or recovers after a crash it will recreate all the entities which were previously running in that `Shard`. To permanently stop entities, a `Passivate` message must be sent to the parent of the entity actor, otherwise the entity will be automatically restarted after the entity restart backoff specified in the configuration.

When `rememberEntities` is set to false, a `Shard` will not automatically restart any entities after a rebalance or recovering from a crash. Entities will only be started once the first message for that entity has been received in the `Shard`. Entities will not be restarted if they stop without using a `Passivate`.

Note that the state of the entities themselves will not be restored unless they have been made persistent, e.g. with *Persistence*.

5.6.8 Supervision

If you need to use another `supervisorStrategy` for the entity actors than the default (restarting) strategy you need to create an intermediate parent actor that defines the `supervisorStrategy` to the child entity actor.

```
class CounterSupervisor extends Actor {
  val counter = context.actorOf(Props[Counter], "theCounter")

  override val supervisorStrategy = OneForOneStrategy() {
    case _: IllegalArgumentException     ⇒ SupervisorStrategy.Resume
    case _: ActorInitializationException ⇒ SupervisorStrategy.Stop
    case _: DeathPactException           ⇒ SupervisorStrategy.Stop
    case _: Exception                    ⇒ SupervisorStrategy.Restart
  }

  def receive = {
    case msg ⇒ counter forward msg
  }
}
```

You start such a supervisor in the same way as if it was the entity actor.

```
ClusterSharding(system).start(
  typeName = "SupervisedCounter",
  entityProps = Props[CounterSupervisor],
  settings = ClusterShardingSettings(system),
  extractEntityId = extractEntityId,
  extractShardId = extractShardId)
```

Note that stopped entities will be started again when a new message is targeted to the entity.

5.6.9 Graceful Shutdown

You can send the message `ShardRegion.GracefulShutdown` message to the `ShardRegion` actor to handoff all shards that are hosted by that `ShardRegion` and then the `ShardRegion` actor will be stopped. You can `watch` the `ShardRegion` actor to know when it is completed. During this period other regions will buffer messages for those shards in the same way as when a rebalance is triggered by the coordinator. When the shards have been stopped the coordinator will allocate these shards elsewhere.

When the `ShardRegion` has terminated you probably want to `leave` the cluster, and shut down the `ActorSystem`.

This is how to do that:

```
class IllustrateGracefulShutdown extends Actor {
  val system = context.system
  val cluster = Cluster(system)
  val region = ClusterSharding(system).shardRegion("Entity")

  def receive = {
    case "leave" ⇒
      context.watch(region)
      region ! ShardRegion.GracefulShutdown

    case Terminated(`region`) ⇒
      cluster.registerOnMemberRemoved(self ! "member-removed")
      cluster.leave(cluster.selfAddress)

    case "member-removed" ⇒
      // Let singletons hand over gracefully before stopping the system
      import context.dispatcher
      system.scheduler.scheduleOnce(10.seconds, self, "stop-system")

    case "stop-system" ⇒
      system.terminate()
  }
}
```

5.6.10 Removal of Internal Cluster Sharding Data

The Cluster Sharding coordinator stores the locations of the shards using Akka Persistence. This data can safely be removed when restarting the whole Akka Cluster. Note that this is not application data.

There is a utility program `akka.cluster.sharding.RemoveInternalClusterShardingData` that removes this data.

Warning: Never use this program while there are running Akka Cluster nodes that are using Cluster Sharding. Stop all Cluster nodes before using this program.

It can be needed to remove the data if the Cluster Sharding coordinator cannot startup because of corrupt data, which may happen if accidentally two clusters were running at the same time, e.g. caused by using auto-down and there was a network partition.

Warning: **Don't use Cluster Sharding together with Automatic Downing**, since it allows the cluster to split up into two separate clusters, which in turn will result in *multiple shards and entities* being started, one in each separate cluster! See *Downing*.

Use this program as a standalone Java main program:

```
java -classpath <jar files, including akka-cluster-sharding>
  akka.cluster.sharding.RemoveInternalClusterShardingData
    -2.3 entityType1 entityType2 entityType3
```

The program is included in the `akka-cluster-sharding` jar file. It is easiest to run it with same classpath and configuration as your ordinary application. It can be run from sbt or maven in similar way.

Specify the entity type names (same as you use in the `start` method of `ClusterSharding`) as program arguments.

If you specify `-2.3` as the first program argument it will also try to remove data that was stored by Cluster Sharding in Akka 2.3.x using different persistenceId.

5.6.11 Dependencies

To use the Cluster Sharding you must add the following dependency in your project.

sbt:

```
"com.typesafe.akka" %% "akka-cluster-sharding" % "2.4.14"
```

maven:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-cluster-sharding_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

5.6.12 Configuration

The `ClusterSharding` extension can be configured with the following properties. These configuration properties are read by the `ClusterShardingSettings` when created with a `ActorSystem` parameter. It is also

possible to amend the `ClusterShardingSettings` or create it from another config section with the same layout as below. `ClusterShardingSettings` is a parameter to the `start` method of the `ClusterSharding` extension, i.e. each each entity type can be configured with different settings if needed.

```
# Settings for the ClusterShardingExtension
akka.cluster.sharding {

  # The extension creates a top level actor with this name in top level system␣
↪scope,
  # e.g. '/system/sharding'
  guardian-name = sharding

  # Specifies that entities runs on cluster nodes with a specific role.
  # If the role is not specified (or empty) all nodes in the cluster are used.
  role = ""

  # When this is set to 'on' the active entity actors will automatically be␣
↪restarted
  # upon Shard restart. i.e. if the Shard is started on a different ShardRegion
  # due to rebalance or crash.
  remember-entities = off

  # If the coordinator can't store state changes it will be stopped
  # and started again after this duration, with an exponential back-off
  # of up to 5 times this duration.
  coordinator-failure-backoff = 5 s

  # The ShardRegion retries registration and shard location requests to the
  # ShardCoordinator with this interval if it does not reply.
  retry-interval = 2 s

  # Maximum number of messages that are buffered by a ShardRegion actor.
  buffer-size = 100000

  # Timeout of the shard rebalancing process.
  handoff-timeout = 60 s

  # Time given to a region to acknowledge it's hosting a shard.
  shard-start-timeout = 10 s

  # If the shard is remembering entities and can't store state changes
  # will be stopped and then started again after this duration. Any messages
  # sent to an affected entity may be lost in this process.
  shard-failure-backoff = 10 s

  # If the shard is remembering entities and an entity stops itself without
  # using passivate. The entity will be restarted after this duration or when
  # the next message for it is received, which ever occurs first.
  entity-restart-backoff = 10 s

  # Rebalance check is performed periodically with this interval.
  rebalance-interval = 10 s

  # Absolute path to the journal plugin configuration entity that is to be
  # used for the internal persistence of ClusterSharding. If not defined
  # the default journal plugin is used. Note that this is not related to
  # persistence used by the entity actors.
  journal-plugin-id = ""

  # Absolute path to the snapshot plugin configuration entity that is to be
  # used for the internal persistence of ClusterSharding. If not defined
  # the default snapshot plugin is used. Note that this is not related to
  # persistence used by the entity actors.
```

```
  snapshot-plugin-id = ""

  # Parameter which determines how the coordinator will be store a state
  # valid values either "persistence" or "ddata"
  # The "ddata" mode is experimental, since it depends on the experimental
  # module akka-distributed-data-experimental.
  state-store-mode = "persistence"

  # The shard saves persistent snapshots after this number of persistent
  # events. Snapshots are used to reduce recovery times.
  snapshot-after = 1000

  # Setting for the default shard allocation strategy
  least-shard-allocation-strategy {
    # Threshold of how large the difference between most and least number of
    # allocated shards must be to begin the rebalancing.
    rebalance-threshold = 10

    # The number of ongoing rebalancing processes is limited to this number.
    max-simultaneous-rebalance = 3
  }

  # Timeout of waiting the initial distributed state (an initial state will be␣
↪queried again if the timeout happened)
  # works only for state-store-mode = "ddata"
  waiting-for-state-timeout = 5 s

  # Timeout of waiting for update the distributed state (update will be retried if␣
↪the timeout happened)
  # works only for state-store-mode = "ddata"
  updating-state-timeout = 5 s

  # The shard uses this strategy to determines how to recover the underlying␣
↪entity actors. The strategy is only used
  # by the persistent shard when rebalancing or restarting. The value can either␣
↪be "all" or "constant". The "all"
  # strategy start all the underlying entity actors at the same time. The constant␣
↪strategy will start the underlying
  # entity actors at a fix rate. The default strategy "all".
  entity-recovery-strategy = "all"

  # Default settings for the constant rate entity recovery strategy
  entity-recovery-constant-rate-strategy {
    # Sets the frequency at which a batch of entity actors is started.
    frequency = 100 ms
    # Sets the number of entity actors to be restart at a particular interval
    number-of-entities = 5
  }

  # Settings for the coordinator singleton. Same layout as akka.cluster.singleton.
  # The "role" of the singleton configuration is not used. The singleton role will
  # be the same as "akka.cluster.sharding.role".
  coordinator-singleton = ${akka.cluster.singleton}

  # The id of the dispatcher to use for ClusterSharding actors.
  # If not specified default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  # This dispatcher for the entity actors is defined by the user provided
  # Props, i.e. this dispatcher is not used for the entity actors.
  use-dispatcher = ""
}
```

Custom shard allocation strategy can be defined in an optional parameter to `ClusterSharding.start`. See

the API documentation of `ShardAllocationStrategy` for details of how to implement a custom shard allocation strategy.

5.6.13 Inspecting cluster sharding state

Two requests to inspect the cluster state are available:

`ShardRegion.GetShardRegionState` which will return a `ShardRegion.CurrentShardRegionState` that contains the identifiers of the shards running in a Region and what entities are alive for each of them.

`ShardRegion.GetClusterShardingStats` which will query all the regions in the cluster and return a `ShardRegion.ClusterShardingStats` containing the identifiers of the shards running in each region and a count of entities that are alive in each shard.

The purpose of these messages is testing and monitoring, they are not provided to give access to directly sending messages to the individual entities.

5.7 Cluster Metrics Extension

5.7.1 Introduction

The member nodes of the cluster can collect system health metrics and publish that to other cluster nodes and to the registered subscribers on the system event bus with the help of Cluster Metrics Extension.

Cluster metrics information is primarily used for load-balancing routers, and can also be used to implement advanced metrics-based node life cycles, such as "Node Let-it-crash" when CPU steal time becomes excessive.

Cluster Metrics Extension is a separate Akka module delivered in `akka-cluster-metrics` jar.

To enable usage of the extension you need to add the following dependency to your project:

```
"com.typesafe.akka" % "akka-cluster-metrics_2.11" % "2.4.14"
```

and add the following configuration stanza to your `application.conf`

```
akka.extensions = [ "akka.cluster.metrics.ClusterMetricsExtension" ]
```

Make sure to disable legacy metrics in akka-cluster: `akka.cluster.metrics.enabled=off`, since it is still enabled in akka-cluster by default (for compatibility with past releases).

Cluster members with status *WeaklyUp*, if that feature is enabled, will participate in Cluster Metrics collection and dissemination.

5.7.2 Metrics Collector

Metrics collection is delegated to an implementation of `akka.cluster.metrics.MetricsCollector`.

Different collector implementations provide different subsets of metrics published to the cluster. Certain message routing and let-it-crash functions may not work when Sigar is not provisioned.

Cluster metrics extension comes with two built-in collector implementations:

1. `akka.cluster.metrics.SigarMetricsCollector`, which requires Sigar provisioning, and is more rich/precise
2. `akka.cluster.metrics.JmxMetricsCollector`, which is used as fall back, and is less rich/precise

You can also plug-in your own metrics collector implementation.

By default, metrics extension will use collector provider fall back and will try to load them in this order:

1. configured user-provided collector
2. built-in `akka.cluster.metrics.SigarMetricsCollector`
3. and finally `akka.cluster.metrics.JmxMetricsCollector`

5.7.3 Metrics Events

Metrics extension periodically publishes current snapshot of the cluster metrics to the node system event bus.

The publication period is controlled by the `akka.cluster.metrics.collector.sample-period` setting.

The payload of the `akka.cluster.metrics.ClusterMetricsChanged` event will contain latest metrics of the node as well as other cluster member nodes metrics gossip which was received during the collector sample period.

You can subscribe your metrics listener actors to these events in order to implement custom node lifecycle

```
ClusterMetricsExtension(system).subscribe(metricsListenerActor)
```

5.7.4 Hyperic Sigar Provisioning

Both user-provided and built-in metrics collectors can optionally use Hyperic Sigar for a wider and more accurate range of metrics compared to what can be retrieved from ordinary JMX MBeans.

Sigar is using a native o/s library, and requires library provisioning, i.e. deployment, extraction and loading of the o/s native library into JVM at runtime.

User can provision Sigar classes and native library in one of the following ways:

1. Use Kamon sigar-loader as a project dependency for the user project. Metrics extension will extract and load sigar library on demand with help of Kamon sigar provisioner.
2. Use Kamon sigar-loader as java agent: `java -javaagent:/path/to/sigar-loader.jar`. Kamon sigar loader agent will extract and load sigar library during JVM start.
3. Place `sigar.jar` on the `classpath` and Sigar native library for the o/s on the `java.library.path`. User is required to manage both project dependency and library deployment manually.

Warning: When using Kamon sigar-loader and running multiple instances of the same application on the same host, you have to make sure that sigar library is extracted to a unique per instance directory. You can control the extract directory with the `akka.cluster.metrics.native-library-extract-folder` configuration setting.

To enable usage of Sigar you can add the following dependency to the user project

```
"io.kamon" % "sigar-loader" % "1.6.6-rev002"
```

You can download Kamon sigar-loader from Maven Central

5.7.5 Adaptive Load Balancing

The `AdaptiveLoadBalancingPool` / `AdaptiveLoadBalancingGroup` performs load balancing of messages to cluster nodes based on the cluster metrics data. It uses random selection of routees with probabilities derived from the remaining capacity of the corresponding node. It can be configured to use a specific MetricsSelector to produce the probabilities, a.k.a. weights:

- `heap` / `HeapMetricsSelector` - Used and max JVM heap memory. Weights based on remaining heap capacity; (max - used) / max

- `load` / `SystemLoadAverageMetricsSelector` - System load average for the past 1 minute, corresponding value can be found in `top` of Linux systems. The system is possibly nearing a bottleneck if the system load average is nearing number of cpus/cores. Weights based on remaining load capacity; 1 - (load / processors)
- `cpu` / `CpuMetricsSelector` - CPU utilization in percentage, sum of User + Sys + Nice + Wait. Weights based on remaining cpu capacity; 1 - utilization
- `mix` / `MixMetricsSelector` - Combines heap, cpu and load. Weights based on mean of remaining capacity of the combined selectors.
- Any custom implementation of `akka.cluster.metrics.MetricsSelector`

The collected metrics values are smoothed with exponential weighted moving average. In the *Configuration* you can adjust how quickly past data is decayed compared to new data.

Let's take a look at this router in action. What can be more demanding than calculating factorials?

The backend worker that performs the factorial calculation:

```
class FactorialBackend extends Actor with ActorLogging {

  import context.dispatcher

  def receive = {
    case (n: Int) =>
      Future(factorial(n)) map { result => (n, result) } pipeTo sender()
  }

  def factorial(n: Int): BigInt = {
    @tailrec def factorialAcc(acc: BigInt, n: Int): BigInt = {
      if (n <= 1) acc
      else factorialAcc(acc * n, n - 1)
    }
    factorialAcc(BigInt(1), n)
  }

}
```

The frontend that receives user jobs and delegates to the backends via the router:

```
class FactorialFrontend(upToN: Int, repeat: Boolean) extends Actor with␣
↪ActorLogging {

  val backend = context.actorOf(FromConfig.props(),
    name = "factorialBackendRouter")

  override def preStart(): Unit = {
    sendJobs()
    if (repeat) {
      context.setReceiveTimeout(10.seconds)
    }
  }

  def receive = {
    case (n: Int, factorial: BigInt) =>
      if (n == upToN) {
        log.debug("{}! = {}", n, factorial)
        if (repeat) sendJobs()
        else context.stop(self)
      }
    case ReceiveTimeout =>
      log.info("Timeout")
      sendJobs()
  }
```

```
  def sendJobs(): Unit = {
    log.info("Starting batch of factorials up to [{}]", upToN)
    1 to upToN foreach { backend ! _ }
  }
}
```

As you can see, the router is defined in the same way as other routers, and in this case it is configured as follows:

```
akka.actor.deployment {
  /factorialFrontend/factorialBackendRouter = {
    # Router type provided by metrics extension.
    router = cluster-metrics-adaptive-group
    # Router parameter specific for metrics extension.
    # metrics-selector = heap
    # metrics-selector = load
    # metrics-selector = cpu
    metrics-selector = mix
    #
    routees.paths = ["/user/factorialBackend"]
    cluster {
      enabled = on
      use-role = backend
      allow-local-routees = off
    }
  }
}
```

It is only `router` type and the `metrics-selector` parameter that is specific to this router, other things work in the same way as other routers.

The same type of router could also have been defined in code:

```
import akka.cluster.routing.ClusterRouterGroup
import akka.cluster.routing.ClusterRouterGroupSettings
import akka.cluster.metrics.AdaptiveLoadBalancingGroup
import akka.cluster.metrics.HeapMetricsSelector

val backend = context.actorOf(
  ClusterRouterGroup(AdaptiveLoadBalancingGroup(HeapMetricsSelector),
    ClusterRouterGroupSettings(
      totalInstances = 100, routeesPaths = List("/user/factorialBackend"),
      allowLocalRoutees = true, useRole = Some("backend"))).props(),
  name = "factorialBackendRouter2")
```

```
import akka.cluster.routing.ClusterRouterPool
import akka.cluster.routing.ClusterRouterPoolSettings
import akka.cluster.metrics.AdaptiveLoadBalancingPool
import akka.cluster.metrics.SystemLoadAverageMetricsSelector

val backend = context.actorOf(
  ClusterRouterPool(AdaptiveLoadBalancingPool(
    SystemLoadAverageMetricsSelector), ClusterRouterPoolSettings(
    totalInstances = 100, maxInstancesPerNode = 3,
    allowLocalRoutees = false, useRole = Some("backend"))).
↪props(Props[FactorialBackend]),
  name = "factorialBackendRouter3")
```

The Lightbend Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Adaptive Load Balancing** sample.

5.7.6 Subscribe to Metrics Events

It is possible to subscribe to the metrics events directly to implement other functionality.

```
import akka.actor.ActorLogging
import akka.actor.Actor
import akka.cluster.Cluster
import akka.cluster.metrics.ClusterMetricsEvent
import akka.cluster.metrics.ClusterMetricsChanged
import akka.cluster.ClusterEvent.CurrentClusterState
import akka.cluster.metrics.NodeMetrics
import akka.cluster.metrics.StandardMetrics.HeapMemory
import akka.cluster.metrics.StandardMetrics.Cpu
import akka.cluster.metrics.ClusterMetricsExtension

class MetricsListener extends Actor with ActorLogging {
  val selfAddress = Cluster(context.system).selfAddress
  val extension = ClusterMetricsExtension(context.system)

  // Subscribe unto ClusterMetricsEvent events.
  override def preStart(): Unit = extension.subscribe(self)

  // Unsubscribe from ClusterMetricsEvent events.
  override def postStop(): Unit = extension.unsubscribe(self)

  def receive = {
    case ClusterMetricsChanged(clusterMetrics) =>
      clusterMetrics.filter(_.address == selfAddress) foreach { nodeMetrics =>
        logHeap(nodeMetrics)
        logCpu(nodeMetrics)
      }
    case state: CurrentClusterState => // Ignore.
  }

  def logHeap(nodeMetrics: NodeMetrics): Unit = nodeMetrics match {
    case HeapMemory(address, timestamp, used, committed, max) =>
      log.info("Used heap: {} MB", used.doubleValue / 1024 / 1024)
    case _ => // No heap info.
  }

  def logCpu(nodeMetrics: NodeMetrics): Unit = nodeMetrics match {
    case Cpu(address, timestamp, Some(systemLoadAverage), cpuCombined, cpuStolen,␣
↪processors) =>
      log.info("Load: {} ({} processors)", systemLoadAverage, processors)
    case _ => // No cpu info.
  }
}
```

5.7.7 Custom Metrics Collector

Metrics collection is delegated to the implementation of `akka.cluster.metrics.MetricsCollector`

You can plug-in your own metrics collector instead of built-in `akka.cluster.metrics.SigarMetricsCollector` or `akka.cluster.metrics.JmxMetricsCollector`.

Look at those two implementations for inspiration.

Custom metrics collector implementation class must be specified in the `akka.cluster.metrics.collector.provider` configuration property.

5.7.8 Configuration

The Cluster metrics extension can be configured with the following properties:

```
############################################
# Akka Cluster Metrics Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits in your application.conf in order to override these settings.

# Sigar provisioning:
#
#  User can provision sigar classes and native library in one of the following ways:
#
#  1) Use https://github.com/kamon-io/sigar-loader Kamon sigar-loader as a project dependency for the user project.
#  Metrics extension will extract and load sigar library on demand with help of Kamon sigar provisioner.
#
#  2) Use https://github.com/kamon-io/sigar-loader Kamon sigar-loader as java agent: `java -javaagent:/path/to/sigar-loader.jar`
#  Kamon sigar loader agent will extract and load sigar library during JVM start.
#
#  3) Place `sigar.jar` on the `classpath` and sigar native library for the o/s on the `java.library.path`
#  User is required to manage both project dependency and library deployment manually.

# Cluster metrics extension.
# Provides periodic statistics collection and publication throughout the cluster.
akka.cluster.metrics {
    # Full path of dispatcher configuration key.
    # Use "" for default key `akka.actor.default-dispatcher`.
    dispatcher = ""
    # How long should any actor wait before starting the periodic tasks.
    periodic-tasks-initial-delay = 1s
    # Sigar native library extract location.
    # Use per-application-instance scoped location, such as program working directory.
    native-library-extract-folder = ${user.dir}"/native"
    # Metrics supervisor actor.
    supervisor {
        # Actor name. Example name space: /system/cluster-metrics
        name = "cluster-metrics"
        # Supervision strategy.
        strategy {
            #
            # FQCN of class providing `akka.actor.SupervisorStrategy`.
            # Must have a constructor with signature `<init>(com.typesafe.config.Config)`.
            # Default metrics strategy provider is a configurable extension of `OneForOneStrategy`.
            provider = "akka.cluster.metrics.ClusterMetricsStrategy"
            #
            # Configuration of the default strategy provider.
            # Replace with custom settings when overriding the provider.
            configuration = {
                # Log restart attempts.
                loggingEnabled = true
                # Child actor restart-on-failure window.
                withinTimeRange = 3s
```

```
                # Maximum number of restart attempts before child actor is stopped.
                maxNrOfRetries = 3
            }
        }
    }
    # Metrics collector actor.
    collector {
        # Enable or disable metrics collector for load-balancing nodes.
        # Metrics collection can also be controlled at runtime by sending control␣
↪messages
        # to /system/cluster-metrics actor: `akka.cluster.metrics.
↪{CollectionStartMessage,CollectionStopMessage}`
        enabled = on
        # FQCN of the metrics collector implementation.
        # It must implement `akka.cluster.metrics.MetricsCollector` and
        # have public constructor with akka.actor.ActorSystem parameter.
        # Will try to load in the following order of priority:
        # 1) configured custom collector 2) internal `SigarMetricsCollector` 3)␣
↪internal `JmxMetricsCollector`
        provider = ""
        # Try all 3 available collector providers, or else fail on the configured␣
↪custom collector provider.
        fallback = true
        # How often metrics are sampled on a node.
        # Shorter interval will collect the metrics more often.
        # Also controls frequency of the metrics publication to the node system␣
↪event bus.
        sample-interval = 3s
        # How often a node publishes metrics information to the other nodes in the␣
↪cluster.
        # Shorter interval will publish the metrics gossip more often.
        gossip-interval = 3s
        # How quickly the exponential weighting of past data is decayed compared to
        # new data. Set lower to increase the bias toward newer values.
        # The relevance of each data sample is halved for every passing half-life
        # duration, i.e. after 4 times the half-life, a data sample's relevance is
        # reduced to 6% of its original relevance. The initial relevance of a data
        # sample is given by 1 - 0.5 ^ (collect-interval / half-life).
        # See http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_
↪average
        moving-average-half-life = 12s
    }
}

# Cluster metrics extension serializers and routers.
akka.actor {
    # Protobuf serializer for remote cluster metrics messages.
    serializers {
       akka-cluster-metrics = "akka.cluster.metrics.protobuf.MessageSerializer"
    }
    # Interface binding for remote cluster metrics messages.
    serialization-bindings {
       "akka.cluster.metrics.ClusterMetricsMessage" = akka-cluster-metrics
    }
    # Globally unique metrics extension serializer identifier.
    serialization-identifiers {
      "akka.cluster.metrics.protobuf.MessageSerializer" = 10
    }
    #  Provide routing of messages based on cluster metrics.
    router.type-mapping {
        cluster-metrics-adaptive-pool  = "akka.cluster.metrics.
↪AdaptiveLoadBalancingPool"
        cluster-metrics-adaptive-group = "akka.cluster.metrics.
↪AdaptiveLoadBalancingGroup"
```

```
    }
}
```

5.8 Distributed Data

Akka Distributed Data is useful when you need to share data between nodes in an Akka Cluster. The data is accessed with an actor providing a key-value store like API. The keys are unique identifiers with type information of the data values. The values are *Conflict Free Replicated Data Types* (CRDTs).

All data entries are spread to all nodes, or nodes with a certain role, in the cluster via direct replication and gossip based dissemination. You have fine grained control of the consistency level for reads and writes.

The nature CRDTs makes it possible to perform updates from any node without coordination. Concurrent updates from different nodes will automatically be resolved by the monotonic merge function, which all data types must provide. The state changes always converge. Several useful data types for counters, sets, maps and registers are provided and you can also implement your own custom data types.

It is eventually consistent and geared toward providing high read and write availability (partition tolerance), with low latency. Note that in an eventually consistent system a read may return an out-of-date value.

Warning: This module is marked as **"experimental"** as of its introduction in Akka 2.4.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply to the contents of the `akka.persistence` package.

5.8.1 Using the Replicator

The `akka.cluster.ddata.Replicator` actor provides the API for interacting with the data. The `Replicator` actor must be started on each node in the cluster, or group of nodes tagged with a specific role. It communicates with other `Replicator` instances with the same path (without address) that are running on other nodes . For convenience it can be used with the `akka.cluster.ddata.DistributedData` extension.

Cluster members with status *WeaklyUp*, if that feature is enabled, will participate in Distributed Data. This means that the data will be replicated to the *WeaklyUp* nodes with the background gossip protocol. Note that it will not participate in any actions where the consistency mode is to read/write from all nodes or the majority of nodes. The *WeaklyUp* node is not counted as part of the cluster. So 3 nodes + 5 *WeaklyUp* is essentially a 3 node cluster as far as consistent actions are concerned.

Below is an example of an actor that schedules tick messages to itself and for each tick adds or removes elements from a `ORSet` (observed-remove set). It also subscribes to changes of this.

```
import java.util.concurrent.ThreadLocalRandom
import akka.actor.Actor
import akka.actor.ActorLogging
import akka.cluster.Cluster
import akka.cluster.ddata.DistributedData
import akka.cluster.ddata.ORSet
import akka.cluster.ddata.ORSetKey
import akka.cluster.ddata.Replicator
import akka.cluster.ddata.Replicator._

object DataBot {
  private case object Tick
}

class DataBot extends Actor with ActorLogging {
  import DataBot._
```

```
  val replicator = DistributedData(context.system).replicator
  implicit val node = Cluster(context.system)

  import context.dispatcher
  val tickTask = context.system.scheduler.schedule(5.seconds, 5.seconds, self,␣
↪Tick)

  val DataKey = ORSetKey[String]("key")

  replicator ! Subscribe(DataKey, self)

  def receive = {
    case Tick =>
      val s = ThreadLocalRandom.current().nextInt(97, 123).toChar.toString
      if (ThreadLocalRandom.current().nextBoolean()) {
        // add
        log.info("Adding: {}", s)
        replicator ! Update(DataKey, ORSet.empty[String], WriteLocal)(_ + s)
      } else {
        // remove
        log.info("Removing: {}", s)
        replicator ! Update(DataKey, ORSet.empty[String], WriteLocal)(_ - s)
      }

    case _: UpdateResponse[_] => // ignore

    case c @ Changed(DataKey) =>
      val data = c.get(DataKey)
      log.info("Current elements: {}", data.elements)
  }

  override def postStop(): Unit = tickTask.cancel()

}
```

Update

To modify and replicate a data value you send a `Replicator.Update` message to the local `Replicator`.

The current data value for the `key` of the `Update` is passed as parameter to the `modify` function of the `Update`. The function is supposed to return the new value of the data, which will then be replicated according to the given consistency level.

The `modify` function is called by the `Replicator` actor and must therefore be a pure function that only uses the data parameter and stable fields from enclosing scope. It must for example not access `sender()` reference of an enclosing actor.

`Update` is intended to only be sent from an actor running in same local `ActorSystem` as

- the *Replicator*, because the *modify* function is typically not serializable.

You supply a write consistency level which has the following meaning:

- `WriteLocal` the value will immediately only be written to the local replica, and later disseminated with gossip
- `WriteTo(n)` the value will immediately be written to at least `n` replicas, including the local replica
- `WriteMajority` the value will immediately be written to a majority of replicas, i.e. at least **N/2 + 1** replicas, where N is the number of nodes in the cluster (or cluster role group)
- `WriteAll` the value will immediately be written to all nodes in the cluster (or all nodes in the cluster role group)

```
implicit val node = Cluster(system)
val replicator = DistributedData(system).replicator

val Counter1Key = PNCounterKey("counter1")
val Set1Key = GSetKey[String]("set1")
val Set2Key = ORSetKey[String]("set2")
val ActiveFlagKey = FlagKey("active")

replicator ! Update(Counter1Key, PNCounter(), WriteLocal)(_ + 1)

val writeTo3 = WriteTo(n = 3, timeout = 1.second)
replicator ! Update(Set1Key, GSet.empty[String], writeTo3)(_ + "hello")

val writeMajority = WriteMajority(timeout = 5.seconds)
replicator ! Update(Set2Key, ORSet.empty[String], writeMajority)(_ + "hello")

val writeAll = WriteAll(timeout = 5.seconds)
replicator ! Update(ActiveFlagKey, Flag.empty, writeAll)(_.switchOn)
```

As reply of the `Update` a `Replicator.UpdateSuccess` is sent to the sender of the `Update` if the value was successfully replicated according to the supplied consistency level within the supplied timeout. Otherwise a `Replicator.UpdateFailure` subclass is sent back. Note that a `Replicator.UpdateTimeout` reply does not mean that the update completely failed or was rolled back. It may still have been replicated to some nodes, and will eventually be replicated to all nodes with the gossip protocol.

```
case UpdateSuccess(Counter1Key, req) => // ok
```

```
case UpdateSuccess(Set1Key, req)   => // ok
case UpdateTimeout(Set1Key, req)   =>
// write to 3 nodes failed within 1.second
```

You will always see your own writes. For example if you send two `Update` messages changing the value of the same `key`, the `modify` function of the second message will see the change that was performed by the first `Update` message.

In the `Update` message you can pass an optional request context, which the `Replicator` does not care about, but is included in the reply messages. This is a convenient way to pass contextual information (e.g. original sender) without having to use `ask` or maintain local correlation data structures.

```
implicit val node = Cluster(system)
val replicator = DistributedData(system).replicator
val writeTwo = WriteTo(n = 2, timeout = 3.second)
val Counter1Key = PNCounterKey("counter1")

def receive: Receive = {
  case "increment" =>
    // incoming command to increase the counter
    val upd = Update(Counter1Key, PNCounter(), writeTwo, request =␣
↪Some(sender()))(_ + 1)
    replicator ! upd

  case UpdateSuccess(Counter1Key, Some(replyTo: ActorRef)) =>
    replyTo ! "ack"
  case UpdateTimeout(Counter1Key, Some(replyTo: ActorRef)) =>
    replyTo ! "nack"
}
```

Get

To retrieve the current value of a data you send `Replicator.Get` message to the `Replicator`. You supply a consistency level which has the following meaning:

- `ReadLocal` the value will only be read from the local replica
- `ReadFrom(n)` the value will be read and merged from `n` replicas, including the local replica
- `ReadMajority` the value will be read and merged from a majority of replicas, i.e. at least **N/2 + 1** replicas, where N is the number of nodes in the cluster (or cluster role group)
- `ReadAll` the value will be read and merged from all nodes in the cluster (or all nodes in the cluster role group)

```
val replicator = DistributedData(system).replicator
val Counter1Key = PNCounterKey("counter1")
val Set1Key = GSetKey[String]("set1")
val Set2Key = ORSetKey[String]("set2")
val ActiveFlagKey = FlagKey("active")

replicator ! Get(Counter1Key, ReadLocal)

val readFrom3 = ReadFrom(n = 3, timeout = 1.second)
replicator ! Get(Set1Key, readFrom3)

val readMajority = ReadMajority(timeout = 5.seconds)
replicator ! Get(Set2Key, readMajority)

val readAll = ReadAll(timeout = 5.seconds)
replicator ! Get(ActiveFlagKey, readAll)
```

As reply of the `Get` a `Replicator.GetSuccess` is sent to the sender of the `Get` if the value was successfully retrieved according to the supplied consistency level within the supplied timeout. Otherwise a `Replicator.GetFailure` is sent. If the key does not exist the reply will be `Replicator.NotFound`.

```
case g @ GetSuccess(Counter1Key, req) =>
  val value = g.get(Counter1Key).value
case NotFound(Counter1Key, req) => // key counter1 does not exist
```

```
case g @ GetSuccess(Set1Key, req) =>
  val elements = g.get(Set1Key).elements
case GetFailure(Set1Key, req) =>
// read from 3 nodes failed within 1.second
case NotFound(Set1Key, req)   => // key set1 does not exist
```

You will always read your own writes. For example if you send a `Update` message followed by a `Get` of the same `key` the `Get` will retrieve the change that was performed by the preceding `Update` message. However, the order of the reply messages are not defined, i.e. in the previous example you may receive the `GetSuccess` before the `UpdateSuccess`.

In the `Get` message you can pass an optional request context in the same way as for the `Update` message, described above. For example the original sender can be passed and replied to after receiving and transforming `GetSuccess`.

```
implicit val node = Cluster(system)
val replicator = DistributedData(system).replicator
val readTwo = ReadFrom(n = 2, timeout = 3.second)
val Counter1Key = PNCounterKey("counter1")

def receive: Receive = {
  case "get-count" =>
    // incoming request to retrieve current value of the counter
```

```
    replicator ! Get(Counter1Key, readTwo, request = Some(sender()))

  case g @ GetSuccess(Counter1Key, Some(replyTo: ActorRef)) =>
    val value = g.get(Counter1Key).value.longValue
    replyTo ! value
  case GetFailure(Counter1Key, Some(replyTo: ActorRef)) =>
    replyTo ! -1L
  case NotFound(Counter1Key, Some(replyTo: ActorRef)) =>
    replyTo ! 0L
}
```

Consistency

The consistency level that is supplied in the *Update* and *Get* specifies per request how many replicas that must respond successfully to a write and read request.

For low latency reads you use `ReadLocal` with the risk of retrieving stale data, i.e. updates from other nodes might not be visible yet.

When using `WriteLocal` the update is only written to the local replica and then disseminated in the background with the gossip protocol, which can take few seconds to spread to all nodes.

`WriteAll` and `ReadAll` is the strongest consistency level, but also the slowest and with lowest availability. For example, it is enough that one node is unavailable for a `Get` request and you will not receive the value.

If consistency is important, you can ensure that a read always reflects the most recent write by using the following formula:

```
(nodes_written + nodes_read) > N
```

where N is the total number of nodes in the cluster, or the number of nodes with the role that is used for the `Replicator`.

For example, in a 7 node cluster this these consistency properties are achieved by writing to 4 nodes and reading from 4 nodes, or writing to 5 nodes and reading from 3 nodes.

By combining `WriteMajority` and `ReadMajority` levels a read always reflects the most recent write. The `Replicator` writes and reads to a majority of replicas, i.e. **N / 2 + 1**. For example, in a 5 node cluster it writes to 3 nodes and reads from 3 nodes. In a 6 node cluster it writes to 4 nodes and reads from 4 nodes.

Here is an example of using `WriteMajority` and `ReadMajority`:

```
private val timeout = 3.seconds
private val readMajority = ReadMajority(timeout)
private val writeMajority = WriteMajority(timeout)
```

```
def receiveGetCart: Receive = {
  case GetCart =>
    replicator ! Get(DataKey, readMajority, Some(sender()))

  case g @ GetSuccess(DataKey, Some(replyTo: ActorRef)) =>
    val data = g.get(DataKey)
    val cart = Cart(data.entries.values.toSet)
    replyTo ! cart

  case NotFound(DataKey, Some(replyTo: ActorRef)) =>
    replyTo ! Cart(Set.empty)

  case GetFailure(DataKey, Some(replyTo: ActorRef)) =>
    // ReadMajority failure, try again with local read
    replicator ! Get(DataKey, ReadLocal, Some(replyTo))
}
```

```
def receiveAddItem: Receive = {
  case cmd @ AddItem(item) =>
    val update = Update(DataKey, LWWMap.empty[LineItem], writeMajority, Some(cmd))
↪{
      cart => updateCart(cart, item)
    }
    replicator ! update
}
```

In some rare cases, when performing an `Update` it is needed to first try to fetch latest data from other nodes. That can be done by first sending a `Get` with `ReadMajority` and then continue with the `Update` when the `GetSuccess`, `GetFailure` or `NotFound` reply is received. This might be needed when you need to base a decision on latest information or when removing entries from `ORSet` or `ORMap`. If an entry is added to an `ORSet` or `ORMap` from one node and removed from another node the entry will only be removed if the added entry is visible on the node where the removal is performed (hence the name observed-removed set).

The following example illustrates how to do that:

```
def receiveRemoveItem: Receive = {
  case cmd @ RemoveItem(productId) =>
    // Try to fetch latest from a majority of nodes first, since ORMap
    // remove must have seen the item to be able to remove it.
    replicator ! Get(DataKey, readMajority, Some(cmd))

  case GetSuccess(DataKey, Some(RemoveItem(productId))) =>
    replicator ! Update(DataKey, LWWMap(), writeMajority, None) {
      _ - productId
    }

  case GetFailure(DataKey, Some(RemoveItem(productId))) =>
    // ReadMajority failed, fall back to best effort local value
    replicator ! Update(DataKey, LWWMap(), writeMajority, None) {
      _ - productId
    }

  case NotFound(DataKey, Some(RemoveItem(productId))) =>
  // nothing to remove
}
```

Warning: *Caveat:* Even if you use `WriteMajority` and `ReadMajority` there is small risk that you may read stale data if the cluster membership has changed between the `Update` and the `Get`. For example, in cluster of 5 nodes when you `Update` and that change is written to 3 nodes: n1, n2, n3. Then 2 more nodes are added and a `Get` request is reading from 4 nodes, which happens to be n4, n5, n6, n7, i.e. the value on n1, n2, n3 is not seen in the response of the `Get` request.

Subscribe

You may also register interest in change notifications by sending `Replicator.Subscribe` message to the `Replicator`. It will send `Replicator.Changed` messages to the registered subscriber when the data for the subscribed key is updated. Subscribers will be notified periodically with the configured `notify-subscribers-interval`, and it is also possible to send an explicit `Replicator.FlushChanges` message to the `Replicator` to notify the subscribers immediately.

The subscriber is automatically removed if the subscriber is terminated. A subscriber can also be deregistered with the `Replicator.Unsubscribe` message.

```
val replicator = DistributedData(system).replicator
val Counter1Key = PNCounterKey("counter1")
// subscribe to changes of the Counter1Key value
```

```
replicator ! Subscribe(Counter1Key, self)
var currentValue = BigInt(0)

def receive: Receive = {
  case c @ Changed(Counter1Key) =>
    currentValue = c.get(Counter1Key).value
  case "get-count" =>
    // incoming request to retrieve current value of the counter
    sender() ! currentValue
}
```

Delete

A data entry can be deleted by sending a `Replicator.Delete` message to the local local `Replicator`. As reply of the `Delete` a `Replicator.DeleteSuccess` is sent to the sender of the `Delete` if the value was successfully deleted according to the supplied consistency level within the supplied timeout. Otherwise a `Replicator.ReplicationDeleteFailure` is sent. Note that `ReplicationDeleteFailure` does not mean that the delete completely failed or was rolled back. It may still have been replicated to some nodes, and may eventually be replicated to all nodes.

A deleted key cannot be reused again, but it is still recommended to delete unused data entries because that reduces the replication overhead when new nodes join the cluster. Subsequent `Delete`, `Update` and `Get` requests will be replied with `Replicator.DataDeleted`. Subscribers will receive `Replicator.DataDeleted`.

```
val replicator = DistributedData(system).replicator
val Counter1Key = PNCounterKey("counter1")
val Set2Key = ORSetKey[String]("set2")

replicator ! Delete(Counter1Key, WriteLocal)

val writeMajority = WriteMajority(timeout = 5.seconds)
replicator ! Delete(Set2Key, writeMajority)
```

Warning: As deleted keys continue to be included in the stored data on each node as well as in gossip messages, a continuous series of updates and deletes of top-level entities will result in growing memory usage until an ActorSystem runs out of memory. To use Akka Distributed Data where frequent adds and removes are required, you should use a fixed number of top-level data types that support both updates and removals, for example `ORMap` or `ORSet`.

5.8.2 Data Types

The data types must be convergent (stateful) CRDTs and implement the `ReplicatedData` trait, i.e. they provide a monotonic merge function and the state changes always converge.

You can use your own custom `ReplicatedData` types, and several types are provided by this package, such as:

- Counters: `GCounter`, `PNCounter`
- Sets: `GSet`, `ORSet`
- Maps: `ORMap`, `ORMultiMap`, `LWWMap`, `PNCounterMap`
- Registers: `LWWRegister`, `Flag`

Counters

`GCounter` is a "grow only counter". It only supports increments, no decrements.

It works in a similar way as a vector clock. It keeps track of one counter per node and the total value is the sum of these counters. The `merge` is implemented by taking the maximum count for each node.

If you need both increments and decrements you can use the `PNCounter` (positive/negative counter).

It is tracking the increments (P) separate from the decrements (N). Both P and N are represented as two internal `GCounter`. Merge is handled by merging the internal P and N counters. The value of the counter is the value of the P counter minus the value of the N counter.

```
implicit val node = Cluster(system)
val c0 = PNCounter.empty
val c1 = c0 + 1
val c2 = c1 + 7
val c3: PNCounter = c2 - 2
println(c3.value) // 6
```

Several related counters can be managed in a map with the `PNCounterMap` data type. When the counters are placed in a `PNCounterMap` as opposed to placing them as separate top level values they are guaranteed to be replicated together as one unit, which is sometimes necessary for related data.

```
implicit val node = Cluster(system)
val m0 = PNCounterMap.empty
val m1 = m0.increment("a", 7)
val m2 = m1.decrement("a", 2)
val m3 = m2.increment("b", 1)
println(m3.get("a")) // 5
m3.entries.foreach { case (key, value) => println(s"$key -> $value") }
```

Sets

If you only need to add elements to a set and not remove elements the `GSet` (grow-only set) is the data type to use. The elements can be any type of values that can be serialized. Merge is simply the union of the two sets.

```
val s0 = GSet.empty[String]
val s1 = s0 + "a"
val s2 = s1 + "b" + "c"
if (s2.contains("a"))
  println(s2.elements) // a, b, c
```

If you need add and remove operations you should use the `ORSet` (observed-remove set). Elements can be added and removed any number of times. If an element is concurrently added and removed, the add will win. You cannot remove an element that you have not seen.

The `ORSet` has a version vector that is incremented when an element is added to the set. The version for the node that added the element is also tracked for each element in a so called "birth dot". The version vector and the dots are used by the `merge` function to track causality of the operations and resolve concurrent updates.

```
implicit val node = Cluster(system)
val s0 = ORSet.empty[String]
val s1 = s0 + "a"
val s2 = s1 + "b"
val s3 = s2 - "a"
println(s3.elements) // b
```

Maps

`ORMap` (observed-remove map) is a map with `String` keys and the values are `ReplicatedData` types themselves. It supports add, remove and delete any number of times for a map entry.

If an entry is concurrently added and removed, the add will win. You cannot remove an entry that you have not seen. This is the same semantics as for the `ORSet`.

If an entry is concurrently updated to different values the values will be merged, hence the requirement that the values must be `ReplicatedData` types.

It is rather inconvenient to use the `ORMap` directly since it does not expose specific types of the values. The `ORMap` is intended as a low level tool for building more specific maps, such as the following specialized maps.

`ORMultiMap` (observed-remove multi-map) is a multi-map implementation that wraps an `ORMap` with an `ORSet` for the map's value.

`PNCounterMap` (positive negative counter map) is a map of named counters. It is a specialized `ORMap` with `PNCounter` values.

`LWWMap` (last writer wins map) is a specialized `ORMap` with `LWWRegister` (last writer wins register) values.

```
implicit val node = Cluster(system)
val m0 = ORMultiMap.empty[Int]
val m1 = m0 + ("a" -> Set(1, 2, 3))
val m2 = m1.addBinding("a", 4)
val m3 = m2.removeBinding("a", 2)
val m4 = m3.addBinding("b", 1)
println(m4.entries)
```

When a data entry is changed the full state of that entry is replicated to other nodes, i.e. when you update a map the whole map is replicated. Therefore, instead of using one `ORMap` with 1000 elements it is more efficient to split that up in 10 top level `ORMap` entries with 100 elements each. Top level entries are replicated individually, which has the trade-off that different entries may not be replicated at the same time and you may see inconsistencies between related entries. Separate top level entries cannot be updated atomically together.

Note that `LWWRegister` and therefore `LWWMap` relies on synchronized clocks and should only be used when the choice of value is not important for concurrent updates occurring within the clock skew. Read more in the below section about `LWWRegister`.

Flags and Registers

`Flag` is a data type for a boolean value that is initialized to `false` and can be switched to `true`. Thereafter it cannot be changed. `true` wins over `false` in merge.

```
val f0 = Flag.empty
val f1 = f0.switchOn
println(f1.enabled)
```

`LWWRegister` (last writer wins register) can hold any (serializable) value.

Merge of a `LWWRegister` takes the register with highest timestamp. Note that this relies on synchronized clocks. *LWWRegister* should only be used when the choice of value is not important for concurrent updates occurring within the clock skew.

Merge takes the register updated by the node with lowest address (`UniqueAddress` is ordered) if the timestamps are exactly the same.

```
implicit val node = Cluster(system)
val r1 = LWWRegister("Hello")
val r2 = r1.withValue("Hi")
println(s"${r1.value} by ${r1.updatedBy} at ${r1.timestamp}")
```

Instead of using timestamps based on `System.currentTimeMillis()` time it is possible to use a timestamp value based on something else, for example an increasing version number from a database record that is used for optimistic concurrency control.

```
case class Record(version: Int, name: String, address: String)

implicit val node = Cluster(system)
implicit val recordClock = new LWWRegister.Clock[Record] {
```

```
  override def apply(currentTimestamp: Long, value: Record): Long =
    value.version
}

val record1 = Record(version = 1, "Alice", "Union Square")
val r1 = LWWRegister(record1)

val record2 = Record(version = 2, "Alice", "Madison Square")
val r2 = LWWRegister(record2)

val r3 = r1.merge(r2)
println(r3.value)
```

For first-write-wins semantics you can use the `LWWRegister#reverseClock` instead of the `LWWRegister#defaultClock`.

The `defaultClock` is using max value of `System.currentTimeMillis()` and `currentTimestamp + 1`. This means that the timestamp is increased for changes on the same node that occurs within the same millisecond. It also means that it is safe to use the `LWWRegister` without synchronized clocks when there is only one active writer, e.g. a Cluster Singleton. Such a single writer should then first read current value with `ReadMajority` (or more) before changing and writing the value with `WriteMajority` (or more).

Custom Data Type

You can rather easily implement your own data types. The only requirement is that it implements the `merge` function of the `ReplicatedData` trait.

A nice property of stateful CRDTs is that they typically compose nicely, i.e. you can combine several smaller data types to build richer data structures. For example, the `PNCounter` is composed of two internal `GCounter` instances to keep track of increments and decrements separately.

Here is s simple implementation of a custom `TwoPhaseSet` that is using two internal `GSet` types to keep track of addition and removals. A `TwoPhaseSet` is a set where an element may be added and removed, but never added again thereafter.

```
case class TwoPhaseSet(
  adds:     GSet[String] = GSet.empty,
  removals: GSet[String] = GSet.empty)
  extends ReplicatedData {
  type T = TwoPhaseSet

  def add(element: String): TwoPhaseSet =
    copy(adds = adds.add(element))

  def remove(element: String): TwoPhaseSet =
    copy(removals = removals.add(element))

  def elements: Set[String] = adds.elements diff removals.elements

  override def merge(that: TwoPhaseSet): TwoPhaseSet =
    copy(
      adds = this.adds.merge(that.adds),
      removals = this.removals.merge(that.removals))
}
```

Data types should be immutable, i.e. "modifying" methods should return a new instance.

Serialization

The data types must be serializable with an *Akka Serializer*. It is highly recommended that you implement efficient serialization with Protobuf or similar for your custom data types. The built in data types are marked with `ReplicatedDataSerialization` and serialized with `akka.cluster.ddata.protobuf.ReplicatedDataSerializer`.

Serialization of the data types are used in remote messages and also for creating message digests (SHA-1) to detect changes. Therefore it is important that the serialization is efficient and produce the same bytes for the same content. For example sets and maps should be sorted deterministically in the serialization.

This is a protobuf representation of the above `TwoPhaseSet`:

```
option java_package = "docs.ddata.protobuf.msg";
option optimize_for = SPEED;

message TwoPhaseSet {
  repeated string adds = 1;
  repeated string removals = 2;
}
```

The serializer for the `TwoPhaseSet`:

```
import java.util.ArrayList
import java.util.Collections
import scala.collection.JavaConverters._
import akka.actor.ExtendedActorSystem
import akka.cluster.ddata.GSet
import akka.cluster.ddata.protobuf.SerializationSupport
import akka.serialization.Serializer
import docs.ddata.TwoPhaseSet
import docs.ddata.protobuf.msg.TwoPhaseSetMessages

class TwoPhaseSetSerializer(val system: ExtendedActorSystem)
  extends Serializer with SerializationSupport {

  override def includeManifest: Boolean = false

  override def identifier = 99999

  override def toBinary(obj: AnyRef): Array[Byte] = obj match {
    case m: TwoPhaseSet => twoPhaseSetToProto(m).toByteArray
    case _ => throw new IllegalArgumentException(
      s"Can't serialize object of type ${obj.getClass}")
  }

  override def fromBinary(bytes: Array[Byte], clazz: Option[Class[_]]): AnyRef = {
    twoPhaseSetFromBinary(bytes)
  }

  def twoPhaseSetToProto(twoPhaseSet: TwoPhaseSet): TwoPhaseSetMessages.
↪TwoPhaseSet = {
    val b = TwoPhaseSetMessages.TwoPhaseSet.newBuilder()
    // using java collections and sorting for performance (avoid conversions)
    val adds = new ArrayList[String]
    twoPhaseSet.adds.elements.foreach(adds.add)
    if (!adds.isEmpty) {
      Collections.sort(adds)
      b.addAllAdds(adds)
    }
    val removals = new ArrayList[String]
    twoPhaseSet.removals.elements.foreach(removals.add)
    if (!removals.isEmpty) {
```

```
      Collections.sort(removals)
      b.addAllRemovals(removals)
    }
    b.build()
  }

  def twoPhaseSetFromBinary(bytes: Array[Byte]): TwoPhaseSet = {
    val msg = TwoPhaseSetMessages.TwoPhaseSet.parseFrom(bytes)
    TwoPhaseSet(
      adds = GSet(msg.getAddsList.iterator.asScala.toSet),
      removals = GSet(msg.getRemovalsList.iterator.asScala.toSet))
  }
}
```

Note that the elements of the sets are sorted so the SHA-1 digests are the same for the same elements.

You register the serializer in configuration:

```
akka.actor {
  serializers {
    two-phase-set = "docs.ddata.protobuf.TwoPhaseSetSerializer"
  }
  serialization-bindings {
    "docs.ddata.TwoPhaseSet" = two-phase-set
  }
}
```

Using compression can sometimes be a good idea to reduce the data size. Gzip compression is provided by the `akka.cluster.ddata.protobuf.SerializationSupport` trait:

```
override def toBinary(obj: AnyRef): Array[Byte] = obj match {
  case m: TwoPhaseSet => compress(twoPhaseSetToProto(m))
  case _ => throw new IllegalArgumentException(
    s"Can't serialize object of type ${obj.getClass}")
}

override def fromBinary(bytes: Array[Byte], clazz: Option[Class[_]]): AnyRef = {
  twoPhaseSetFromBinary(decompress(bytes))
}
```

The two embedded `GSet` can be serialized as illustrated above, but in general when composing new data types from the existing built in types it is better to make use of the existing serializer for those types. This can be done by declaring those as bytes fields in protobuf:

```
message TwoPhaseSet2 {
  optional bytes adds = 1;
  optional bytes removals = 2;
}
```

and use the methods `otherMessageToProto` and `otherMessageFromBinary` that are provided by the `SerializationSupport` trait to serialize and deserialize the `GSet` instances. This works with any type that has a registered Akka serializer. This is how such an serializer would look like for the `TwoPhaseSet`:

```
import akka.actor.ExtendedActorSystem
import akka.cluster.ddata.GSet
import akka.cluster.ddata.protobuf.ReplicatedDataSerializer
import akka.cluster.ddata.protobuf.SerializationSupport
import akka.serialization.Serializer
import docs.ddata.TwoPhaseSet
import docs.ddata.protobuf.msg.TwoPhaseSetMessages

class TwoPhaseSetSerializer2(val system: ExtendedActorSystem)
```

```
  extends Serializer with SerializationSupport {

  override def includeManifest: Boolean = false

  override def identifier = 99999

  val replicatedDataSerializer = new ReplicatedDataSerializer(system)

  override def toBinary(obj: AnyRef): Array[Byte] = obj match {
    case m: TwoPhaseSet => twoPhaseSetToProto(m).toByteArray
    case _ => throw new IllegalArgumentException(
      s"Can't serialize object of type ${obj.getClass}")
  }

  override def fromBinary(bytes: Array[Byte], clazz: Option[Class[_]]): AnyRef = {
    twoPhaseSetFromBinary(bytes)
  }

  def twoPhaseSetToProto(twoPhaseSet: TwoPhaseSet): TwoPhaseSetMessages.
↪TwoPhaseSet2 = {
    val b = TwoPhaseSetMessages.TwoPhaseSet2.newBuilder()
    if (!twoPhaseSet.adds.isEmpty)
      b.setAdds(otherMessageToProto(twoPhaseSet.adds).toByteString())
    if (!twoPhaseSet.removals.isEmpty)
      b.setRemovals(otherMessageToProto(twoPhaseSet.removals).toByteString())
    b.build()
  }

  def twoPhaseSetFromBinary(bytes: Array[Byte]): TwoPhaseSet = {
    val msg = TwoPhaseSetMessages.TwoPhaseSet2.parseFrom(bytes)
    val adds =
      if (msg.hasAdds)
        otherMessageFromBinary(msg.getAdds.toByteArray).asInstanceOf[GSet[String]]
      else
        GSet.empty[String]
    val removals =
      if (msg.hasRemovals)
        otherMessageFromBinary(msg.getRemovals.toByteArray).
↪asInstanceOf[GSet[String]]
      else
        GSet.empty[String]
    TwoPhaseSet(adds, removals)
  }
}
```

CRDT Garbage

One thing that can be problematic with CRDTs is that some data types accumulate history (garbage). For example a `GCounter` keeps track of one counter per node. If a `GCounter` has been updated from one node it will associate the identifier of that node forever. That can become a problem for long running systems with many cluster nodes being added and removed. To solve this problem the `Replicator` performs pruning of data associated with nodes that have been removed from the cluster. Data types that need pruning have to implement the `RemovedNodePruning` trait.

5.8.3 Samples

Several interesting samples are included and described in the Lightbend Activator tutorial named Akka Distributed Data Samples with Scala.

- Low Latency Voting Service

- Highly Available Shopping Cart
- Distributed Service Registry
- Replicated Cache
- Replicated Metrics

5.8.4 Limitations

There are some limitations that you should be aware of.

CRDTs cannot be used for all types of problems, and eventual consistency does not fit all domains. Sometimes you need strong consistency.

It is not intended for *Big Data*. The number of top level entries should not exceed 100000. When a new node is added to the cluster all these entries are transferred (gossiped) to the new node. The entries are split up in chunks and all existing nodes collaborate in the gossip, but it will take a while (tens of seconds) to transfer all entries and this means that you cannot have too many top level entries. The current recommended limit is 100000. We will be able to improve this if needed, but the design is still not intended for billions of entries.

All data is held in memory, which is another reason why it is not intended for *Big Data*.

When a data entry is changed the full state of that entry is replicated to other nodes. For example, if you add one element to a Set with 100 existing elements, all 101 elements are transferred to other nodes. This means that you cannot have too large data entries, because then the remote message size will be too large. We might be able to make this more efficient by implementing Efficient State-based CRDTs by Delta-Mutation.

The data is only kept in memory. It is redundant since it is replicated to other nodes in the cluster, but if you stop all nodes the data is lost, unless you have saved it elsewhere. Making the data durable is a possible future feature, but even if we implement that it is not intended to be a full featured database.

5.8.5 Learn More about CRDTs

- The Final Causal Frontier talk by Sean Cribbs
- Eventually Consistent Data Structures talk by Sean Cribbs
- Strong Eventual Consistency and Conflict-free Replicated Data Types talk by Mark Shapiro
- A comprehensive study of Convergent and Commutative Replicated Data Types paper by Mark Shapiro et. al.

Dependencies

To use Distributed Data you must add the following dependency in your project.

sbt:

```
"com.typesafe.akka" %% "akka-distributed-data-experimental" % "2.4.14"
```

maven:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-distributed-data-experimental_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

5.8.6 Configuration

The `DistributedData` extension can be configured with the following properties:

```
# Settings for the DistributedData extension
akka.cluster.distributed-data {
  # Actor name of the Replicator actor, /system/ddataReplicator
  name = ddataReplicator

  # Replicas are running on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # How often the Replicator should send out gossip information
  gossip-interval = 2 s

  # How often the subscribers will be notified of changes, if any
  notify-subscribers-interval = 500 ms

  # Maximum number of entries to transfer in one gossip message when synchronizing
  # the replicas. Next chunk will be transferred in next round of gossip.
  max-delta-elements = 1000

  # The id of the dispatcher to use for Replicator actors. If not specified
  # default dispatcher is used.
  # If specified you need to define the settings of the actual dispatcher.
  use-dispatcher = ""

  # How often the Replicator checks for pruning of data associated with
  # removed cluster nodes.
  pruning-interval = 30 s

  # How long time it takes (worst case) to spread the data to all other replica␣
↪nodes.
  # This is used when initiating and completing the pruning process of data␣
↪associated
  # with removed cluster nodes. The time measurement is stopped when any replica is
  # unreachable, so it should be configured to worst case in a healthy cluster.
  max-pruning-dissemination = 60 s

  # Serialized Write and Read messages are cached when they are sent to
  # several nodes. If no further activity they are removed from the cache
  # after this duration.
  serializer-cache-time-to-live = 10s

}
```

5.9 Remoting

For an introduction of remoting capabilities of Akka please see *Location Transparency*.

Note: As explained in that chapter Akka remoting is designed for communication in a peer-to-peer fashion and it has limitations for client-server setups. In particular Akka Remoting does not work transparently with Network Address Translation, Load Balancers, or in Docker containers. For symmetric communication in these situations network and/or Akka configuration will have to be changed as described in *Akka behind NAT or in a Docker container*.

5.9.1 Preparing your ActorSystem for Remoting

The Akka remoting is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-remote" % "2.4.14"
```

To enable remote capabilities in your Akka project you should, at a minimum, add the following changes to your `application.conf` file:

```
akka {
  actor {
    provider = remote
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

As you can see in the example above there are four things you need to add to get started:

- Change provider from `local` to `remote`
- Add host name - the machine you want to run the actor system on; this host name is exactly what is passed to remote systems in order to identify this system and consequently used for connecting back to this system if need be, hence set it to a reachable IP address or resolvable name in case you want to communicate across the network.
- Add port number - the port the actor system should listen on, set to 0 to have it chosen automatically

---

Note: The port number needs to be unique for each actor system on the same machine even if the actor systems have different names. This is because each actor system has its own networking subsystem listening for connections and handling messages as not to interfere with other actor systems.

The example above only illustrates the bare minimum of properties you have to add to enable remoting. All settings are described in *Remote Configuration*.

5.9.2 Types of Remote Interaction

Akka has two ways of using remoting:

- Lookup : used to look up an actor on a remote node with `actorSelection(path)`
- Creation : used to create an actor on a remote node with `actorOf(Props(...),actorName)`

In the next sections the two alternatives are described in detail.

5.9.3 Looking up Remote Actors

`actorSelection(path)` will obtain an `ActorSelection` to an Actor on a remote node, e.g.:

```
val selection =
  context.actorSelection("akka.tcp://actorSystemName@10.0.0.1:2552/user/actorName")
```

As you can see from the example above the following pattern is used to find an actor on a remote node:

```
akka.<protocol>://<actor system>@<hostname>:<port>/<actor path>
```

Once you obtained a selection to the actor you can interact with it in the same way you would with a local actor, e.g.:

```
selection ! "Pretty awesome feature"
```

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

Note: For more details on how actor addresses and paths are formed and used, please refer to *Actor References, Paths and Addresses*.

Note: Message sends to actors that are actually in the sending actor system do not get delivered via the remote actor ref provider. They're delivered directly, by the local actor ref provider.

Aside from providing better performance, this also means that if the hostname you configure remoting to listen as cannot actually be resolved from within the very same actor system, such messages will (perhaps counterintuitively) be delivered just fine.

5.9.4 Creating Actors Remotely

If you want to use the creation functionality in Akka remoting you have to further amend the `application.conf` file in the following way (only showing deployment section):

```
akka {
  actor {
    deployment {
      /sampleActor {
        remote = "akka.tcp://sampleActorSystem@127.0.0.1:2553"
      }
    }
  }
}
```

The configuration above instructs Akka to react when an actor with path `/sampleActor` is created, i.e. using `system.actorOf(Props(...),"sampleActor")`. This specific actor will not be directly instantiated, but instead the remote daemon of the remote system will be asked to create the actor, which in this sample corresponds to `sampleActorSystem@127.0.0.1:2553`.

Once you have configured the properties above you would do the following in code:

```
val actor = system.actorOf(Props[SampleActor], "sampleActor")
actor ! "Pretty slick"
```

The actor class `SampleActor` has to be available to the runtimes using it, i.e. the classloader of the actor systems has to have a JAR containing the class.

Note: In order to ensure serializability of `Props` when passing constructor arguments to the actor being created, do not make the factory an inner class: this will inherently capture a reference to its enclosing object, which in most cases is not serializable. It is best to create a factory method in the companion object of the actor's class.

Serializability of all Props can be tested by setting the configuration item `akka.actor.serialize-creators=on`. Only Props whose `deploy` has `LocalScope` are exempt from this check.

Note: You can use asterisks as wildcard matches for the actor paths, so you could specify: `/*/sampleActor` and that would match all `sampleActor` on that level in the hierarchy. You can also use wildcard in the last position to match all actors at a certain level: `/someParent/*`. Non-wildcard matches always have higher priority to match than wildcards, so: `/foo/bar` is considered **more specific** than `/foo/*` and only the highest priority match is used. Please note that it **cannot** be used to partially match section, like this: `/foo*/bar`, `/f*o/bar` etc.

Programmatic Remote Deployment

To allow dynamically deployed systems, it is also possible to include deployment configuration in the `Props` which are used to create an actor: this information is the equivalent of a deployment section from the configuration file, and if both are given, the external configuration takes precedence.

With these imports:

```
import akka.actor.{ Props, Deploy, Address, AddressFromURIString }
import akka.remote.RemoteScope
```

and a remote address like this:

```
val one = AddressFromURIString("akka.tcp://sys@host:1234")
val two = Address("akka.tcp", "sys", "host", 1234) // this gives the same
```

you can advise the system to create a child on that remote node like so:

```
val ref = system.actorOf(Props[SampleActor].
  withDeploy(Deploy(scope = RemoteScope(address))))
```

Remote deployment whitelist

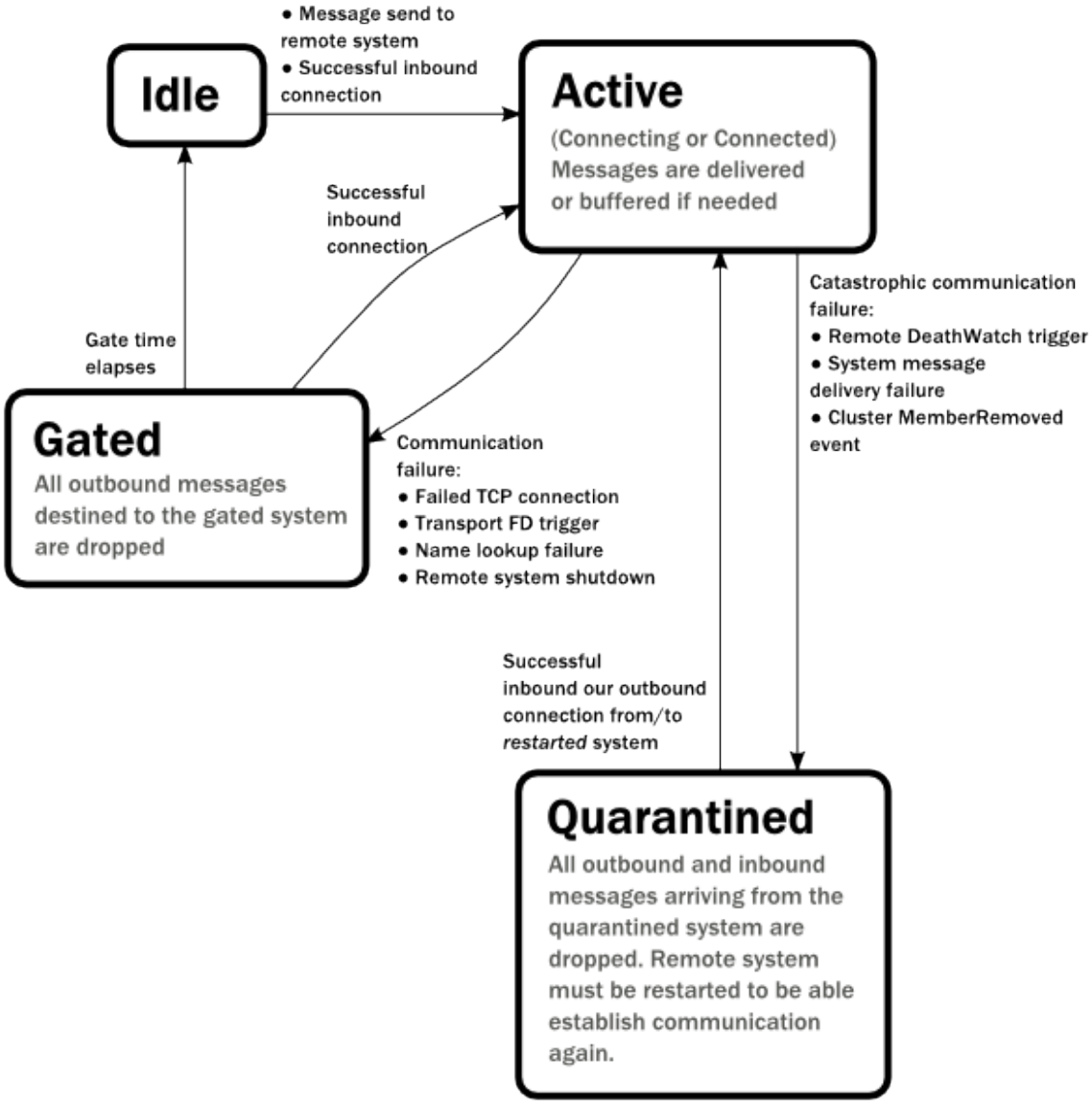
As remote deployment can potentially be abused by both users and even attackers a whitelist feature is available to guard the ActorSystem from deploying unexpected actors. Please note that remote deployment is *not* remote code loading, the Actors class to be deployed onto a remote system needs to be present on that remote system. This still however may pose a security risk, and one may want to restrict remote deployment to only a specific set of known actors by enabling the whitelist feature.

To enable remote deployment whitelisting set the `akka.remote.deployment.enable-whitelist` value to `on`. The list of allowed classes has to be configured on the "remote" system, in other words on the system onto which others will be attempting to remote deploy Actors. That system, locally, knows best which Actors it should or should not allow others to remote deploy onto it. The full settings section may for example look like this:

```
akka.remote.deployment {
  enable-whitelist = on

  whitelist = [
    "NOT_ON_CLASSPATH", # verify we don't throw if a class not on classpath is␣
↪listed here
    "akka.remote.RemoteDeploymentWhitelistSpec.EchoWhitelisted"
  ]
}
```

Actor classes not included in the whitelist will not be allowed to be remote deployed onto this system.

5.9.5 Lifecycle and Failure Recovery Model



Each link with a remote system can be in one of the four states as illustrated above. Before any communication happens with a remote system at a given `Address` the state of the association is `Idle`. The first time a message is attempted to be sent to the remote system or an inbound connection is accepted the state of the link transitions to `Active` denoting that the two systems has messages to send or receive and no failures were encountered so far. When a communication failure happens and the connection is lost between the two systems the link becomes `Gated`.

In this state the system will not attempt to connect to the remote host and all outbound messages will be dropped. The time while the link is in the `Gated` state is controlled by the setting `akka.remote.retry-gate-closed-for`: after this time elapses the link state transitions to `Idle` again. `Gate` is one-sided in the sense that whenever a successful *inbound* connection is accepted from a remote system during `Gate` it automatically transitions to `Active` and communication resumes immediately.

In the face of communication failures that are unrecoverable because the state of the participating systems are inconsistent, the remote system becomes `Quarantined`. Unlike `Gate`, quarantining is permanent and lasts until one of the systems is restarted. After a restart communication can be resumed again and the link can become `Active` again.

5.9.6 Watching Remote Actors

Watching a remote actor is not different than watching a local actor, as described in *Lifecycle Monitoring aka DeathWatch*.

Failure Detector

Under the hood remote death watch uses heartbeat messages and a failure detector to generate `Terminated` message from network failures and JVM crashes, in addition to graceful termination of watched actor.

The heartbeat arrival times is interpreted by an implementation of The Phi Accrual Failure Detector.

The suspicion level of failure is given by a value called *phi*. The basic idea of the phi failure detector is to express the value of *phi* on a scale that is dynamically adjusted to reflect current network conditions.

The value of *phi* is calculated as:

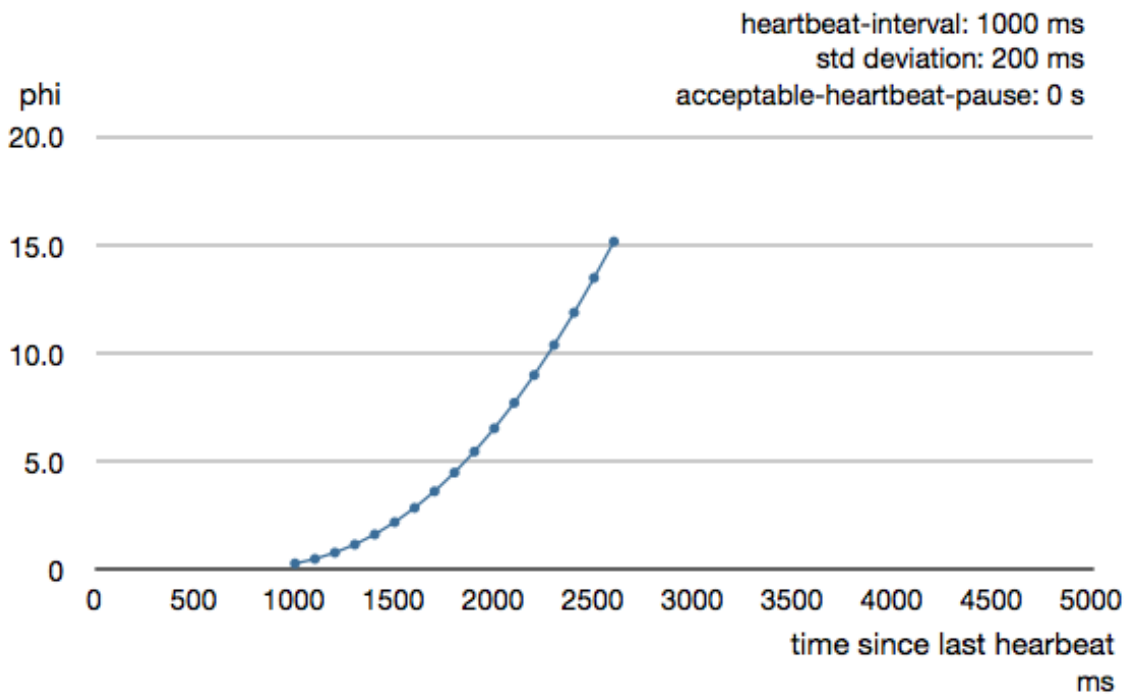
```
phi = -log10(1 - F(timeSinceLastHeartbeat))
```

where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times.

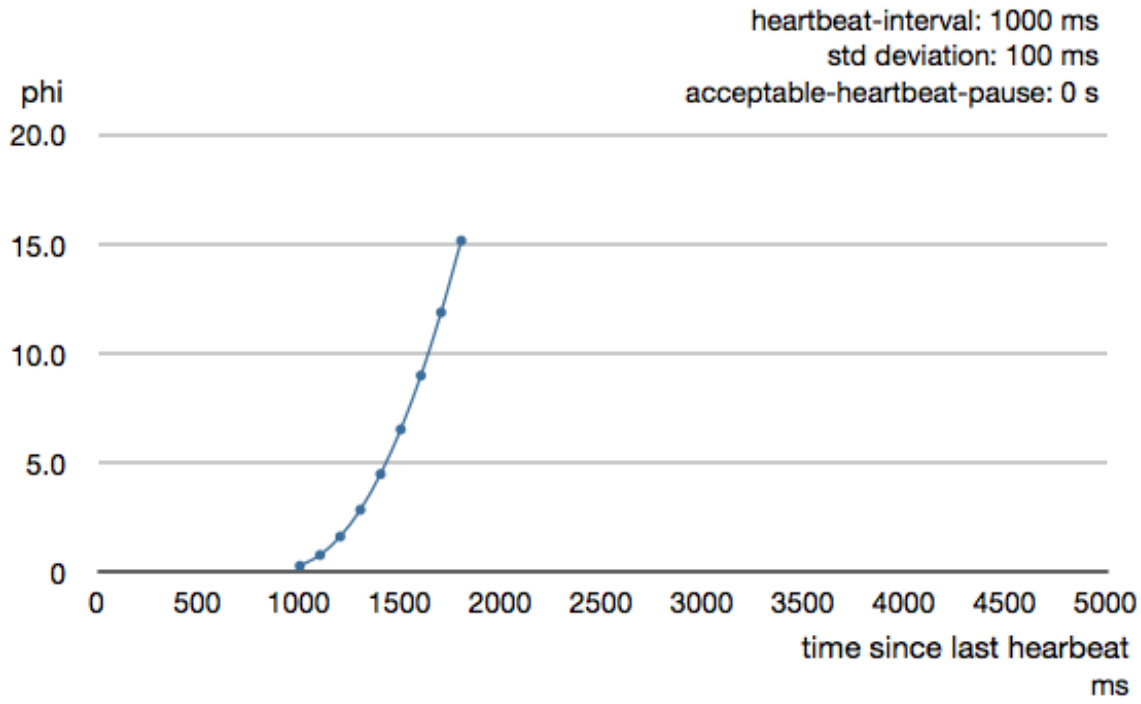
In the *Remote Configuration* you can adjust the `akka.remote.watch-failure-detector.threshold` to define when a *phi* value is considered to be a failure.

A low `threshold` is prone to generate many false positives but ensures a quick detection in the event of a real crash. Conversely, a high `threshold` generates fewer mistakes but needs more time to detect actual crashes. The default `threshold` is 10 and is appropriate for most situations. However in cloud environments, such as Amazon EC2, the value could be increased to 12 in order to account for network issues that sometimes occur on such platforms.

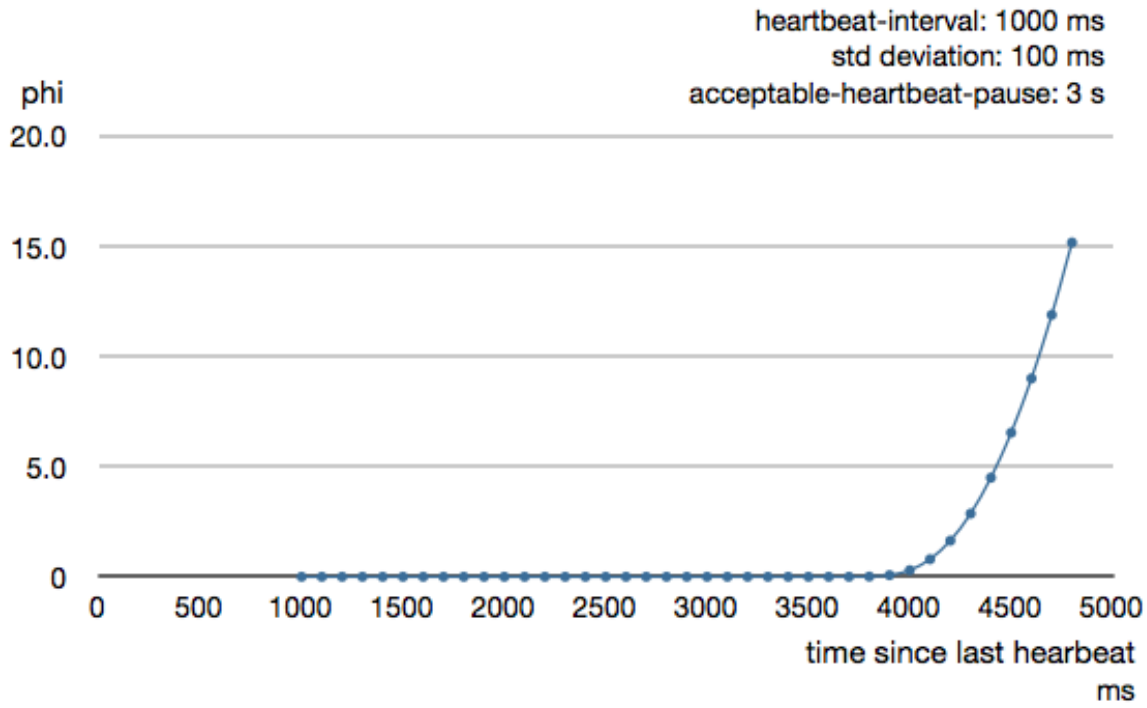
The following chart illustrates how *phi* increase with increasing time since the previous heartbeat.



Phi is calculated from the mean and standard deviation of historical inter arrival times. The previous chart is an example for standard deviation of 200 ms. If the heartbeats arrive with less deviation the curve becomes steeper, i.e. it is possible to determine failure more quickly. The curve looks like this for a standard deviation of 100 ms.



To be able to survive sudden abnormalities, such as garbage collection pauses and transient network failures the failure detector is configured with a margin, `akka.remote.watch-failure-detector.acceptable-heartbeat-pause`. You may want to adjust the *Remote Configuration* of this depending on you environment. This is how the curve looks like for `acceptable-heartbeat-pause` configured to 3 seconds.



5.9.7 Serialization

When using remoting for actors you must ensure that the `props` and `messages` used for those actors are serializable. Failing to do so will cause the system to behave in an unintended way.

For more information please see *Serialization*.

Disabling the Java Serializer

Since the `2.4.11` release of Akka it is possible to entirely disable the default Java Serialization mechanism. Please note that *new remoting implementation (codename Artery)* does not use Java serialization for internal messages by default. For compatibility reasons, the current remoting still uses Java serialization for some classes, however you can disable it in this remoting implementation as well by following the steps below.

The first step is to enable some additional serializers that replace previous Java serialization of some internal messages. This is recommended also when you can't disable Java serialization completely. Those serializers are enabled with this configuration:

```
akka.actor {
  # Set this to on to enable serialization-bindings define in
  # additional-serialization-bindings. Those are by default not included
  # for backwards compatibility reasons. They are enabled by default if
  # akka.remote.artery.enabled=on.
  enable-additional-serialization-bindings = on
}
```

The reason these are not enabled by default is wire-level compatibility between any 2.4.x Actor Systems. If you roll out a new cluster, all on the same Akka version that can enable these serializers it is recommended to enable this setting. When using *Remoting (codename Artery)* these serializers are enabled by default.

Warning: Please note that when enabling the additional-serialization-bindings when using the old remoting, you must do so on all nodes participating in a cluster, otherwise the mis-aligned serialization configurations will cause deserialization errors on the receiving nodes.

Java serialization is known to be slow and prone to attacks of various kinds - it never was designed for high throughput messaging after all. However it is very convenient to use, thus it remained the default serialization mechanism that Akka used to serialize user messages as well as some of its internal messages in previous versions. Since the release of Artery, Akka internals do not rely on Java serialization anymore (one exception being `java.lang.Throwable`).

Note: When using the new remoting implementation (codename Artery), Akka does not use Java Serialization for any of it's internal messages. It is highly encouraged to disable java serialization, so please plan to do so at the earliest possibility you have in your project.

One may think that network bandwidth and latency limit the performance of remote messaging, but serialization is a more typical bottleneck.

For user messages, the default serializer, implemented using Java serialization, remains available and enabled in Artery. We do however recommend to disable it entirely and utilise a proper serialization library instead in order effectively utilise the improved performance and ability for rolling deployments using Artery. Libraries that we recommend to use include, but are not limited to, Kryo by using the akka-kryo-serialization library or Google Protocol Buffers if you want more control over the schema evolution of your messages.

In order to completely disable Java Serialization in your Actor system you need to add the following configuration to your `application.conf`:

```
akka {
  actor {
```

```
    serialization-bindings {
      "java.io.Serializable" = none
    }
  }
}
```

Please note that this means that you will have to configure different serializers which will able to handle all of your remote messages. Please refer to the *Serialization* documentation as well as *ByteBuffer based serialization* to learn how to do this.

5.9.8 Routers with Remote Destinations

It is absolutely feasible to combine remoting with *Routing*.

A pool of remote deployed routees can be configured as:

```
akka.actor.deployment {
  /parent/remotePool {
    router = round-robin-pool
    nr-of-instances = 10
    target.nodes = ["akka.tcp://app@10.0.0.2:2552", "akka.tcp://app@10.0.0.3:2552"]
  }
}
```

This configuration setting will clone the actor defined in the `Props` of the `remotePool` 10 times and deploy it evenly distributed across the two given target nodes.

A group of remote actors can be configured as:

```
akka.actor.deployment {
  /parent/remoteGroup {
    router = round-robin-group
    routees.paths = [
      "akka.tcp://app@10.0.0.1:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.2:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.3:2552/user/workers/w1"]
  }
}
```

This configuration setting will send messages to the defined remote actor paths. It requires that you create the destination actors on the remote nodes with matching paths. That is not done by the router.

5.9.9 Remoting Sample

There is a more extensive remote example that comes with Lightbend Activator. The tutorial named Akka Remote Samples with Scala demonstrates both remote deployment and look-up of remote actors.

Remote Events

It is possible to listen to events that occur in Akka Remote, and to subscribe/unsubscribe to these events you simply register as listener to the below described types in on the `ActorSystem.eventStream`.

Note: To subscribe to any remote event, subscribe to `RemotingLifecycleEvent`. To subscribe to events related only to the lifecycle of associations, subscribe to `akka.remote.AssociationEvent`.

Note: The use of term "Association" instead of "Connection" reflects that the remoting subsystem may use connectionless transports, but an association similar to transport layer connections is maintained between endpoints by the Akka protocol.

By default an event listener is registered which logs all of the events described below. This default was chosen to help setting up a system, but it is quite common to switch this logging off once that phase of the project is finished.

Note: In order to switch off the logging, set `akka.remote.log-remote-lifecycle-events = off` in your `application.conf`.

To be notified when an association is over ("disconnected") listen to `DisassociatedEvent` which holds the direction of the association (inbound or outbound) and the addresses of the involved parties.

To be notified when an association is successfully established ("connected") listen to `AssociatedEvent` which holds the direction of the association (inbound or outbound) and the addresses of the involved parties.

To intercept errors directly related to associations, listen to `AssociationErrorEvent` which holds the direction of the association (inbound or outbound), the addresses of the involved parties and the `Throwable` cause.

To be notified when the remoting subsystem is ready to accept associations, listen to `RemotingListenEvent` which contains the addresses the remoting listens on.

To be notified when the current system is quarantined by the remote system, listen to `ThisActorSystemQuarantinedEvent`, which includes the addresses of local and remote ActorSystems.

To be notified when the remoting subsystem has been shut down, listen to `RemotingShutdownEvent`.

To intercept generic remoting related errors, listen to `RemotingErrorEvent` which holds the `Throwable` cause.

5.9.10 Remote Security

Untrusted Mode

As soon as an actor system can connect to another remotely, it may in principle send any possible message to any actor contained within that remote system. One example may be sending a `PoisonPill` to the system guardian, shutting that system down. This is not always desired, and it can be disabled with the following setting:

```
akka.remote.untrusted-mode = on
```

This disallows sending of system messages (actor life-cycle commands, DeathWatch, etc.) and any message extending `PossiblyHarmful` to the system on which this flag is set. Should a client send them nonetheless they are dropped and logged (at DEBUG level in order to reduce the possibilities for a denial of service attack). `PossiblyHarmful` covers the predefined messages like `PoisonPill` and `Kill`, but it can also be added as a marker trait to user-defined messages.

Messages sent with actor selection are by default discarded in untrusted mode, but permission to receive actor selection messages can be granted to specific actors defined in configuration:

```
akka.remote.trusted-selection-paths = ["/user/receptionist", "/user/namingService"]
```

The actual message must still not be of type `PossiblyHarmful`.

In summary, the following operations are ignored by a system configured in untrusted mode when incoming via the remoting layer:

- remote deployment (which also means no remote supervision)
- remote DeathWatch

- `system.stop()`, `PoisonPill`, `Kill`
- sending any message which extends from the `PossiblyHarmful` marker interface, which includes `Terminated`
- messages sent with actor selection, unless destination defined in `trusted-selection-paths`.

Note: Enabling the untrusted mode does not remove the capability of the client to freely choose the target of its message sends, which means that messages not prohibited by the above rules can be sent to any actor in the remote system. It is good practice for a client-facing system to only contain a well-defined set of entry point actors, which then forward requests (possibly after performing validation) to another actor system containing the actual worker actors. If messaging between these two server-side systems is done using local `ActorRef` (they can be exchanged safely between actor systems within the same JVM), you can restrict the messages on this interface by marking them `PossiblyHarmful` so that a client cannot forge them.

Configuring SSL/TLS for Akka Remoting

SSL can be used as the remote transport by adding `akka.remote.netty.ssl` to the `enabled-transport` configuration section. An example of setting up the default Netty based SSL driver as default:

```
akka {
  remote {
    enabled-transports = [akka.remote.netty.ssl]
  }
}
```

Next the actual SSL/TLS parameters have to be configured:

```
akka {
  remote {
    netty.ssl.security {
      key-store = "/example/path/to/mykeystore.jks"
      trust-store = "/example/path/to/mytruststore.jks"

      key-store-password = "changeme"
      key-password = "changeme"
      trust-store-password = "changeme"

      protocol = "TLSv1.2"

      enabled-algorithms = [TLS_DHE_RSA_WITH_AES_128_GCM_SHA256]

      random-number-generator = "AES128CounterSecureRNG"
    }
  }
}
```

According to RFC 7525 the recommended algorithms to use with TLS 1.2 (as of writing this document) are:

- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

You should always check the latest information about security and algorithm recommendations though before you configure your system.

Creating and working with keystores and certificates is well documented in the Generating X.509 Certificates section of Lightbend's SSL-Config library.

Since an Akka remoting is inherently *peer-to-peer* both the key-store as well as trust-store need to be configured on each remoting node participating in the cluster.

The official Java Secure Socket Extension documentation as well as the Oracle documentation on creating KeyStore and TrustStores are both great resources to research when setting up security on the JVM. Please consult those resources when troubleshooting and configuring SSL.

See also a description of the settings in the *Remote Configuration* section.

Note: When using SHA1PRNG on Linux it's recommended specify `-Djava.security.egd=file:/dev/urandom` as argument to the JVM to prevent blocking. It is NOT as secure because it reuses the seed.

5.9.11 Remote Configuration

There are lots of configuration properties that are related to remoting in Akka. We refer to the *reference configuration* for more information.

Note: Setting properties like the listening IP and port number programmatically is best done by using something like the following:

```
ConfigFactory.parseString("akka.remote.netty.tcp.hostname=\"1.2.3.4\"")
    .withFallback(ConfigFactory.load());
```

Akka behind NAT or in a Docker container

In setups involving Network Address Translation (NAT), Load Balancers or Docker containers the hostname and port pair that Akka binds to will be different than the "logical" host name and port pair that is used to connect to the system from the outside. This requires special configuration that sets both the logical and the bind pairs for remoting.

```
akka {
  remote {
    netty.tcp {
      hostname = my.domain.com      # external (logical) hostname
      port = 8000                   # external (logical) port

      bind-hostname = local.address # internal (bind) hostname
      bind-port = 2552              # internal (bind) port
    }
 }
}
```

5.10 Remoting (codename Artery)

Note: This page describes the experimental remoting subsystem, codenamed *Artery* that will eventually replace the old remoting implementation. For the current stable remoting system please refer to *Remoting*.

Remoting enables Actor systems on different hosts or JVMs to communicate with each other. By enabling remoting the system will start listening on a provided network address and also gains the ability to connect to other systems through the network. From the application's perspective there is no API difference between local or remote systems, `ActorRef` instances that point to remote systems look exactly the same as local ones: they can

be sent messages to, watched, etc. Every `ActorRef` contains hostname and port information and can be passed around even on the network. This means that on a network every `ActorRef` is a unique identifier of an actor on that network.

Remoting is not a server-client technology. All systems using remoting can contact any other system on the network if they possess an `ActorRef` pointing to those system. This means that every system that is remoting enabled acts as a "server" to which arbitrary systems on the same network can connect to.

5.10.1 What is new in Artery

Artery is a reimplementation of the old remoting module aimed at improving performance and stability. It is mostly backwards compatible with the old implementation and it is a drop-in replacement in many cases. Main features of Artery compared to the previous implementation:

- Based on Aeron (UDP) instead of TCP
- Focused on high-throughput, low-latency communication
- Isolation of internal control messages from user messages improving stability and reducing false failure detection in case of heavy traffic by using a dedicated subchannel.
- Mostly allocation-free operation
- Support for a separate subchannel for large messages to avoid interference with smaller messages
- Compression of actor paths on the wire to reduce overhead for smaller messages
- Support for faster serialization/deserialization using ByteBuffers directly
- Built-in Flight-Recorder to help debugging implementation issues without polluting users logs with implementaiton specific events
- Providing protocol stability across major Akka versions to support rolling updates of large-scale systems

The main incompatible change from the previous implementation that the protocol field of the string representation of an `ActorRef` is always *akka* instead of the previously used *akka.tcp* or *akka.ssl.tcp*. Configuration properties are also different.

5.10.2 Preparing your ActorSystem for Remoting

The Akka remoting is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-remote" % "2.4.14"
```

To enable remote capabilities in your Akka project you should, at a minimum, add the following changes to your `application.conf` file:

```
akka {
  actor {
    provider = remote
  }
  remote {
    artery {
      enabled = on
      canonical.hostname = "127.0.0.1"
      canonical.port = 25520
    }
  }
}
```

As you can see in the example above there are four things you need to add to get started:

- Change provider from `local` to `remote`

- Enable Artery to use it as the remoting implementation
- Add host name - the machine you want to run the actor system on; this host name is exactly what is passed to remote systems in order to identify this system and consequently used for connecting back to this system if need be, hence set it to a reachable IP address or resolvable name in case you want to communicate across the network.
- Add port number - the port the actor system should listen on, set to 0 to have it chosen automatically

---

Note: The port number needs to be unique for each actor system on the same machine even if the actor systems have different names. This is because each actor system has its own networking subsystem listening for connections and handling messages as not to interfere with other actor systems.

The example above only illustrates the bare minimum of properties you have to add to enable remoting. All settings are described in *Remote Configuration*.

Canonical address

In order to remoting to work properly, where each system can send messages to any other system on the same network (for example a system forwards a message to a third system, and the third replies directly to the sender system) it is essential for every system to have a *unique, globally reachable* address and port. This address is part of the unique name of the system and will be used by other systems to open a connection to it and send messages. This means that if a host has multiple names (different DNS records pointing to the same IP address) then only one of these can be *canonical*. If a message arrives to a system but it contains a different hostname than the expected canonical name then the message will be dropped. If multiple names for a system would be allowed, then equality checks among `ActorRef` instances would no longer to be trusted and this would violate the fundamental assumption that an actor has a globally unique reference on a given network. As a consequence, this also means that localhost addresses (e.g. *127.0.0.1*) cannot be used in general (apart from local development) since they are not unique addresses in a real network.

In cases, where Network Address Translation (NAT) is used or other network bridging is involved, it is important to configure the system so that it understands that there is a difference between his externally visible, canonical address and between the host-port pair that is used to listen for connections. See *Akka behind NAT or in a Docker container* for details.

5.10.3 Acquiring references to remote actors

In order to communicate with an actor, it is necessary to have its `ActorRef`. In the local case it is usually the creator of the actor (the caller of `actorOf()`) is who gets the `ActorRef` for an actor that it can then send to other actors. In other words:

- An Actor can get a remote Actor's reference simply by receiving a message from it (as it's available as *sender()* then), or inside of a remote message (e.g. *PleaseReply(message: String, remoteActorRef: ActorRef)*)

Alternatively, an actor can look up another located at a known path using `ActorSelection`. These methods are available even in remoting enabled systems:

- Remote Lookup : used to look up an actor on a remote node with `actorSelection(path)`
- Remote Creation : used to create an actor on a remote node with `actorOf(Props(...),actorName)`

In the next sections the two alternatives are described in detail.

Looking up Remote Actors

`actorSelection(path)` will obtain an `ActorSelection` to an Actor on a remote node, e.g.:

```
val selection =
  context.actorSelection("akka://actorSystemName@10.0.0.1:25520/user/actorName")
```

As you can see from the example above the following pattern is used to find an actor on a remote node:

```
akka://<actor system>@<hostname>:<port>/<actor path>
```

Note: Unlike with earlier remoting, the protocol field is always *akka* as pluggable transports are no longer supported.

Once you obtained a selection to the actor you can interact with it in the same way you would with a local actor, e.g.:

```
selection ! "Pretty awesome feature"
```

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

For more details on how actor addresses and paths are formed and used, please refer to *Actor References, Paths and Addresses*.

Note: Message sends to actors that are actually in the sending actor system do not get delivered via the remote actor ref provider. They're delivered directly, by the local actor ref provider.

Aside from providing better performance, this also means that if the hostname you configure remoting to listen as cannot actually be resolved from within the very same actor system, such messages will (perhaps counterintuitively) be delivered just fine.

Creating Actors Remotely

If you want to use the creation functionality in Akka remoting you have to further amend the `application.conf` file in the following way (only showing deployment section):

```
akka {
  actor {
    deployment {
      /sampleActor {
        remote = "akka://sampleActorSystem@127.0.0.1:2553"
      }
    }
  }
}
```

The configuration above instructs Akka to react when an actor with path `/sampleActor` is created, i.e. using `system.actorOf(Props(...),"sampleActor")`. This specific actor will not be directly instantiated, but instead the remote daemon of the remote system will be asked to create the actor, which in this sample corresponds to `sampleActorSystem@127.0.0.1:2553`.

Once you have configured the properties above you would do the following in code:

```
val actor = system.actorOf(Props[SampleActor], "sampleActor")
actor ! "Pretty slick"
```

The actor class `SampleActor` has to be available to the runtimes using it, i.e. the classloader of the actor systems has to have a JAR containing the class.

Note: In order to ensure serializability of `Props` when passing constructor arguments to the actor being created, do not make the factory an inner class: this will inherently capture a reference to its enclosing object, which in most cases is not serializable. It is best to create a factory method in the companion object of the actor's class.

Serializability of all Props can be tested by setting the configuration item `akka.actor.serialize-creators=on`. Only Props whose `deploy` has `LocalScope` are exempt from this check.

You can use asterisks as wildcard matches for the actor paths, so you could specify: `/*/sampleActor` and that would match all `sampleActor` on that level in the hierarchy. You can also use wildcard in the last position to match all actors at a certain level: `/someParent/*`. Non-wildcard matches always have higher priority to match than wildcards, so: `/foo/bar` is considered **more specific** than `/foo/*` and only the highest priority match is used. Please note that it **cannot** be used to partially match section, like this: `/foo*/bar`, `/f*o/bar` etc.

Programmatic Remote Deployment

To allow dynamically deployed systems, it is also possible to include deployment configuration in the `Props` which are used to create an actor: this information is the equivalent of a deployment section from the configuration file, and if both are given, the external configuration takes precedence.

With these imports:

```
import akka.actor.{ Props, Deploy, Address, AddressFromURIString }
import akka.remote.RemoteScope
```

and a remote address like this:

```
val one = AddressFromURIString("akka://sys@host:1234")
val two = Address("akka", "sys", "host", 1234) // this gives the same
```

you can advise the system to create a child on that remote node like so:

```
val ref = system.actorOf(Props[SampleActor].
  withDeploy(Deploy(scope = RemoteScope(address))))
```

Untrusted Mode

As soon as an actor system can connect to another remotely, it may in principle send any possible message to any actor contained within that remote system. One example may be sending a `PoisonPill` to the system guardian, shutting that system down. This is not always desired, and it can be disabled with the following setting:

```
akka.remote.artery.untrusted-mode = on
```

This disallows sending of system messages (actor life-cycle commands, DeathWatch, etc.) and any message extending `PossiblyHarmful` to the system on which this flag is set. Should a client send them nonetheless they are dropped and logged (at DEBUG level in order to reduce the possibilities for a denial of service attack). `PossiblyHarmful` covers the predefined messages like `PoisonPill` and `Kill`, but it can also be added as a marker trait to user-defined messages.

Messages sent with actor selection are by default discarded in untrusted mode, but permission to receive actor selection messages can be granted to specific actors defined in configuration:

```
akka.remote.artery..trusted-selection-paths = ["/user/receptionist", "/user/
↪namingService"]
```

The actual message must still not be of type `PossiblyHarmful`.

In summary, the following operations are ignored by a system configured in untrusted mode when incoming via the remoting layer:

- remote deployment (which also means no remote supervision)
- remote DeathWatch
- `system.stop()`, `PoisonPill`, `Kill`
- sending any message which extends from the `PossiblyHarmful` marker interface, which includes `Terminated`
- messages sent with actor selection, unless destination defined in `trusted-selection-paths`.

Note: Enabling the untrusted mode does not remove the capability of the client to freely choose the target of its message sends, which means that messages not prohibited by the above rules can be sent to any actor in the remote system. It is good practice for a client-facing system to only contain a well-defined set of entry point actors, which then forward requests (possibly after performing validation) to another actor system containing the actual worker actors. If messaging between these two server-side systems is done using local `ActorRef` (they can be exchanged safely between actor systems within the same JVM), you can restrict the messages on this interface by marking them `PossiblyHarmful` so that a client cannot forge them.

5.10.4 Quarantine

Akka remoting is using Aeron as underlying message transport. Aeron is using UDP and adds among other things reliable delivery and session semantics, very similar to TCP. This means that the order of the messages are preserved, which is needed for the *Actor message ordering guarantees*. Under normal circumstances all messages will be delivered but there are cases when messages may not be delivered to the destination:

- during a network partition and the Aeron session is broken, this automatically recovered once the partition is over
- when sending too many messages without flow control and thereby filling up the outbound send queue (`outbound-message-queue-size` config)
- if serialization or deserialization of a message fails (only that message will be dropped)
- if an unexpected exception occurs in the remoting infrastructure

In short, Actor message delivery is "at-most-once" as described in *Message Delivery Reliability*

Some messages in Akka are called system messages and those cannot be dropped because that would result in an inconsistent state between the systems. Such messages are used for essentially two features; remote death watch and remote deployment. These messages are delivered by Akka remoting with "exactly-once" guarantee by confirming each message and resending unconfirmed messages. If a system message anyway cannot be delivered the association with the destination system is irrecoverable failed, and Terminated is signaled for all watched actors on the remote system. It is placed in a so called quarantined state. Quarantine usually does not happen if remote watch or remote deployment is not used.

Each `ActorSystem` instance has an unique identifier (UID), which is important for differentiating between incarnations of a system when it is restarted with the same hostname and port. It is the specific incarnation (UID) that is quarantined. The only way to recover from this state is to restart one of the actor systems.

Messages that are sent to and received from a quarantined system will be dropped. However, it is possible to send messages with `actorSelection` to the address of a quarantined system, which is useful to probe if the system has been restarted.

An association will be quarantined when:

- Cluster node is removed from the cluster membership.

- Remote failure detector triggers, i.e. remote watch is used. This is different when *Akka Cluster* is used. The unreachable observation by the cluster failure detector can go back to reachable if the network partition heals. A cluster member is not quarantined when the failure detector triggers.
- Overflow of the system message delivery buffer, e.g. because of too many `watch` requests at the same time (`system-message-buffer-size` config).
- Unexpected exception occurs in the control subchannel of the remoting infrastructure.

The UID of the `ActorSystem` is exchanged in a two-way handshake when the first message is sent to a destination. The handshake will be retried until the other system replies and no other messages will pass through until the handshake is completed. If the handshake cannot be established within a timeout (`handshake-timeout` config) the association is stopped (freeing up resources). Queued messages will be dropped if the handshake cannot be established. It will not be quarantined, because the UID is unknown. New handshake attempt will start when next message is sent to the destination.

Handshake requests are actually also sent periodically to be able to establish a working connection when the destination system has been restarted.

Watching Remote Actors

Watching a remote actor is API wise not different than watching a local actor, as described in *Lifecycle Monitoring aka DeathWatch*. However, it is important to note, that unlike in the local case, remoting has to handle when a remote actor does not terminate in a graceful way sending a system message to notify the watcher actor about the event, but instead being hosted on a system which stopped abruptly (crashed). These situations are handled by the built-in failure detector.

Failure Detector

Under the hood remote death watch uses heartbeat messages and a failure detector to generate `Terminated` message from network failures and JVM crashes, in addition to graceful termination of watched actor.

The heartbeat arrival times is interpreted by an implementation of The Phi Accrual Failure Detector.

The suspicion level of failure is given by a value called *phi*. The basic idea of the phi failure detector is to express the value of *phi* on a scale that is dynamically adjusted to reflect current network conditions.

The value of *phi* is calculated as:

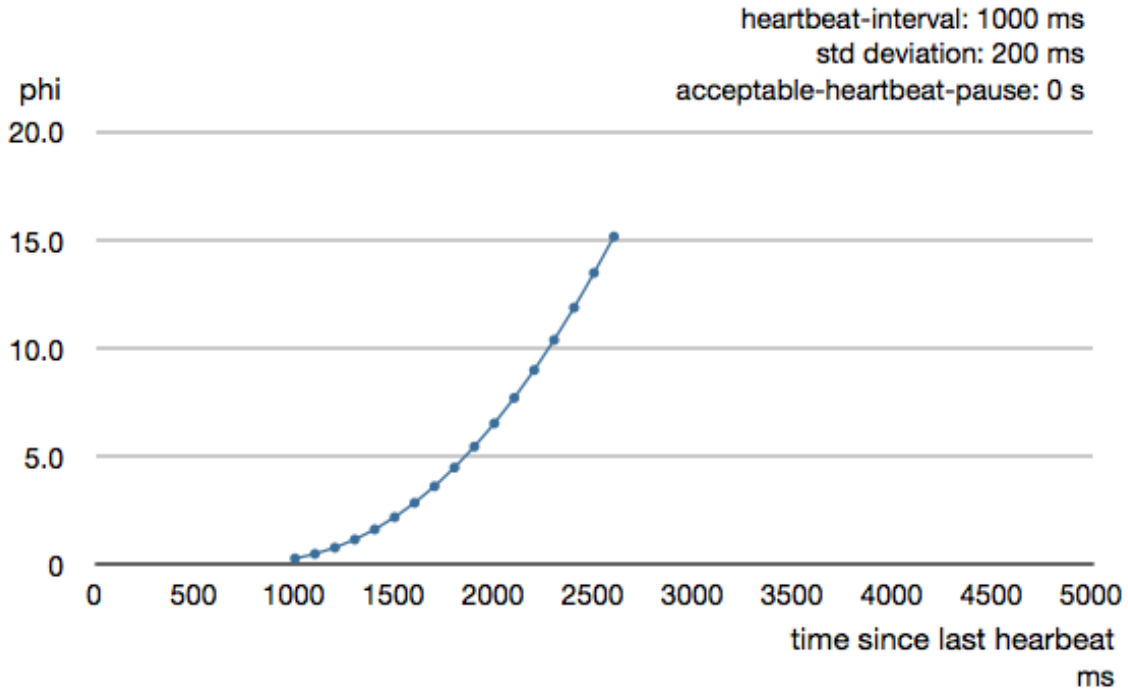
```
phi = -log10(1 - F(timeSinceLastHeartbeat))
```

where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times.

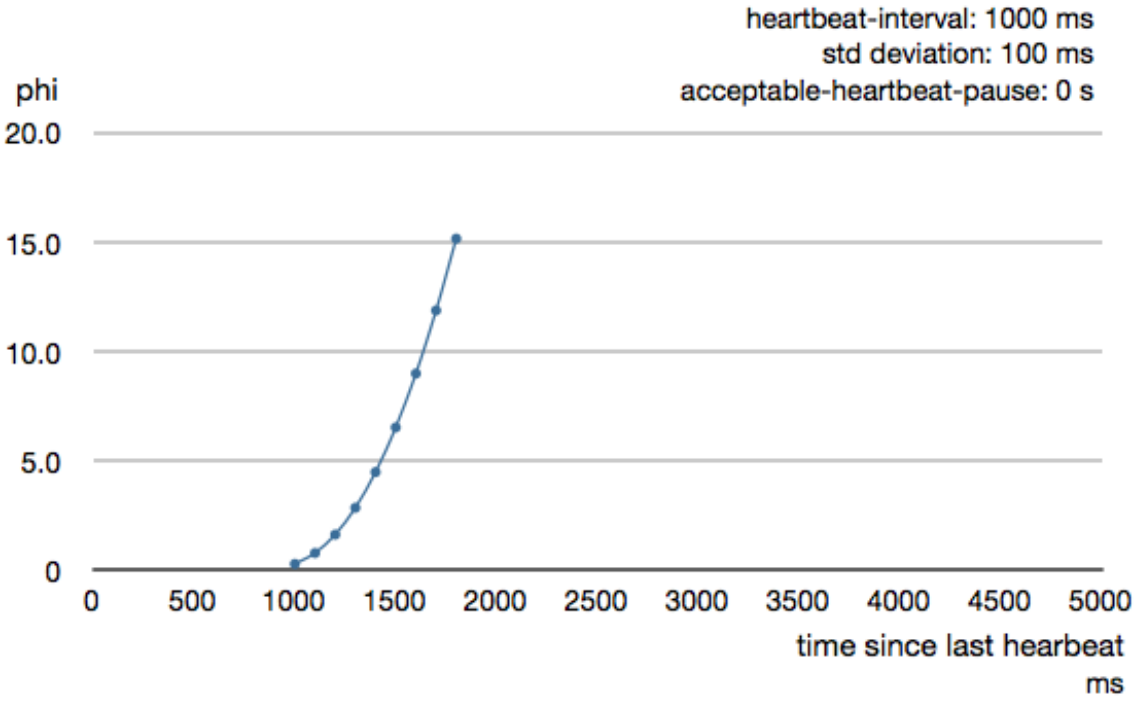
In the *Remote Configuration* you can adjust the `akka.remote.watch-failure-detector.threshold` to define when a *phi* value is considered to be a failure.

A low `threshold` is prone to generate many false positives but ensures a quick detection in the event of a real crash. Conversely, a high `threshold` generates fewer mistakes but needs more time to detect actual crashes. The default `threshold` is 10 and is appropriate for most situations. However in cloud environments, such as Amazon EC2, the value could be increased to 12 in order to account for network issues that sometimes occur on such platforms.

The following chart illustrates how *phi* increase with increasing time since the previous heartbeat.

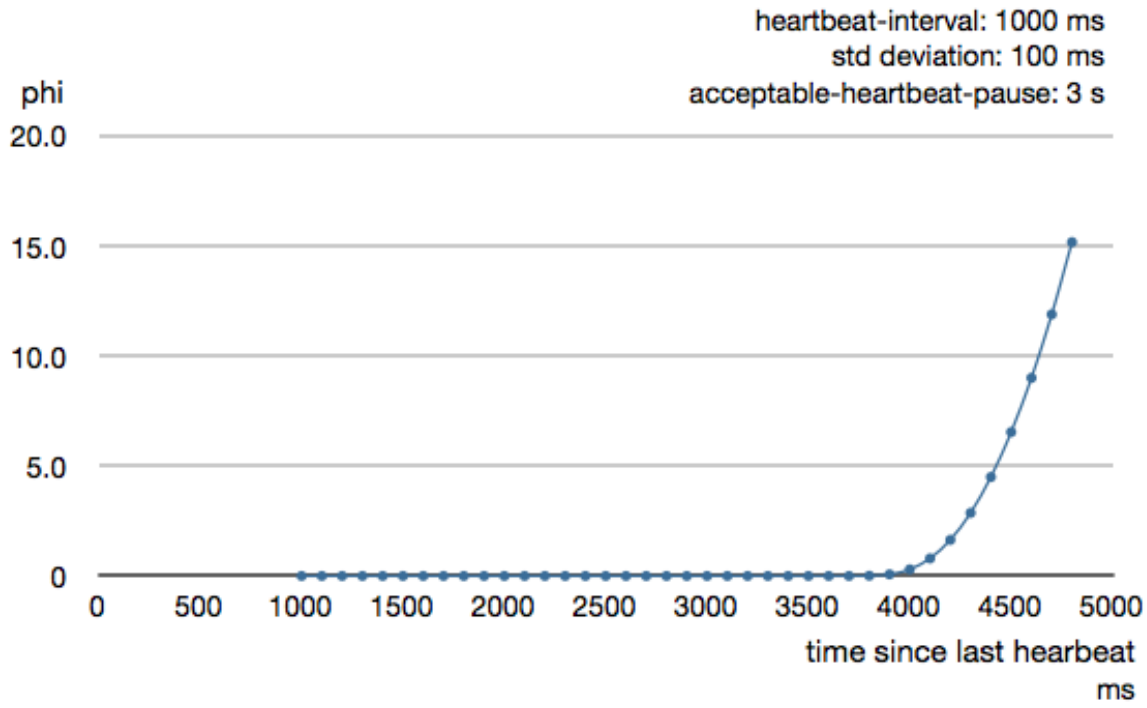


Phi is calculated from the mean and standard deviation of historical inter arrival times. The previous chart is an example for standard deviation of 200 ms. If the heartbeats arrive with less deviation the curve becomes steeper, i.e. it is possible to determine failure more quickly. The curve looks like this for a standard deviation of 100 ms.



To be able to survive sudden abnormalities, such as garbage collection pauses and transient network failures the failure detector is configured with a margin, `akka.remote.watch-failure-detector.acceptable-heartbeat-pause`. You may want to adjust the *Remote Configuration* of this depending on you environment. This is how the curve looks like for

`acceptable-heartbeat-pause` configured to 3 seconds.



5.10.5 Serialization

When using remoting for actors you must ensure that the `props` and `messages` used for those actors are serializable. Failing to do so will cause the system to behave in an unintended way.

For more information please see *Serialization*.

ByteBuffer based serialization

Artery introduces a new serialization mechanism which allows the `ByteBufferSerializer` to directly write into a shared `java.nio.ByteBuffer` instead of being forced to allocate and return an `Array[Byte]` for each serialized message. For high-throughput messaging this API change can yield significant performance benefits, so we recommend changing your serializers to use this new mechanism.

This new API also plays well with new versions of Google Protocol Buffers and other serialization libraries, which gained the ability to serialize directly into and from ByteBuffers.

As the new feature only changes how bytes are read and written, and the rest of the serializatio infrastructure remained the same, we recommend reading the *Serialization* documentation first.

Implementing an `akka.serialization.ByteBufferSerializer` works the same way as any other serializer,

```
trait ByteBufferSerializer {

  /**
   * Serializes the given object into the `ByteBuffer`.
   */
  def toBinary(o: AnyRef, buf: ByteBuffer): Unit

  /**
   * Produces an object from a `ByteBuffer`, with an optional type-hint;
   * the class should be loaded using ActorSystem.dynamicAccess.
```

```
   */
  def fromBinary(buf: ByteBuffer, manifest: String): AnyRef

}
```

Implementing a serializer for Artery is therefore as simple as implementing this interface, and binding the serializer as usual (which is explained in *Serialization*).

Implementations should typically extend `SerializerWithStringManifest` and in addition to the `ByteBuffer` based `toBinary` and `fromBinary` methods also implement the array based `toBinary` and `fromBinary` methods. The array based methods will be used when `ByteBuffer` is not used, e.g. in Akka Persistence.

Note that the array based methods can be implemented by delegation like this:

```
import java.nio.ByteBuffer
import akka.serialization.ByteBufferSerializer
import akka.serialization.SerializerWithStringManifest

  class ExampleByteBufSerializer extends SerializerWithStringManifest with␣
↪ByteBufferSerializer {
    override def identifier: Int = 1337
    override def manifest(o: AnyRef): String = "naive-toStringImpl"

    // Implement this method for compatibility with `SerializerWithStringManifest`.
    override def toBinary(o: AnyRef): Array[Byte] = {
      // in production code, aquire this from a BufferPool
      val buf = ByteBuffer.allocate(256)

      toBinary(o, buf)
      buf.flip()
      val bytes = Array.ofDim[Byte](buf.remaining)
      buf.get(bytes)
      bytes
    }

    // Implement this method for compatibility with `SerializerWithStringManifest`.
    override def fromBinary(bytes: Array[Byte], manifest: String): AnyRef =
      fromBinary(ByteBuffer.wrap(bytes), manifest)

    // Actual implementation in the ByteBuffer versions of to/fromBinary:
    override def toBinary(o: AnyRef, buf: ByteBuffer): Unit = ??? // implement␣
↪actual logic here
    override def fromBinary(buf: ByteBuffer, manifest: String): AnyRef = ??? //␣
↪implement actual logic here
  }
```

Disabling the Java Serializer

With Artery it is possible to completely disable Java Serialization for the entire Actor system.

Java serialization is known to be slow and prone to attacks of various kinds - it never was designed for high throughput messaging after all. However it is very convenient to use, thus it remained the default serialization mechanism that Akka used to serialize user messages as well as some of its internal messages in previous versions. Since the release of Artery, Akka internals do not rely on Java serialization anymore (exceptions to that being `java.lang.Throwable` and "remote deployment").

Note: When using Artery, Akka does not use Java Serialization for any of it's internal messages. It is highly encouraged to disable java serialization, so please plan to do so at the earliest possibility you have in your project.

One may think that network bandwidth and latency limit the performance of remote messaging, but serialization is a more typical bottleneck.

For user messages, the default serializer, implemented using Java serialization, remains available and enabled in Artery. We do however recommend to disable it entirely and utilise a proper serialization library instead in order effectively utilise the improved performance and ability for rolling deployments using Artery. Libraries that we recommend to use include, but are not limited to, Kryo by using the akka-kryo-serialization library or Google Protocol Buffers if you want more control over the schema evolution of your messages.

In order to completely disable Java Serialization in your Actor system you need to add the following configuration to your `application.conf`:

```
akka {
  actor {
    serialization-bindings {
      "java.io.Serializable" = none
    }
  }
}
```

Please note that this means that you will have to configure different serializers which will able to handle all of your remote messages. Please refer to the *Serialization* documentation as well as *ByteBuffer based serialization* to learn how to do this.

5.10.6 Routers with Remote Destinations

It is absolutely feasible to combine remoting with *Routing*.

A pool of remote deployed routees can be configured as:

```
akka.actor.deployment {
  /parent/remotePool {
    router = round-robin-pool
    nr-of-instances = 10
    target.nodes = ["tcp://app@10.0.0.2:2552", "akka://app@10.0.0.3:2552"]
  }
}
```

This configuration setting will clone the actor defined in the `Props` of the `remotePool` 10 times and deploy it evenly distributed across the two given target nodes.

A group of remote actors can be configured as:

```
akka.actor.deployment {
  /parent/remoteGroup2 {
    router = round-robin-group
    routees.paths = [
      "akka://app@10.0.0.1:2552/user/workers/w1",
      "akka://app@10.0.0.2:2552/user/workers/w1",
      "akka://app@10.0.0.3:2552/user/workers/w1"]
  }
}
```

This configuration setting will send messages to the defined remote actor paths. It requires that you create the destination actors on the remote nodes with matching paths. That is not done by the router.

5.10.7 Remoting Sample

There is a more extensive remote example that comes with Lightbend Activator. The tutorial named Akka Remote Samples with Scala demonstrates both remote deployment and look-up of remote actors.

5.10.8 Performance tuning

Dedicated subchannel for large messages

All the communication between user defined remote actors are isolated from the channel of Akka internal messages so a large user message cannot block an urgent system message. While this provides good isolation for Akka services, all user communications by default happen through a shared network connection (an Aeron stream). When some actors send large messages this can cause other messages to suffer higher latency as they need to wait until the full message has been transported on the shared channel (and hence, shared bottleneck). In these cases it is usually helpful to separate actors that have different QoS requirements: large messages vs. low latency.

Akka remoting provides a dedicated channel for large messages if configured. Since actor message ordering must not be violated the channel is actually dedicated for *actors* instead of messages, to ensure all of the messages arrive in send order. It is possible to assign actors on given paths to use this dedicated channel by using path patterns:

```
akka.remote.artery.large-message-destinations = [
   "/user/largeMessageActor",
   "/user/largeMessagesGroup/*",
   "/user/anotherGroup/*/largeMesssages",
   "/user/thirdGroup/**",
]
```

This means that all messages sent to the following actors will pass through the dedicated, large messages channel:

- `/user/largeMessageActor`
- `/user/largeMessageActorGroup/actor1`
- `/user/largeMessageActorGroup/actor2`
- `/user/anotherGroup/actor1/largeMessages`
- `/user/anotherGroup/actor2/largeMessages`
- `/user/thirdGroup/actor3/`
- `/user/thirdGroup/actor4/actor5`

Messages destined for actors not matching any of these patterns are sent using the default channel as before.

External, shared Aeron media driver

The Aeron transport is running in a so called media driver. By default, Akka starts the media driver embedded in the same JVM process as application. This is convenient and simplifies operational concerns by only having one process to start and monitor.

The media driver may use rather much CPU resources. If you run more than one Akka application JVM on the same machine it can therefore be wise to share the media driver by running it as a separate process.

The media driver has also different resource usage characteristics than a normal application and it can therefore be more efficient and stable to run the media driver as a separate process.

Given that Aeron jar files are in the classpath the standalone media driver can be started with:

```
java io.aeron.driver.MediaDriver
```

The needed classpath:

```
Agrona-0.5.4.jar:aeron-driver-1.0.1.jar:aeron-client-1.0.1.jar
```

You find those jar files on maven central, or you can create a package with your preferred build tool.

You can pass Aeron properties as command line *-D* system properties:

```
-Daeron.dir=/dev/shm/aeron
```

You can also define Aeron properties in a file:

```
java io.aeron.driver.MediaDriver config/aeron.properties
```

An example of such a properties file:

```
aeron.mtu.length=16384
aeron.socket.so_sndbuf=2097152
aeron.socket.so_rcvbuf=2097152
aeron.rcv.buffer.length=16384
aeron.rcv.initial.window.length=2097152
agrona.disable.bounds.checks=true

aeron.threading.mode=SHARED_NETWORK

# low latency settings
#aeron.threading.mode=DEDICATED
#aeron.sender.idle.strategy=org.agrona.concurrent.BusySpinIdleStrategy
#aeron.receiver.idle.strategy=org.agrona.concurrent.BusySpinIdleStrategy

# use same director in akka.remote.artery.advanced.aeron-dir config
# of the Akka application
aeron.dir=/dev/shm/aeron
```

Read more about the media driver in the Aeron documentation.

To use the external media driver from the Akka application you need to define the following two configuration properties:

```
akka.remote.artery.advanced {
  embedded-media-driver = off
  aeron-dir = /dev/shm/aeron
}
```

The `aeron-dir` must match the directory you started the media driver with, i.e. the `aeron.dir` property.

Several Akka applications can then be configured to use the same media driver by pointing to the same directory.

Note that if the media driver process is stopped the Akka applications that are using it will also be stopped.

Aeron Tuning

See Aeron documentation about Performance Testing.

Fine-tuning CPU usage latency tradeoff

Artery has been designed for low latency and as a result it can be CPU hungry when the system is mostly idle. This is not always desirable. It is possible to tune the tradeoff between CPU usage and latency with the following configuration:

> # Values can be from 1 to 10, where 10 strongly prefers low latency # and 1 strongly prefers less CPU usage akka.remote.artery.advanced.idle-cpu-level = 1

By setting this value to a lower number, it tells Akka to do longer "sleeping" periods on its thread dedicated for spin-waiting and hence reducing CPU load when there is no immediate task to execute at the cost of a longer reaction time to an event when it actually happens. It is worth to be noted though that during a continuously high-throughput period this setting makes not much difference as the thread mostly has tasks to execute. This also means that under high throughput (but below maximum capacity) the system might have less latency than at low message rates.

5.10.9 Internal Event Log for Debugging (Flight Recorder)

Note: In this version (2.4.14) the flight-recorder is disabled by default because there is no automatic file name and path calculation implemented to make it possible to reuse the same file for every restart of the same actor system without clashing with files produced by other systems (possibly running on the same machine). Currently, you have to set the path and file names yourself to avoid creating an unbounded number of files and enable flight recorder manually by adding *akka.remote.artery.advanced.flight-recorder.enabled=on* to your configuration file. This a limitation of the current version and will not be necessary in the future.

Emitting event information (logs) from internals is always a tradeoff. The events that are usable for the Akka developers are usually too low level to be of any use for users and usually need to be fine-grained enough to provide enough information to be able to debug issues in the internal implementation. This usually means that these logs are hidden behind special flags and emitted at low log levels to not clutter the log output of the user system. Unfortunately this means that during production or integration testing these flags are usually off and events are not available when an actual failure happens - leaving maintainers in the dark about details of the event. To solve this contradiction, remoting has an internal, high-performance event store for debug events which is always on. This log and the events that it contains are highly specialized and not directly exposed to users, their primary purpose is to help the maintainers of Akka to identify and solve issues discovered during daily usage. When you encounter production issues involving remoting, you can include the flight recorder log file in your bug report to give us more insight into the nature of the failure.

There are various important features of this event log:

- Flight Recorder produces a fixed size file completely encapsulating log rotation. This means that this file will never grow in size and will not cause any unexpected disk space shortage in production.
- This file is crash resistant, i.e. its contents can be recovered even if the JVM hosting the `ActorSystem` crashes unexpectedly.
- Very low overhead, specialized, binary logging that has no significant overhead and can be safely left enabled for production systems.

The location of the file can be controlled via the *akka.remote.artery.advanced.flight-recoder.destination* setting (see *akka-remote (artery)* for details). By default, a file with the *.afr* extension is produced in the temporary directory of the operating system. In cases where the flight recorder casuses issues, it can be disabled by adding the setting *akka.remote.artery.advanced.flight-recorder.enabled=off*, although this is not recommended.

5.10.10 Remote Configuration

There are lots of configuration properties that are related to remoting in Akka. We refer to the *reference configuration* for more information.

Note: Setting properties like the listening IP and port number programmatically is best done by using something like the following:

```
ConfigFactory.parseString("akka.remote.artery.canonical.hostname=\"1.2.3.4\"")
    .withFallback(ConfigFactory.load());
```

Akka behind NAT or in a Docker container

In setups involving Network Address Translation (NAT), Load Balancers or Docker containers the hostname and port pair that Akka binds to will be different than the "logical" host name and port pair that is used to connect to the system from the outside. This requires special configuration that sets both the logical and the bind pairs for remoting.

```
akka {
  remote {
    artery {
      canonical.hostname = my.domain.com      # external (logical) hostname
      canonical.port = 8000                   # external (logical) port

      bind.hostname = local.address # internal (bind) hostname
      bind.port = 25520             # internal (bind) port
    }
 }
}
```

5.11 Serialization

Akka has a built-in Extension for serialization, and it is both possible to use the built-in serializers and to write your own.

The serialization mechanism is both used by Akka internally to serialize messages, and available for ad-hoc serialization of whatever you might need it for.

5.11.1 Usage

Configuration

For Akka to know which `Serializer` to use for what, you need edit your *Configuration*, in the "akka.actor.serializers"-section you bind names to implementations of the `akka.serialization.Serializer` you wish to use, like this:

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      myown = "docs.serialization.MyOwnSerializer"
    }
  }
}
```

After you've bound names to different implementations of `Serializer` you need to wire which classes should be serialized using which `Serializer`, this is done in the "akka.actor.serialization-bindings"-section:

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      myown = "docs.serialization.MyOwnSerializer"
    }

    serialization-bindings {
      "java.lang.String" = java
      "docs.serialization.Customer" = java
      "com.google.protobuf.Message" = proto
      "docs.serialization.MyOwnSerializable" = myown
      "java.lang.Boolean" = myown
    }
  }
}
```

You only need to specify the name of an interface or abstract base class of the messages. In case of ambiguity, i.e. the message implements several of the configured classes, the most specific configured class will be used, i.e. the one of which all other candidates are superclasses. If this condition cannot be met, because e.g. `java.io.Serializable` and `MyOwnSerializable` both apply and neither is a subtype of the other, a warning will be issued

Note: If your messages are contained inside of a Scala object, then in order to reference those messages, you will need use the fully qualified Java class name. For a message named `Message` contained inside the object named `Wrapper` you would need to reference it as `Wrapper$Message` instead of `Wrapper.Message`.

Akka provides serializers for `java.io.Serializable` and protobuf `com.google.protobuf.GeneratedMessage` by default (the latter only if depending on the akka-remote module), so normally you don't need to add configuration for that; since `com.google.protobuf.GeneratedMessage` implements `java.io.Serializable`, protobuf messages will always be serialized using the protobuf protocol unless specifically overridden. In order to disable a default serializer, map its marker type to "none":

```
akka.actor.serialization-bindings {
  "java.io.Serializable" = none
}
```

Verification

If you want to verify that your messages are serializable you can enable the following config option:

```
akka {
  actor {
    serialize-messages = on
  }
}
```

Warning: We only recommend using the config option turned on when you're running tests. It is completely pointless to have it turned on in other scenarios.

If you want to verify that your `Props` are serializable you can enable the following config option:

```
akka {
  actor {
    serialize-creators = on
  }
}
```

Warning: We only recommend using the config option turned on when you're running tests. It is completely pointless to have it turned on in other scenarios.

Programmatic

If you want to programmatically serialize/deserialize using Akka Serialization, here's some examples:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory
```

```
    val system = ActorSystem("example")

    // Get the Serialization Extension
    val serialization = SerializationExtension(system)

    // Have something to serialize
    val original = "woohoo"

    // Find the Serializer for it
    val serializer = serialization.findSerializerFor(original)

    // Turn it into bytes
    val bytes = serializer.toBinary(original)

    // Turn it back into an object
    val back = serializer.fromBinary(bytes, manifest = None)

    // Voilá!
    back should be(original)
```

For more information, have a look at the `ScalaDoc` for `akka.serialization._`

5.11.2 Customization

So, lets say that you want to create your own `Serializer`, you saw the `docs.serialization.MyOwnSerializer` in the config example above?

Creating new Serializers

First you need to create a class definition of your `Serializer` like so:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory

class MyOwnSerializer extends Serializer {

  // This is whether "fromBinary" requires a "clazz" or not
  def includeManifest: Boolean = true

  // Pick a unique identifier for your Serializer,
  // you've got a couple of billions to choose from,
  // 0 - 16 is reserved by Akka itself
  def identifier = 1234567

  // "toBinary" serializes the given object to an Array of Bytes
  def toBinary(obj: AnyRef): Array[Byte] = {
    // Put the code that serializes the object here
    // ... ...
  }

  // "fromBinary" deserializes the given array,
  // using the type hint (if any, see "includeManifest" above)
  def fromBinary(
    bytes: Array[Byte],
    clazz: Option[Class[_]]): AnyRef = {
    // Put your code that deserializes here
    // ... ...
  }
}
```

The manifest is a type hint so that the same serializer can be used for different classes. The manifest parameter in `fromBinary` is the class of the object that was serialized. In `fromBinary` you can match on the class and deserialize the bytes to different objects.

Then you only need to fill in the blanks, bind it to a name in your *Configuration* and then list which classes that should be serialized using it.

Serializer with String Manifest

The `Serializer` illustrated above supports a class based manifest (type hint). For serialization of data that need to evolve over time the `SerializerWithStringManifest` is recommended instead of `Serializer` because the manifest (type hint) is a `String` instead of a `Class`. That means that the class can be moved/removed and the serializer can still deserialize old data by matching on the `String`. This is especially useful for *Persistence*.

The manifest string can also encode a version number that can be used in `fromBinary` to deserialize in different ways to migrate old data to new domain objects.

If the data was originally serialized with `Serializer` and in a later version of the system you change to `SerializerWithStringManifest` the manifest string will be the full class name if you used `includeManifest=true`, otherwise it will be the empty string.

This is how a `SerializerWithStringManifest` looks like:

```
class MyOwnSerializer2 extends SerializerWithStringManifest {

  val CustomerManifest = "customer"
  val UserManifest = "user"
  val UTF_8 = StandardCharsets.UTF_8.name()

  // Pick a unique identifier for your Serializer,
  // you've got a couple of billions to choose from,
  // 0 - 16 is reserved by Akka itself
  def identifier = 1234567

  // The manifest (type hint) that will be provided in the fromBinary method
  // Use `""` if manifest is not needed.
  def manifest(obj: AnyRef): String =
    obj match {
      case _: Customer => CustomerManifest
      case _: User     => UserManifest
    }

  // "toBinary" serializes the given object to an Array of Bytes
  def toBinary(obj: AnyRef): Array[Byte] = {
    // Put the real code that serializes the object here
    obj match {
      case Customer(name) => name.getBytes(UTF_8)
      case User(name)     => name.getBytes(UTF_8)
    }
  }

  // "fromBinary" deserializes the given array,
  // using the type hint
  def fromBinary(bytes: Array[Byte], manifest: String): AnyRef = {
    // Put the real code that deserializes here
    manifest match {
      case CustomerManifest =>
        Customer(new String(bytes, UTF_8))
      case UserManifest =>
        User(new String(bytes, UTF_8))
    }
```

```
  }
}
```

You must also bind it to a name in your *Configuration* and then list which classes that should be serialized using it.

Serializing ActorRefs

All ActorRefs are serializable using JavaSerializer, but in case you are writing your own serializer, you might want to know how to serialize and deserialize them properly. In the general case, the local address to be used depends on the type of remote address which shall be the recipient of the serialized information. Use `Serialization.serializedActorPath(actorRef)` like this:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory

    // Serialize
    // (beneath toBinary)
    val identifier: String = Serialization.serializedActorPath(theActorRef)

    // Then just serialize the identifier however you like

    // Deserialize
    // (beneath fromBinary)
    val deserializedActorRef = extendedSystem.provider.resolveActorRef(identifier)
    // Then just use the ActorRef
```

This assumes that serialization happens in the context of sending a message through the remote transport. There are other uses of serialization, though, e.g. storing actor references outside of an actor application (database, etc.). In this case, it is important to keep in mind that the address part of an actor's path determines how that actor is communicated with. Storing a local actor path might be the right choice if the retrieval happens in the same logical context, but it is not enough when deserializing it on a different network host: for that it would need to include the system's remote transport address. An actor system is not limited to having just one remote transport per se, which makes this question a bit more interesting. To find out the appropriate address to use when sending to `remoteAddr` you can use `ActorRefProvider.getExternalAddressFor(remoteAddr)` like this:

```
object ExternalAddress extends ExtensionKey[ExternalAddressExt]

class ExternalAddressExt(system: ExtendedActorSystem) extends Extension {
  def addressFor(remoteAddr: Address): Address =
    system.provider.getExternalAddressFor(remoteAddr) getOrElse
      (throw new UnsupportedOperationException("cannot send to " + remoteAddr))
}

def serializeTo(ref: ActorRef, remote: Address): String =
  ref.path.toSerializationFormatWithAddress(ExternalAddress(extendedSystem).
    addressFor(remote))
```

Note: `ActorPath.toSerializationFormatWithAddress` differs from `toString` if the address does not already have `host` and `port` components, i.e. it only inserts address information for local addresses.

`toSerializationFormatWithAddress` also adds the unique id of the actor, which will change when the actor is stopped and then created again with the same name. Sending messages to a reference pointing the old actor will not be delivered to the new actor. If you don't want this behavior, e.g. in case of long term storage of the reference, you can use `toStringWithAddress`, which doesn't include the unique id.

This requires that you know at least which type of address will be supported by the system which will deserialize the resulting actor reference; if you have no concrete address handy you can create a dummy one for the right

protocol using `Address(protocol, "", "", 0)` (assuming that the actual transport used is as lenient as Akka's RemoteActorRefProvider).

There is also a default remote address which is the one used by cluster support (and typical systems have just this one); you can get it like this:

```
object ExternalAddress extends ExtensionKey[ExternalAddressExt]

class ExternalAddressExt(system: ExtendedActorSystem) extends Extension {
  def addressForAkka: Address = system.provider.getDefaultAddress
}

def serializeAkkaDefault(ref: ActorRef): String =
  ref.path.toSerializationFormatWithAddress(ExternalAddress(theActorSystem).
    addressForAkka)
```

Deep serialization of Actors

The recommended approach to do deep serialization of internal actor state is to use Akka *Persistence*.

5.11.3 A Word About Java Serialization

When using Java serialization without employing the `JavaSerializer` for the task, you must make sure to supply a valid `ExtendedActorSystem` in the dynamic variable `JavaSerializer.currentSystem`. This is used when reading in the representation of an `ActorRef` for turning the string representation into a real reference. `DynamicVariable` is a thread-local variable, so be sure to have it set while deserializing anything which might contain actor references.

5.11.4 External Akka Serializers

Akka-protostuff by Roman Levenstein

Akka-quickser by Roman Levenstein

Akka-kryo by Roman Levenstein

Twitter Chill Scala extensions for Kryo (based on Akka Version 2.3.x but due to backwards compatibility of the Serializer Interface this extension also works with 2.4.x)

5.12 I/O

5.12.1 Introduction

The `akka.io` package has been developed in collaboration between the Akka and spray.io teams. Its design combines experiences from the `spray-io` module with improvements that were jointly developed for more general consumption as an actor-based service.

The guiding design goal for this I/O implementation was to reach extreme scalability, make no compromises in providing an API correctly matching the underlying transport mechanism and to be fully event-driven, non-blocking and asynchronous. The API is meant to be a solid foundation for the implementation of network protocols and building higher abstractions; it is not meant to be a full-service high-level NIO wrapper for end users.

5.12.2 Terminology, Concepts

The I/O API is completely actor based, meaning that all operations are implemented with message passing instead of direct method calls. Every I/O driver (TCP, UDP) has a special actor, called a *manager* that serves as an entry point for the API. I/O is broken into several drivers. The manager for a particular driver is accessible through the `IO` entry point. For example the following code looks up the TCP manager and returns its `ActorRef`:

```
import akka.io.{ IO, Tcp }
import context.system // implicitly used by IO(Tcp)

val manager = IO(Tcp)
```

The manager receives I/O command messages and instantiates worker actors in response. The worker actors present themselves to the API user in the reply to the command that was sent. For example after a `Connect` command sent to the TCP manager the manager creates an actor representing the TCP connection. All operations related to the given TCP connections can be invoked by sending messages to the connection actor which announces itself by sending a `Connected` message.

DeathWatch and Resource Management

I/O worker actors receive commands and also send out events. They usually need a user-side counterpart actor listening for these events (such events could be inbound connections, incoming bytes or acknowledgements for writes). These worker actors *watch* their listener counterparts. If the listener stops then the worker will automatically release any resources that it holds. This design makes the API more robust against resource leaks.

Thanks to the completely actor based approach of the I/O API the opposite direction works as well: a user actor responsible for handling a connection can watch the connection actor to be notified if it unexpectedly terminates.

Write models (Ack, Nack)

I/O devices have a maximum throughput which limits the frequency and size of writes. When an application tries to push more data than a device can handle, the driver has to buffer bytes until the device is able to write them. With buffering it is possible to handle short bursts of intensive writes — but no buffer is infinite. "Flow control" is needed to avoid overwhelming device buffers.

Akka supports two types of flow control:

- *Ack-based*, where the driver notifies the writer when writes have succeeded.
- *Nack-based*, where the driver notifies the writer when writes have failed.

Each of these models is available in both the TCP and the UDP implementations of Akka I/O.

Individual writes can be acknowledged by providing an ack object in the write message (`Write` in the case of TCP and `Send` for UDP). When the write is complete the worker will send the ack object to the writing actor. This can be used to implement *ack-based* flow control; sending new data only when old data has been acknowledged.

If a write (or any other command) fails, the driver notifies the actor that sent the command with a special message (`CommandFailed` in the case of UDP and TCP). This message will also notify the writer of a failed write, serving as a nack for that write. Please note, that in a nack-based flow-control setting the writer has to be prepared for the fact that the failed write might not be the most recent write it sent. For example, the failure notification for a write `W1` might arrive after additional write commands `W2` and `W3` have been sent. If the writer wants to resend any nacked messages it may need to keep a buffer of pending messages.

> **Warning:** An acknowledged write does not mean acknowledged delivery or storage; receiving an ack for a write simply signals that the I/O driver has successfully processed the write. The Ack/Nack protocol described here is a means of flow control not error handling. In other words, data may still be lost, even if every write is acknowledged.

ByteString

To maintain isolation, actors should communicate with immutable objects only. `ByteString` is an immutable container for bytes. It is used by Akka's I/O system as an efficient, immutable alternative the traditional byte containers used for I/O on the JVM, such as `Array[Byte]` and `ByteBuffer`.

`ByteString` is a rope-like data structure that is immutable and provides fast concatenation and slicing operations (perfect for I/O). When two `ByteString`s are concatenated together they are both stored within the resulting `ByteString` instead of copying both to a new `Array`. Operations such as `drop` and `take` return `ByteString`s that still reference the original `Array`, but just change the offset and length that is visible. Great care has also been taken to make sure that the internal `Array` cannot be modified. Whenever a potentially unsafe `Array` is used to create a new `ByteString` a defensive copy is created. If you require a `ByteString` that only blocks as much memory as necessary for it's content, use the `compact` method to get a `CompactByteString` instance. If the `ByteString` represented only a slice of the original array, this will result in copying all bytes in that slice.

`ByteString` inherits all methods from `IndexedSeq`, and it also has some new ones. For more information, look up the `akka.util.ByteString` class and it's companion object in the ScalaDoc.

`ByteString` also comes with its own optimized builder and iterator classes `ByteStringBuilder` and `ByteIterator` which provide extra features in addition to those of normal builders and iterators.

Compatibility with java.io

A `ByteStringBuilder` can be wrapped in a `java.io.OutputStream` via the `asOutputStream` method. Likewise, `ByteIterator` can be wrapped in a `java.io.InputStream` via `asInputStream`. Using these, `akka.io` applications can integrate legacy code based on `java.io` streams.

5.12.3 Architecture in-depth

For further details on the design and internal architecture see *I/O Layer Design*.

5.13 Using TCP

The code snippets through-out this section assume the following imports:

```
import akka.actor.{ Actor, ActorRef, Props }
import akka.io.{ IO, Tcp }
import akka.util.ByteString
import java.net.InetSocketAddress
```

All of the Akka I/O APIs are accessed through manager objects. When using an I/O API, the first step is to acquire a reference to the appropriate manager. The code below shows how to acquire a reference to the `Tcp` manager.

```
import akka.io.{ IO, Tcp }
import context.system // implicitly used by IO(Tcp)

val manager = IO(Tcp)
```

The manager is an actor that handles the underlying low level I/O resources (selectors, channels) and instantiates workers for specific tasks, such as listening to incoming connections.

5.13.1 Connecting

```
object Client {
  def props(remote: InetSocketAddress, replies: ActorRef) =
    Props(classOf[Client], remote, replies)
}

class Client(remote: InetSocketAddress, listener: ActorRef) extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  def receive = {
    case CommandFailed(_: Connect) =>
      listener ! "connect failed"
      context stop self

    case c @ Connected(remote, local) =>
      listener ! c
      val connection = sender()
      connection ! Register(self)
      context become {
        case data: ByteString =>
          connection ! Write(data)
        case CommandFailed(w: Write) =>
          // O/S buffer was full
          listener ! "write failed"
        case Received(data) =>
          listener ! data
        case "close" =>
          connection ! Close
        case _: ConnectionClosed =>
          listener ! "connection closed"
          context stop self
      }
  }
}
```

The first step of connecting to a remote address is sending a `Connect` message to the TCP manager; in addition to the simplest form shown above there is also the possibility to specify a local `InetSocketAddress` to bind to and a list of socket options to apply.

Note: The SO_NODELAY (TCP_NODELAY on Windows) socket option defaults to true in Akka, independently of the OS default settings. This setting disables Nagle's algorithm, considerably improving latency for most applications. This setting could be overridden by passing `SO.TcpNoDelay(false)` in the list of socket options of the `Connect` message.

The TCP manager will then reply either with a `CommandFailed` or it will spawn an internal actor representing the new connection. This new actor will then send a `Connected` message to the original sender of the `Connect` message.

In order to activate the new connection a `Register` message must be sent to the connection actor, informing that one about who shall receive data from the socket. Before this step is done the connection cannot be used, and there is an internal timeout after which the connection actor will shut itself down if no `Register` message is received.

The connection actor watches the registered handler and closes the connection when that one terminates, thereby cleaning up all internal resources associated with that connection.

The actor in the example above uses `become` to switch from unconnected to connected operation, demonstrating the commands and events which are observed in that state. For a discussion on `CommandFailed` see *Throttling*

Reads and Writes below. `ConnectionClosed` is a trait, which marks the different connection close events. The last line handles all connection close events in the same way. It is possible to listen for more fine-grained connection close events, see *Closing Connections* below.

5.13.2 Accepting connections

```
class Server extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0))

  def receive = {
    case b @ Bound(localAddress) =>
      // do some logging or setup ...

    case CommandFailed(_: Bind) => context stop self

    case c @ Connected(remote, local) =>
      val handler = context.actorOf(Props[SimplisticHandler])
      val connection = sender()
      connection ! Register(handler)
  }

}
```

To create a TCP server and listen for inbound connections, a `Bind` command has to be sent to the TCP manager. This will instruct the TCP manager to listen for TCP connections on a particular `InetSocketAddress`; the port may be specified as `0` in order to bind to a random port.

The actor sending the `Bind` message will receive a `Bound` message signaling that the server is ready to accept incoming connections; this message also contains the `InetSocketAddress` to which the socket was actually bound (i.e. resolved IP address and correct port number).

From this point forward the process of handling connections is the same as for outgoing connections. The example demonstrates that handling the reads from a certain connection can be delegated to another actor by naming it as the handler when sending the `Register` message. Writes can be sent from any actor in the system to the connection actor (i.e. the actor which sent the `Connected` message). The simplistic handler is defined as:

```
class SimplisticHandler extends Actor {
  import Tcp._
  def receive = {
    case Received(data) => sender() ! Write(data)
    case PeerClosed     => context stop self
  }
}
```

For a more complete sample which also takes into account the possibility of failures when sending please see *Throttling Reads and Writes* below.

The only difference to outgoing connections is that the internal actor managing the listen port—the sender of the `Bound` message—watches the actor which was named as the recipient for `Connected` messages in the `Bind` message. When that actor terminates the listen port will be closed and all resources associated with it will be released; existing connections will not be terminated at this point.

5.13.3 Closing connections

A connection can be closed by sending one of the commands `Close`, `ConfirmedClose` or `Abort` to the connection actor.

`Close` will close the connection by sending a `FIN` message, but without waiting for confirmation from the remote endpoint. Pending writes will be flushed. If the close is successful, the listener will be notified with `Closed`.

`ConfirmedClose` will close the sending direction of the connection by sending a `FIN` message, but data will continue to be received until the remote endpoint closes the connection, too. Pending writes will be flushed. If the close is successful, the listener will be notified with `ConfirmedClosed`.

`Abort` will immediately terminate the connection by sending a `RST` message to the remote endpoint. Pending writes will be not flushed. If the close is successful, the listener will be notified with `Aborted`.

`PeerClosed` will be sent to the listener if the connection has been closed by the remote endpoint. Per default, the connection will then automatically be closed from this endpoint as well. To support half-closed connections set the `keepOpenOnPeerClosed` member of the `Register` message to `true` in which case the connection stays open until it receives one of the above close commands.

`ErrorClosed` will be sent to the listener whenever an error happened that forced the connection to be closed.

All close notifications are sub-types of `ConnectionClosed` so listeners who do not need fine-grained close events may handle all close events in the same way.

5.13.4 Writing to a connection

Once a connection has been established data can be sent to it from any actor in the form of a `Tcp.WriteCommand`. `Tcp.WriteCommand` is an abstract class with three concrete implementations:

Tcp.Write The simplest `WriteCommand` implementation which wraps a `ByteString` instance and an "ack" event. A `ByteString` (as explained in *this section*) models one or more chunks of immutable in-memory data with a maximum (total) size of 2 GB (2^31 bytes).

Tcp.WriteFile If you want to send "raw" data from a file you can do so efficiently with the `Tcp.WriteFile` command. This allows you do designate a (contiguous) chunk of on-disk bytes for sending across the connection without the need to first load them into the JVM memory. As such `Tcp.WriteFile` can "hold" more than 2GB of data and an "ack" event if required.

Tcp.CompoundWrite Sometimes you might want to group (or interleave) several `Tcp.Write` and/or `Tcp.WriteFile` commands into one atomic write command which gets written to the connection in one go. The `Tcp.CompoundWrite` allows you to do just that and offers three benefits:

1. As explained in the following section the TCP connection actor can only handle one single write command at a time. By combining several writes into one `CompoundWrite` you can have them be sent across the connection with minimum overhead and without the need to spoon feed them to the connection actor via an *ACK-based* message protocol.
2. Because a `WriteCommand` is atomic you can be sure that no other actor can "inject" other writes into your series of writes if you combine them into one single `CompoundWrite`. In scenarios where several actors write to the same connection this can be an important feature which can be somewhat hard to achieve otherwise.
3. The "sub writes" of a `CompoundWrite` are regular `Write` or `WriteFile` commands that themselves can request "ack" events. These ACKs are sent out as soon as the respective "sub write" has been completed. This allows you to attach more than one ACK to a `Write` or `WriteFile` (by combining it with an empty write that itself requests an ACK) or to have the connection actor acknowledge the progress of transmitting the `CompoundWrite` by sending out intermediate ACKs at arbitrary points.

5.13.5 Throttling Reads and Writes

The basic model of the TCP connection actor is that it has no internal buffering (i.e. it can only process one write at a time, meaning it can buffer one write until it has been passed on to the O/S kernel in full). Congestion needs to be handled at the user level, for both writes and reads.

For back-pressuring writes there are three modes of operation

- *ACK-based:* every `Write` command carries an arbitrary object, and if this object is not `Tcp.NoAck` then it will be returned to the sender of the `Write` upon successfully writing all contained data to the socket. If no other write is initiated before having received this acknowledgement then no failures can happen due to buffer overrun.
- *NACK-based:* every write which arrives while a previous write is not yet completed will be replied to with a `CommandFailed` message containing the failed write. Just relying on this mechanism requires the implemented protocol to tolerate skipping writes (e.g. if each write is a valid message on its own and it is not required that all are delivered). This mode is enabled by setting the `useResumeWriting` flag to `false` within the `Register` message during connection activation.
- *NACK-based with write suspending:* this mode is very similar to the NACK-based one, but once a single write has failed no further writes will succeed until a `ResumeWriting` message is received. This message will be answered with a `WritingResumed` message once the last accepted write has completed. If the actor driving the connection implements buffering and resends the NACK'ed messages after having awaited the `WritingResumed` signal then every message is delivered exactly once to the network socket.

These write back-pressure models (with the exception of the second which is rather specialised) are demonstrated in complete examples below. The full and contiguous source is available on GitHub.

For back-pressuring reads there are two modes of operation

- *Push-reading:* in this mode the connection actor sends the registered reader actor incoming data as soon as available as `Received` events. Whenever the reader actor wants to signal back-pressure to the remote TCP endpoint it can send a `SuspendReading` message to the connection actor to indicate that it wants to suspend the reception of new data. No `Received` events will arrive until a corresponding `ResumeReading` is sent indicating that the receiver actor is ready again.
- *Pull-reading:* after sending a `Received` event the connection actor automatically suspends accepting data from the socket until the reader actor signals with a `ResumeReading` message that it is ready to process more input data. Hence new data is "pulled" from the connection by sending `ResumeReading` messages.

Note: It should be obvious that all these flow control schemes only work between one writer/reader and one connection actor; as soon as multiple actors send write commands to a single connection no consistent result can be achieved.

5.13.6 ACK-Based Write Back-Pressure

For proper function of the following example it is important to configure the connection to remain half-open when the remote side closed its writing end: this allows the example `EchoHandler` to write all outstanding data back to the client before fully closing the connection. This is enabled using a flag upon connection activation (observe the `Register` message):

```
case Connected(remote, local) =>
  log.info("received connection from {}", remote)
  val handler = context.actorOf(Props(handlerClass, sender(), remote))
  sender() ! Register(handler, keepOpenOnPeerClosed = true)
```

With this preparation let us dive into the handler itself:

```
  // storage omitted ...
class SimpleEchoHandler(connection: ActorRef, remote: InetSocketAddress)
  extends Actor with ActorLogging {

  import Tcp._

  // sign death pact: this actor terminates when connection breaks
  context watch connection

  case object Ack extends Event
```

```
  def receive = {
    case Received(data) =>
      buffer(data)
      connection ! Write(data, Ack)

      context.become({
        case Received(data) => buffer(data)
        case Ack            => acknowledge()
        case PeerClosed     => closing = true
      }, discardOld = false)

    case PeerClosed => context stop self
  }

  // storage omitted ...
}
```

The principle is simple: when having written a chunk always wait for the `Ack` to come back before sending the next chunk. While waiting we switch behavior such that new incoming data are buffered. The helper functions used are a bit lengthy but not complicated:

```
private def buffer(data: ByteString): Unit = {
  storage :+= data
  stored += data.size

  if (stored > maxStored) {
    log.warning(s"drop connection to [$remote] (buffer overrun)")
    context stop self

  } else if (stored > highWatermark) {
    log.debug(s"suspending reading")
    connection ! SuspendReading
    suspended = true
  }
}

private def acknowledge(): Unit = {
  require(storage.nonEmpty, "storage was empty")

  val size = storage(0).size
  stored -= size
  transferred += size

  storage = storage drop 1

  if (suspended && stored < lowWatermark) {
    log.debug("resuming reading")
    connection ! ResumeReading
    suspended = false
  }

  if (storage.isEmpty) {
    if (closing) context stop self
    else context.unbecome()
  } else connection ! Write(storage(0), Ack)
}
```

The most interesting part is probably the last: an `Ack` removes the oldest data chunk from the buffer, and if that was the last chunk then we either close the connection (if the peer closed its half already) or return to the idle behavior; otherwise we just send the next buffered chunk and stay waiting for the next `Ack`.

Back-pressure can be propagated also across the reading side back to the writer on the other end of the connection

by sending the `SuspendReading` command to the connection actor. This will lead to no data being read from the socket anymore (although this does happen after a delay because it takes some time until the connection actor processes this command, hence appropriate head-room in the buffer should be present), which in turn will lead to the O/S kernel buffer filling up on our end, then the TCP window mechanism will stop the remote side from writing, filling up its write buffer, until finally the writer on the other side cannot push any data into the socket anymore. This is how end-to-end back-pressure is realized across a TCP connection.

5.13.7 NACK-Based Write Back-Pressure with Suspending

```
object EchoHandler {
  final case class Ack(offset: Int) extends Tcp.Event

  def props(connection: ActorRef, remote: InetSocketAddress): Props =
    Props(classOf[EchoHandler], connection, remote)
}

class EchoHandler(connection: ActorRef, remote: InetSocketAddress)
  extends Actor with ActorLogging {

  import Tcp._
  import EchoHandler._

  // sign death pact: this actor terminates when connection breaks
  context watch connection

  // start out in optimistic write-through mode
  def receive = writing

  def writing: Receive = {
    case Received(data) =>
      connection ! Write(data, Ack(currentOffset))
      buffer(data)

    case Ack(ack) =>
      acknowledge(ack)

    case CommandFailed(Write(_, Ack(ack))) =>
      connection ! ResumeWriting
      context become buffering(ack)

    case PeerClosed =>
      if (storage.isEmpty) context stop self
      else context become closing
  }

  // buffering ...

  // closing ...

  override def postStop(): Unit = {
    log.info(s"transferred $transferred bytes from/to [$remote]")
  }

  // storage omitted ...
}
  // storage omitted ...
```

The principle here is to keep writing until a `CommandFailed` is received, using acknowledgements only to prune the resend buffer. When a such a failure was received, transition into a different state for handling and handle resending of all queued data:

```
def buffering(nack: Int): Receive = {
  var toAck = 10
  var peerClosed = false

  {
    case Received(data)          => buffer(data)
    case WritingResumed          => writeFirst()
    case PeerClosed              => peerClosed = true
    case Ack(ack) if ack < nack => acknowledge(ack)
    case Ack(ack) =>
      acknowledge(ack)
      if (storage.nonEmpty) {
        if (toAck > 0) {
          // stay in ACK-based mode for a while
          writeFirst()
          toAck -= 1
        } else {
          // then return to NACK-based again
          writeAll()
          context become (if (peerClosed) closing else writing)
        }
      } else if (peerClosed) context stop self
      else context become writing
  }
}
```

It should be noted that all writes which are currently buffered have also been sent to the connection actor upon entering this state, which means that the `ResumeWriting` message is enqueued after those writes, leading to the reception of all outstanding `CommandFailed` messages (which are ignored in this state) before receiving the `WritingResumed` signal. That latter message is sent by the connection actor only once the internally queued write has been fully completed, meaning that a subsequent write will not fail. This is exploited by the `EchoHandler` to switch to an ACK-based approach for the first ten writes after a failure before resuming the optimistic write-through behavior.

```
def closing: Receive = {
  case CommandFailed(_: Write) =>
    connection ! ResumeWriting
    context.become({

      case WritingResumed =>
        writeAll()
        context.unbecome()

      case ack: Int => acknowledge(ack)

    }, discardOld = false)

  case Ack(ack) =>
    acknowledge(ack)
    if (storage.isEmpty) context stop self
}
```

Closing the connection while still sending all data is a bit more involved than in the ACK-based approach: the idea is to always send all outstanding messages and acknowledge all successful writes, and if a failure happens then switch behavior to await the `WritingResumed` event and start over.

The helper functions are very similar to the ACK-based case:

```
private def buffer(data: ByteString): Unit = {
  storage :+= data
  stored += data.size
```

```
  if (stored > maxStored) {
    log.warning(s"drop connection to [$remote] (buffer overrun)")
    context stop self

  } else if (stored > highWatermark) {
    log.debug(s"suspending reading at $currentOffset")
    connection ! SuspendReading
    suspended = true
  }
}

private def acknowledge(ack: Int): Unit = {
  require(ack == storageOffset, s"received ack $ack at $storageOffset")
  require(storage.nonEmpty, s"storage was empty at ack $ack")

  val size = storage(0).size
  stored -= size
  transferred += size

  storageOffset += 1
  storage = storage drop 1

  if (suspended && stored < lowWatermark) {
    log.debug("resuming reading")
    connection ! ResumeReading
    suspended = false
  }
}
```

5.13.8 Read Back-Pressure with Pull Mode

When using push based reading, data coming from the socket is sent to the actor as soon as it is available. In the case of the previous Echo server example this meant that we needed to maintain a buffer of incoming data to keep it around since the rate of writing might be slower than the rate of the arrival of new data.

With the Pull mode this buffer can be completely eliminated as the following snippet demonstrates:

```
override def preStart: Unit = connection ! ResumeReading

def receive = {
  case Received(data) => connection ! Write(data, Ack)
  case Ack            => connection ! ResumeReading
}
```

The idea here is that reading is not resumed until the previous write has been completely acknowledged by the connection actor. Every pull mode connection actor starts from suspended state. To start the flow of data we send a `ResumeReading` in the `preStart` method to tell the connection actor that we are ready to receive the first chunk of data. Since we only resume reading when the previous data chunk has been completely written there is no need for maintaining a buffer.

To enable pull reading on an outbound connection the `pullMode` parameter of the `Connect` should be set to `true`:

```
IO(Tcp) ! Connect(listenAddress, pullMode = true)
```

Pull Mode Reading for Inbound Connections

The previous section demonstrated how to enable pull reading mode for outbound connections but it is possible to create a listener actor with this mode of reading by setting the `pullMode` parameter of the `Bind` command to

`true`:

```
IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0), pullMode = true)
```

One of the effects of this setting is that all connections accepted by this listener actor will use pull mode reading.

Another effect of this setting is that in addition of setting all inbound connections to pull mode, accepting connections becomes pull based, too. This means that after handling one (or more) `Connected` events the listener actor has to be resumed by sending it a `ResumeAccepting` message.

Listener actors with pull mode start suspended so to start accepting connections a `ResumeAccepting` command has to be sent to the listener actor after binding was successful:

```
case Bound(localAddress) =>
  // Accept connections one by one
  sender() ! ResumeAccepting(batchSize = 1)
  context.become(listening(sender()))
```

After handling an incoming connection we need to resume accepting again:

```
def listening(listener: ActorRef): Receive = {
  case Connected(remote, local) =>
    val handler = context.actorOf(Props(classOf[PullEcho], sender()))
    sender() ! Register(handler, keepOpenOnPeerClosed = true)
    listener ! ResumeAccepting(batchSize = 1)
}
```

The `ResumeAccepting` accepts a `batchSize` parameter that specifies how many new connections are accepted before a next `ResumeAccepting` message is needed to resume handling of new connections.

5.14 Using UDP

UDP is a connectionless datagram protocol which offers two different ways of communication on the JDK level:

- sockets which are free to send datagrams to any destination and receive datagrams from any origin
- sockets which are restricted to communication with one specific remote socket address

In the low-level API the distinction is made—confusingly—by whether or not `connect` has been called on the socket (even when connect has been called the protocol is still connectionless). These two forms of UDP usage are offered using distinct IO extensions described below.

5.14.1 Unconnected UDP

Simple Send

```
class SimpleSender(remote: InetSocketAddress) extends Actor {
  import context.system
  IO(Udp) ! Udp.SimpleSender

  def receive = {
    case Udp.SimpleSenderReady =>
      context.become(ready(sender()))
  }

  def ready(send: ActorRef): Receive = {
    case msg: String =>
      send ! Udp.Send(ByteString(msg), remote)
  }
}
```

The simplest form of UDP usage is to just send datagrams without the need of getting a reply. To this end a "simple sender" facility is provided as demonstrated above. The UDP extension is queried using the `SimpleSender` message, which is answered by a `SimpleSenderReady` notification. The sender of this message is the newly created sender actor which from this point onward can be used to send datagrams to arbitrary destinations; in this example it will just send any UTF-8 encoded `String` it receives to a predefined remote address.

Note: The simple sender will not shut itself down because it cannot know when you are done with it. You will need to send it a `PoisonPill` when you want to close the ephemeral port the sender is bound to.

Bind (and Send)

```
class Listener(nextActor: ActorRef) extends Actor {
  import context.system
  IO(Udp) ! Udp.Bind(self, new InetSocketAddress("localhost", 0))

  def receive = {
    case Udp.Bound(local) =>
      context.become(ready(sender()))
  }

  def ready(socket: ActorRef): Receive = {
    case Udp.Received(data, remote) =>
      val processed = // parse data etc., e.g. using PipelineStage
      socket ! Udp.Send(data, remote) // example server echoes back
      nextActor ! processed
    case Udp.Unbind  => socket ! Udp.Unbind
    case Udp.Unbound => context.stop(self)
  }
}
```

If you want to implement a UDP server which listens on a socket for incoming datagrams then you need to use the `Bind` command as shown above. The local address specified may have a zero port in which case the operating system will automatically choose a free port and assign it to the new socket. Which port was actually bound can be found out by inspecting the `Bound` message.

The sender of the `Bound` message is the actor which manages the new socket. Sending datagrams is achieved by using the `Send` message type and the socket can be closed by sending a `Unbind` command, in which case the socket actor will reply with a `Unbound` notification.

Received datagrams are sent to the actor designated in the `Bind` message, whereas the `Bound` message will be sent to the sender of the `Bind`.

5.14.2 Connected UDP

The service provided by the connection based UDP API is similar to the bind-and-send service we saw earlier, but the main difference is that a connection is only able to send to the `remoteAddress` it was connected to, and will receive datagrams only from that address.

```
class Connected(remote: InetSocketAddress) extends Actor {
  import context.system
  IO(UdpConnected) ! UdpConnected.Connect(self, remote)

  def receive = {
    case UdpConnected.Connected =>
      context.become(ready(sender()))
  }

  def ready(connection: ActorRef): Receive = {
```

```
    case UdpConnected.Received(data) =>
      // process data, send it on, etc.
    case msg: String =>
      connection ! UdpConnected.Send(ByteString(msg))
    case UdpConnected.Disconnect =>
      connection ! UdpConnected.Disconnect
    case UdpConnected.Disconnected => context.stop(self)
  }
}
```

Consequently the example shown here looks quite similar to the previous one, the biggest difference is the absence of remote address information in `Send` and `Received` messages.

Note: There is a small performance benefit in using connection based UDP API over the connectionless one. If there is a SecurityManager enabled on the system, every connectionless message send has to go through a security check, while in the case of connection-based UDP the security check is cached after connect, thus writes do not suffer an additional performance penalty.

5.14.3 UDP Multicast

If you want to use UDP multicast you will need to use Java 7. Akka provides a way to control various options of `DatagramChannel` through the `akka.io.Inet.SocketOption` interface. The example below shows how to setup a receiver of multicast messages using IPv6 protocol.

To select a Protocol Family you must extend `akka.io.Inet.DatagramChannelCreator` class which extends `akka.io.Inet.SocketOption`. Provide custom logic for opening a datagram channel by overriding `create` method.

```
final case class Inet6ProtocolFamily() extends DatagramChannelCreator {
  override def create() =
    DatagramChannel.open(StandardProtocolFamily.INET6)
}
```

Another socket option will be needed to join a multicast group.

```
final case class MulticastGroup(address: String, interface: String) extends␣
↪SocketOptionV2 {
  override def afterBind(s: DatagramSocket) {
    val group = InetAddress.getByName(address)
    val networkInterface = NetworkInterface.getByName(interface)
    s.getChannel.join(group, networkInterface)
  }
}
```

Socket options must be provided to `UdpMessage.Bind` message.

```
import context.system
val opts = List(Inet6ProtocolFamily(), MulticastGroup(group, iface))
IO(Udp) ! Udp.Bind(self, new InetSocketAddress(port), opts)
```

5.15 Camel

5.15.1 Introduction

The akka-camel module allows Untyped Actors to receive and send messages over a great variety of protocols and APIs. In addition to the native Scala and Java actor API, actors can now exchange messages with other systems

over large number of protocols and APIs such as HTTP, SOAP, TCP, FTP, SMTP or JMS, to mention a few. At the moment, approximately 80 protocols and APIs are supported.

Apache Camel

The akka-camel module is based on Apache Camel, a powerful and light-weight integration framework for the JVM. For an introduction to Apache Camel you may want to read this Apache Camel article. Camel comes with a large number of components that provide bindings to different protocols and APIs. The camel-extra project provides further components.

Consumer

Usage of Camel's integration components in Akka is essentially a one-liner. Here's an example.

```
import akka.camel.{ CamelMessage, Consumer }

class MyEndpoint extends Consumer {
  def endpointUri = "mina2:tcp://localhost:6200?textline=true"

  def receive = {
    case msg: CamelMessage => { /* ... */ }
    case _                 => { /* ... */ }
  }
}

// start and expose actor via tcp
import akka.actor.{ ActorSystem, Props }

val system = ActorSystem("some-system")
val mina = system.actorOf(Props[MyEndpoint])
```

The above example exposes an actor over a TCP endpoint via Apache Camel's Mina component. The actor implements the endpointUri method to define an endpoint from which it can receive messages. After starting the actor, TCP clients can immediately send messages to and receive responses from that actor. If the message exchange should go over HTTP (via Camel's Jetty component), only the actor's endpointUri method must be changed.

```
import akka.camel.{ CamelMessage, Consumer }

class MyEndpoint extends Consumer {
  def endpointUri = "jetty:http://localhost:8877/example"

  def receive = {
    case msg: CamelMessage => { /* ... */ }
    case _                 => { /* ... */ }
  }
}
```

Producer

Actors can also trigger message exchanges with external systems i.e. produce to Camel endpoints.

```
import akka.actor.Actor
import akka.camel.{ Producer, Oneway }
import akka.actor.{ ActorSystem, Props }

class Orders extends Actor with Producer with Oneway {
  def endpointUri = "jms:queue:Orders"
}
```

```
val sys = ActorSystem("some-system")
val orders = sys.actorOf(Props[Orders])

orders ! <order amount="100" currency="PLN" itemId="12345"/>
```

In the above example, any message sent to this actor will be sent to the JMS queue `orders`. Producer actors may choose from the same set of Camel components as Consumer actors do.

CamelMessage

The number of Camel components is constantly increasing. The akka-camel module can support these in a plug-and-play manner. Just add them to your application's classpath, define a component-specific endpoint URI and use it to exchange messages over the component-specific protocols or APIs. This is possible because Camel components bind protocol-specific message formats to a Camel-specific normalized message format. The normalized message format hides protocol-specific details from Akka and makes it therefore very easy to support a large number of protocols through a uniform Camel component interface. The akka-camel module further converts mutable Camel messages into immutable representations which are used by Consumer and Producer actors for pattern matching, transformation, serialization or storage. In the above example of the Orders Producer, the XML message is put in the body of a newly created Camel Message with an empty set of headers. You can also create a CamelMessage yourself with the appropriate body and headers as you see fit.

CamelExtension

The akka-camel module is implemented as an Akka Extension, the `CamelExtension` object. Extensions will only be loaded once per `ActorSystem`, which will be managed by Akka. The `CamelExtension` object provides access to the Camel trait. The Camel trait in turn provides access to two important Apache Camel objects, the CamelContext and the ProducerTemplate. Below you can see how you can get access to these Apache Camel objects.

```
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val camelContext = camel.context
val producerTemplate = camel.template
```

One `CamelExtension` is only loaded once for every one `ActorSystem`, which makes it safe to call the `CamelExtension` at any point in your code to get to the Apache Camel objects associated with it. There is one CamelContext and one ProducerTemplate for every one `ActorSystem` that uses a `CamelExtension`. By Default, a new CamelContext is created when the `CamelExtension` starts. If you want to inject your own context instead, you can extend the ContextProvider trait and add the FQCN of your implementation in the config, as the value of the "akka.camel.context-provider". This interface define a single method `getContext` used to load the CamelContext.

Below an example on how to add the ActiveMQ component to the CamelContext, which is required when you would like to use the ActiveMQ component.

```
// import org.apache.activemq.camel.component.ActiveMQComponent
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val camelContext = camel.context
// camelContext.addComponent("activemq", ActiveMQComponent.activeMQComponent(
//   "vm://localhost?broker.persistent=false"))
```

The CamelContext joins the lifecycle of the `ActorSystem` and `CamelExtension` it is associated with; the CamelContext is started when the `CamelExtension` is created, and it is shut down when the associated `ActorSystem` is shut down. The same is true for the ProducerTemplate.

The `CamelExtension` is used by both *Producer* and *Consumer* actors to interact with Apache Camel internally. You can access the `CamelExtension` inside a *Producer* or a *Consumer* using the `camel` definition, or get

straight at the *CamelContext* using the `camelContext` definition. Actors are created and started asynchronously. When a *Consumer* actor is created, the *Consumer* is published at its Camel endpoint (more precisely, the route is added to the CamelContext from the Endpoint to the actor). When a *Producer* actor is created, a SendProcessor and Endpoint are created so that the Producer can send messages to it. Publication is done asynchronously; setting up an endpoint may still be in progress after you have requested the actor to be created. Some Camel components can take a while to startup, and in some cases you might want to know when the endpoints are activated and ready to be used. The Camel trait allows you to find out when the endpoint is activated or deactivated.

```
import akka.camel.{ CamelMessage, Consumer }
import scala.concurrent.duration._

class MyEndpoint extends Consumer {
  def endpointUri = "mina2:tcp://localhost:6200?textline=true"

  def receive = {
    case msg: CamelMessage => { /* ... */ }
    case _                 => { /* ... */ }
  }
}
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val actorRef = system.actorOf(Props[MyEndpoint])
// get a future reference to the activation of the endpoint of the Consumer Actor
val activationFuture = camel.activationFutureFor(actorRef)(
  timeout = 10 seconds,
  executor = system.dispatcher)
```

The above code shows that you can get a `Future` to the activation of the route from the endpoint to the actor, or you can wait in a blocking fashion on the activation of the route. An `ActivationTimeoutException` is thrown if the endpoint could not be activated within the specified timeout. Deactivation works in a similar fashion:

```
system.stop(actorRef)
// get a future reference to the deactivation of the endpoint of the Consumer Actor
val deactivationFuture = camel.deactivationFutureFor(actorRef)(
  timeout = 10 seconds,
  executor = system.dispatcher)
```

Deactivation of a Consumer or a Producer actor happens when the actor is terminated. For a Consumer, the route to the actor is stopped. For a Producer, the SendProcessor is stopped. A `DeActivationTimeoutException` is thrown if the associated camel objects could not be deactivated within the specified timeout.

5.15.2 Consumer Actors

For objects to receive messages, they must mixin the Consumer trait. For example, the following actor class (Consumer1) implements the endpointUri method, which is declared in the Consumer trait, in order to receive messages from the `file:data/input/actor` Camel endpoint.

```
import akka.camel.{ CamelMessage, Consumer }

class Consumer1 extends Consumer {
  def endpointUri = "file:data/input/actor"

  def receive = {
    case msg: CamelMessage => println("received %s" format msg.bodyAs[String])
  }
}
```

Whenever a file is put into the data/input/actor directory, its content is picked up by the Camel file component and sent as message to the actor. Messages consumed by actors from Camel endpoints are of type CamelMessage. These are immutable representations of Camel messages.

Here's another example that sets the endpointUri to `jetty:http://localhost:8877/camel/default`. It causes Camel's Jetty component to start an embedded Jetty server, accepting HTTP connections from localhost on port 8877.

```
import akka.camel.{ CamelMessage, Consumer }

class Consumer2 extends Consumer {
  def endpointUri = "jetty:http://localhost:8877/camel/default"

  def receive = {
    case msg: CamelMessage => sender() ! ("Hello %s" format msg.bodyAs[String])
  }
}
```

After starting the actor, clients can send messages to that actor by POSTing to `http://localhost:8877/camel/default`. The actor sends a response by using the sender *!* method. For returning a message body and headers to the HTTP client the response type should be CamelMessage. For any other response type, a new CamelMessage object is created by akka-camel with the actor response as message body.

Delivery acknowledgements

With in-out message exchanges, clients usually know that a message exchange is done when they receive a reply from a consumer actor. The reply message can be a CamelMessage (or any object which is then internally converted to a CamelMessage) on success, and a Failure message on failure.

With in-only message exchanges, by default, an exchange is done when a message is added to the consumer actor's mailbox. Any failure or exception that occurs during processing of that message by the consumer actor cannot be reported back to the endpoint in this case. To allow consumer actors to positively or negatively acknowledge the receipt of a message from an in-only message exchange, they need to override the `autoAck` method to return false. In this case, consumer actors must reply either with a special akka.camel.Ack message (positive acknowledgement) or a akka.actor.Status.Failure (negative acknowledgement).

```
import akka.camel.{ CamelMessage, Consumer }
import akka.camel.Ack
import akka.actor.Status.Failure

class Consumer3 extends Consumer {
  override def autoAck = false

  def endpointUri = "jms:queue:test"

  def receive = {
    case msg: CamelMessage =>
      sender() ! Ack
      // on success
      // ..
      val someException = new Exception("e1")
      // on failure
      sender() ! Failure(someException)
  }
}
```

Consumer timeout

Camel Exchanges (and their corresponding endpoints) that support two-way communications need to wait for a response from an actor before returning it to the initiating client. For some endpoint types, timeout values can be defined in an endpoint-specific way which is described in the documentation of the individual Camel components. Another option is to configure timeouts on the level of consumer actors.

Two-way communications between a Camel endpoint and an actor are initiated by sending the request message to the actor with the ask pattern and the actor replies to the endpoint when the response is ready. The ask request to the actor can timeout, which will result in the Exchange failing with a TimeoutException set on the failure of the Exchange. The timeout on the consumer actor can be overridden with the `replyTimeout`, as shown below.

```
import akka.camel.{ CamelMessage, Consumer }
import scala.concurrent.duration._

class Consumer4 extends Consumer {
  def endpointUri = "jetty:http://localhost:8877/camel/default"
  override def replyTimeout = 500 millis
  def receive = {
    case msg: CamelMessage => sender() ! ("Hello %s" format msg.bodyAs[String])
  }
}
```

5.15.3 Producer Actors

For sending messages to Camel endpoints, actors need to mixin the Producer trait and implement the endpointUri method.

```
import akka.actor.Actor
import akka.actor.{ Props, ActorSystem }
import akka.camel.{ Producer, CamelMessage }
import akka.util.Timeout

class Producer1 extends Actor with Producer {
  def endpointUri = "http://localhost:8080/news"
}
```

Producer1 inherits a default implementation of the receive method from the Producer trait. To customize a producer actor's default behavior you must override the Producer.transformResponse and Producer.transformOutgoingMessage methods. This is explained later in more detail. Producer Actors cannot override the default Producer.receive method.

Any message sent to a Producer actor will be sent to the associated Camel endpoint, in the above example to `http://localhost:8080/news`. The Producer always sends messages asynchronously. Response messages (if supported by the configured endpoint) will, by default, be returned to the original sender. The following example uses the ask pattern to send a message to a Producer actor and waits for a response.

```
import akka.pattern.ask
import scala.concurrent.duration._
implicit val timeout = Timeout(10 seconds)

val system = ActorSystem("some-system")
val producer = system.actorOf(Props[Producer1])
val future = producer.ask("some request").mapTo[CamelMessage]
```

The future contains the response CamelMessage, or an `AkkaCamelException` when an error occurred, which contains the headers of the response.

Custom Processing

Instead of replying to the initial sender, producer actors can implement custom response processing by overriding the routeResponse method. In the following example, the response message is forwarded to a target actor instead of being replied to the original sender.

```
import akka.actor.{ Actor, ActorRef }
import akka.camel.{ Producer, CamelMessage }
```

```
import akka.actor.{ Props, ActorSystem }

class ResponseReceiver extends Actor {
  def receive = {
    case msg: CamelMessage =>
    // do something with the forwarded response
  }
}

class Forwarder(uri: String, target: ActorRef) extends Actor with Producer {
  def endpointUri = uri

  override def routeResponse(msg: Any) { target forward msg }
}
val system = ActorSystem("some-system")
val receiver = system.actorOf(Props[ResponseReceiver])
val forwardResponse = system.actorOf(
  Props(classOf[Forwarder], this, "http://localhost:8080/news/akka", receiver))
// the Forwarder sends out a request to the web page and forwards the response to
// the ResponseReceiver
forwardResponse ! "some request"
```

Before producing messages to endpoints, producer actors can pre-process them by overriding the Producer.transformOutgoingMessage method.

```
import akka.actor.Actor
import akka.camel.{ Producer, CamelMessage }

class Transformer(uri: String) extends Actor with Producer {
  def endpointUri = uri

  def upperCase(msg: CamelMessage) = msg.mapBody {
    body: String => body.toUpperCase
  }

  override def transformOutgoingMessage(msg: Any) = msg match {
    case msg: CamelMessage => upperCase(msg)
  }
}
```

Producer configuration options

The interaction of producer actors with Camel endpoints can be configured to be one-way or two-way (by initiating in-only or in-out message exchanges, respectively). By default, the producer initiates an in-out message exchange with the endpoint. For initiating an in-only exchange, producer actors have to override the oneway method to return true.

```
import akka.actor.{ Actor, Props, ActorSystem }
import akka.camel.Producer

class OnewaySender(uri: String) extends Actor with Producer {
  def endpointUri = uri
  override def oneway: Boolean = true
}

val system = ActorSystem("some-system")
val producer = system.actorOf(Props(classOf[OnewaySender], this, "activemq:FOO.BAR
↪"))
producer ! "Some message"
```

Message correlation

To correlate request with response messages, applications can set the *Message.MessageExchangeId* message header.

```
import akka.camel.{ Producer, CamelMessage }
import akka.actor.Actor
import akka.actor.{ Props, ActorSystem }

class Producer2 extends Actor with Producer {
  def endpointUri = "activemq:FOO.BAR"
}
val system = ActorSystem("some-system")
val producer = system.actorOf(Props[Producer2])

producer ! CamelMessage("bar", Map(CamelMessage.MessageExchangeId -> "123"))
```

ProducerTemplate

The Producer trait is a very convenient way for actors to produce messages to Camel endpoints. Actors may also use a Camel ProducerTemplate for producing messages to endpoints.

```
import akka.actor.Actor
class MyActor extends Actor {
  def receive = {
    case msg =>
      val template = CamelExtension(context.system).template
      template.sendBody("direct:news", msg)
  }
}
```

For initiating a two-way message exchange, one of the `ProducerTemplate.request*` methods must be used.

```
import akka.actor.Actor
class MyActor extends Actor {
  def receive = {
    case msg =>
      val template = CamelExtension(context.system).template
      sender() ! template.requestBody("direct:news", msg)
  }
}
```

5.15.4 Asynchronous routing

In-out message exchanges between endpoints and actors are designed to be asynchronous. This is the case for both, consumer and producer actors.

- A consumer endpoint sends request messages to its consumer actor using the `!` (tell) operator and the actor returns responses with `sender !` once they are ready.
- A producer actor sends request messages to its endpoint using Camel's asynchronous routing engine. Asynchronous responses are wrapped and added to the producer actor's mailbox for later processing. By default, response messages are returned to the initial sender but this can be overridden by Producer implementations (see also description of the `routeResponse` method in *Custom Processing*).

However, asynchronous two-way message exchanges, without allocating a thread for the full duration of exchange, cannot be generically supported by Camel's asynchronous routing engine alone. This must be supported by the individual Camel components (from which endpoints are created) as well. They must be able to suspend any work started for request processing (thereby freeing threads to do other work) and resume processing when the response

is ready. This is currently the case for a subset of components such as the Jetty component. All other Camel components can still be used, of course, but they will cause allocation of a thread for the duration of an in-out message exchange. There's also *Examples* that implements both, an asynchronous consumer and an asynchronous producer, with the jetty component.

If the used Camel component is blocking it might be necessary to use a separate *dispatcher* for the producer. The Camel processor is invoked by a child actor of the producer and the dispatcher can be defined in the deployment section of the configuration. For example, if your producer actor has path `/user/integration/output` the dispatcher of the child actor can be defined with:

```
akka.actor.deployment {
  /integration/output/* {
    dispatcher = my-dispatcher
  }
}
```

5.15.5 Custom Camel routes

In all the examples so far, routes to consumer actors have been automatically constructed by akka-camel, when the actor was started. Although the default route construction templates, used by akka-camel internally, are sufficient for most use cases, some applications may require more specialized routes to actors. The akka-camel module provides two mechanisms for customizing routes to actors, which will be explained in this section. These are:

- Usage of *Akka Camel components* to access actors. Any Camel route can use these components to access Akka actors.
- *Intercepting route construction* to actors. This option gives you the ability to change routes that have already been added to Camel. Consumer actors have a hook into the route definition process which can be used to change the route.

Akka Camel components

Akka actors can be accessed from Camel routes using the actor Camel component. This component can be used to access any Akka actor (not only consumer actors) from Camel routes, as described in the following sections.

Access to actors

To access actors from custom Camel routes, the actor Camel component should be used. It fully supports Camel's asynchronous routing engine.

This component accepts the following endpoint URI format:

- `[<actor-path>]?<options>`

where `<actor-path>` is the `ActorPath` to the actor. The `<options>` are name-value pairs separated by `&` (i.e. `name1=value1&name2=value2&...`).

URI options

The following URI options are supported:

Name	Type	Default	Description
replyTimeout	Duration	false	The reply timeout, specified in the same way that you use the duration in akka, for instance `10 seconds` except that in the url it is handy to use a + between the amount and the unit, like for example `200+millis` See also *Consumer timeout*.
autoAck	Boolean	true	If set to true, in-only message exchanges are auto-acknowledged when the message is added to the actor's mailbox. If set to false, actors must acknowledge the receipt of the message. See also *Delivery acknowledgements*.

Here's an actor endpoint URI example containing an actor path:

```
akka://some-system/user/myconsumer?autoAck=false&replyTimeout=100+millis
```

In the following example, a custom route to an actor is created, using the actor's path. the Akka camel package contains an implicit `toActorRouteDefinition` that allows for a route to reference an `ActorRef` directly as shown in the below example, The route starts from a Jetty endpoint and ends at the target actor.

```
import akka.actor.{ Props, ActorSystem, Actor, ActorRef }
import akka.camel.{ CamelMessage, CamelExtension }
import org.apache.camel.builder.RouteBuilder
import akka.camel._
class Responder extends Actor {
  def receive = {
    case msg: CamelMessage =>
      sender() ! (msg.mapBody {
        body: String => "received %s" format body
      })
  }
}

class CustomRouteBuilder(system: ActorSystem, responder: ActorRef)
  extends RouteBuilder {
  def configure {
    from("jetty:http://localhost:8877/camel/custom").to(responder)
  }
}
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val responder = system.actorOf(Props[Responder], name = "TestResponder")
camel.context.addRoutes(new CustomRouteBuilder(system, responder))
```

When a message is received on the jetty endpoint, it is routed to the Responder actor, which in return replies back to the client of the HTTP request.

Intercepting route construction

The previous section, *Akka Camel components*, explained how to setup a route to an actor manually. It was the application's responsibility to define the route and add it to the current CamelContext. This section explains a more convenient way to define custom routes: akka-camel is still setting up the routes to consumer actors (and adds these routes to the current CamelContext) but applications can define extensions to these routes. Extensions can be defined with Camel's Java DSL or Scala DSL. For example, an extension could be a custom error handler that redelivers messages from an endpoint to an actor's bounded mailbox when the mailbox was full.

The following examples demonstrate how to extend a route to a consumer actor for handling exceptions thrown by that actor.

```
import akka.camel.Consumer

import org.apache.camel.builder.Builder
```

```
import org.apache.camel.model.RouteDefinition

class ErrorThrowingConsumer(override val endpointUri: String) extends Consumer {
  def receive = {
    case msg: CamelMessage => throw new Exception("error: %s" format msg.body)
  }
  override def onRouteDefinition = (rd) => rd.onException(classOf[Exception]).
    handled(true).transform(Builder.exceptionMessage).end

  final override def preRestart(reason: Throwable, message: Option[Any]) {
    sender() ! Failure(reason)
  }
}
```

The above ErrorThrowingConsumer sends the Failure back to the sender in preRestart because the Exception that is thrown in the actor would otherwise just crash the actor, by default the actor would be restarted, and the response would never reach the client of the Consumer.

The akka-camel module creates a RouteDefinition instance by calling from(endpointUri) on a Camel RouteBuilder (where endpointUri is the endpoint URI of the consumer actor) and passes that instance as argument to the route definition handler *). The route definition handler then extends the route and returns a ProcessorDefinition (in the above example, the ProcessorDefinition returned by the end method. See the org.apache.camel.model package for details). After executing the route definition handler, akka-camel finally calls a to(targetActorUri) on the returned ProcessorDefinition to complete the route to the consumer actor (where targetActorUri is the actor component URI as described in *Access to actors*). If the actor cannot be found, a *ActorNotRegisteredException* is thrown.

*) Before passing the RouteDefinition instance to the route definition handler, akka-camel may make some further modifications to it.

5.15.6 Examples

The Lightbend Activator tutorial named Akka Camel Samples with Scala contains 3 samples:

- Asynchronous routing and transformation - This example demonstrates how to implement consumer and producer actors that support *Asynchronous routing* with their Camel endpoints.
- Custom Camel route - Demonstrates the combined usage of a `Producer` and a `Consumer` actor as well as the inclusion of a custom Camel route.
- Quartz Scheduler Example - Showing how simple is to implement a cron-style scheduler by using the Camel Quartz component

5.15.7 Configuration

There are several configuration properties for the Camel module, please refer to the *reference configuration*.

5.15.8 Additional Resources

For an introduction to akka-camel 2, see also the Peter Gabryanczyk's talk Migrating akka-camel module to Akka 2.x.

For an introduction to akka-camel 1, see also the Appendix E - Akka and Camel (pdf) of the book Camel in Action.

Other, more advanced external articles (for version 1) are:

- Akka Consumer Actors: New Features and Best Practices
- Akka Producer Actors: New Features and Best Practices

CHAPTER

SIX

UTILITIES

6.1 Event Bus

Originally conceived as a way to send messages to groups of actors, the `EventBus` has been generalized into a set of composable traits implementing a simple interface:

```
/**
 * Attempts to register the subscriber to the specified Classifier
 * @return true if successful and false if not (because it was already
 *   subscribed to that Classifier, or otherwise)
 */
def subscribe(subscriber: Subscriber, to: Classifier): Boolean

/**
 * Attempts to deregister the subscriber from the specified Classifier
 * @return true if successful and false if not (because it wasn't subscribed
 *   to that Classifier, or otherwise)
 */
def unsubscribe(subscriber: Subscriber, from: Classifier): Boolean

/**
 * Attempts to deregister the subscriber from all Classifiers it may be subscribed␣
↪to
 */
def unsubscribe(subscriber: Subscriber): Unit

/**
 * Publishes the specified Event to this bus
 */
def publish(event: Event): Unit
```

Note: Please note that the EventBus does not preserve the sender of the published messages. If you need a reference to the original sender you have to provide it inside the message.

This mechanism is used in different places within Akka, e.g. the *Event Stream*. Implementations can make use of the specific building blocks presented below.

An event bus must define the following three abstract types:

- `Event` is the type of all events published on that bus
- `Subscriber` is the type of subscribers allowed to register on that event bus
- `Classifier` defines the classifier to be used in selecting subscribers for dispatching events

The traits below are still generic in these types, but they need to be defined for any concrete implementation.

6.1.1 Classifiers

The classifiers presented here are part of the Akka distribution, but rolling your own in case you do not find a perfect match is not difficult, check the implementation of the existing ones on github

Lookup Classification

The simplest classification is just to extract an arbitrary classifier from each event and maintaining a set of subscribers for each possible classifier. This can be compared to tuning in on a radio station. The trait `LookupClassification` is still generic in that it abstracts over how to compare subscribers and how exactly to classify.

The necessary methods to be implemented are illustrated with the following example:

```
import akka.event.EventBus
import akka.event.LookupClassification

final case class MsgEnvelope(topic: String, payload: Any)

/**
 * Publishes the payload of the MsgEnvelope when the topic of the
 * MsgEnvelope equals the String specified when subscribing.
 */
class LookupBusImpl extends EventBus with LookupClassification {
  type Event = MsgEnvelope
  type Classifier = String
  type Subscriber = ActorRef

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): Classifier = event.topic

  // will be invoked for each event for all subscribers which registered themselves
  // for the event's classifier
  override protected def publish(event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event.payload
  }

  // must define a full order over the subscribers, expressed as expected from
  // `java.lang.Comparable.compare`
  override protected def compareSubscribers(a: Subscriber, b: Subscriber): Int =
    a.compareTo(b)

  // determines the initial size of the index data structure
  // used internally (i.e. the expected number of different classifiers)
  override protected def mapSize: Int = 128

}
```

A test for this implementation may look like this:

```
val lookupBus = new LookupBusImpl
lookupBus.subscribe(testActor, "greetings")
lookupBus.publish(MsgEnvelope("time", System.currentTimeMillis()))
lookupBus.publish(MsgEnvelope("greetings", "hello"))
expectMsg("hello")
```

This classifier is efficient in case no subscribers exist for a particular event.

Subchannel Classification

If classifiers form a hierarchy and it is desired that subscription be possible not only at the leaf nodes, this classification may be just the right one. It can be compared to tuning in on (possibly multiple) radio channels by genre. This classification has been developed for the case where the classifier is just the JVM class of the event and subscribers may be interested in subscribing to all subclasses of a certain class, but it may be used with any classifier hierarchy.

The necessary methods to be implemented are illustrated with the following example:

```
import akka.util.Subclassification

class StartsWithSubclassification extends Subclassification[String] {
  override def isEqual(x: String, y: String): Boolean =
    x == y

  override def isSubclass(x: String, y: String): Boolean =
    x.startsWith(y)
}

import akka.event.SubchannelClassification

/**
 * Publishes the payload of the MsgEnvelope when the topic of the
 * MsgEnvelope starts with the String specified when subscribing.
 */
class SubchannelBusImpl extends EventBus with SubchannelClassification {
  type Event = MsgEnvelope
  type Classifier = String
  type Subscriber = ActorRef

  // Subclassification is an object providing `isEqual` and `isSubclass`
  // to be consumed by the other methods of this classifier
  override protected val subclassification: Subclassification[Classifier] =
    new StartsWithSubclassification

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): Classifier = event.topic

  // will be invoked for each event for all subscribers which registered
  // themselves for the event's classifier
  override protected def publish(event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event.payload
  }
}
```

A test for this implementation may look like this:

```
val subchannelBus = new SubchannelBusImpl
subchannelBus.subscribe(testActor, "abc")
subchannelBus.publish(MsgEnvelope("xyzabc", "x"))
subchannelBus.publish(MsgEnvelope("bcdef", "b"))
subchannelBus.publish(MsgEnvelope("abc", "c"))
expectMsg("c")
subchannelBus.publish(MsgEnvelope("abcdef", "d"))
expectMsg("d")
```

This classifier is also efficient in case no subscribers are found for an event, but it uses conventional locking to synchronize an internal classifier cache, hence it is not well-suited to use cases in which subscriptions change with very high frequency (keep in mind that "opening" a classifier by sending the first message will also have to re-check all previous subscriptions).

Scanning Classification

The previous classifier was built for multi-classifier subscriptions which are strictly hierarchical, this classifier is useful if there are overlapping classifiers which cover various parts of the event space without forming a hierarchy. It can be compared to tuning in on (possibly multiple) radio stations by geographical reachability (for old-school radio-wave transmission).

The necessary methods to be implemented are illustrated with the following example:

```
import akka.event.ScanningClassification

/**
 * Publishes String messages with length less than or equal to the length
 * specified when subscribing.
 */
class ScanningBusImpl extends EventBus with ScanningClassification {
  type Event = String
  type Classifier = Int
  type Subscriber = ActorRef

  // is needed for determining matching classifiers and storing them in an
  // ordered collection
  override protected def compareClassifiers(a: Classifier, b: Classifier): Int =
    if (a < b) -1 else if (a == b) 0 else 1

  // is needed for storing subscribers in an ordered collection
  override protected def compareSubscribers(a: Subscriber, b: Subscriber): Int =
    a.compareTo(b)

  // determines whether a given classifier shall match a given event; it is invoked
  // for each subscription for all received events, hence the name of the␣
↪classifier
  override protected def matches(classifier: Classifier, event: Event): Boolean =
    event.length <= classifier

  // will be invoked for each event for all subscribers which registered themselves
  // for a classifier matching this event
  override protected def publish(event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event
  }
}
```

A test for this implementation may look like this:

```
val scanningBus = new ScanningBusImpl
scanningBus.subscribe(testActor, 3)
scanningBus.publish("xyzabc")
scanningBus.publish("ab")
expectMsg("ab")
scanningBus.publish("abc")
expectMsg("abc")
```

This classifier takes always a time which is proportional to the number of subscriptions, independent of how many actually match.

Actor Classification

This classification was originally developed specifically for implementing *DeathWatch*: subscribers as well as classifiers are of type `ActorRef`.

This classification requires an `ActorSystem` in order to perform book-keeping operations related to the subscribers being Actors, which can terminate without first unsubscribing from the EventBus. ManagedActorClassi-

fication maintains a system Actor which takes care of unsubscribing terminated actors automatically.

The necessary methods to be implemented are illustrated with the following example:

```
import akka.event.ActorEventBus
import akka.event.ManagedActorClassification
import akka.event.ActorClassifier

final case class Notification(ref: ActorRef, id: Int)

class ActorBusImpl(val system: ActorSystem) extends ActorEventBus with␣
↪ActorClassifier with ManagedActorClassification {
  type Event = Notification

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): ActorRef = event.ref

  // determines the initial size of the index data structure
  // used internally (i.e. the expected number of different classifiers)
  override protected def mapSize: Int = 128
}
```

A test for this implementation may look like this:

```
val observer1 = TestProbe().ref
val observer2 = TestProbe().ref
val probe1 = TestProbe()
val probe2 = TestProbe()
val subscriber1 = probe1.ref
val subscriber2 = probe2.ref
val actorBus = new ActorBusImpl(system)
actorBus.subscribe(subscriber1, observer1)
actorBus.subscribe(subscriber2, observer1)
actorBus.subscribe(subscriber2, observer2)
actorBus.publish(Notification(observer1, 100))
probe1.expectMsg(Notification(observer1, 100))
probe2.expectMsg(Notification(observer1, 100))
actorBus.publish(Notification(observer2, 101))
probe2.expectMsg(Notification(observer2, 101))
probe1.expectNoMsg(500.millis)
```

This classifier is still is generic in the event type, and it is efficient for all use cases.

6.1.2 Event Stream

The event stream is the main event bus of each actor system: it is used for carrying *log messages* and *Dead Letters* and may be used by the user code for other purposes as well. It uses *Subchannel Classification* which enables registering to related sets of channels (as is used for `RemotingLifecycleEvent`). The following example demonstrates how a simple subscription works:

```
import akka.actor.{ Actor, DeadLetter, Props }

class Listener extends Actor {
  def receive = {
    case d: DeadLetter => println(d)
  }
}

val listener = system.actorOf(Props(classOf[Listener], this))
system.eventStream.subscribe(listener, classOf[DeadLetter])
```

It is also worth pointing out that thanks to the way the subchannel classification is implemented in the event stream, it is possible to subscribe to a group of events, by subscribing to their common superclass as demonstrated in the following example:

```
abstract class AllKindsOfMusic { def artist: String }
case class Jazz(artist: String) extends AllKindsOfMusic
case class Electronic(artist: String) extends AllKindsOfMusic

class Listener extends Actor {
  def receive = {
    case m: Jazz       => println(s"${self.path.name} is listening to: ${m.artist}
↪")
    case m: Electronic => println(s"${self.path.name} is listening to: ${m.artist}
↪")
  }
}

val jazzListener = system.actorOf(Props(classOf[Listener], this))
val musicListener = system.actorOf(Props(classOf[Listener], this))
system.eventStream.subscribe(jazzListener, classOf[Jazz])
system.eventStream.subscribe(musicListener, classOf[AllKindsOfMusic])

// only musicListener gets this message, since it listens to *all* kinds of music:
system.eventStream.publish(Electronic("Parov Stelar"))

// jazzListener and musicListener will be notified about Jazz:
system.eventStream.publish(Jazz("Sonny Rollins"))
```

Similarly to *Actor Classification*, `EventStream` will automatically remove subscribers when they terminate.

Note: The event stream is a *local facility*, meaning that it will *not* distribute events to other nodes in a clustered environment (unless you subscribe a Remote Actor to the stream explicitly). If you need to broadcast events in an Akka cluster, *without* knowing your recipients explicitly (i.e. obtaining their ActorRefs), you may want to look into: *Distributed Publish Subscribe in Cluster*.

Default Handlers

Upon start-up the actor system creates and subscribes actors to the event stream for logging: these are the handlers which are configured for example in `application.conf`:

```
akka {
  loggers = ["akka.event.Logging$DefaultLogger"]
}
```

The handlers listed here by fully-qualified class name will be subscribed to all log event classes with priority higher than or equal to the configured log-level and their subscriptions are kept in sync when changing the log-level at runtime:

```
system.eventStream.setLogLevel(Logging.DebugLevel)
```

This means that log events for a level which will not be logged are not typically not dispatched at all (unless manual subscriptions to the respective event class have been done)

Dead Letters

As described at *Stopping actors*, messages queued when an actor terminates or sent after its death are re-routed to the dead letter mailbox, which by default will publish the messages wrapped in `DeadLetter`. This wrapper holds the original sender, receiver and message of the envelope which was redirected.

Some internal messages (marked with the `DeadLetterSuppression` trait) will not end up as dead letters like normal messages. These are by design safe and expected to sometimes arrive at a terminated actor and since they are nothing to worry about, they are suppressed from the default dead letters logging mechanism.

However, in case you find yourself in need of debugging these kinds of low level suppressed dead letters, it's still possible to subscribe to them explicitly:

```
import akka.actor.SuppressedDeadLetter
system.eventStream.subscribe(listener, classOf[SuppressedDeadLetter])
```

or all dead letters (including the suppressed ones):

```
import akka.actor.AllDeadLetters
system.eventStream.subscribe(listener, classOf[AllDeadLetters])
```

Other Uses

The event stream is always there and ready to be used, just publish your own events (it accepts `AnyRef`) and subscribe listeners to the corresponding JVM classes.

6.2 Logging

Logging in Akka is not tied to a specific logging backend. By default log messages are printed to STDOUT, but you can plug-in a SLF4J logger or your own logger. Logging is performed asynchronously to ensure that logging has minimal performance impact. Logging generally means IO and locks, which can slow down the operations of your code if it was performed synchronously.

6.2.1 How to Log

Create a `LoggingAdapter` and use the `error`, `warning`, `info`, or `debug` methods, as illustrated in this example:

```
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  override def preStart() = {
    log.debug("Starting")
  }
  override def preRestart(reason: Throwable, message: Option[Any]) {
    log.error(reason, "Restarting due to [{}] when processing [{}]",
      reason.getMessage, message.getOrElse(""))
  }
  def receive = {
    case "test" => log.info("Received test")
    case x      => log.warning("Received unknown message: {}", x)
  }
}
```

For convenience you can mixin the `log` member into actors, instead of defining it as above.

```
class MyActor extends Actor with akka.actor.ActorLogging {
  ...
}
```

The second parameter to the `Logging` is the source of this logging channel. The source object is translated to a String according to the following rules:

- if it is an Actor or ActorRef, its path is used
- in case of a String it is used as is
- in case of a class an approximation of its simpleName
- and in all other cases a compile error occurs unless and implicit `LogSource[T]` is in scope for the type in question.

The log message may contain argument placeholders `{}`, which will be substituted if the log level is enabled. Giving more arguments as there are placeholders results in a warning being appended to the log statement (i.e. on the same line with the same severity). You may pass a Java array as the only substitution argument to have its elements be treated individually:

```
val args = Array("The", "brown", "fox", "jumps", 42)
system.log.debug("five parameters: {}, {}, {}, {}, {}", args)
```

The Java `Class` of the log source is also included in the generated `LogEvent`. In case of a simple string this is replaced with a "marker" class `akka.event.DummyClassForStringSources` in order to allow special treatment of this case, e.g. in the SLF4J event listener which will then use the string instead of the class' name for looking up the logger instance to use.

Logging of Dead Letters

By default messages sent to dead letters are logged at info level. Existence of dead letters does not necessarily indicate a problem, but it might be, and therefore they are logged by default. After a few messages this logging is turned off, to avoid flooding the logs. You can disable this logging completely or adjust how many dead letters that are logged. During system shutdown it is likely that you see dead letters, since pending messages in the actor mailboxes are sent to dead letters. You can also disable logging of dead letters during shutdown.

```
akka {
  log-dead-letters = 10
  log-dead-letters-during-shutdown = on
}
```

To customize the logging further or take other actions for dead letters you can subscribe to the *Event Stream*.

Auxiliary logging options

Akka has a couple of configuration options for very low level debugging, that makes most sense in for developers and not for operations.

You almost definitely need to have logging set to DEBUG to use any of the options below:

```
akka {
  loglevel = "DEBUG"
}
```

This config option is very good if you want to know what config settings are loaded by Akka:

```
akka {
  # Log the complete configuration at INFO level when the actor system is started.
  # This is useful when you are uncertain of what configuration is used.
  log-config-on-start = on
}
```

If you want very detailed logging of user-level messages then wrap your actors' behaviors with `akka.event.LoggingReceive` and enable the `receive` option:

```
akka {
  actor {
    debug {
      # enable function of LoggingReceive, which is to log any received message at
      # DEBUG level
      receive = on
    }
  }
}
```

If you want very detailed logging of all automatically received messages that are processed by Actors:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of all AutoReceiveMessages (Kill, PoisonPill et.c.)
      autoreceive = on
    }
  }
}
```

If you want very detailed logging of all lifecycle changes of Actors (restarts, deaths etc):

```
akka {
  actor {
    debug {
      # enable DEBUG logging of actor lifecycle changes
      lifecycle = on
    }
  }
}
```

If you want unhandled messages logged at DEBUG:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of unhandled messages
      unhandled = on
    }
  }
}
```

If you want very detailed logging of all events, transitions and timers of FSM Actors that extend LoggingFSM:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of all LoggingFSMs for events, transitions and timers
      fsm = on
    }
  }
}
```

If you want to monitor subscriptions (subscribe/unsubscribe) on the ActorSystem.eventStream:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of subscription changes on the eventStream
      event-stream = on
    }
```

```
  }
}
```

Auxiliary remote logging options

If you want to see all messages that are sent through remoting at DEBUG log level: (This is logged as they are sent by the transport layer, not by the Actor)

```
akka {
  remote {
    # If this is "on", Akka will log all outbound messages at DEBUG level,
    # if off then they are not logged
    log-sent-messages = on
  }
}
```

If you want to see all messages that are received through remoting at DEBUG log level: (This is logged as they are received by the transport layer, not by any Actor)

```
akka {
  remote {
    # If this is "on", Akka will log all inbound messages at DEBUG level,
    # if off then they are not logged
    log-received-messages = on
  }
}
```

If you want to see message types with payload size in bytes larger than a specified limit at INFO log level:

```
akka {
  remote {
    # Logging of message types with payload size in bytes larger than
    # this value. Maximum detected size per message type is logged once,
    # with an increase threshold of 10%.
    # By default this feature is turned off. Activate it by setting the property to
    # a value in bytes, such as 1000b. Note that for all messages larger than this
    # limit there will be extra performance and scalability cost.
    log-frame-size-exceeding = 1000b
  }
}
```

Also see the logging options for TestKit: *Tracing Actor Invocations*.

Translating Log Source to String and Class

The rules for translating the source object to the source string and class which are inserted into the `LogEvent` during runtime are implemented using implicit parameters and thus fully customizable: simply create your own instance of `LogSource[T]` and have it in scope when creating the logger.

```
import akka.actor.ActorSystem
import akka.event.LogSource

object MyType {
  implicit val logSource: LogSource[AnyRef] = new LogSource[AnyRef] {
    def genString(o: AnyRef): String = o.getClass.getName
    override def getClazz(o: AnyRef): Class[_] = o.getClass
  }
}
```

```
class MyType(system: ActorSystem) {
  import MyType._
  import akka.event.Logging

  val log = Logging(system, this)
}
```

This example creates a log source which mimics traditional usage of Java loggers, which are based upon the originating object's class name as log category. The override of `getClazz` is only included for demonstration purposes as it contains exactly the default behavior.

Note: You may also create the string representation up front and pass that in as the log source, but be aware that then the `Class[_]` which will be put in the `LogEvent` is `akka.event.DummyClassForStringSources`.

The SLF4J event listener treats this case specially (using the actual string to look up the logger instance to use instead of the class' name), and you might want to do this also in case you implement your own logging adapter.

Turn Off Logging

To turn off logging you can configure the log levels to be `OFF` like this.

```
akka {
  stdout-loglevel = "OFF"
  loglevel = "OFF"
}
```

The `stdout-loglevel` is only in effect during system startup and shutdown, and setting it to `OFF` as well, ensures that nothing gets logged during system startup or shutdown.

6.2.2 Loggers

Logging is performed asynchronously through an event bus. Log events are processed by an event handler actor and it will receive the log events in the same order as they were emitted.

Note: The event handler actor does not have a bounded inbox and is run on the default dispatcher. This means that logging extreme amounts of data may affect your application badly. It can be somewhat mitigated by making sure to use an async logging backend though. (See *Using the SLF4J API directly*)

You can configure which event handlers are created at system start-up and listen to logging events. That is done using the `loggers` element in the *Configuration*. Here you can also define the log level. More fine grained filtering based on the log source can be implemented in a custom `LoggingFilter`, which can be defined in the `logging-filter` configuration property.

```
akka {
  # Loggers to register at boot time (akka.event.Logging$DefaultLogger logs
  # to STDOUT)
  loggers = ["akka.event.Logging$DefaultLogger"]
  # Options: OFF, ERROR, WARNING, INFO, DEBUG
  loglevel = "DEBUG"
}
```

The default one logs to STDOUT and is registered by default. It is not intended to be used for production. There is also an *SLF4J* logger available in the 'akka-slf4j' module.

Example of creating a listener:

```
import akka.event.Logging.Debug
import akka.event.Logging.Error
import akka.event.Logging.Info
import akka.event.Logging.InitializeLogger
import akka.event.Logging.LoggerInitialized
import akka.event.Logging.Warning

class MyEventListener extends Actor {
  def receive = {
    case InitializeLogger(_)                          => sender() ! LoggerInitialized
    case Error(cause, logSource, logClass, message) => // ...
    case Warning(logSource, logClass, message)      => // ...
    case Info(logSource, logClass, message)         => // ...
    case Debug(logSource, logClass, message)        => // ...
  }
}
```

6.2.3 Logging to stdout during startup and shutdown

When the actor system is starting up and shutting down the configured `loggers` are not used. Instead log messages are printed to stdout (System.out). The default log level for this stdout logger is `WARNING` and it can be silenced completely by setting `akka.stdout-loglevel=OFF`.

6.2.4 SLF4J

Akka provides a logger for SL4FJ. This module is available in the 'akka-slf4j.jar'. It has one single dependency; the slf4j-api jar. In runtime you also need a SLF4J backend, we recommend Logback:

```
libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.1.3"
```

You need to enable the Slf4jLogger in the `loggers` element in the *Configuration*. Here you can also define the log level of the event bus. More fine grained log levels can be defined in the configuration of the SLF4J backend (e.g. logback.xml). You should also define `akka.event.slf4j.Slf4jLoggingFilter` in the `logging-filter` configuration property. It will filter the log events using the backend configuration (e.g. logback.xml) before they are published to the event bus.

Warning: If you set the `loglevel` to a higher level than "DEBUG", any DEBUG events will be filtered out already at the source and will never reach the logging backend, regardless of how the backend is configured.

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"
}
```

One gotcha is that the timestamp is attributed in the event handler, not when actually doing the logging.

The SLF4J logger selected for each log event is chosen based on the `Class[_]` of the log source specified when creating the `LoggingAdapter`, unless that was given directly as a string in which case that string is used (i.e. `LoggerFactory.getLogger(c: Class[_])` is used in the first case and `LoggerFactory.getLogger(s: String)` in the second).

Note: Beware that the actor system's name is appended to a `String` log source if the LoggingAdapter was created giving an `ActorSystem` to the factory. If this is not intended, give a `LoggingBus` instead as shown below:

```
val log = Logging(system.eventStream, "my.nice.string")
```

Using the SLF4J API directly

If you use the SLF4J API directly in your application, remember that the logging operations will block while the underlying infrastructure writes the log statements.

This can be avoided by configuring the logging implementation to use a non-blocking appender. Logback provides AsyncAppender that does this. It also contains a feature which will drop `INFO` and `DEBUG` messages if the logging load is high.

Logging Thread, Akka Source and Actor System in MDC

Since the logging is done asynchronously the thread in which the logging was performed is captured in Mapped Diagnostic Context (MDC) with attribute name `sourceThread`. With Logback the thread name is available with `%X{sourceThread}` specifier within the pattern layout configuration:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{sourceThread} - %msg%n</
↪pattern>
  </encoder>
</appender>
```

Note: It will probably be a good idea to use the `sourceThread` MDC value also in non-Akka parts of the application in order to have this property consistently available in the logs.

Another helpful facility is that Akka captures the actor's address when instantiating a logger within it, meaning that the full instance identification is available for associating log messages e.g. with members of a router. This information is available in the MDC with attribute name `akkaSource`:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{akkaSource} - %msg%n</pattern>
  </encoder>
</appender>
```

Finally, the actor system in which the logging was performed is available in the MDC with attribute name `sourceActorSystem`:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{sourceActorSystem} - %msg%n</
↪pattern>
  </encoder>
</appender>
```

For more details on what this attribute contains—also for non-actors—please see *How to Log*.

More accurate timestamps for log output in MDC

Akka's logging is asynchronous which means that the timestamp of a log entry is taken from when the underlying logger implementation is called, which can be surprising at first. If you want to more accurately output the timestamp, use the MDC attribute `akkaTimestamp`:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%X{akkaTimestamp} %-5level %logger{36} %X{akkaSource} - %msg%n</
↪pattern>
  </encoder>
</appender>
```

MDC values defined by the application

One useful feature available in Slf4j is MDC, Akka has a way for let the application specify custom values, you just need to get a specialized `LoggingAdapter`, the `DiagnosticLoggingAdapter`. In order to get it you will use the factory receiving an Actor as logSource:

```
// Within your Actor
val log: DiagnosticLoggingAdapter = Logging(this);
```

Once you have the logger, you just need to add the custom values before you log something. This way, the values will dologbe put in the SLF4J MDC right before appending the log and removed after.

Note: The cleanup (removal) should be done in the actor at the end, otherwise, next message will log with same mdc values, if it is not set to a new map. Use `log.clearMDC()`.

```
val mdc = Map("requestId" -> 1234, "visitorId" -> 5678)
log.mdc(mdc)

// Log something
log.info("Starting new request")

log.clearMDC()
```

For convenience you can mixin the `log` member into actors, instead of defining it as above. This trait also lets you override `def mdc(msg: Any): MDC` for specifying MDC values depending on current message and lets you forget about the cleanup as well, since it already does it for you.

```
import Logging.MDC

final case class Req(work: String, visitorId: Int)

class MdcActorMixin extends Actor with akka.actor.DiagnosticActorLogging {
  var reqId = 0

  override def mdc(currentMessage: Any): MDC = {
    reqId += 1
    val always = Map("requestId" -> reqId)
    val perMessage = currentMessage match {
      case r: Req => Map("visitorId" -> r.visitorId)
      case _      => Map()
    }
    always ++ perMessage
  }

  def receive: Receive = {
    case r: Req => {
      log.info(s"Starting new request: ${r.work}")
    }
  }
}
```

Now, the values will be available in the MDC, so you can use them in the layout pattern:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>
      %-5level %logger{36} [req: %X{requestId}, visitor: %X{visitorId}] - %msg%n
    </pattern>
  </encoder>
</appender>
```

Using Markers

Some logging libraries allow, in addition to MDC data, attaching so called "markers" to log statements. These are used to filter out rare and special events, for example you might want to mark logs that detect some malicious activity and mark them with a `SECURITY` tag, and in your appender configuration make these trigger emails and other notifications immediately.

Markers are available through the LoggingAdapters, when obtained via `Logging.withMarker`. The first argument passed into all log calls then should be a `akka.event.LogMarker`.

The slf4j bridge provided by akka in `akka-slf4j` will automatically pick up this marker value and make it available to SLF4J. For example you could use it like this:

```
<pattern>%date{ISO8601} [%marker][%level] [%msg]%n</pattern>
```

A more advanced (including most Akka added information) example pattern would be:

```
<pattern>%date{ISO8601} level=[%level] marker=[%marker] logger=[%logger]␣
↪akkaSource=[%X{akkaSource}] sourceActorSystem=[%X{sourceActorSystem}]␣
↪sourceThread=[%X{sourceThread}] mdc=[ticket-#%X{ticketNumber}: %X{ticketDesc}] -␣
↪msg=[%msg]%n----%n</pattern>
```

6.3 Scheduler

Sometimes the need for making things happen in the future arises, and where do you go look then? Look no further than `ActorSystem`! There you find the `scheduler` method that returns an instance of `akka.actor.Scheduler`, this instance is unique per ActorSystem and is used internally for scheduling things to happen at specific points in time.

You can schedule sending of messages to actors and execution of tasks (functions or Runnable). You will get a `Cancellable` back that you can call `cancel` on to cancel the execution of the scheduled operation.

Warning: The default implementation of `Scheduler` used by Akka is based on job buckets which are emptied according to a fixed schedule. It does not execute tasks at the exact time, but on every tick, it will run everything that is (over)due. The accuracy of the default Scheduler can be modified by the `akka.scheduler.tick-duration` configuration property.

6.3.1 Some examples

```
import akka.actor.Actor
import akka.actor.Props
import scala.concurrent.duration._

    //Use the system's dispatcher as ExecutionContext
    import system.dispatcher
```

```
    //Schedules to send the "foo"-message to the testActor after 50ms
    system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")
```

```
//Schedules a function to be executed (send a message to the testActor) after 50ms
system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}
```

```
val Tick = "tick"
class TickActor extends Actor {
  def receive = {
    case Tick => //Do something
  }
}
val tickActor = system.actorOf(Props(classOf[TickActor], this))
//Use system's dispatcher as ExecutionContext
import system.dispatcher

//This will schedule to send the Tick-message
//to the tickActor after 0ms repeating every 50ms
val cancellable =
  system.scheduler.schedule(
    0 milliseconds,
    50 milliseconds,
    tickActor,
    Tick)

//This cancels further Ticks to be sent
cancellable.cancel()
```

Warning: If you schedule functions or Runnable instances you should be extra careful to not close over unstable references. In practice this means not using `this` inside the closure in the scope of an Actor instance, not accessing `sender()` directly and not calling the methods of the Actor instance directly. If you need to schedule an invocation schedule a message to `self` instead (containing the necessary parameters) and then call the method when the message is received.

6.3.2 From `akka.actor.ActorSystem`

```
/**
 * Light-weight scheduler for running asynchronous tasks after some deadline
 * in the future. Not terribly precise but cheap.
 */
def scheduler: Scheduler
```

Warning: All scheduled task will be executed when the `ActorSystem` is terminated, i.e. the task may execute before its timeout.

6.3.3 The Scheduler interface

The actual scheduler implementation is loaded reflectively upon `ActorSystem` start-up, which means that it is possible to provide a different one using the `akka.scheduler.implementation` configuration property. The referenced class must implement the following interface:

```
/**
 * An Akka scheduler service. This one needs one special behavior: if
 * Closeable, it MUST execute all outstanding tasks upon .close() in order
 * to properly shutdown all dispatchers.
 *
 * Furthermore, this timer service MUST throw IllegalStateException if it
 * cannot schedule a task. Once scheduled, the task MUST be executed. If
 * executed upon close(), the task may execute before its timeout.
 *
 * Scheduler implementation are loaded reflectively at ActorSystem start-up
 * with the following constructor arguments:
 *  1) the system's com.typesafe.config.Config (from system.settings.config)
 *  2) a akka.event.LoggingAdapter
 *  3) a java.util.concurrent.ThreadFactory
 */
trait Scheduler {
  /**
   * Schedules a message to be sent repeatedly with an initial delay and
   * frequency. E.g. if you would like a message to be sent immediately and
   * thereafter every 500ms you would set delay=Duration.Zero and
   * interval=Duration(500, TimeUnit.MILLISECONDS)
   *
   * Java & Scala API
   */
  final def schedule(
    initialDelay: FiniteDuration,
    interval:     FiniteDuration,
    receiver:     ActorRef,
    message:      Any)(implicit
    executor: ExecutionContext,
                      sender: ActorRef = Actor.noSender): Cancellable =
    schedule(initialDelay, interval, new Runnable {
      def run = {
        receiver ! message
        if (receiver.isTerminated)
          throw new SchedulerException("timer active for terminated actor")
      }
    })

  /**
   * Schedules a function to be run repeatedly with an initial delay and a
   * frequency. E.g. if you would like the function to be run after 2 seconds
   * and thereafter every 100ms you would set delay = Duration(2, TimeUnit.SECONDS)
   * and interval = Duration(100, TimeUnit.MILLISECONDS). If the execution of
   * the function takes longer than the interval, the subsequent execution will
   * start immediately after the prior one completes (there will be no overlap
   * of the function executions). In such cases, the actual execution interval
   * will differ from the interval passed to this method.
   *
   * If the function throws an exception the repeated scheduling is aborted,
   * i.e. the function will not be invoked any more.
   *
   * Scala API
   */
  final def schedule(
    initialDelay: FiniteDuration,
    interval:     FiniteDuration)(f: ⇒ Unit)(
    implicit
    executor: ExecutionContext): Cancellable =
    schedule(initialDelay, interval, new Runnable { override def run = f })

  /**
   * Schedules a `Runnable` to be run repeatedly with an initial delay and
```

```
   * a frequency. E.g. if you would like the function to be run after 2
   * seconds and thereafter every 100ms you would set delay = Duration(2,
   * TimeUnit.SECONDS) and interval = Duration(100, TimeUnit.MILLISECONDS). If
   * the execution of the runnable takes longer than the interval, the
   * subsequent execution will start immediately after the prior one completes
   * (there will be no overlap of executions of the runnable). In such cases,
   * the actual execution interval will differ from the interval passed to this
   * method.
   *
   * If the `Runnable` throws an exception the repeated scheduling is aborted,
   * i.e. the function will not be invoked any more.
   *
   * Java API
   */
  def schedule(
    initialDelay: FiniteDuration,
    interval:     FiniteDuration,
    runnable:     Runnable)(implicit executor: ExecutionContext): Cancellable

  /**
   * Schedules a message to be sent once with a delay, i.e. a time period that has
   * to pass before the message is sent.
   *
   * Java & Scala API
   */
  final def scheduleOnce(
    delay:    FiniteDuration,
    receiver: ActorRef,
    message:  Any)(implicit
    executor: ExecutionContext,
                  sender: ActorRef = Actor.noSender): Cancellable =
    scheduleOnce(delay, new Runnable {
      override def run = receiver ! message
    })

  /**
   * Schedules a function to be run once with a delay, i.e. a time period that has
   * to pass before the function is run.
   *
   * Scala API
   */
  final def scheduleOnce(delay: FiniteDuration)(f: ⇒ Unit)(
    implicit
    executor: ExecutionContext): Cancellable =
    scheduleOnce(delay, new Runnable { override def run = f })

  /**
   * Schedules a Runnable to be run once with a delay, i.e. a time period that
   * has to pass before the runnable is executed.
   *
   * Java & Scala API
   */
  def scheduleOnce(
    delay:    FiniteDuration,
    runnable: Runnable)(implicit executor: ExecutionContext): Cancellable

  /**
   * The maximum supported task frequency of this scheduler, i.e. the inverse
   * of the minimum time interval between executions of a recurring task, in Hz.
   */
  def maxFrequency: Double

}
```

6.3.4 The Cancellable interface

Scheduling a task will result in a `Cancellable` (or throw an `IllegalStateException` if attempted after the scheduler's shutdown). This allows you to cancel something that has been scheduled for execution.

Warning: This does not abort the execution of the task, if it had already been started. Check the return value of `cancel` to detect whether the scheduled task was canceled or will (eventually) have run.

```
/**
 * Signifies something that can be cancelled
 * There is no strict guarantee that the implementation is thread-safe,
 * but it should be good practice to make it so.
 */
trait Cancellable {
  /**
   * Cancels this Cancellable and returns true if that was successful.
   * If this cancellable was (concurrently) cancelled already, then this method
   * will return false although isCancelled will return true.
   *
   * Java & Scala API
   */
  def cancel(): Boolean

  /**
   * Returns true if and only if this Cancellable has been successfully cancelled
   *
   * Java & Scala API
   */
  def isCancelled: Boolean
}
```

6.4 Duration

Durations are used throughout the Akka library, wherefore this concept is represented by a special data type, `scala.concurrent.duration.Duration`. Values of this type may represent infinite (`Duration.Inf`, `Duration.MinusInf`) or finite durations, or be `Duration.Undefined`.

6.4.1 Finite vs. Infinite

Since trying to convert an infinite duration into a concrete time unit like seconds will throw an exception, there are different types available for distinguishing the two kinds at compile time:

- `FiniteDuration` is guaranteed to be finite, calling `toNanos` and friends is safe
- `Duration` can be finite or infinite, so this type should only be used when finite-ness does not matter; this is a supertype of `FiniteDuration`

6.4.2 Scala

In Scala durations are constructable using a mini-DSL and support all expected arithmetic operations:

```
import scala.concurrent.duration._

val fivesec = 5.seconds
val threemillis = 3.millis
val diff = fivesec - threemillis
assert(diff < fivesec)
val fourmillis = threemillis * 4 / 3 // you cannot write it the other way around
val n = threemillis / (1 millisecond)
```

Note: You may leave out the dot if the expression is clearly delimited (e.g. within parentheses or in an argument list), but it is recommended to use it if the time unit is the last token on a line, otherwise semi-colon inference might go wrong, depending on what starts the next line.

6.4.3 Java

Java provides less syntactic sugar, so you have to spell out the operations as method calls instead:

```
import scala.concurrent.duration.Duration;
import scala.concurrent.duration.Deadline;
```

```
final Duration fivesec = Duration.create(5, "seconds");
final Duration threemillis = Duration.create("3 millis");
final Duration diff = fivesec.minus(threemillis);
assert diff.lt(fivesec);
assert Duration.Zero().lt(Duration.Inf());
```

6.4.4 Deadline

Durations have a brother named `Deadline`, which is a class holding a representation of an absolute point in time, and support deriving a duration from this by calculating the difference between now and the deadline. This is useful when you want to keep one overall deadline without having to take care of the book-keeping wrt. the passing of time yourself:

```
val deadline = 10.seconds.fromNow
// do something
val rest = deadline.timeLeft
```

In Java you create these from durations:

```
final Deadline deadline = Duration.create(10, "seconds").fromNow();
final Duration rest = deadline.timeLeft();
```

6.5 Circuit Breaker

6.5.1 Why are they used?

A circuit breaker is used to provide stability and prevent cascading failures in distributed systems. These should be used in conjunction with judicious timeouts at the interfaces between remote systems to prevent the failure of a single component from bringing down all components.

As an example, we have a web application interacting with a remote third party web service. Let's say the third party has oversold their capacity and their database melts down under load. Assume that the database fails in such a way that it takes a very long time to hand back an error to the third party web service. This in turn makes calls

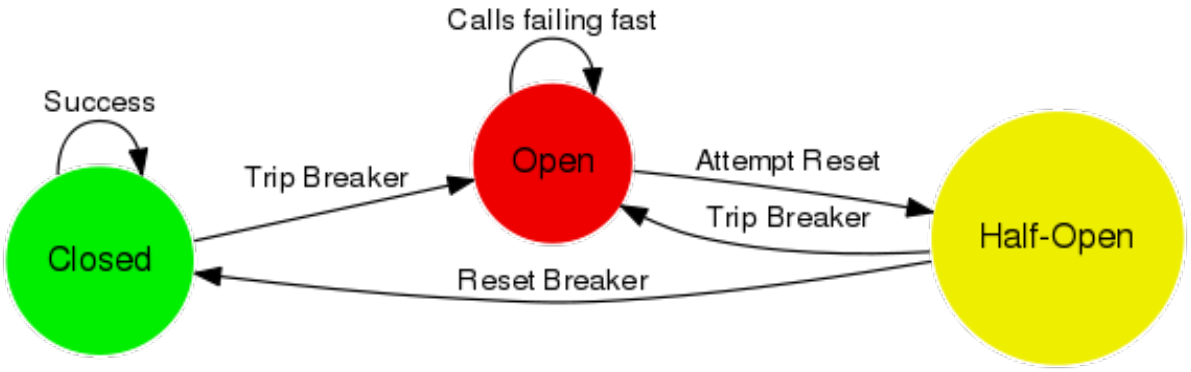
fail after a long period of time. Back to our web application, the users have noticed that their form submissions take much longer seeming to hang. Well the users do what they know to do which is use the refresh button, adding more requests to their already running requests. This eventually causes the failure of the web application due to resource exhaustion. This will affect all users, even those who are not using functionality dependent on this third party web service.

Introducing circuit breakers on the web service call would cause the requests to begin to fail-fast, letting the user know that something is wrong and that they need not refresh their request. This also confines the failure behavior to only those users that are using functionality dependent on the third party, other users are no longer affected as there is no resource exhaustion. Circuit breakers can also allow savvy developers to mark portions of the site that use the functionality unavailable, or perhaps show some cached content as appropriate while the breaker is open.

The Akka library provides an implementation of a circuit breaker called `akka.pattern.CircuitBreaker` which has the behavior described below.

6.5.2 What do they do?

- **During normal operation, a circuit breaker is in the *Closed* state:**
  - Exceptions or calls exceeding the configured *callTimeout* increment a failure counter
  - Successes reset the failure count to zero
  - When the failure counter reaches a *maxFailures* count, the breaker is tripped into *Open* state
- **While in *Open* state:**
  - All calls fail-fast with a `CircuitBreakerOpenException`
  - After the configured *resetTimeout*, the circuit breaker enters a *Half-Open* state
- **In *Half-Open* state:**
  - The first call attempted is allowed through without failing fast
  - All other calls fail-fast with an exception just as in *Open* state
  - If the first call succeeds, the breaker is reset back to *Closed* state and the *resetTimeout* is reset
  - If the first call fails, the breaker is tripped again into the *Open* state (as for exponential backoff circuit breaker, the *resetTimeout* is multiplied by the exponential backoff factor)
- **State transition listeners:**
  - Callbacks can be provided for every state entry via *onOpen*, *onClose*, and *onHalfOpen*
  - These are executed in the `ExecutionContext` provided.



6.5.3 Examples

Initialization

Here's how a `CircuitBreaker` would be configured for:

- 5 maximum failures
- a call timeout of 10 seconds
- a reset timeout of 1 minute

Scala

```
import scala.concurrent.duration._
import akka.pattern.CircuitBreaker
import akka.pattern.pipe
import akka.actor.{ Actor, ActorLogging, ActorRef }

import scala.concurrent.Future

class DangerousActor extends Actor with ActorLogging {
  import context.dispatcher

  val breaker =
    new CircuitBreaker(
      context.system.scheduler,
      maxFailures = 5,
      callTimeout = 10.seconds,
      resetTimeout = 1.minute).onOpen(notifyMeOnOpen())

  def notifyMeOnOpen(): Unit =
    log.warning("My CircuitBreaker is now open, and will not close for one minute")
```

Java

```
import akka.actor.UntypedActor;
import scala.concurrent.Future;
import akka.event.LoggingAdapter;
import scala.concurrent.duration.Duration;
import akka.pattern.CircuitBreaker;
import akka.event.Logging;

import static akka.pattern.Patterns.pipe;
import static akka.dispatch.Futures.future;

import java.util.concurrent.Callable;

public class DangerousJavaActor extends UntypedActor {

  private final CircuitBreaker breaker;
  private final LoggingAdapter log = Logging.getLogger(getContext().system(),␣
↪this);

  public DangerousJavaActor() {
    this.breaker = new CircuitBreaker(
      getContext().dispatcher(), getContext().system().scheduler(),
      5, Duration.create(10, "s"), Duration.create(1, "m"))
      .onOpen(new Runnable() {
        public void run() {
          notifyMeOnOpen();
        }
```

```
      });
  }

  public void notifyMeOnOpen() {
    log.warning("My CircuitBreaker is now open, and will not close for one minute
↪");
  }
```

Call Protection

Here's how the `CircuitBreaker` would be used to protect an asynchronous call as well as a synchronous one:

Scala

```
def dangerousCall: String = "This really isn't that dangerous of a call after all"

def receive = {
  case "is my middle name" =>
    breaker.withCircuitBreaker(Future(dangerousCall)) pipeTo sender()
  case "block for me" =>
    sender() ! breaker.withSyncCircuitBreaker(dangerousCall)
}
```

Java

```
public String dangerousCall() {
  return "This really isn't that dangerous of a call after all";
}

@Override
public void onReceive(Object message) {
  if (message instanceof String) {
    String m = (String) message;
    if ("is my middle name".equals(m)) {
      pipe(
        breaker.callWithCircuitBreaker(() ->
          future(() -> dangerousCall(), getContext().dispatcher())
        ), getContext().dispatcher()
      ).to(getSender());
    }
    if ("block for me".equals(m)) {
      getSender().tell(breaker
        .callWithSyncCircuitBreaker(
          () -> dangerousCall()), getSelf());
    }
  }
}
```

Note: Using the `CircuitBreaker` companion object's *apply* or *create* methods will return a `CircuitBreaker` where callbacks are executed in the caller's thread. This can be useful if the asynchronous `Future` behavior is unnecessary, for example invoking a synchronous-only API.

Tell Pattern

The above `Call Protection` pattern works well when the return from a remote call is wrapped in a `Future`. However, when a remote call sends back a message or timeout to the caller `Actor`, the `Call Protection` pattern is awkward. CircuitBreaker doesn't support it natively at the moment, so you need to use below low-level power-user APIs, `succeed` and `fail` methods, as well as `isClose`, `isOpen`, `isHalfOpen`.

Note: The below examples doesn't make a remote call when the state is *HalfOpen*. Using the power-user APIs, it is your responsibility to judge when to make remote calls in *HalfOpen*.

Scala

```
import akka.actor.ReceiveTimeout

def receive = {
  case "call" if breaker.isClosed => {
    recipient ! "message"
  }
  case "response" => {
    breaker.succeed()
  }
  case err: Throwable => {
    breaker.fail()
  }
  case ReceiveTimeout => {
    breaker.fail()
  }
}
```

Java

```
@Override
public void onReceive(Object payload) {
  if ( "call".equals(payload) && breaker.isClosed() ) {
    target.tell("message", getSelf());
  } else if ( "response".equals(payload) ) {
    breaker.succeed();
  } else if ( payload instanceof Throwable ) {
    breaker.fail();
  } else if ( payload instanceof ReceiveTimeout ) {
    breaker.fail();
  }
}
```

6.6 Akka Extensions

If you want to add features to Akka, there is a very elegant, but powerful mechanism for doing so. It's called Akka Extensions and is comprised of 2 basic components: an `Extension` and an `ExtensionId`.

Extensions will only be loaded once per `ActorSystem`, which will be managed by Akka. You can choose to have your Extension loaded on-demand or at `ActorSystem` creation time through the Akka configuration. Details on how to make that happens are below, in the "Loading from Configuration" section.

Warning: Since an extension is a way to hook into Akka itself, the implementor of the extension needs to ensure the thread safety of his/her extension.

6.6.1 Building an Extension

So let's create a sample extension that just lets us count the number of times something has happened.

First, we define what our `Extension` should do:

```
import akka.actor.Extension

class CountExtensionImpl extends Extension {
  //Since this Extension is a shared instance
  // per ActorSystem we need to be threadsafe
  private val counter = new AtomicLong(0)

  //This is the operation this Extension provides
  def increment() = counter.incrementAndGet()
}
```

Then we need to create an `ExtensionId` for our extension so we can grab a hold of it.

```
import akka.actor.ActorSystem
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem

object CountExtension
  extends ExtensionId[CountExtensionImpl]
  with ExtensionIdProvider {
  //The lookup method is required by ExtensionIdProvider,
  // so we return ourselves here, this allows us
  // to configure our extension to be loaded when
  // the ActorSystem starts up
  override def lookup = CountExtension

  //This method will be called by Akka
  // to instantiate our Extension
  override def createExtension(system: ExtendedActorSystem) = new␣
↪CountExtensionImpl

  /**
   * Java API: retrieve the Count extension for the given system.
   */
  override def get(system: ActorSystem): CountExtensionImpl = super.get(system)
}
```

Wicked! Now all we need to do is to actually use it:

```
CountExtension(system).increment
```

Or from inside of an Akka Actor:

```
class MyActor extends Actor {
  def receive = {
    case someMessage =>
      CountExtension(context.system).increment()
  }
}
```

You can also hide extension behind traits:

```
trait Counting { self: Actor =>
  def increment() = CountExtension(context.system).increment()
}
class MyCounterActor extends Actor with Counting {
  def receive = {
    case someMessage => increment()
  }
}
```

That's all there is to it!

6.6.2 Loading from Configuration

To be able to load extensions from your Akka configuration you must add FQCNs of implementations of either `ExtensionId` or `ExtensionIdProvider` in the `akka.extensions` section of the config you provide to your `ActorSystem`.

```
akka {
  extensions = ["docs.extension.CountExtension"]
}
```

6.6.3 Applicability

The sky is the limit! By the way, did you know that Akka's `Typed Actors`, `Serialization` and other features are implemented as Akka Extensions?

Application specific settings

The *Configuration* can be used for application specific settings. A good practice is to place those settings in an Extension.

Sample configuration:

```
myapp {
  db {
    uri = "mongodb://example1.com:27017,example2.com:27017"
  }
  circuit-breaker {
    timeout = 30 seconds
  }
}
```

The `Extension`:

```
import akka.actor.ActorSystem
import akka.actor.Extension
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem
import scala.concurrent.duration.Duration
import com.typesafe.config.Config
import java.util.concurrent.TimeUnit

class SettingsImpl(config: Config) extends Extension {
  val DbUri: String = config.getString("myapp.db.uri")
  val CircuitBreakerTimeout: Duration =
    Duration(
```

```
      config.getMilliseconds("myapp.circuit-breaker.timeout"),
      TimeUnit.MILLISECONDS)
}
object Settings extends ExtensionId[SettingsImpl] with ExtensionIdProvider {

  override def lookup = Settings

  override def createExtension(system: ExtendedActorSystem) =
    new SettingsImpl(system.settings.config)

  /**
   * Java API: retrieve the Settings extension for the given system.
   */
  override def get(system: ActorSystem): SettingsImpl = super.get(system)
}
```

Use it:

```
class MyActor extends Actor {
  val settings = Settings(context.system)
  val connection = connect(settings.DbUri, settings.CircuitBreakerTimeout)
```

6.6.4 Library extensions

A third part library may register it's extension for auto-loading on actor system startup by appending it to `akka.library-extensions` in its `reference.conf`.

```
akka.library-extensions += "docs.extension.ExampleExtension"
```

As there is no way to selectively remove such extensions, it should be used with care and only when there is no case where the user would ever want it disabled or have specific support for disabling such sub-features. One example where this could be important is in tests.

Warning: The‘‘akka.library-extensions‘‘ must never be assigned (`= ["Extension"]`) instead of appending as this will break the library-extension mechanism and make behavior depend on class path ordering.

CHAPTER

SEVEN

STREAMS

7.1 Introduction

7.1.1 Motivation

The way we consume services from the Internet today includes many instances of streaming data, both downloading from a service as well as uploading to it or peer-to-peer data transfers. Regarding data as a stream of elements instead of in its entirety is very useful because it matches the way computers send and receive them (for example via TCP), but it is often also a necessity because data sets frequently become too large to be handled as a whole. We spread computations or analyses over large clusters and call it "big data", where the whole principle of processing them is by feeding those data sequentially—as a stream—through some CPUs.

Actors can be seen as dealing with streams as well: they send and receive series of messages in order to transfer knowledge (or data) from one place to another. We have found it tedious and error-prone to implement all the proper measures in order to achieve stable streaming between actors, since in addition to sending and receiving we also need to take care to not overflow any buffers or mailboxes in the process. Another pitfall is that Actor messages can be lost and must be retransmitted in that case lest the stream have holes on the receiving side. When dealing with streams of elements of a fixed given type, Actors also do not currently offer good static guarantees that no wiring errors are made: type-safety could be improved in this case.

For these reasons we decided to bundle up a solution to these problems as an Akka Streams API. The purpose is to offer an intuitive and safe way to formulate stream processing setups such that we can then execute them efficiently and with bounded resource usage—no more OutOfMemoryErrors. In order to achieve this our streams need to be able to limit the buffering that they employ, they need to be able to slow down producers if the consumers cannot keep up. This feature is called back-pressure and is at the core of the Reactive Streams initiative of which Akka is a founding member. For you this means that the hard problem of propagating and reacting to back-pressure has been incorporated in the design of Akka Streams already, so you have one less thing to worry about; it also means that Akka Streams interoperate seamlessly with all other Reactive Streams implementations (where Reactive Streams interfaces define the interoperability SPI while implementations like Akka Streams offer a nice user API).

Relationship with Reactive Streams

The Akka Streams API is completely decoupled from the Reactive Streams interfaces. While Akka Streams focus on the formulation of transformations on data streams the scope of Reactive Streams is just to define a common mechanism of how to move data across an asynchronous boundary without losses, buffering or resource exhaustion.

The relationship between these two is that the Akka Streams API is geared towards end-users while the Akka Streams implementation uses the Reactive Streams interfaces internally to pass data between the different processing stages. For this reason you will not find any resemblance between the Reactive Streams interfaces and the Akka Streams API. This is in line with the expectations of the Reactive Streams project, whose primary purpose is to define interfaces such that different streaming implementation can interoperate; it is not the purpose of Reactive Streams to describe an end-user API.

7.1.2 How to read these docs

Stream processing is a different paradigm to the Actor Model or to Future composition, therefore it may take some careful study of this subject until you feel familiar with the tools and techniques. The documentation is here to help and for best results we recommend the following approach:

- Read the *Quick Start Guide* to get a feel for how streams look like and what they can do.
- The top-down learners may want to peruse the *Design Principles behind Akka Streams* at this point.
- The bottom-up learners may feel more at home rummaging through the *Streams Cookbook*.
- For a complete overview of the built-in processing stages you can look at the table in *Overview of built-in stages and their semantics*
- The other sections can be read sequentially or as needed during the previous steps, each digging deeper into specific topics.

7.2 Quick Start Guide

A stream usually begins at a source, so this is also how we start an Akka Stream. Before we create one, we import the full complement of streaming tools:

```
import akka.stream._
import akka.stream.scaladsl._
```

If you want to execute the code samples while you read through the quick start guide, you will also need the following imports:

```
import akka.{ NotUsed, Done }
import akka.actor.ActorSystem
import akka.util.ByteString
import scala.concurrent._
import scala.concurrent.duration._
import java.nio.file.Paths
```

Now we will start with a rather simple source, emitting the integers 1 to 100:

```
val source: Source[Int, NotUsed] = Source(1 to 100)
```

The `Source` type is parameterized with two types: the first one is the type of element that this source emits and the second one may signal that running the source produces some auxiliary value (e.g. a network source may provide information about the bound port or the peer's address). Where no auxiliary information is produced, the type `akka.NotUsed` is used—and a simple range of integers surely falls into this category.

Having created this source means that we have a description of how to emit the first 100 natural numbers, but this source is not yet active. In order to get those numbers out we have to run it:

```
source.runForeach(i => println(i))(materializer)
```

This line will complement the source with a consumer function—in this example we simply print out the numbers to the console—and pass this little stream setup to an Actor that runs it. This activation is signaled by having "run" be part of the method name; there are other methods that run Akka Streams, and they all follow this pattern.

You may wonder where the Actor gets created that runs the stream, and you are probably also asking yourself what this `materializer` means. In order to get this value we first need to create an Actor system:

```
implicit val system = ActorSystem("QuickStart")
implicit val materializer = ActorMaterializer()
```

There are other ways to create a materializer, e.g. from an `ActorContext` when using streams from within Actors. The `Materializer` is a factory for stream execution engines, it is the thing that makes streams run—you

don't need to worry about any of the details just now apart from that you need one for calling any of the `run` methods on a `Source`. The materializer is picked up implicitly if it is omitted from the `run` method call arguments, which we will do in the following.

The nice thing about Akka Streams is that the `Source` is just a description of what you want to run, and like an architect's blueprint it can be reused, incorporated into a larger design. We may choose to transform the source of integers and write it to a file instead:

```
val factorials = source.scan(BigInt(1))((acc, next) => acc * next)

val result: Future[IOResult] =
  factorials
    .map(num => ByteString(s"$num\n"))
    .runWith(FileIO.toPath(Paths.get("factorials.txt")))
```

First we use the `scan` combinator to run a computation over the whole stream: starting with the number 1 (`BigInt(1)`) we multiple by each of the incoming numbers, one after the other; the scan operation emits the initial value and then every calculation result. This yields the series of factorial numbers which we stash away as a `Source` for later reuse—it is important to keep in mind that nothing is actually computed yet, this is just a description of what we want to have computed once we run the stream. Then we convert the resulting series of numbers into a stream of `ByteString` objects describing lines in a text file. This stream is then run by attaching a file as the receiver of the data. In the terminology of Akka Streams this is called a `Sink`. `IOResult` is a type that IO operations return in Akka Streams in order to tell you how many bytes or elements were consumed and whether the stream terminated normally or exceptionally.

7.2.1 Reusable Pieces

One of the nice parts of Akka Streams—and something that other stream libraries do not offer—is that not only sources can be reused like blueprints, all other elements can be as well. We can take the file-writing `Sink`, prepend the processing steps necessary to get the `ByteString` elements from incoming strings and package that up as a reusable piece as well. Since the language for writing these streams always flows from left to right (just like plain English), we need a starting point that is like a source but with an "open" input. In Akka Streams this is called a `Flow`:

```
def lineSink(filename: String): Sink[String, Future[IOResult]] =
  Flow[String]
    .map(s => ByteString(s + "\n"))
    .toMat(FileIO.toPath(Paths.get(filename)))(Keep.right)
```

Starting from a flow of strings we convert each to `ByteString` and then feed to the already known file-writing `Sink`. The resulting blueprint is a `Sink[String,Future[IOResult]]`, which means that it accepts strings as its input and when materialized it will create auxiliary information of type `Future[IOResult]` (when chaining operations on a `Source` or `Flow` the type of the auxiliary information—called the "materialized value"—is given by the leftmost starting point; since we want to retain what the `FileIO.toPath` sink has to offer, we need to say `Keep.right`).

We can use the new and shiny `Sink` we just created by attaching it to our `factorials` source—after a small adaptation to turn the numbers into strings:

```
factorials.map(_.toString).runWith(lineSink("factorial2.txt"))
```

7.2.2 Time-Based Processing

Before we start looking at a more involved example we explore the streaming nature of what Akka Streams can do. Starting from the `factorials` source we transform the stream by zipping it together with another stream, represented by a `Source` that emits the number 0 to 100: the first number emitted by the `factorials` source is the factorial of zero, the second is the factorial of one, and so on. We combine these two by forming strings like `"3! = 6"`.

```
val done: Future[Done] =
  factorials
    .zipWith(Source(0 to 100))((num, idx) => s"$idx! = $num")
    .throttle(1, 1.second, 1, ThrottleMode.shaping)
    .runForeach(println)
```

All operations so far have been time-independent and could have been performed in the same fashion on strict collections of elements. The next line demonstrates that we are in fact dealing with streams that can flow at a certain speed: we use the `throttle` combinator to slow down the stream to 1 element per second (the second `1` in the argument list is the maximum size of a burst that we want to allow—passing `1` means that the first element gets through immediately and the second then has to wait for one second and so on).

If you run this program you will see one line printed per second. One aspect that is not immediately visible deserves mention, though: if you try and set the streams to produce a billion numbers each then you will notice that your JVM does not crash with an OutOfMemoryError, even though you will also notice that running the streams happens in the background, asynchronously (this is the reason for the auxiliary information to be provided as a `Future`). The secret that makes this work is that Akka Streams implicitly implement pervasive flow control, all combinators respect back-pressure. This allows the throttle combinator to signal to all its upstream sources of data that it can only accept elements at a certain rate—when the incoming rate is higher than one per second the throttle combinator will assert *back-pressure* upstream.

This is basically all there is to Akka Streams in a nutshell—glossing over the fact that there are dozens of sources and sinks and many more stream transformation combinators to choose from, see also *Overview of built-in stages and their semantics*.

7.3 Reactive Tweets

A typical use case for stream processing is consuming a live stream of data that we want to extract or aggregate some other data from. In this example we'll consider consuming a stream of tweets and extracting information concerning Akka from them.

We will also consider the problem inherent to all non-blocking streaming solutions: *"What if the subscriber is too slow to consume the live stream of data?"*. Traditionally the solution is often to buffer the elements, but this can—and usually will—cause eventual buffer overflows and instability of such systems. Instead Akka Streams depend on internal backpressure signals that allow to control what should happen in such scenarios.

Here's the data model we'll be working with throughout the quickstart examples:

```
final case class Author(handle: String)

final case class Hashtag(name: String)

final case class Tweet(author: Author, timestamp: Long, body: String) {
  def hashtags: Set[Hashtag] =
    body.split(" ").collect { case t if t.startsWith("#") => Hashtag(t) }.toSet
}

val akkaTag = Hashtag("#akka")
```

Note: If you would like to get an overview of the used vocabulary first instead of diving head-first into an actual example you can have a look at the *Core concepts* and *Defining and running streams* sections of the docs, and then come back to this quickstart to see it all pieced together into a simple example application.

7.3.1 Transforming and consuming simple streams

The example application we will be looking at is a simple Twitter feed stream from which we'll want to extract certain information, like for example finding all twitter handles of users who tweet about `#akka`.

In order to prepare our environment by creating an `ActorSystem` and `ActorMaterializer`, which will be responsible for materializing and running the streams we are about to create:

```
implicit val system = ActorSystem("reactive-tweets")
implicit val materializer = ActorMaterializer()
```

The `ActorMaterializer` can optionally take `ActorMaterializerSettings` which can be used to define materialization properties, such as default buffer sizes (see also *Buffers for asynchronous stages*), the dispatcher to be used by the pipeline etc. These can be overridden with `withAttributes` on `Flow`, `Source`, `Sink` and `Graph`.

Let's assume we have a stream of tweets readily available. In Akka this is expressed as a `Source[Out,M]`:

```
val tweets: Source[Tweet, NotUsed]
```

Streams always start flowing from a `Source[Out,M1]` then can continue through `Flow[In,Out,M2]` elements or more advanced graph elements to finally be consumed by a `Sink[In,M3]` (ignore the type parameters `M1`, `M2` and `M3` for now, they are not relevant to the types of the elements produced/consumed by these classes – they are "materialized types", which we'll talk about *below*).

The operations should look familiar to anyone who has used the Scala Collections library, however they operate on streams and not collections of data (which is a very important distinction, as some operations only make sense in streaming and vice versa):

```
val authors: Source[Author, NotUsed] =
  tweets
    .filter(_.hashtags.contains(akkaTag))
    .map(_.author)
```

Finally in order to *materialize* and run the stream computation we need to attach the Flow to a `Sink` that will get the Flow running. The simplest way to do this is to call `runWith(sink)` on a `Source`. For convenience a number of common Sinks are predefined and collected as methods on the `Sink` companion object. For now let's simply print each author:

```
authors.runWith(Sink.foreach(println))
```

or by using the shorthand version (which are defined only for the most popular Sinks such as `Sink.fold` and `Sink.foreach`):

```
authors.runForeach(println)
```

Materializing and running a stream always requires a `Materializer` to be in implicit scope (or passed in explicitly, like this: `.run(materializer)`).

The complete snippet looks like this:

```
implicit val system = ActorSystem("reactive-tweets")
implicit val materializer = ActorMaterializer()

val authors: Source[Author, NotUsed] =
  tweets
    .filter(_.hashtags.contains(akkaTag))
    .map(_.author)

authors.runWith(Sink.foreach(println))
```

7.3.2 Flattening sequences in streams

In the previous section we were working on 1:1 relationships of elements which is the most common case, but sometimes we might want to map from one element to a number of elements and receive a "flattened" stream,

similarly like `flatMap` works on Scala Collections. In order to get a flattened stream of hashtags from our stream of tweets we can use the `mapConcat` combinator:

```
val hashtags: Source[Hashtag, NotUsed] = tweets.mapConcat(_.hashtags.toList)
```

Note: The name `flatMap` was consciously avoided due to its proximity with for-comprehensions and monadic composition. It is problematic for two reasons: first, flattening by concatenation is often undesirable in bounded stream processing due to the risk of deadlock (with merge being the preferred strategy), and second, the monad laws would not hold for our implementation of flatMap (due to the liveness issues).

Please note that the `mapConcat` requires the supplied function to return a strict collection (`f:Out=>immutable.Seq[T]`), whereas `flatMap` would have to operate on streams all the way through.

7.3.3 Broadcasting a stream

Now let's say we want to persist all hashtags, as well as all author names from this one live stream. For example we'd like to write all author handles into one file, and all hashtags into another file on disk. This means we have to split the source stream into two streams which will handle the writing to these different files.

Elements that can be used to form such "fan-out" (or "fan-in") structures are referred to as "junctions" in Akka Streams. One of these that we'll be using in this example is called `Broadcast`, and it simply emits elements from its input port to all of its output ports.

Akka Streams intentionally separate the linear stream structures (Flows) from the non-linear, branching ones (Graphs) in order to offer the most convenient API for both of these cases. Graphs can express arbitrarily complex stream setups at the expense of not reading as familiarly as collection transformations.

Graphs are constructed using `GraphDSL` like this:

```
val writeAuthors: Sink[Author, Unit] = ???
val writeHashtags: Sink[Hashtag, Unit] = ???
val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val bcast = b.add(Broadcast[Tweet](2))
  tweets ~> bcast.in
  bcast.out(0) ~> Flow[Tweet].map(_.author) ~> writeAuthors
  bcast.out(1) ~> Flow[Tweet].mapConcat(_.hashtags.toList) ~> writeHashtags
  ClosedShape
})
g.run()
```

As you can see, inside the `GraphDSL` we use an implicit graph builder `b` to mutably construct the graph using the `~>` "edge operator" (also read as "connect" or "via" or "to"). The operator is provided implicitly by importing `GraphDSL.Implicits._`.

`GraphDSL.create` returns a `Graph`, in this example a `Graph[ClosedShape,Unit]` where `ClosedShape` means that it is *a fully connected graph* or "closed" - there are no unconnected inputs or outputs. Since it is closed it is possible to transform the graph into a `RunnableGraph` using `RunnableGraph.fromGraph`. The runnable graph can then be `run()` to materialize a stream out of it.

Both `Graph` and `RunnableGraph` are *immutable, thread-safe, and freely shareable*.

A graph can also have one of several other shapes, with one or more unconnected ports. Having unconnected ports expresses a graph that is a *partial graph*. Concepts around composing and nesting graphs in large structures are explained in detail in *Modularity, Composition and Hierarchy*. It is also possible to wrap complex computation graphs as Flows, Sinks or Sources, which will be explained in detail in *Constructing Sources, Sinks and Flows from Partial Graphs*.

7.3.4 Back-pressure in action

One of the main advantages of Akka Streams is that they *always* propagate back-pressure information from stream Sinks (Subscribers) to their Sources (Publishers). It is not an optional feature, and is enabled at all times. To learn more about the back-pressure protocol used by Akka Streams and all other Reactive Streams compatible implementations read *Back-pressure explained*.

A typical problem applications (not using Akka Streams) like this often face is that they are unable to process the incoming data fast enough, either temporarily or by design, and will start buffering incoming data until there's no more space to buffer, resulting in either `OutOfMemoryError` s or other severe degradations of service responsiveness. With Akka Streams buffering can and must be handled explicitly. For example, if we are only interested in the "*most recent tweets, with a buffer of 10 elements*" this can be expressed using the `buffer` element:

```
tweets
  .buffer(10, OverflowStrategy.dropHead)
  .map(slowComputation)
  .runWith(Sink.ignore)
```

The `buffer` element takes an explicit and required `OverflowStrategy`, which defines how the buffer should react when it receives another element while it is full. Strategies provided include dropping the oldest element (`dropHead`), dropping the entire buffer, signalling errors etc. Be sure to pick and choose the strategy that fits your use case best.

7.3.5 Materialized values

So far we've been only processing data using Flows and consuming it into some kind of external Sink - be it by printing values or storing them in some external system. However sometimes we may be interested in some value that can be obtained from the materialized processing pipeline. For example, we want to know how many tweets we have processed. While this question is not as obvious to give an answer to in case of an infinite stream of tweets (one way to answer this question in a streaming setting would be to create a stream of counts described as "*up until now*, we've processed N tweets"), but in general it is possible to deal with finite streams and come up with a nice result such as a total count of elements.

First, let's write such an element counter using `Sink.fold` and see how the types look like:

```
val count: Flow[Tweet, Int, NotUsed] = Flow[Tweet].map(_ => 1)

val sumSink: Sink[Int, Future[Int]] = Sink.fold[Int, Int](0)(_ + _)

val counterGraph: RunnableGraph[Future[Int]] =
  tweets
    .via(count)
    .toMat(sumSink)(Keep.right)

val sum: Future[Int] = counterGraph.run()

sum.foreach(c => println(s"Total tweets processed: $c"))
```

First we prepare a reusable `Flow` that will change each incoming tweet into an integer of value `1`. We'll use this in order to combine those with a `Sink.fold` that will sum all `Int` elements of the stream and make its result available as a `Future[Int]`. Next we connect the `tweets` stream to `count` with `via`. Finally we connect the Flow to the previously prepared Sink using `toMat`.

Remember those mysterious `Mat` type parameters on `Source[+Out,+Mat]`, `Flow[-In,+Out,+Mat]` and `Sink[-In,+Mat]`? They represent the type of values these processing parts return when materialized. When you chain these together, you can explicitly combine their materialized values. In our example we used the `Keep.right` predefined function, which tells the implementation to only care about the materialized type of the stage currently appended to the right. The materialized type of `sumSink` is `Future[Int]` and because of using `Keep.right`, the resulting `RunnableGraph` has also a type parameter of `Future[Int]`.

This step does *not* yet materialize the processing pipeline, it merely prepares the description of the Flow, which is now connected to a Sink, and therefore can be `run()`, as indicated by its type: `RunnableGraph[Future[Int]]`. Next we call `run()` which uses the implicit `ActorMaterializer` to materialize and run the Flow. The value returned by calling `run()` on a `RunnableGraph[T]` is of type `T`. In our case this type is `Future[Int]` which, when completed, will contain the total length of our `tweets` stream. In case of the stream failing, this future would complete with a Failure.

A `RunnableGraph` may be reused and materialized multiple times, because it is just the "blueprint" of the stream. This means that if we materialize a stream, for example one that consumes a live stream of tweets within a minute, the materialized values for those two materializations will be different, as illustrated by this example:

```
val sumSink = Sink.fold[Int, Int](0)(_ + _)
val counterRunnableGraph: RunnableGraph[Future[Int]] =
  tweetsInMinuteFromNow
    .filter(_.hashtags contains akkaTag)
    .map(t => 1)
    .toMat(sumSink)(Keep.right)

// materialize the stream once in the morning
val morningTweetsCount: Future[Int] = counterRunnableGraph.run()
// and once in the evening, reusing the flow
val eveningTweetsCount: Future[Int] = counterRunnableGraph.run()
```

Many elements in Akka Streams provide materialized values which can be used for obtaining either results of computation or steering these elements which will be discussed in detail in *Stream Materialization*. Summing up this section, now we know what happens behind the scenes when we run this one-liner, which is equivalent to the multi line version above:

```
val sum: Future[Int] = tweets.map(t => 1).runWith(sumSink)
```

Note: `runWith()` is a convenience method that automatically ignores the materialized value of any other stages except those appended by the `runWith()` itself. In the above example it translates to using `Keep.right` as the combiner for materialized values.

7.4 Design Principles behind Akka Streams

It took quite a while until we were reasonably happy with the look and feel of the API and the architecture of the implementation, and while being guided by intuition the design phase was very much exploratory research. This section details the findings and codifies them into a set of principles that have emerged during the process.

Note: As detailed in the introduction keep in mind that the Akka Streams API is completely decoupled from the Reactive Streams interfaces which are just an implementation detail for how to pass stream data between individual processing stages.

7.4.1 What shall users of Akka Streams expect?

Akka is built upon a conscious decision to offer APIs that are minimal and consistent—as opposed to easy or intuitive. The credo is that we favor explicitness over magic, and if we provide a feature then it must work always, no exceptions. Another way to say this is that we minimize the number of rules a user has to learn instead of trying to keep the rules close to what we think users might expect.

From this follows that the principles implemented by Akka Streams are:

- all features are explicit in the API, no magic
- supreme compositionality: combined pieces retain the function of each part

- exhaustive model of the domain of distributed bounded stream processing

This means that we provide all the tools necessary to express any stream processing topology, that we model all the essential aspects of this domain (back-pressure, buffering, transformations, failure recovery, etc.) and that whatever the user builds is reusable in a larger context.

Akka Streams does not send dropped stream elements to the dead letter office

One important consequence of offering only features that can be relied upon is the restriction that Akka Streams cannot ensure that all objects sent through a processing topology will be processed. Elements can be dropped for a number of reasons:

- plain user code can consume one element in a *map(...)* stage and produce an entirely different one as its result
- common stream operators drop elements intentionally, e.g. take/drop/filter/conflate/buffer/...
- stream failure will tear down the stream without waiting for processing to finish, all elements that are in flight will be discarded
- stream cancellation will propagate upstream (e.g. from a *take* operator) leading to upstream processing steps being terminated without having processed all of their inputs

This means that sending JVM objects into a stream that need to be cleaned up will require the user to ensure that this happens outside of the Akka Streams facilities (e.g. by cleaning them up after a timeout or when their results are observed on the stream output, or by using other means like finalizers etc.).

Resulting Implementation Constraints

Compositionality entails reusability of partial stream topologies, which led us to the lifted approach of describing data flows as (partial) graphs that can act as composite sources, flows (a.k.a. pipes) and sinks of data. These building blocks shall then be freely shareable, with the ability to combine them freely to form larger graphs. The representation of these pieces must therefore be an immutable blueprint that is materialized in an explicit step in order to start the stream processing. The resulting stream processing engine is then also immutable in the sense of having a fixed topology that is prescribed by the blueprint. Dynamic networks need to be modeled by explicitly using the Reactive Streams interfaces for plugging different engines together.

The process of materialization will often create specific objects that are useful to interact with the processing engine once it is running, for example for shutting it down or for extracting metrics. This means that the materialization function produces a result termed the *materialized value of a graph*.

7.4.2 Interoperation with other Reactive Streams implementations

Akka Streams fully implement the Reactive Streams specification and interoperate with all other conformant implementations. We chose to completely separate the Reactive Streams interfaces from the user-level API because we regard them to be an SPI that is not targeted at endusers. In order to obtain a `Publisher` or `Subscriber` from an Akka Stream topology, a corresponding `Sink.asPublisher` or `Source.asSubscriber` element must be used.

All stream Processors produced by the default materialization of Akka Streams are restricted to having a single Subscriber, additional Subscribers will be rejected. The reason for this is that the stream topologies described using our DSL never require fan-out behavior from the Publisher sides of the elements, all fan-out is done using explicit elements like `Broadcast[T]`.

This means that `Sink.asPublisher(true)` (for enabling fan-out support) must be used where broadcast behavior is needed for interoperation with other Reactive Streams implementations.

7.4.3 What shall users of streaming libraries expect?

We expect libraries to be built on top of Akka Streams, in fact Akka HTTP is one such example that lives within the Akka project itself. In order to allow users to profit from the principles that are described for Akka Streams above, the following rules are established:

- libraries shall provide their users with reusable pieces, i.e. expose factories that return graphs, allowing full compositionality
- libraries may optionally and additionally provide facilities that consume and materialize graphs

The reasoning behind the first rule is that compositionality would be destroyed if different libraries only accepted graphs and expected to materialize them: using two of these together would be impossible because materialization can only happen once. As a consequence, the functionality of a library must be expressed such that materialization can be done by the user, outside of the library's control.

The second rule allows a library to additionally provide nice sugar for the common case, an example of which is the Akka HTTP API that provides a `handleWith` method for convenient materialization.

Note: One important consequence of this is that a reusable flow description cannot be bound to "live" resources, any connection to or allocation of such resources must be deferred until materialization time. Examples of "live" resources are already existing TCP connections, a multicast Publisher, etc.; a TickSource does not fall into this category if its timer is created only upon materialization (as is the case for our implementation).

Exceptions from this need to be well-justified and carefully documented.

Resulting Implementation Constraints

Akka Streams must enable a library to express any stream processing utility in terms of immutable blueprints. The most common building blocks are

- Source: something with exactly one output stream
- Sink: something with exactly one input stream
- Flow: something with exactly one input and one output stream
- BidiFlow: something with exactly two input streams and two output streams that conceptually behave like two Flows of opposite direction
- Graph: a packaged stream processing topology that exposes a certain set of input and output ports, characterized by an object of type `Shape`.

Note: A source that emits a stream of streams is still just a normal Source, the kind of elements that are produced does not play a role in the static stream topology that is being expressed.

7.4.4 The difference between Error and Failure

The starting point for this discussion is the definition given by the Reactive Manifesto. Translated to streams this means that an error is accessible within the stream as a normal data element, while a failure means that the stream itself has failed and is collapsing. In concrete terms, on the Reactive Streams interface level data elements (including errors) are signaled via `onNext` while failures raise the `onError` signal.

Note: Unfortunately the method name for signaling *failure* to a Subscriber is called `onError` for historical reasons. Always keep in mind that the Reactive Streams interfaces (Publisher/Subscription/Subscriber) are modeling the low-level infrastructure for passing streams between execution units, and errors on this level are precisely the failures that we are talking about on the higher level that is modeled by Akka Streams.

There is only limited support for treating `onError` in Akka Streams compared to the operators that are available for the transformation of data elements, which is intentional in the spirit of the previous paragraph. Since `onError` signals that the stream is collapsing, its ordering semantics are not the same as for stream completion: transformation stages of any kind will just collapse with the stream, possibly still holding elements in implicit or explicit buffers. This means that data elements emitted before a failure can still be lost if the `onError` overtakes them.

The ability for failures to propagate faster than data elements is essential for tearing down streams that are back-pressured—especially since back-pressure can be the failure mode (e.g. by tripping upstream buffers which then abort because they cannot do anything else; or if a dead-lock occurred).

The semantics of stream recovery

A recovery element (i.e. any transformation that absorbs an `onError` signal and turns that into possibly more data elements followed normal stream completion) acts as a bulkhead that confines a stream collapse to a given region of the stream topology. Within the collapsed region buffered elements may be lost, but the outside is not affected by the failure.

This works in the same fashion as a `try–catch` expression: it marks a region in which exceptions are caught, but the exact amount of code that was skipped within this region in case of a failure might not be known precisely—the placement of statements matters.

7.5 Basics and working with Flows

7.5.1 Core concepts

Akka Streams is a library to process and transfer a sequence of elements using bounded buffer space. This latter property is what we refer to as *boundedness* and it is the defining feature of Akka Streams. Translated to everyday terms it is possible to express a chain (or as we see later, graphs) of processing entities, each executing independently (and possibly concurrently) from the others while only buffering a limited number of elements at any given time. This property of bounded buffers is one of the differences from the actor model, where each actor usually has an unbounded, or a bounded, but dropping mailbox. Akka Stream processing entities have bounded "mailboxes" that do not drop.

Before we move on, let's define some basic terminology which will be used throughout the entire documentation:

Stream An active process that involves moving and transforming data.

Element An element is the processing unit of streams. All operations transform and transfer elements from upstream to downstream. Buffer sizes are always expressed as number of elements independently form the actual size of the elements.

Back-pressure A means of flow-control, a way for consumers of data to notify a producer about their current availability, effectively slowing down the upstream producer to match their consumption speeds. In the context of Akka Streams back-pressure is always understood as *non-blocking* and *asynchronous*.

Non-Blocking Means that a certain operation does not hinder the progress of the calling thread, even if it takes long time to finish the requested operation.

Graph A description of a stream processing topology, defining the pathways through which elements shall flow when the stream is running.

Processing Stage The common name for all building blocks that build up a Graph. Examples of a processing stage would be operations like `map()`, `filter()`, custom `GraphStage` s and graph junctions like `Merge` or `Broadcast`. For the full list of built-in processing stages see *Overview of built-in stages and their semantics*

When we talk about *asynchronous, non-blocking backpressure* we mean that the processing stages available in Akka Streams will not use blocking calls but asynchronous message passing to exchange messages between each other, and they will use asynchronous means to slow down a fast producer, without blocking its thread. This is a

thread-pool friendly design, since entities that need to wait (a fast producer waiting on a slow consumer) will not block the thread but can hand it back for further use to an underlying thread-pool.

7.5.2 Defining and running streams

Linear processing pipelines can be expressed in Akka Streams using the following core abstractions:

Source A processing stage with *exactly one output*, emitting data elements whenever downstream processing stages are ready to receive them.

Sink A processing stage with *exactly one input*, requesting and accepting data elements possibly slowing down the upstream producer of elements

Flow A processing stage which has *exactly one input and output*, which connects its up- and downstreams by transforming the data elements flowing through it.

RunnableGraph A Flow that has both ends "attached" to a Source and Sink respectively, and is ready to be `run()`.

It is possible to attach a `Flow` to a `Source` resulting in a composite source, and it is also possible to prepend a `Flow` to a `Sink` to get a new sink. After a stream is properly terminated by having both a source and a sink, it will be represented by the `RunnableGraph` type, indicating that it is ready to be executed.

It is important to remember that even after constructing the `RunnableGraph` by connecting all the source, sink and different processing stages, no data will flow through it until it is materialized. Materialization is the process of allocating all resources needed to run the computation described by a Graph (in Akka Streams this will often involve starting up Actors). Thanks to Flows being simply a description of the processing pipeline they are *immutable, thread-safe, and freely shareable*, which means that it is for example safe to share and send them between actors, to have one actor prepare the work, and then have it be materialized at some completely different place in the code.

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)

// connect the Source to the Sink, obtaining a RunnableGraph
val runnable: RunnableGraph[Future[Int]] = source.toMat(sink)(Keep.right)

// materialize the flow and get the value of the FoldSink
val sum: Future[Int] = runnable.run()
```

After running (materializing) the `RunnableGraph[T]` we get back the materialized value of type T. Every stream processing stage can produce a materialized value, and it is the responsibility of the user to combine them to a new type. In the above example we used `toMat` to indicate that we want to transform the materialized value of the source and sink, and we used the convenience function `Keep.right` to say that we are only interested in the materialized value of the sink. In our example the `FoldSink` materializes a value of type `Future` which will represent the result of the folding process over the stream. In general, a stream can expose multiple materialized values, but it is quite common to be interested in only the value of the Source or the Sink in the stream. For this reason there is a convenience method called `runWith()` available for `Sink`, `Source` or `Flow` requiring, respectively, a supplied `Source` (in order to run a `Sink`), a `Sink` (in order to run a `Source`) or both a `Source` and a `Sink` (in order to run a `Flow`, since it has neither attached yet).

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)

// materialize the flow, getting the Sinks materialized value
val sum: Future[Int] = source.runWith(sink)
```

It is worth pointing out that since processing stages are *immutable*, connecting them returns a new processing stage, instead of modifying the existing instance, so while constructing long flows, remember to assign the new value to a variable or run it:

```
val source = Source(1 to 10)
source.map(_ => 0) // has no effect on source, since it's immutable
source.runWith(Sink.fold(0)(_ + _)) // 55

val zeroes = source.map(_ => 0) // returns new Source[Int], with `map()` appended
zeroes.runWith(Sink.fold(0)(_ + _)) // 0
```

Note: By default Akka Streams elements support **exactly one** downstream processing stage. Making fan-out (supporting multiple downstream processing stages) an explicit opt-in feature allows default stream elements to be less complex and more efficient. Also it allows for greater flexibility on *how exactly* to handle the multicast scenarios, by providing named fan-out elements such as broadcast (signals all down-stream elements) or balance (signals one of available down-stream elements).

In the above example we used the `runWith` method, which both materializes the stream and returns the materialized value of the given sink or source.

Since a stream can be materialized multiple times, the materialized value will also be calculated anew for each such materialization, usually leading to different values being returned each time. In the example below we create two running materialized instance of the stream that we described in the `runnable` variable, and both materializations give us a different `Future` from the map even though we used the same `sink` to refer to the future:

```
// connect the Source to the Sink, obtaining a RunnableGraph
val sink = Sink.fold[Int, Int](0)(_ + _)
val runnable: RunnableGraph[Future[Int]] =
  Source(1 to 10).toMat(sink)(Keep.right)

// get the materialized value of the FoldSink
val sum1: Future[Int] = runnable.run()
val sum2: Future[Int] = runnable.run()

// sum1 and sum2 are different Futures!
```

Defining sources, sinks and flows

The objects `Source` and `Sink` define various ways to create sources and sinks of elements. The following examples show some of the most useful constructs (refer to the API documentation for more details):

```
// Create a source from an Iterable
Source(List(1, 2, 3))

// Create a source from a Future
Source.fromFuture(Future.successful("Hello Streams!"))

// Create a source from a single element
Source.single("only one element")

// an empty source
Source.empty

// Sink that folds over the stream and returns a Future
// of the final result as its materialized value
Sink.fold[Int, Int](0)(_ + _)

// Sink that returns a Future as its materialized value,
// containing the first element of the stream
Sink.head

// A Sink that consumes a stream without doing anything with the elements
```

```
Sink.ignore

// A Sink that executes a side-effecting call for every element of the stream
Sink.foreach[String](println(_))
```

There are various ways to wire up different parts of a stream, the following examples show some of the available options:

```
// Explicitly creating and wiring up a Source, Sink and Flow
Source(1 to 6).via(Flow[Int].map(_ * 2)).to(Sink.foreach(println(_)))

// Starting from a Source
val source = Source(1 to 6).map(_ * 2)
source.to(Sink.foreach(println(_)))

// Starting from a Sink
val sink: Sink[Int, NotUsed] = Flow[Int].map(_ * 2).to(Sink.foreach(println(_)))
Source(1 to 6).to(sink)

// Broadcast to a sink inline
val otherSink: Sink[Int, NotUsed] =
  Flow[Int].alsoTo(Sink.foreach(println(_))).to(Sink.ignore)
Source(1 to 6).to(otherSink)
```

Illegal stream elements

In accordance to the Reactive Streams specification (Rule 2.13) Akka Streams do not allow `null` to be passed through the stream as an element. In case you want to model the concept of absence of a value we recommend using `scala.Option` or `scala.util.Either`.

7.5.3 Back-pressure explained

Akka Streams implement an asynchronous non-blocking back-pressure protocol standardised by the Reactive Streams specification, which Akka is a founding member of.

The user of the library does not have to write any explicit back-pressure handling code — it is built in and dealt with automatically by all of the provided Akka Streams processing stages. It is possible however to add explicit buffer stages with overflow strategies that can influence the behaviour of the stream. This is especially important in complex processing graphs which may even contain loops (which *must* be treated with very special care, as explained in *Graph cycles, liveness and deadlocks*).

The back pressure protocol is defined in terms of the number of elements a downstream `Subscriber` is able to receive and buffer, referred to as `demand`. The source of data, referred to as `Publisher` in Reactive Streams terminology and implemented as `Source` in Akka Streams, guarantees that it will never emit more elements than the received total demand for any given `Subscriber`.

Note: The Reactive Streams specification defines its protocol in terms of `Publisher` and `Subscriber`. These types are **not** meant to be user facing API, instead they serve as the low level building blocks for different Reactive Streams implementations.

Akka Streams implements these concepts as `Source`, `Flow` (referred to as `Processor` in Reactive Streams) and `Sink` without exposing the Reactive Streams interfaces directly. If you need to integrate with other Reactive Stream libraries read *Integrating with Reactive Streams*.

The mode in which Reactive Streams back-pressure works can be colloquially described as "dynamic push / pull mode", since it will switch between push and pull based back-pressure models depending on the downstream being able to cope with the upstream production rate or not.

To illustrate this further let us consider both problem situations and how the back-pressure protocol handles them:

Slow Publisher, fast Subscriber

This is the happy case of course – we do not need to slow down the Publisher in this case. However signalling rates are rarely constant and could change at any point in time, suddenly ending up in a situation where the Subscriber is now slower than the Publisher. In order to safeguard from these situations, the back-pressure protocol must still be enabled during such situations, however we do not want to pay a high penalty for this safety net being enabled.

The Reactive Streams protocol solves this by asynchronously signalling from the Subscriber to the Publisher `Request(n:Int)` signals. The protocol guarantees that the Publisher will never signal *more* elements than the signalled demand. Since the Subscriber however is currently faster, it will be signalling these Request messages at a higher rate (and possibly also batching together the demand - requesting multiple elements in one Request signal). This means that the Publisher should not ever have to wait (be back-pressured) with publishing its incoming elements.

As we can see, in this scenario we effectively operate in so called push-mode since the Publisher can continue producing elements as fast as it can, since the pending demand will be recovered just-in-time while it is emitting elements.

Fast Publisher, slow Subscriber

This is the case when back-pressuring the `Publisher` is required, because the `Subscriber` is not able to cope with the rate at which its upstream would like to emit data elements.

Since the `Publisher` is not allowed to signal more elements than the pending demand signalled by the `Subscriber`, it will have to abide to this back-pressure by applying one of the below strategies:

- not generate elements, if it is able to control their production rate,
- try buffering the elements in a *bounded* manner until more demand is signalled,
- drop elements until more demand is signalled,
- tear down the stream if unable to apply any of the above strategies.

As we can see, this scenario effectively means that the `Subscriber` will *pull* the elements from the Publisher – this mode of operation is referred to as pull-based back-pressure.

7.5.4 Stream Materialization

When constructing flows and graphs in Akka Streams think of them as preparing a blueprint, an execution plan. Stream materialization is the process of taking a stream description (the graph) and allocating all the necessary resources it needs in order to run. In the case of Akka Streams this often means starting up Actors which power the processing, but is not restricted to that—it could also mean opening files or socket connections etc.—depending on what the stream needs.

Materialization is triggered at so called "terminal operations". Most notably this includes the various forms of the `run()` and `runWith()` methods defined on `Source` and `Flow` elements as well as a small number of special syntactic sugars for running with well-known sinks, such as `runForeach(el => ...)` (being an alias to `runWith(Sink.foreach(el => ...))`.

Materialization is currently performed synchronously on the materializing thread. The actual stream processing is handled by actors started up during the streams materialization, which will be running on the thread pools they have been configured to run on - which defaults to the dispatcher set in `MaterializationSettings` while constructing the `ActorMaterializer`.

Note: Reusing *instances* of linear computation stages (Source, Sink, Flow) inside composite Graphs is legal, yet will materialize that stage multiple times.

Operator Fusion

Akka Streams 2.0 contains an initial version of stream operator fusion support. This means that the processing steps of a flow or stream graph can be executed within the same Actor and has three consequences:

- starting up a stream may take longer than before due to executing the fusion algorithm
- passing elements from one processing stage to the next is a lot faster between fused stages due to avoiding the asynchronous messaging overhead
- fused stream processing stages do no longer run in parallel to each other, meaning that only up to one CPU core is used for each fused part

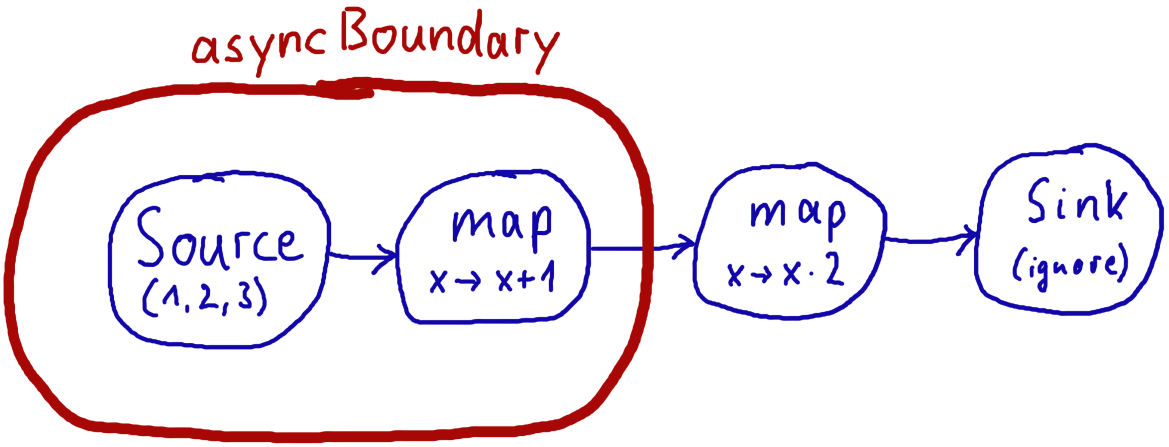
The first point can be countered by pre-fusing and then reusing a stream blueprint as sketched below:

```
import akka.stream.Fusing

val flow = Flow[Int].map(_ * 2).filter(_ > 500)
val fused = Fusing.aggressive(flow)

Source.fromIterator { () => Iterator from 0 }
  .via(fused)
  .take(1000)
```

In order to balance the effects of the second and third bullet points you will have to insert asynchronous boundaries manually into your flows and graphs by way of adding `Attributes.asyncBoundary` using the method `async` on `Source`, `Sink` and `Flow` to pieces that shall communicate with the rest of the graph in an asynchronous fashion.

```
Source(List(1, 2, 3))
  .map(_ + 1).async
  .map(_ * 2)
  .to(Sink.ignore)
```

In this example we create two regions within the flow which will be executed in one Actor each—assuming that adding and multiplying integers is an extremely costly operation this will lead to a performance gain since two CPUs can work on the tasks in parallel. It is important to note that asynchronous boundaries are not singular places within a flow where elements are passed asynchronously (as in other streaming libraries), but instead attributes always work by adding information to the flow graph that has been constructed up to this point:



This means that everything that is inside the red bubble will be executed by one actor and everything outside of it by another. This scheme can be applied successively, always having one such boundary enclose the previous ones plus all processing stages that have been added since them.

Warning: Without fusing (i.e. up to version 2.0-M2) each stream processing stage had an implicit input buffer that holds a few elements for efficiency reasons. If your flow graphs contain cycles then these buffers may have been crucial in order to avoid deadlocks. With fusing these implicit buffers are no longer there, data elements are passed without buffering between fused stages. In those cases where buffering is needed in order to allow the stream to run at all, you will have to insert explicit buffers with the `.buffer()` combinator—typically a buffer of size 2 is enough to allow a feedback loop to function.

The new fusing behavior can be disabled by setting the configuration parameter `akka.stream.materializer.auto-fusing=off`. In that case you can still manually fuse those graphs which shall run on less Actors. With the exception of the `SslTlsStage` and the `groupBy` operator all built-in processing stages can be fused.

Combining materialized values

Since every processing stage in Akka Streams can provide a materialized value after being materialized, it is necessary to somehow express how these values should be composed to a final value when we plug these stages together. For this, many combinator methods have variants that take an additional argument, a function, that will be used to combine the resulting values. Some examples of using these combiners are illustrated in the example below.

```
// An source that can be signalled explicitly from the outside
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]

// A flow that internally throttles elements to 1/second, and returns a Cancellable
// which can be used to shut down the stream
val flow: Flow[Int, Int, Cancellable] = throttler

// A sink that returns the first element of a stream in the returned Future
val sink: Sink[Int, Future[Int]] = Sink.head[Int]

// By default, the materialized value of the leftmost stage is preserved
val r1: RunnableGraph[Promise[Option[Int]]] = source.via(flow).to(sink)

// Simple selection of materialized values by using Keep.right
val r2: RunnableGraph[Cancellable] = source.viaMat(flow)(Keep.right).to(sink)
val r3: RunnableGraph[Future[Int]] = source.via(flow).toMat(sink)(Keep.right)

// Using runWith will always give the materialized values of the stages added
// by runWith() itself
val r4: Future[Int] = source.via(flow).runWith(sink)
val r5: Promise[Option[Int]] = flow.to(sink).runWith(source)
val r6: (Promise[Option[Int]], Future[Int]) = flow.runWith(source, sink)

// Using more complex combinations
val r7: RunnableGraph[(Promise[Option[Int]], Cancellable)] =
  source.viaMat(flow)(Keep.both).to(sink)

val r8: RunnableGraph[(Promise[Option[Int]], Future[Int])] =
  source.via(flow).toMat(sink)(Keep.both)

val r9: RunnableGraph[((Promise[Option[Int]], Cancellable), Future[Int])] =
  source.viaMat(flow)(Keep.both).toMat(sink)(Keep.both)

val r10: RunnableGraph[(Cancellable, Future[Int])] =
  source.viaMat(flow)(Keep.right).toMat(sink)(Keep.both)
```

```
// It is also possible to map over the materialized values. In r9 we had a
// doubly nested pair, but we want to flatten it out
val r11: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
  r9.mapMaterializedValue {
    case ((promise, cancellable), future) =>
      (promise, cancellable, future)
  }

// Now we can use pattern matching to get the resulting materialized values
val (promise, cancellable, future) = r11.run()

// Type inference works as expected
promise.success(None)
cancellable.cancel()
future.map(_ + 3)

// The result of r11 can be also achieved by using the Graph API
val r12: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
  RunnableGraph.fromGraph(GraphDSL.create(source, flow, sink)((_, _, _)) {␣
↪implicit builder => (src, f, dst) =>
    import GraphDSL.Implicits._
    src ~> f ~> dst
    ClosedShape
  })
```

Note: In Graphs it is possible to access the materialized value from inside the stream processing graph. For details see *Accessing the materialized value inside the Graph*.

7.5.5 Stream ordering

In Akka Streams almost all computation stages *preserve input order* of elements. This means that if inputs `{IA1,IA2,...,IAn}` "cause" outputs `{OA1,OA2,...,OAk}` and inputs `{IB1,IB2,...,IBm}` "cause" outputs `{OB1,OB2,...,OBl}` and all of `IAi` happened before all `IBi` then `OAi` happens before `OBi`.

This property is even uphold by async operations such as `mapAsync`, however an unordered version exists called `mapAsyncUnordered` which does not preserve this ordering.

However, in the case of Junctions which handle multiple input streams (e.g. `Merge`) the output order is, in general, *not defined* for elements arriving on different input ports. That is a merge-like operation may emit `Ai` before emitting `Bi`, and it is up to its internal logic to decide the order of emitted elements. Specialized elements such as `Zip` however *do guarantee* their outputs order, as each output element depends on all upstream elements having been signalled already – thus the ordering in the case of zipping is defined by this property.

If you find yourself in need of fine grained control over order of emitted elements in fan-in scenarios consider using `MergePreferred` or `GraphStage` – which gives you full control over how the merge is performed.

7.6 Working with Graphs

In Akka Streams computation graphs are not expressed using a fluent DSL like linear computations are, instead they are written in a more graph-resembling DSL which aims to make translating graph drawings (e.g. from notes taken from design discussions, or illustrations in protocol specifications) to and from code simpler. In this section we'll dive into the multiple ways of constructing and re-using graphs, as well as explain common pitfalls and how to avoid them.

Graphs are needed whenever you want to perform any kind of fan-in ("multiple inputs") or fan-out ("multiple outputs") operations. Considering linear Flows to be like roads, we can picture graph operations as junctions:

multiple flows being connected at a single point. Some graph operations which are common enough and fit the linear style of Flows, such as `concat` (which concatenates two streams, such that the second one is consumed after the first one has completed), may have shorthand methods defined on `Flow` or `Source` themselves, however you should keep in mind that those are also implemented as graph junctions.

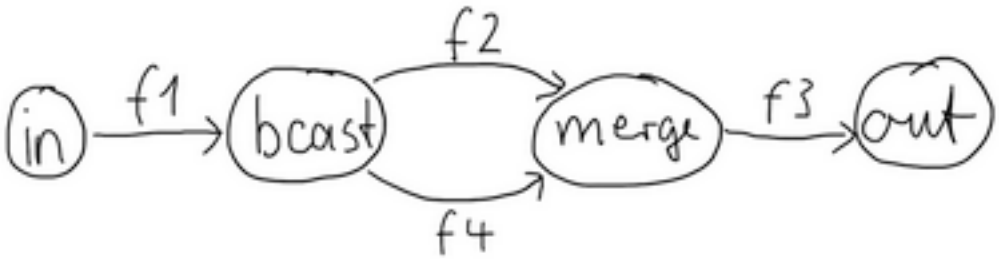
7.6.1 Constructing Graphs

Graphs are built from simple Flows which serve as the linear connections within the graphs as well as junctions which serve as fan-in and fan-out points for Flows. Thanks to the junctions having meaningful types based on their behaviour and making them explicit elements these elements should be rather straightforward to use.

Akka Streams currently provide these junctions (for a detailed list see *Overview of built-in stages and their semantics*):

- **Fan-out**
- `Broadcast[T]` – *(1 input, N outputs)* given an input element emits to each output
- `Balance[T]` – *(1 input, N outputs)* given an input element emits to one of its output ports
- `UnzipWith[In,A,B,...]` – *(1 input, N outputs)* takes a function of 1 input that given a value for each input emits N output elements (where N <= 20)
- `UnZip[A,B]` – *(1 input, 2 outputs)* splits a stream of `(A,B)` tuples into two streams, one of type `A` and one of type `B`
- **Fan-in**
- `Merge[In]` – *(N inputs , 1 output)* picks randomly from inputs pushing them one by one to its output
- `MergePreferred[In]` – like `Merge` but if elements are available on `preferred` port, it picks from it, otherwise randomly from `others`
- `ZipWith[A,B,...,Out]` – *(N inputs, 1 output)* which takes a function of N inputs that given a value for each input emits 1 output element
- `Zip[A,B]` – *(2 inputs, 1 output)* is a `ZipWith` specialised to zipping input streams of `A` and `B` into an `(A,B)` tuple stream
- `Concat[A]` – *(2 inputs, 1 output)* concatenates two streams (first consume one, then the second one)

One of the goals of the GraphDSL DSL is to look similar to how one would draw a graph on a whiteboard, so that it is simple to translate a design from whiteboard to code and be able to relate those two. Let's illustrate this by translating the below hand drawn graph into Akka Streams:



Such graph is simple to translate to the Graph DSL since each linear element corresponds to a `Flow`, and each circle corresponds to either a `Junction` or a `Source` or `Sink` if it is beginning or ending a `Flow`. Junctions must always be created with defined type parameters, as otherwise the `Nothing` type will be inferred.

```
val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder: GraphDSL.
↪Builder[NotUsed] =>
  import GraphDSL.Implicits._
  val in = Source(1 to 10)
  val out = Sink.ignore
```

```
  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))

  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

  in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
  bcast ~> f4 ~> merge
  ClosedShape
})
```

Note: Junction *reference equality* defines *graph node equality* (i.e. the same merge *instance* used in a GraphDSL refers to the same location in the resulting graph).

Notice the `import GraphDSL.Implicits._` which brings into scope the `~>` operator (read as "edge", "via" or "to") and its inverted counterpart `<~` (for noting down flows in the opposite direction where appropriate).

By looking at the snippets above, it should be apparent that the `GraphDSL.Builder` object is *mutable*. It is used (implicitly) by the `~>` operator, also making it a mutable operation as well. The reason for this design choice is to enable simpler creation of complex graphs, which may even contain cycles. Once the GraphDSL has been constructed though, the `GraphDSL` instance *is immutable, thread-safe, and freely shareable*. The same is true of all graph pieces—sources, sinks, and flows—once they are constructed. This means that you can safely re-use one given Flow or junction in multiple places in a processing graph.

We have seen examples of such re-use already above: the merge and broadcast junctions were imported into the graph using `builder.add(...)`, an operation that will make a copy of the blueprint that is passed to it and return the inlets and outlets of the resulting copy so that they can be wired up. Another alternative is to pass existing graphs—of any shape—into the factory method that produces a new graph. The difference between these approaches is that importing using `builder.add(...)` ignores the materialized value of the imported graph while importing via the factory method allows its inclusion; for more details see *Stream Materialization*.

In the example below we prepare a graph that consists of two parallel streams, in which we re-use the same instance of `Flow`, yet it will properly be materialized as two connections between the corresponding Sources and Sinks:

```
val topHeadSink = Sink.head[Int]
val bottomHeadSink = Sink.head[Int]
val sharedDoubler = Flow[Int].map(_ * 2)

RunnableGraph.fromGraph(GraphDSL.create(topHeadSink, bottomHeadSink)((_, _)) {␣
↪implicit builder =>
  (topHS, bottomHS) =>
  import GraphDSL.Implicits._
  val broadcast = builder.add(Broadcast[Int](2))
  Source.single(1) ~> broadcast.in

  broadcast.out(0) ~> sharedDoubler ~> topHS.in
  broadcast.out(1) ~> sharedDoubler ~> bottomHS.in
  ClosedShape
})
```

7.6.2 Constructing and combining Partial Graphs

Sometimes it is not possible (or needed) to construct the entire computation graph in one place, but instead construct all of its different phases in different places and in the end connect them all into a complete graph and run it.

This can be achieved by returning a different `Shape` than `ClosedShape`, for example `FlowShape(in,out)`, from the function given to `GraphDSL.create`. See *Predefined shapes*) for a list of such predefined shapes.

Making a `Graph` a `RunnableGraph` requires all ports to be connected, and if they are not it will throw an exception at construction time, which helps to avoid simple wiring errors while working with graphs. A partial graph however allows you to return the set of yet to be connected ports from the code block that performs the internal wiring.

Let's imagine we want to provide users with a specialized element that given 3 inputs will pick the greatest int value of each zipped triple. We'll want to expose 3 input ports (unconnected sources) and one output port (unconnected sink).

```
val pickMaxOfThree = GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val zip1 = b.add(ZipWith[Int, Int, Int](math.max _))
  val zip2 = b.add(ZipWith[Int, Int, Int](math.max _))
  zip1.out ~> zip2.in0

  UniformFanInShape(zip2.out, zip1.in0, zip1.in1, zip2.in1)
}

val resultSink = Sink.head[Int]

val g = RunnableGraph.fromGraph(GraphDSL.create(resultSink) { implicit b => sink =>
  import GraphDSL.Implicits._

  // importing the partial graph will return its shape (inlets & outlets)
  val pm3 = b.add(pickMaxOfThree)

  Source.single(1) ~> pm3.in(0)
  Source.single(2) ~> pm3.in(1)
  Source.single(3) ~> pm3.in(2)
  pm3.out ~> sink.in
  ClosedShape
})

val max: Future[Int] = g.run()
Await.result(max, 300.millis) should equal(3)
```

As you can see, first we construct the partial graph that contains all the zipping and comparing of stream elements. This partial graph will have three inputs and one output, wherefore we use the `UniformFanInShape`. Then we import it (all of its nodes and connections) explicitly into the closed graph built in the second step in which all the undefined elements are rewired to real sources and sinks. The graph can then be run and yields the expected result.

Warning: Please note that `GraphDSL` is not able to provide compile time type-safety about whether or not all elements have been properly connected—this validation is performed as a runtime check during the graph's instantiation.

A partial graph also verifies that all ports are either connected or part of the returned `Shape`.

7.6.3 Constructing Sources, Sinks and Flows from Partial Graphs

Instead of treating a partial graph as simply a collection of flows and junctions which may not yet all be connected it is sometimes useful to expose such a complex graph as a simpler structure, such as a `Source`, `Sink` or `Flow`.

In fact, these concepts can be easily expressed as special cases of a partially connected graph:

- `Source` is a partial graph with *exactly one* output, that is it returns a `SourceShape`.
- `Sink` is a partial graph with *exactly one* input, that is it returns a `SinkShape`.
- `Flow` is a partial graph with *exactly one* input and *exactly one* output, that is it returns a `FlowShape`.

Being able to hide complex graphs inside of simple elements such as Sink / Source / Flow enables you to easily create one complex element and from there on treat it as simple compound stage for linear computations.

In order to create a Source from a graph the method `Source.fromGraph` is used, to use it we must have a `Graph[SourceShape,T]`. This is constructed using `GraphDSL.create` and returning a `SourceShape` from the function passed in . The single outlet must be provided to the `SourceShape.of` method and will become "the sink that must be attached before this Source can run".

Refer to the example below, in which we create a Source that zips together two numbers, to see this graph construction in action:

```
val pairs = Source.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  // prepare graph elements
  val zip = b.add(Zip[Int, Int]())
  def ints = Source.fromIterator(() => Iterator.from(1))

  // connect the graph
  ints.filter(_ % 2 != 0) ~> zip.in0
  ints.filter(_ % 2 == 0) ~> zip.in1

  // expose port
  SourceShape(zip.out)
})

val firstPair: Future[(Int, Int)] = pairs.runWith(Sink.head)
```

Similarly the same can be done for a `Sink[T]`, using `SinkShape.of` in which case the provided value must be an `Inlet[T]`. For defining a `Flow[T]` we need to expose both an inlet and an outlet:

```
val pairUpWithToString =
  Flow.fromGraph(GraphDSL.create() { implicit b =>
    import GraphDSL.Implicits._

    // prepare graph elements
    val broadcast = b.add(Broadcast[Int](2))
    val zip = b.add(Zip[Int, String]())

    // connect the graph
    broadcast.out(0).map(identity) ~> zip.in0
    broadcast.out(1).map(_.toString) ~> zip.in1

    // expose ports
    FlowShape(broadcast.in, zip.out)
  })

pairUpWithToString.runWith(Source(List(1)), Sink.head)
```

7.6.4 Combining Sources and Sinks with simplified API

There is a simplified API you can use to combine sources and sinks with junctions like: `Broadcast[T]`, `Balance[T]`, `Merge[In]` and `Concat[A]` without the need for using the Graph DSL. The combine method takes care of constructing the necessary graph underneath. In following example we combine two sources into one (fan-in):

```
val sourceOne = Source(List(1))
val sourceTwo = Source(List(2))
val merged = Source.combine(sourceOne, sourceTwo)(Merge(_))

val mergedResult: Future[Int] = merged.runWith(Sink.fold(0)(_ + _))
```

The same can be done for a `Sink[T]` but in this case it will be fan-out:

```
val sendRmotely = Sink.actorRef(actorRef, "Done")
val localProcessing = Sink.foreach[Int](_ => /* do something usefull */ ())

val sink = Sink.combine(sendRmotely, localProcessing)(Broadcast[Int](_))

Source(List(0, 1, 2)).runWith(sink)
```

7.6.5 Building reusable Graph components

It is possible to build reusable, encapsulated components of arbitrary input and output ports using the graph DSL.

As an example, we will build a graph junction that represents a pool of workers, where a worker is expressed as a `Flow[I,O,_]`, i.e. a simple transformation of jobs of type `I` to results of type `O` (as you have seen already, this flow can actually contain a complex graph inside). Our reusable worker pool junction will not preserve the order of the incoming jobs (they are assumed to have a proper ID field) and it will use a `Balance` junction to schedule jobs to available workers. On top of this, our junction will feature a "fastlane", a dedicated port where jobs of higher priority can be sent.

Altogether, our junction will have two input ports of type `I` (for the normal and priority jobs) and an output port of type `O`. To represent this interface, we need to define a custom `Shape`. The following lines show how to do that.

```
// A shape represents the input and output ports of a reusable
// processing module
case class PriorityWorkerPoolShape[In, Out](
  jobsIn:         Inlet[In],
  priorityJobsIn: Inlet[In],
  resultsOut:     Outlet[Out]) extends Shape {

  // It is important to provide the list of all input and output
  // ports with a stable order. Duplicates are not allowed.
  override val inlets: immutable.Seq[Inlet[_]] =
    jobsIn :: priorityJobsIn :: Nil
  override val outlets: immutable.Seq[Outlet[_]] =
    resultsOut :: Nil

  // A Shape must be able to create a copy of itself. Basically
  // it means a new instance with copies of the ports
  override def deepCopy() = PriorityWorkerPoolShape(
    jobsIn.carbonCopy(),
    priorityJobsIn.carbonCopy(),
    resultsOut.carbonCopy())

  // A Shape must also be able to create itself from existing ports
  override def copyFromPorts(
    inlets:  immutable.Seq[Inlet[_]],
    outlets: immutable.Seq[Outlet[_]]) = {
    assert(inlets.size == this.inlets.size)
    assert(outlets.size == this.outlets.size)
    // This is why order matters when overriding inlets and outlets.
    PriorityWorkerPoolShape[In, Out](inlets(0).as[In], inlets(1).as[In],␣
↪outlets(0).as[Out])
  }
}
```

7.6.6 Predefined shapes

In general a custom `Shape` needs to be able to provide all its input and output ports, be able to copy itself, and also be able to create a new instance from given ports. There are some predefined shapes provided to avoid unnecessary

boilerplate:

- `SourceShape`, `SinkShape`, `FlowShape` for simpler shapes,
- `UniformFanInShape` and `UniformFanOutShape` for junctions with multiple input (or output) ports of the same type,
- `FanInShape1`, `FanInShape2`, ..., `FanOutShape1`, `FanOutShape2`, ... for junctions with multiple input (or output) ports of different types.

Since our shape has two input ports and one output port, we can just use the `FanInShape` DSL to define our custom shape:

```
import FanInShape.{ Init, Name }

class PriorityWorkerPoolShape2[In, Out](_init: Init[Out] = Name("PriorityWorkerPool
↪"))
  extends FanInShape[Out](_init) {
  protected override def construct(i: Init[Out]) = new PriorityWorkerPoolShape2(i)

  val jobsIn = newInlet[In]("jobsIn")
  val priorityJobsIn = newInlet[In]("priorityJobsIn")
  // Outlet[Out] with name "out" is automatically created
}
```

Now that we have a `Shape` we can wire up a Graph that represents our worker pool. First, we will merge incoming normal and priority jobs using `MergePreferred`, then we will send the jobs to a `Balance` junction which will fan-out to a configurable number of workers (flows), finally we merge all these results together and send them out through our only output port. This is expressed by the following code:

```
object PriorityWorkerPool {
  def apply[In, Out](
    worker:      Flow[In, Out, Any],
    workerCount: Int): Graph[PriorityWorkerPoolShape[In, Out], NotUsed] = {

    GraphDSL.create() { implicit b =>
      import GraphDSL.Implicits._

      val priorityMerge = b.add(MergePreferred[In](1))
      val balance = b.add(Balance[In](workerCount))
      val resultsMerge = b.add(Merge[Out](workerCount))

      // After merging priority and ordinary jobs, we feed them to the balancer
      priorityMerge ~> balance

      // Wire up each of the outputs of the balancer to a worker flow
      // then merge them back
      for (i <- 0 until workerCount)
        balance.out(i) ~> worker ~> resultsMerge.in(i)

      // We now expose the input ports of the priorityMerge and the output
      // of the resultsMerge as our PriorityWorkerPool ports
      // -- all neatly wrapped in our domain specific Shape
      PriorityWorkerPoolShape(
        jobsIn = priorityMerge.in(0),
        priorityJobsIn = priorityMerge.preferred,
        resultsOut = resultsMerge.out)
    }

  }

}
```

All we need to do now is to use our custom junction in a graph. The following code simulates some simple workers

and jobs using plain strings and prints out the results. Actually we used *two* instances of our worker pool junction using `add()` twice.

```
val worker1 = Flow[String].map("step 1 " + _)
val worker2 = Flow[String].map("step 2 " + _)

RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val priorityPool1 = b.add(PriorityWorkerPool(worker1, 4))
  val priorityPool2 = b.add(PriorityWorkerPool(worker2, 2))

  Source(1 to 100).map("job: " + _) ~> priorityPool1.jobsIn
  Source(1 to 100).map("priority job: " + _) ~> priorityPool1.priorityJobsIn

  priorityPool1.resultsOut ~> priorityPool2.jobsIn
  Source(1 to 100).map("one-step, priority " + _) ~> priorityPool2.priorityJobsIn

  priorityPool2.resultsOut ~> Sink.foreach(println)
  ClosedShape
}).run()
```

7.6.7 Bidirectional Flows

A graph topology that is often useful is that of two flows going in opposite directions. Take for example a codec stage that serializes outgoing messages and deserializes incoming octet streams. Another such stage could add a framing protocol that attaches a length header to outgoing data and parses incoming frames back into the original octet stream chunks. These two stages are meant to be composed, applying one atop the other as part of a protocol stack. For this purpose exists the special type `BidiFlow` which is a graph that has exactly two open inlets and two open outlets. The corresponding shape is called `BidiShape` and is defined like this:

```
/**
 * A bidirectional flow of elements that consequently has two inputs and two
 * outputs, arranged like this:
 *
 * {{{
 *        +------+
 *  In1 ~>|      |~> Out1
 *        | bidi |
 * Out2 <~|      |<~ In2
 *        +------+
 * }}}
 */
final case class BidiShape[-In1, +Out1, -In2, +Out2](
  in1:  Inlet[In1 @uncheckedVariance],
  out1: Outlet[Out1 @uncheckedVariance],
  in2:  Inlet[In2 @uncheckedVariance],
  out2: Outlet[Out2 @uncheckedVariance]) extends Shape {
  // implementation details elided ...
}
```

A bidirectional flow is defined just like a unidirectional `Flow` as demonstrated for the codec mentioned above:

```
trait Message
case class Ping(id: Int) extends Message
case class Pong(id: Int) extends Message

def toBytes(msg: Message): ByteString = {
  // implementation details elided ...
}
```

```
def fromBytes(bytes: ByteString): Message = {
  // implementation details elided ...
}

val codecVerbose = BidiFlow.fromGraph(GraphDSL.create() { b =>
  // construct and add the top flow, going outbound
  val outbound = b.add(Flow[Message].map(toBytes))
  // construct and add the bottom flow, going inbound
  val inbound = b.add(Flow[ByteString].map(fromBytes))
  // fuse them together into a BidiShape
  BidiShape.fromFlows(outbound, inbound)
})

// this is the same as the above
val codec = BidiFlow.fromFunctions(toBytes _, fromBytes _)
```

The first version resembles the partial graph constructor, while for the simple case of a functional 1:1 transformation there is a concise convenience method as shown on the last line. The implementation of the two functions is not difficult either:

```
def toBytes(msg: Message): ByteString = {
  implicit val order = ByteOrder.LITTLE_ENDIAN
  msg match {
    case Ping(id) => ByteString.newBuilder.putByte(1).putInt(id).result()
    case Pong(id) => ByteString.newBuilder.putByte(2).putInt(id).result()
  }
}

def fromBytes(bytes: ByteString): Message = {
  implicit val order = ByteOrder.LITTLE_ENDIAN
  val it = bytes.iterator
  it.getByte match {
    case 1     => Ping(it.getInt)
    case 2     => Pong(it.getInt)
    case other => throw new RuntimeException(s"parse error: expected 1|2 got $other
↪")
  }
}
```

In this way you could easily integrate any other serialization library that turns an object into a sequence of bytes.

The other stage that we talked about is a little more involved since reversing a framing protocol means that any received chunk of bytes may correspond to zero or more messages. This is best implemented using a `GraphStage` (see also *Custom processing with GraphStage*).

```
val framing = BidiFlow.fromGraph(GraphDSL.create() { b =>
  implicit val order = ByteOrder.LITTLE_ENDIAN

  def addLengthHeader(bytes: ByteString) = {
    val len = bytes.length
    ByteString.newBuilder.putInt(len).append(bytes).result()
  }

  class FrameParser extends GraphStage[FlowShape[ByteString, ByteString]] {

    val in = Inlet[ByteString]("FrameParser.in")
    val out = Outlet[ByteString]("FrameParser.out")
    override val shape = FlowShape.of(in, out)

    override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =␣
↪new GraphStageLogic(shape) {

      // this holds the received but not yet parsed bytes
```

```
      var stash = ByteString.empty
      // this holds the current message length or -1 if at a boundary
      var needed = -1

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (isClosed(in)) run()
          else pull(in)
        }
      })
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val bytes = grab(in)
          stash = stash ++ bytes
          run()
        }

        override def onUpstreamFinish(): Unit = {
          // either we are done
          if (stash.isEmpty) completeStage()
          // or we still have bytes to emit
          // wait with completion and let run() complete when the
          // rest of the stash has been sent downstream
          else if (isAvailable(out)) run()
        }
      })

      private def run(): Unit = {
        if (needed == -1) {
          // are we at a boundary? then figure out next length
          if (stash.length < 4) {
            if (isClosed(in)) completeStage()
            else pull(in)
          } else {
            needed = stash.iterator.getInt
            stash = stash.drop(4)
            run() // cycle back to possibly already emit the next chunk
          }
        } else if (stash.length < needed) {
          // we are in the middle of a message, need more bytes,
          // or have to stop if input closed
          if (isClosed(in)) completeStage()
          else pull(in)
        } else {
          // we have enough to emit at least one message, so do it
          val emit = stash.take(needed)
          stash = stash.drop(needed)
          needed = -1
          push(out, emit)
        }
      }
    }
  }

  val outbound = b.add(Flow[ByteString].map(addLengthHeader))
  val inbound = b.add(Flow[ByteString].via(new FrameParser))
  BidiShape.fromFlows(outbound, inbound)
})
```

With these implementations we can build a protocol stack and test it:

```
/* construct protocol stack
 *         +------------------------------------+
 *         | stack                              |
 *         |                                    |
 *         |  +-------+            +---------+  |
 *    ~>   O~~o       |     ~>     |         o~~O    ~>
 * Message |  | codec | ByteString | framing |  | ByteString
 *    <~   O~~o       |     <~     |         o~~O    <~
 *         |  +-------+            +---------+  |
 *         +------------------------------------+
 */
val stack = codec.atop(framing)

// test it by plugging it into its own inverse and closing the right end
val pingpong = Flow[Message].collect { case Ping(id) => Pong(id) }
val flow = stack.atop(stack.reversed).join(pingpong)
val result = Source((0 to 9).map(Ping)).via(flow).limit(20).runWith(Sink.seq)
Await.result(result, 1.second) should ===((0 to 9).map(Pong))
```

This example demonstrates how `BidiFlow` subgraphs can be hooked together and also turned around with the `.reversed` method. The test simulates both parties of a network communication protocol without actually having to open a network connection—the flows can just be connected directly.

7.6.8 Accessing the materialized value inside the Graph

In certain cases it might be necessary to feed back the materialized value of a Graph (partial, closed or backing a Source, Sink, Flow or BidiFlow). This is possible by using `builder.materializedValue` which gives an `Outlet` that can be used in the graph as an ordinary source or outlet, and which will eventually emit the materialized value. If the materialized value is needed at more than one place, it is possible to call `materializedValue` any number of times to acquire the necessary number of outlets.

```
import GraphDSL.Implicits._
val foldFlow: Flow[Int, Int, Future[Int]] = Flow.fromGraph(GraphDSL.create(Sink.
↪fold[Int, Int](0)(_ + _)) { implicit builder => fold =>
  FlowShape(fold.in, builder.materializedValue.mapAsync(4)(identity).outlet)
})
```

Be careful not to introduce a cycle where the materialized value actually contributes to the materialized value. The following example demonstrates a case where the materialized `Future` of a fold is fed back to the fold itself.

```
import GraphDSL.Implicits._
// This cannot produce any value:
val cyclicFold: Source[Int, Future[Int]] = Source.fromGraph(GraphDSL.create(Sink.
↪fold[Int, Int](0)(_ + _)) { implicit builder => fold =>
  // - Fold cannot complete until its upstream mapAsync completes
  // - mapAsync cannot complete until the materialized Future produced by
  //   fold completes
  // As a result this Source will never emit anything, and its materialited
  // Future will never complete
  builder.materializedValue.mapAsync(4)(identity) ~> fold
  SourceShape(builder.materializedValue.mapAsync(4)(identity).outlet)
})
```

7.6.9 Graph cycles, liveness and deadlocks

Cycles in bounded stream topologies need special considerations to avoid potential deadlocks and other liveness issues. This section shows several examples of problems that can arise from the presence of feedback arcs in stream processing graphs.

In the following examples runnable graphs are created but do not run because each have some issue and will deadlock after start. `Source` variable is not defined as the nature and number of element does not matter for described problems.

The first example demonstrates a graph that contains a naïve cycle. The graph takes elements from the source, prints them, then broadcasts those elements to a consumer (we just used `Sink.ignore` for now) and to a feedback arc that is merged back into the main stream via a `Merge` junction.

Note: The graph DSL allows the connection arrows to be reversed, which is particularly handy when writing cycles—as we will see there are cases where this is very helpful.

```
// WARNING! The graph below deadlocks!
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val merge = b.add(Merge[Int](2))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
            merge                  <~                      bcast
  ClosedShape
})
```

Running this we observe that after a few numbers have been printed, no more elements are logged to the console - all processing stops after some time. After some investigation we observe that:

- through merging from `source` we increase the number of elements flowing in the cycle
- by broadcasting back to the cycle we do not decrease the number of elements in the cycle

Since Akka Streams (and Reactive Streams in general) guarantee bounded processing (see the "Buffering" section for more details) it means that only a bounded number of elements are buffered over any time span. Since our cycle gains more and more elements, eventually all of its internal buffers become full, backpressuring `source` forever. To be able to process more elements from `source` elements would need to leave the cycle somehow.

If we modify our feedback loop by replacing the `Merge` junction with a `MergePreferred` we can avoid the deadlock. `MergePreferred` is unfair as it always tries to consume from a preferred input port if there are elements available before trying the other lower priority input ports. Since we feed back through the preferred port it is always guaranteed that the elements in the cycles can flow.

```
// WARNING! The graph below stops consuming from "source" after a few steps
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val merge = b.add(MergePreferred[Int](1))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
            merge.preferred              <~                  bcast
  ClosedShape
})
```

If we run the example we see that the same sequence of numbers are printed over and over again, but the processing does not stop. Hence, we avoided the deadlock, but `source` is still back-pressured forever, because buffer space is never recovered: the only action we see is the circulation of a couple of initial elements from `source`.

Note: What we see here is that in certain cases we need to choose between boundedness and liveness. Our first example would not deadlock if there would be an infinite buffer in the loop, or vice versa, if the elements in the cycle would be balanced (as many elements are removed as many are injected) then there would be no deadlock.

To make our cycle both live (not deadlocking) and fair we can introduce a dropping element on the feedback arc. In this case we chose the `buffer()` operation giving it a dropping strategy `OverflowStrategy.dropHead`.

```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val merge = b.add(Merge[Int](2))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
      merge <~ Flow[Int].buffer(10, OverflowStrategy.dropHead) <~ bcast
  ClosedShape
})
```

If we run this example we see that

- The flow of elements does not stop, there are always elements printed
- We see that some of the numbers are printed several times over time (due to the feedback loop) but on average the numbers are increasing in the long term

This example highlights that one solution to avoid deadlocks in the presence of potentially unbalanced cycles (cycles where the number of circulating elements are unbounded) is to drop elements. An alternative would be to define a larger buffer with `OverflowStrategy.fail` which would fail the stream instead of deadlocking it after all buffer space has been consumed.

As we discovered in the previous examples, the core problem was the unbalanced nature of the feedback loop. We circumvented this issue by adding a dropping element, but now we want to build a cycle that is balanced from the beginning instead. To achieve this we modify our first graph by replacing the `Merge` junction with a `ZipWith`. Since `ZipWith` takes one element from `source` *and* from the feedback arc to inject one element into the cycle, we maintain the balance of elements.

```
// WARNING! The graph below never processes any elements
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val zip = b.add(ZipWith[Int, Int, Int]((left, right) => right))
  val bcast = b.add(Broadcast[Int](2))

  source ~> zip.in0
  zip.out.map { s => println(s); s } ~> bcast ~> Sink.ignore
  zip.in1             <~                bcast
  ClosedShape
})
```

Still, when we try to run the example it turns out that no element is printed at all! After some investigation we realize that:

- In order to get the first element from `source` into the cycle we need an already existing element in the cycle
- In order to get an initial element in the cycle we need an element from `source`

These two conditions are a typical "chicken-and-egg" problem. The solution is to inject an initial element into the cycle that is independent from `source`. We do this by using a `Concat` junction on the backwards arc that injects a single element using `Source.single`.

```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val zip = b.add(ZipWith((left: Int, right: Int) => left))
  val bcast = b.add(Broadcast[Int](2))
  val concat = b.add(Concat[Int]())
  val start = Source.single(0)
```

```
  source ~> zip.in0
  zip.out.map { s => println(s); s } ~> bcast ~> Sink.ignore
  zip.in1 <~ concat <~ start
             concat         <~          bcast
  ClosedShape
})
```

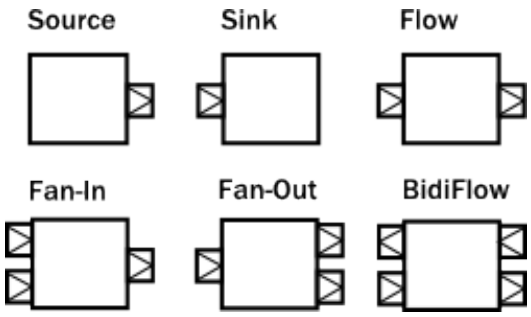
When we run the above example we see that processing starts and never stops. The important takeaway from this example is that balanced cycles often need an initial "kick-off" element to be injected into the cycle.

7.7 Modularity, Composition and Hierarchy

Akka Streams provide a uniform model of stream processing graphs, which allows flexible composition of reusable components. In this chapter we show how these look like from the conceptual and API perspective, demonstrating the modularity aspects of the library.

7.7.1 Basics of composition and modularity

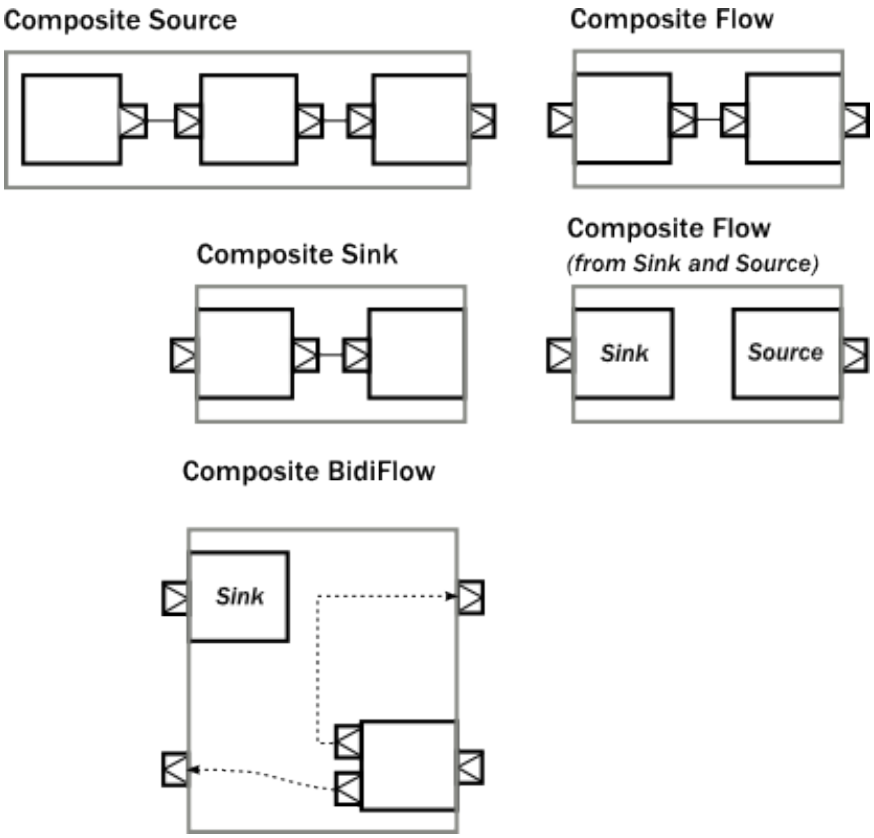
Every processing stage used in Akka Streams can be imagined as a "box" with input and output ports where elements to be processed arrive and leave the stage. In this view, a `Source` is nothing else than a "box" with a single output port, or, a `BidiFlow` is a "box" with exactly two input and two output ports. In the figure below we illustrate the most common used stages viewed as "boxes".



The *linear* stages are `Source`, `Sink` and `Flow`, as these can be used to compose strict chains of processing stages. Fan-in and Fan-out stages have usually multiple input or multiple output ports, therefore they allow to build more complex graph layouts, not just chains. `BidiFlow` stages are usually useful in IO related tasks, where there are input and output channels to be handled. Due to the specific shape of `BidiFlow` it is easy to stack them on top of each other to build a layered protocol for example. The `TLS` support in Akka is for example implemented as a `BidiFlow`.

These reusable components already allow the creation of complex processing networks. What we have seen so far does not implement modularity though. It is desirable for example to package up a larger graph entity into a reusable component which hides its internals only exposing the ports that are meant to the users of the module to interact with. One good example is the `Http` server component, which is encoded internally as a `BidiFlow` which interfaces with the client TCP connection using an input-output port pair accepting and sending `ByteString` s, while its upper ports emit and receive `HttpRequest` and `HttpResponse` instances.

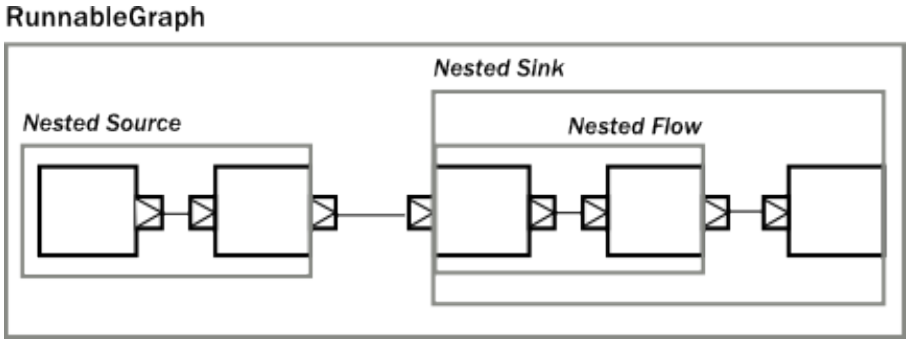
The following figure demonstrates various composite stages, that contain various other type of stages internally, but hiding them behind a *shape* that looks like a `Source`, `Flow`, etc.



One interesting example above is a `Flow` which is composed of a disconnected `Sink` and `Source`. This can be achieved by using the `fromSinkAndSource()` constructor method on `Flow` which takes the two parts as parameters.

The example `BidiFlow` demonstrates that internally a module can be of arbitrary complexity, and the exposed ports can be wired in flexible ways. The only constraint is that all the ports of enclosed modules must be either connected to each other, or exposed as interface ports, and the number of such ports needs to match the requirement of the shape, for example a `Source` allows only one exposed output port, the rest of the internal ports must be properly connected.

These mechanics allow arbitrary nesting of modules. For example the following figure demonstrates a `RunnableGraph` that is built from a composite `Source` and a composite `Sink` (which in turn contains a composite `Flow`).



The above diagram contains one more shape that we have not seen yet, which is called `RunnableGraph`. It turns out, that if we wire all exposed ports together, so that no more open ports remain, we get a module that is *closed*. This is what the `RunnableGraph` class represents. This is the shape that a `Materializer` can take and turn into a network of running entities that perform the task described. In fact, a `RunnableGraph` is a module itself, and (maybe somewhat surprisingly) it can be used as part of larger graphs. It is rarely useful to embed a closed graph shape in a larger graph (since it becomes an isolated island as there are no open port for communication with the rest of the graph), but this demonstrates the uniform underlying model.

If we try to build a code snippet that corresponds to the above diagram, our first try might look like this:

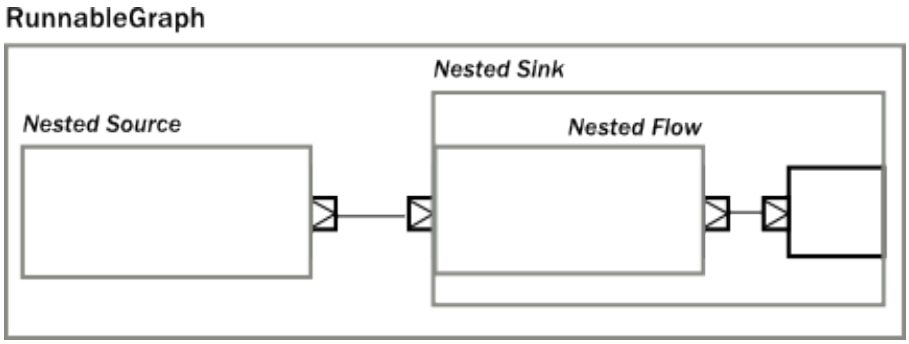
```
Source.single(0)
  .map(_ + 1)
  .filter(_ != 0)
  .map(_ - 2)
  .to(Sink.fold(0)(_ + _))

// ... where is the nesting?
```

It is clear however that there is no nesting present in our first attempt, since the library cannot figure out where we intended to put composite module boundaries, it is our responsibility to do that. If we are using the DSL provided by the `Flow`, `Source`, `Sink` classes then nesting can be achieved by calling one of the methods `withAttributes()` or `named()` (where the latter is just a shorthand for adding a name attribute).

The following code demonstrates how to achieve the desired nesting:

```
val nestedSource =
  Source.single(0) // An atomic source
    .map(_ + 1) // an atomic processing stage
    .named("nestedSource") // wraps up the current Source and gives it a name

val nestedFlow =
  Flow[Int].filter(_ != 0) // an atomic processing stage
    .map(_ - 2) // another atomic processing stage
    .named("nestedFlow") // wraps up the Flow, and gives it a name

val nestedSink =
  nestedFlow.to(Sink.fold(0)(_ + _)) // wire an atomic sink to the nestedFlow
    .named("nestedSink") // wrap it up

// Create a RunnableGraph
val runnableGraph = nestedSource.to(nestedSink)
```

Once we have hidden the internals of our components, they act like any other built-in component of similar shape. If we hide some of the internals of our composites, the result looks just like if any other predefine component has been used:



If we look at usage of built-in components, and our custom components, there is no difference in usage as the code snippet below demonstrates.

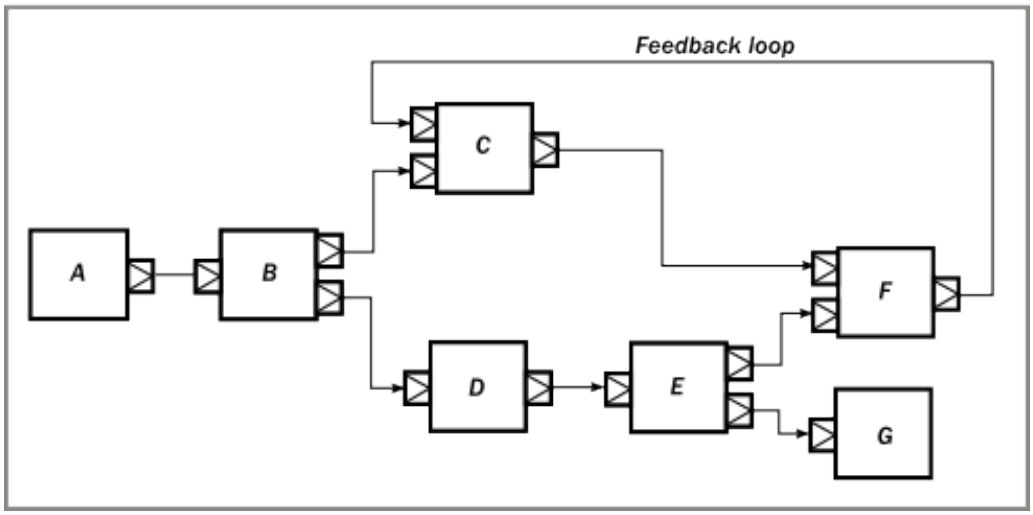
```
// Create a RunnableGraph from our components
val runnableGraph = nestedSource.to(nestedSink)

// Usage is uniform, no matter if modules are composite or atomic
val runnableGraph2 = Source.single(0).to(Sink.fold(0)(_ + _))
```

7.7.2 Composing complex systems

In the previous section we explored the possibility of composition, and hierarchy, but we stayed away from non-linear, generalized graph components. There is nothing in Akka Streams though that enforces that stream processing layouts can only be linear. The DSL for `Source` and friends is optimized for creating such linear chains, as they are the most common in practice. There is a more advanced DSL for building complex graphs, that can be used if more flexibility is needed. We will see that the difference between the two DSLs is only on the surface: the concepts they operate on are uniform across all DSLs and fit together nicely.

As a first example, let's look at a more complex layout:

RunnableGraph



The diagram shows a `RunnableGraph` (remember, if there are no unwired ports, the graph is closed, and therefore can be materialized) that encapsulates a non-trivial stream processing network. It contains fan-in, fan-out stages, directed and non-directed cycles. The `runnable()` method of the `GraphDSL` object allows the creation of a general, closed, and runnable graph. For example the network on the diagram can be realized like this:

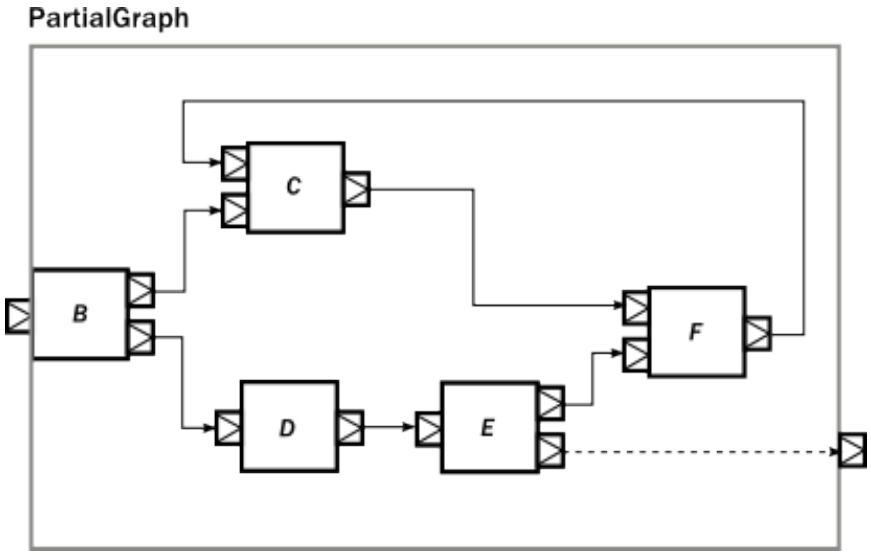
```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val A: Outlet[Int]                  = builder.add(Source.single(0)).out
  val B: UniformFanOutShape[Int, Int] = builder.add(Broadcast[Int](2))
  val C: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val D: FlowShape[Int, Int]          = builder.add(Flow[Int].map(_ + 1))
  val E: UniformFanOutShape[Int, Int] = builder.add(Balance[Int](2))
  val F: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val G: Inlet[Any]                   = builder.add(Sink.foreach(println)).in

                C     <~      F
  A  ~>  B  ~>  C     ~>      F
         B  ~>  D  ~>  E  ~>  F
                       E  ~>  G

  ClosedShape
})
```

In the code above we used the implicit port numbering feature (to make the graph more readable and similar to the diagram) and we imported `Source` s, `Sink` s and `Flow` s explicitly. It is possible to refer to the ports explicitly, and it is not necessary to import our linear stages via `add()`, so another version might look like this:

```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val B = builder.add(Broadcast[Int](2))
  val C = builder.add(Merge[Int](2))
  val E = builder.add(Balance[Int](2))
  val F = builder.add(Merge[Int](2))

  Source.single(0) ~> B.in; B.out(0) ~> C.in(1); C.out ~> F.in(0)
  C.in(0) <~ F.out
```

```
  B.out(1).map(_ + 1) ~> E.in; E.out(0) ~> F.in(1)
  E.out(1) ~> Sink.foreach(println)
  ClosedShape
})
```

Similar to the case in the first section, so far we have not considered modularity. We created a complex graph, but the layout is flat, not modularized. We will modify our example, and create a reusable component with the graph DSL. The way to do it is to use the `create()` factory method on `GraphDSL`. If we remove the sources and sinks from the previous example, what remains is a partial graph:



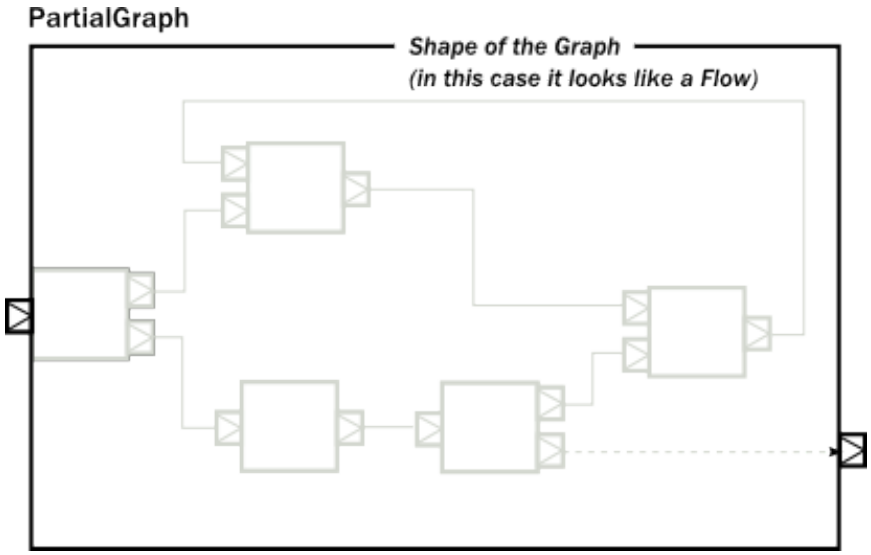
We can recreate a similar graph in code, using the DSL in a similar way than before:

```
import GraphDSL.Implicits._
val partial = GraphDSL.create() { implicit builder =>
  val B = builder.add(Broadcast[Int](2))
  val C = builder.add(Merge[Int](2))
  val E = builder.add(Balance[Int](2))
  val F = builder.add(Merge[Int](2))

                                  C  <~  F
  B  ~>                           C  ~>  F
  B  ~>  Flow[Int].map(_ + 1)  ~>  E  ~>  F
  FlowShape(B.in, E.out(1))
}.named("partial")
```

The only new addition is the return value of the builder block, which is a `Shape`. All graphs (including `Source`, `BidiFlow`, etc) have a shape, which encodes the *typed* ports of the module. In our example there is exactly one

input and output port left, so we can declare it to have a `FlowShape` by returning an instance of it. While it is possible to create new `Shape` types, it is usually recommended to use one of the matching built-in ones.

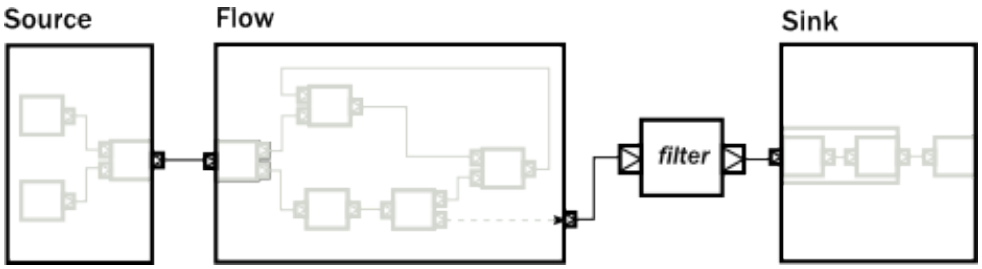
The resulting graph is already a properly wrapped module, so there is no need to call *named()* to encapsulate the graph, but it is a good practice to give names to modules to help debugging.



Since our partial graph has the right shape, it can be already used in the simpler, linear DSL:

```
Source.single(0).via(partial).to(Sink.ignore)
```

It is not possible to use it as a `Flow` yet, though (i.e. we cannot call `.filter()` on it), but `Flow` has a `fromGraph()` method that just adds the DSL to a `FlowShape`. There are similar methods on `Source`, `Sink` and `BidiShape`, so it is easy to get back to the simpler DSL if a graph has the right shape. For convenience, it is also possible to skip the partial graph creation, and use one of the convenience creator methods. To demonstrate this, we will create the following graph:



The code version of the above closed graph might look like this:

```
// Convert the partial graph of FlowShape to a Flow to get
// access to the fluid DSL (for example to be able to call .filter())
val flow = Flow.fromGraph(partial)

// Simple way to create a graph backed Source
val source = Source.fromGraph( GraphDSL.create() { implicit builder =>
  val merge = builder.add(Merge[Int](2))
  Source.single(0)      ~> merge
  Source(List(2, 3, 4)) ~> merge

  // Exposing exactly one output port
  SourceShape(merge.out)
})

// Building a Sink with a nested Flow, using the fluid DSL
val sink = {
  val nestedFlow = Flow[Int].map(_ * 2).drop(10).named("nestedFlow")
  nestedFlow.to(Sink.head)
}

// Putting all together
val closed = source.via(flow.filter(_ > 1)).to(sink)
```

Note: All graph builder sections check if the resulting graph has all ports connected except the exposed ones and will throw an exception if this is violated.

We are still in debt of demonstrating that `RunnableGraph` is a component just like any other, which can be embedded in graphs. In the following snippet we embed one closed graph in another:

```
val closed1 = Source.single(0).to(Sink.foreach(println))
val closed2 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val embeddedClosed: ClosedShape = builder.add(closed1)
  // ...
  embeddedClosed
})
```

The type of the imported module indicates that the imported module has a `ClosedShape`, and so we are not able to wire it to anything else inside the enclosing closed graph. Nevertheless, this "island" is embedded properly, and will be materialized just like any other module that is part of the graph.

As we have demonstrated, the two DSLs are fully interoperable, as they encode a similar nested structure of "boxes with ports", it is only the DSLs that differ to be as much powerful as possible on the given abstraction level. It is possible to embed complex graphs in the fluid DSL, and it is just as easy to import and embed a `Flow`, etc, in a larger, complex structure.

We have also seen, that every module has a `Shape` (for example a `Sink` has a `SinkShape`) independently which DSL was used to create it. This uniform representation enables the rich composability of various stream processing entities in a convenient way.

7.7.3 Materialized values

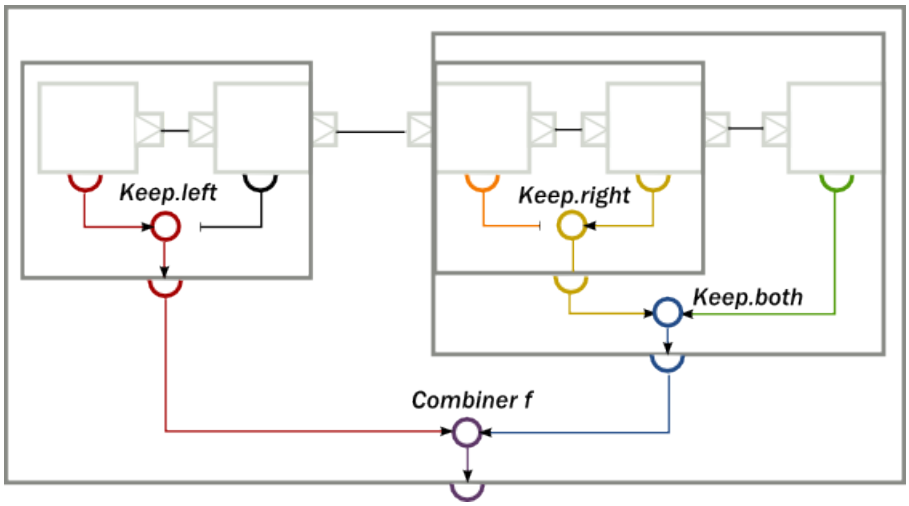
After realizing that `RunnableGraph` is nothing more than a module with no unused ports (it is an island), it becomes clear that after materialization the only way to communicate with the running stream processing logic is via some side-channel. This side channel is represented as a *materialized value*. The situation is similar to `Actor` s, where the `Props` instance describes the actor logic, but it is the call to `actorOf()` that creates an actually running actor, and returns an `ActorRef` that can be used to communicate with the running actor itself. Since the `Props` can be reused, each call will return a different reference.

When it comes to streams, each materialization creates a new running network corresponding to the blueprint that was encoded in the provided `RunnableGraph`. To be able to interact with the running network, each materialization needs to return a different object that provides the necessary interaction capabilities. In other words, the `RunnableGraph` can be seen as a factory, which creates:

- a network of running processing entities, inaccessible from the outside
- a materialized value, optionally providing a controlled interaction capability with the network

Unlike actors though, each of the processing stages might provide a materialized value, so when we compose multiple stages or modules, we need to combine the materialized value as well (there are default rules which make this easier, for example *to()* and *via()* takes care of the most common case of taking the materialized value to the left. See *Combining materialized values* for details). We demonstrate how this works by a code example and a diagram which graphically demonstrates what is happening.

The propagation of the individual materialized values from the enclosed modules towards the top will look like this:



To implement the above, first, we create a composite `Source`, where the enclosed `Source` have a materialized type of `Promise[Unit]`. By using the combiner function `Keep.left`, the resulting materialized type is of the nested module (indicated by the color *red* on the diagram):

```
// Materializes to Promise[Option[Int]]                                (red)
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]

// Materializes to Unit                                                (black)
val flow1: Flow[Int, Int, NotUsed] = Flow[Int].take(100)

// Materializes to Promise[Int]                                        (red)
val nestedSource: Source[Int, Promise[Option[Int]]] =
  source.viaMat(flow1)(Keep.left).named("nestedSource")
```

Next, we create a composite `Flow` from two smaller components. Here, the second enclosed `Flow` has a materialized type of `Future[OutgoingConnection]`, and we propagate this to the parent by using `Keep.right` as the combiner function (indicated by the color *yellow* on the diagram):

```
// Materializes to Unit                                                    (orange)
val flow2: Flow[Int, ByteString, NotUsed] = Flow[Int].map { i => ByteString(i.
↪toString) }

// Materializes to Future[OutgoingConnection]                              (yellow)
val flow3: Flow[ByteString, ByteString, Future[OutgoingConnection]] =
  Tcp().outgoingConnection("localhost", 8080)

// Materializes to Future[OutgoingConnection]                              (yellow)
val nestedFlow: Flow[Int, ByteString, Future[OutgoingConnection]] =
  flow2.viaMat(flow3)(Keep.right).named("nestedFlow")
```

As a third step, we create a composite `Sink`, using our `nestedFlow` as a building block. In this snippet, both the enclosed `Flow` and the folding `Sink` has a materialized value that is interesting for us, so we use `Keep.both` to get a `Pair` of them as the materialized type of `nestedSink` (indicated by the color *blue* on the diagram)

```
// Materializes to Future[String]                                          (green)
val sink: Sink[ByteString, Future[String]] = Sink.fold("")(_ + _.utf8String)

// Materializes to (Future[OutgoingConnection], Future[String])            (blue)
val nestedSink: Sink[Int, (Future[OutgoingConnection], Future[String])] =
  nestedFlow.toMat(sink)(Keep.both)
```

As the last example, we wire together `nestedSource` and `nestedSink` and we use a custom combiner function to create a yet another materialized type of the resulting `RunnableGraph`. This combiner function just ignores the `Future[Sink]` part, and wraps the other two values in a custom case class `MyClass` (indicated by color *purple* on the diagram):

```
case class MyClass(private val p: Promise[Option[Int]], conn: OutgoingConnection) {
  def close() = p.trySuccess(None)
}

def f(
  p:    Promise[Option[Int]],
  rest: (Future[OutgoingConnection], Future[String])): Future[MyClass] = {

  val connFuture = rest._1
  connFuture.map(MyClass(p, _))
}

// Materializes to Future[MyClass]                                         (purple)
val runnableGraph: RunnableGraph[Future[MyClass]] =
  nestedSource.toMat(nestedSink)(f)
```

Note: The nested structure in the above example is not necessary for combining the materialized values, it just demonstrates how the two features work together. See *Combining materialized values* for further examples of combining materialized values without nesting and hierarchy involved.

7.7.4 Attributes

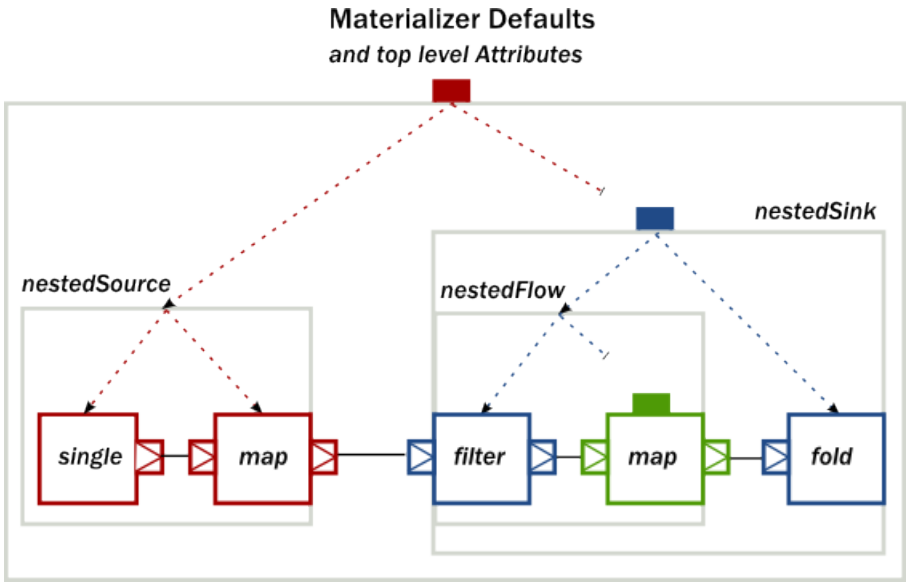
We have seen that we can use `named()` to introduce a nesting level in the fluid DSL (and also explicit nesting by using `create()` from `GraphDSL`). Apart from having the effect of adding a nesting level, `named()` is actually a shorthand for calling `withAttributes(Attributes.name("someName"))`. Attributes provide a way to fine-tune certain aspects of the materialized running entity. For example buffer sizes for asynchronous stages can be controlled via attributes (see *Buffers for asynchronous stages*). When it comes to hierarchic composition, attributes are inherited by nested modules, unless they override them with a custom value.

The code below, a modification of an earlier example sets the `inputBuffer` attribute on certain modules, but

not on others:

```
import Attributes._
val nestedSource =
  Source.single(0)
    .map(_ + 1)
    .named("nestedSource") // Wrap, no inputBuffer set

val nestedFlow =
  Flow[Int].filter(_ != 0)
    .via(Flow[Int].map(_ - 2).withAttributes(inputBuffer(4, 4))) // override
    .named("nestedFlow") // Wrap, no inputBuffer set

val nestedSink =
  nestedFlow.to(Sink.fold(0)(_ + _)) // wire an atomic sink to the nestedFlow
    .withAttributes(name("nestedSink") and inputBuffer(3, 3)) // override
```

The effect is, that each module inherits the `inputBuffer` attribute from its enclosing parent, unless it has the same attribute explicitly set. `nestedSource` gets the default attributes from the materializer itself. `nestedSink` on the other hand has this attribute set, so it will be used by all nested modules. `nestedFlow` will inherit from `nestedSink` except the `map` stage which has again an explicitly provided attribute overriding the inherited one.



This diagram illustrates the inheritance process for the example code (representing the materializer default attributes as the color *red*, the attributes set on `nestedSink` as *blue* and the attributes set on `nestedFlow` as *green*).

7.8 Buffers and working with rate

When upstream and downstream rates differ, especially when the throughput has spikes, it can be useful to introduce buffers in a stream. In this chapter we cover how buffers are used in Akka Streams.

7.8.1 Buffers for asynchronous stages

In this section we will discuss internal buffers that are introduced as an optimization when using asynchronous stages.

To run a stage asynchronously it has to be marked explicitly as such using the `.async` method. Being run asynchronously means that a stage, after handing out an element to its downstream consumer is able to immediately process the next message. To demonstrate what we mean by this, let's take a look at the following example:

```
Source(1 to 3)
  .map { i => println(s"A: $i"); i }.async
  .map { i => println(s"B: $i"); i }.async
  .map { i => println(s"C: $i"); i }.async
  .runWith(Sink.ignore)
```

Running the above example, one of the possible outputs looks like this:

```
A: 1
A: 2
B: 1
A: 3
B: 2
C: 1
B: 3
C: 2
C: 3
```

Note that the order is *not* `A:1,B:1,C:1,A:2,B:2,C:2,` which would correspond to the normal fused synchronous execution model of flows where an element completely passes through the processing pipeline before the next element enters the flow. The next element is processed by an asynchronous stage as soon as it is emitted the previous one.

While pipelining in general increases throughput, in practice there is a cost of passing an element through the asynchronous (and therefore thread crossing) boundary which is significant. To amortize this cost Akka Streams uses a *windowed*, *batching* backpressure strategy internally. It is windowed because as opposed to a Stop-And-Wait protocol multiple elements might be "in-flight" concurrently with requests for elements. It is also batching because a new element is not immediately requested once an element has been drained from the window-buffer but multiple elements are requested after multiple elements have been drained. This batching strategy reduces the communication cost of propagating the backpressure signal through the asynchronous boundary.

While this internal protocol is mostly invisible to the user (apart form its throughput increasing effects) there are situations when these details get exposed. In all of our previous examples we always assumed that the rate of the processing chain is strictly coordinated through the backpressure signal causing all stages to process no faster than the throughput of the connected chain. There are tools in Akka Streams however that enable the rates of different segments of a processing chain to be "detached" or to define the maximum throughput of the stream through external timing sources. These situations are exactly those where the internal batching buffering strategy suddenly becomes non-transparent.

Internal buffers and their effect

As we have explained, for performance reasons Akka Streams introduces a buffer for every asynchronous processing stage. The purpose of these buffers is solely optimization, in fact the size of 1 would be the most natural choice if there would be no need for throughput improvements. Therefore it is recommended to keep these buffer sizes small, and increase them only to a level suitable for the throughput requirements of the application. Default buffer sizes can be set through configuration:

```
akka.stream.materializer.max-input-buffer-size = 16
```

Alternatively they can be set by passing a `ActorMaterializerSettings` to the materializer:

```
val materializer = ActorMaterializer(
  ActorMaterializerSettings(system)
    .withInputBuffer(
      initialSize = 64,
      maxSize = 64))
```

If the buffer size needs to be set for segments of a `Flow` only, it is possible by defining a separate `Flow` with these attributes:

```
val section = Flow[Int].map(_ * 2).async
  .addAttributes(Attributes.inputBuffer(initial = 1, max = 1)) // the buffer size␣
↪of this map is 1
val flow = section.via(Flow[Int].map(_ / 2)).async // the buffer size of this map␣
↪is the default
```

Here is an example of a code that demonstrate some of the issues caused by internal buffers:

```
import scala.concurrent.duration._
case class Tick()

RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  // this is the asynchronous stage in this graph
  val zipper = b.add(ZipWith[Tick, Int, Int]((tick, count) => count).async)

  Source.tick(initialDelay = 3.second, interval = 3.second, Tick()) ~> zipper.in0

  Source.tick(initialDelay = 1.second, interval = 1.second, "message!")
    .conflateWithSeed(seed = (_) => 1)((count, _) => count + 1) ~> zipper.in1

  zipper.out ~> Sink.foreach(println)
  ClosedShape
})
```

Running the above example one would expect the number *3* to be printed in every 3 seconds (the `conflateWithSeed` step here is configured so that it counts the number of elements received before the downstream `ZipWith` consumes them). What is being printed is different though, we will see the number *1*. The reason for this is the internal buffer which is by default 16 elements large, and prefetches elements before the `ZipWith` starts consuming them. It is possible to fix this issue by changing the buffer size of `ZipWith` (or the whole graph) to 1. We will still see a leading 1 though which is caused by an initial prefetch of the `ZipWith` element.

Note: In general, when time or rate driven processing stages exhibit strange behavior, one of the first solutions to try should be to decrease the input buffer of the affected elements to 1.

7.8.2 Buffers in Akka Streams

In this section we will discuss *explicit* user defined buffers that are part of the domain logic of the stream processing pipeline of an application.

The example below will ensure that 1000 jobs (but not more) are dequeued from an external (imaginary) system and stored locally in memory - relieving the external system:

```
// Getting a stream of jobs from an imaginary external system as a Source
val jobs: Source[Job, NotUsed] = inboundJobsConnector()
jobs.buffer(1000, OverflowStrategy.backpressure)
```

The next example will also queue up 1000 jobs locally, but if there are more jobs waiting in the imaginary external systems, it makes space for the new element by dropping one element from the *tail* of the buffer. Dropping from the tail is a very common strategy but it must be noted that this will drop the *youngest* waiting job. If some "fairness" is desired in the sense that we want to be nice to jobs that has been waiting for long, then this option can be useful.

```
jobs.buffer(1000, OverflowStrategy.dropTail)
```

Instead of dropping the youngest element from the tail of the buffer a new element can be dropped without enqueueing it to the buffer at all.

```
jobs.buffer(1000, OverflowStrategy.dropNew)
```

Here is another example with a queue of 1000 jobs, but it makes space for the new element by dropping one element from the *head* of the buffer. This is the *oldest* waiting job. This is the preferred strategy if jobs are expected to be resent if not processed in a certain period. The oldest element will be retransmitted soon, (in fact a retransmitted duplicate might be already in the queue!) so it makes sense to drop it first.

```
jobs.buffer(1000, OverflowStrategy.dropHead)
```

Compared to the dropping strategies above, dropBuffer drops all the 1000 jobs it has enqueued once the buffer gets full. This aggressive strategy is useful when dropping jobs is preferred to delaying jobs.

```
jobs.buffer(1000, OverflowStrategy.dropBuffer)
```

If our imaginary external job provider is a client using our API, we might want to enforce that the client cannot have more than 1000 queued jobs otherwise we consider it flooding and terminate the connection. This is easily achievable by the error strategy which simply fails the stream once the buffer gets full.

```
jobs.buffer(1000, OverflowStrategy.fail)
```

7.8.3 Rate transformation

Understanding conflate

When a fast producer can not be informed to slow down by backpressure or some other signal, `conflate` might be useful to combine elements from a producer until a demand signal comes from a consumer.

Below is an example snippet that summarizes fast stream of elements to a standart deviation, mean and count of elements that have arrived while the stats have been calculated.

```
val statsFlow = Flow[Double]
  .conflateWithSeed(Seq(_))(_ :+ _)
  .map { s =>
    val μ = s.sum / s.size
    val se = s.map(x => pow(x - μ, 2))
    val σ = sqrt(se.sum / se.size)
    (σ, μ, s.size)
  }
```

This example demonstrates that such flow's rate is decoupled. The element rate at the start of the flow can be much higher that the element rate at the end of the flow.

Another possible use of `conflate` is to not consider all elements for summary when producer starts getting too fast. Example below demonstrates how `conflate` can be used to implement random drop of elements when consumer is not able to keep up with the producer.

```
val p = 0.01
val sampleFlow = Flow[Double]
  .conflateWithSeed(Seq(_)) {
```

```
    case (acc, elem) if Random.nextDouble < p => acc :+ elem
    case (acc, _)                             => acc
  }
  .mapConcat(identity)
```

Understanding expand

Expand helps to deal with slow producers which are unable to keep up with the demand coming from consumers. Expand allows to extrapolate a value to be sent as an element to a consumer.

As a simple use of `expand` here is a flow that sends the same element to consumer when producer does not send any new elements.

```
val lastFlow = Flow[Double]
  .expand(Iterator.continually(_))
```

Expand also allows to keep some state between demand requests from the downstream. Leveraging this, here is a flow that tracks and reports a drift between fast consumer and slow producer.

```
val driftFlow = Flow[Double]
  .expand(i => Iterator.from(0).map(i -> _))
```

Note that all of the elements coming from upstream will go through `expand` at least once. This means that the output of this flow is going to report a drift of zero if producer is fast enough, or a larger drift otherwise.

7.9 Dynamic stream handling

7.9.1 Controlling graph completion with KillSwitch

A `KillSwitch` allows the completion of graphs of `FlowShape` from the outside. It consists of a flow element that can be linked to a graph of `FlowShape` needing completion control. The `KillSwitch` trait allows to complete or fail the graph(s).

```
trait KillSwitch {
  /**
   * After calling [[KillSwitch#shutdown()]] the linked [[Graph]]s of␣
↪[[FlowShape]] are completed normally.
   */
  def shutdown(): Unit
  /**
   * After calling [[KillSwitch#abort()]] the linked [[Graph]]s of [[FlowShape]]␣
↪are failed.
   */
  def abort(ex: Throwable): Unit
}
```

After the first call to either `shutdown` or `abort`, all subsequent calls to any of these methods will be ignored. Graph completion is performed by both

- completing its downstream
- cancelling (in case of `shutdown`) or failing (in case of `abort`) its upstream.

A `KillSwitch` can control the completion of one or multiple streams, and therefore comes in two different flavours.

UniqueKillSwitch

`UniqueKillSwitch` allows to control the completion of **one** materialized `Graph` of `FlowShape`. Refer to the below for usage examples.

- **Shutdown**

```
val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.
↪backpressure)
val lastSnk = Sink.last[Int]

val (killSwitch, last) = countingSrc
  .viaMat(KillSwitches.single)(Keep.right)
  .toMat(lastSnk)(Keep.both)
  .run()

doSomethingElse()

killSwitch.shutdown()

Await.result(last, 1.second) shouldBe 2
```

- **Abort**

```
val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.
↪backpressure)
val lastSnk = Sink.last[Int]

val (killSwitch, last) = countingSrc
  .viaMat(KillSwitches.single)(Keep.right)
  .toMat(lastSnk)(Keep.both).run()

val error = new RuntimeException("boom!")
killSwitch.abort(error)

Await.result(last.failed, 1.second) shouldBe error
```

SharedKillSwitch

A `SharedKillSwitch` allows to control the completion of an arbitrary number graphs of `FlowShape`. It can be materialized multiple times via its `flow` method, and all materialized graphs linked to it are controlled by the switch. Refer to the below for usage examples.

- **Shutdown**

```
val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.
↪backpressure)
val lastSnk = Sink.last[Int]
val sharedKillSwitch = KillSwitches.shared("my-kill-switch")

val last = countingSrc
  .via(sharedKillSwitch.flow)
  .runWith(lastSnk)

val delayedLast = countingSrc
  .delay(1.second, DelayOverflowStrategy.backpressure)
  .via(sharedKillSwitch.flow)
  .runWith(lastSnk)

doSomethingElse()

sharedKillSwitch.shutdown()
```

```
Await.result(last, 1.second) shouldBe 2
Await.result(delayedLast, 1.second) shouldBe 1
```

- **Abort**

```
val countingSrc = Source(Stream.from(1)).delay(1.second)
val lastSnk = Sink.last[Int]
val sharedKillSwitch = KillSwitches.shared("my-kill-switch")

val last1 = countingSrc.via(sharedKillSwitch.flow).runWith(lastSnk)
val last2 = countingSrc.via(sharedKillSwitch.flow).runWith(lastSnk)

val error = new RuntimeException("boom!")
sharedKillSwitch.abort(error)

Await.result(last1.failed, 1.second) shouldBe error
Await.result(last2.failed, 1.second) shouldBe error
```

Note: A `UniqueKillSwitch` is always a result of a materialization, whilst `SharedKillSwitch` needs to be constructed before any materialization takes place.

7.9.2 Dynamic fan-in and fan-out with MergeHub and BroadcastHub

There are many cases when consumers or producers of a certain service (represented as a Sink, Source, or possibly Flow) are dynamic and not known in advance. The Graph DSL does not allow to represent this, all connections of the graph must be known in advance and must be connected upfront. To allow dynamic fan-in and fan-out streaming, the Hubs should be used. They provide means to construct `Sink` and `Source` pairs that are "attached" to each other, but one of them can be materialized multiple times to implement dynamic fan-in or fan-out.

Using the MergeHub

A `MergeHub` allows to implement a dynamic fan-in junction point in a graph where elements coming from different producers are emitted in a First-Comes-First-Served fashion. If the consumer cannot keep up then *all* of the producers are backpressured. The hub itself comes as a `Source` to which the single consumer can be attached. It is not possible to attach any producers until this `Source` has been materialized (started). This is ensured by the fact that we only get the corresponding `Sink` as a materialized value. Usage might look like this:

```
// A simple consumer that will print to the console for now
val consumer = Sink.foreach(println)

// Attach a MergeHub Source to the consumer. This will materialize to a
// corresponding Sink.
val runnableGraph: RunnableGraph[Sink[String, NotUsed]] =
  MergeHub.source[String](perProducerBufferSize = 16).to(consumer)

// By running/materializing the consumer we get back a Sink, and hence
// now have access to feed elements into it. This Sink can be materialized
// any number of times, and every element that enters the Sink will
// be consumed by our consumer.
val toConsumer: Sink[String, NotUsed] = runnableGraph.run()

// Feeding two independent sources into the hub.
Source.single("Hello!").runWith(toConsumer)
Source.single("Hub!").runWith(toConsumer)
```

This sequence, while might look odd at first, ensures proper startup order. Once we get the `Sink`, we can use it as many times as wanted. Everything that is fed to it will be delivered to the consumer we attached previously until it cancels.

Using the BroadcastHub

A `BroadcastHub` can be used to consume elements from a common producer by a dynamic set of consumers. The rate of the producer will be automatically adapted to the slowest consumer. In this case, the hub is a `Sink` to which the single producer must be attached first. Consumers can only be attached once the `Sink` has been materialized (i.e. the producer has been started). One example of using the `BroadcastHub`:

```
// A simple producer that publishes a new "message" every second
val producer = Source.tick(1.second, 1.second, "New message")

// Attach a BroadcastHub Sink to the producer. This will materialize to a
// corresponding Source.
// (We need to use toMat and Keep.right since by default the materialized
// value to the left is used)
val runnableGraph: RunnableGraph[Source[String, NotUsed]] =
  producer.toMat(BroadcastHub.sink(bufferSize = 256))(Keep.right)

// By running/materializing the producer, we get back a Source, which
// gives us access to the elements published by the producer.
val fromProducer: Source[String, NotUsed] = runnableGraph.run()

// Print out messages from the producer in two independent consumers
fromProducer.runForeach(msg => println("consumer1: " + msg))
fromProducer.runForeach(msg => println("consumer2: " + msg))
```

The resulting `Source` can be materialized any number of times, each materialization effectively attaching a new subscriber. If there are no subscribers attached to this hub then it will not drop any elements but instead backpressure the upstream producer until subscribers arrive. This behavior can be tweaked by using the combinators `.buffer` for example with a drop strategy, or just attaching a subscriber that drops all messages. If there are no other subscribers, this will ensure that the producer is kept drained (dropping all elements) and once a new subscriber arrives it will adaptively slow down, ensuring no more messages are dropped.

Combining dynamic stages to build a simple Publish-Subscribe service

The features provided by the Hub implementations are limited by default. This is by design, as various combinations can be used to express additional features like unsubscribing producers or consumers externally. We show here an example that builds a `Flow` representing a publish-subscribe channel. The input of the `Flow` is published to all subscribers while the output streams all the elements published.

First, we connect a `MergeHub` and a `BroadcastHub` together to form a publish-subscribe channel. Once we materialize this small stream, we get back a pair of `Source` and `Sink` that together define the publish and subscribe sides of our channel.

```
// Obtain a Sink and Source which will publish and receive from the "bus"␣
↪respectively.
val (sink, source) =
  MergeHub.source[String](perProducerBufferSize = 16)
    .toMat(BroadcastHub.sink(bufferSize = 256))(Keep.both)
    .run()
```

We now use a few tricks to add more features. First of all, we attach a `Sink.ignore` at the broadcast side of the channel to keep it drained when there are no subscribers. If this behavior is not the desired one this line can be simply dropped.

```
// Ensure that the Broadcast output is dropped if there are no listening parties.
// If this dropping Sink is not attached, then the broadcast hub will not drop any
```

```
// elements itself when there are no subscribers, backpressuring the producer␣
↪instead.
source.runWith(Sink.ignore)
```

We now wrap the `Sink` and `Source` in a `Flow` using `Flow.fromSinkAndSource`. This bundles up the two sides of the channel into one and forces users of it to always define a publisher and subscriber side (even if the subscriber side is just dropping). It also allows us to very simply attach a `KillSwitch` as a `BidiStage` which in turn makes it possible to close both the original `Sink` and `Source` at the same time. Finally, we add `backpressureTimeout` on the consumer side to ensure that subscribers that block the channel for more than 3 seconds are forcefully removed (and their stream failed).

```
// We create now a Flow that represents a publish-subscribe channel using the above
// started stream as its "topic". We add two more features, external cancellation␣
↪of
// the registration and automatic cleanup for very slow subscribers.
val busFlow: Flow[String, String, UniqueKillSwitch] =
  Flow.fromSinkAndSource(sink, source)
    .joinMat(KillSwitches.singleBidi[String, String])(Keep.right)
    .backpressureTimeout(3.seconds)
```

The resulting Flow now has a type of `Flow[String,String,UniqueKillSwitch]` representing a publish-subscribe channel which can be used any number of times to attach new producers or consumers. In addition, it materializes to a `UniqueKillSwitch` (see *UniqueKillSwitch*) that can be used to deregister a single user externally:

```
val switch: UniqueKillSwitch =
  Source.repeat("Hello world!")
    .viaMat(busFlow)(Keep.right)
    .to(Sink.foreach(println))
    .run()

// Shut down externally
switch.shutdown()
```

7.10 Custom stream processing

While the processing vocabulary of Akka Streams is quite rich (see the *Streams Cookbook* for examples) it is sometimes necessary to define new transformation stages either because some functionality is missing from the stock operations, or for performance reasons. In this part we show how to build custom processing stages and graph junctions of various kinds.

Note: A custom graph stage should not be the first tool you reach for, defining graphs using flows and the graph DSL is in general easier and does to a larger extent protect you from mistakes that might be easy to make with a custom `GraphStage`

7.10.1 Custom processing with GraphStage

The `GraphStage` abstraction can be used to create arbitrary graph processing stages with any number of input or output ports. It is a counterpart of the `GraphDSL.create()` method which creates new stream processing stages by composing others. Where `GraphStage` differs is that it creates a stage that is itself not divisible into smaller ones, and allows state to be maintained inside it in a safe way.

As a first motivating example, we will build a new `Source` that will simply emit numbers from 1 until it is cancelled. To start, we need to define the "interface" of our stage, which is called *shape* in Akka Streams terminology (this is explained in more detail in the section *Modularity, Composition and Hierarchy*). This is how this looks like:

```
import akka.stream.SourceShape
import akka.stream.stage.GraphStage

class NumbersSource extends GraphStage[SourceShape[Int]] {
  // Define the (sole) output port of this stage
  val out: Outlet[Int] = Outlet("NumbersSource")
  // Define the shape of this stage, which is SourceShape with the port we defined␣
↪above
  override val shape: SourceShape[Int] = SourceShape(out)

  // This is where the actual (possibly stateful) logic will live
  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = ???
}
```

As you see, in itself the `GraphStage` only defines the ports of this stage and a shape that contains the ports. It also has, a currently unimplemented method called `createLogic`. If you recall, stages are reusable in multiple materializations, each resulting in a different executing entity. In the case of `GraphStage` the actual running logic is modeled as an instance of a `GraphStageLogic` which will be created by the materializer by calling the `createLogic` method. In other words, all we need to do is to create a suitable logic that will emit the numbers we want.

Note: It is very important to keep the GraphStage object itself immutable and reusable. All mutable state needs to be confined to the GraphStageLogic that is created for every materialization.

In order to emit from a `Source` in a backpressured stream one needs first to have demand from downstream. To receive the necessary events one needs to register a subclass of `OutHandler` with the output port (`Outlet`). This handler will receive events related to the lifecycle of the port. In our case we need to override `onPull()` which indicates that we are free to emit a single element. There is another callback, `onDownstreamFinish()` which is called if the downstream cancelled. Since the default behavior of that callback is to stop the stage, we don't need to override it. In the `onPull` callback we will simply emit the next number. This is how it looks like in the end:

```
import akka.stream.SourceShape
import akka.stream.Graph
import akka.stream.stage.GraphStage
import akka.stream.stage.OutHandler

class NumbersSource extends GraphStage[SourceShape[Int]] {
  val out: Outlet[Int] = Outlet("NumbersSource")
  override val shape: SourceShape[Int] = SourceShape(out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // All state MUST be inside the GraphStageLogic,
      // never inside the enclosing GraphStage.
      // This state is safe to access and modify from all the
      // callbacks that are provided by GraphStageLogic and the
      // registered handlers.
      private var counter = 1

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          push(out, counter)
          counter += 1
        }
      })
    }
}
```

Instances of the above `GraphStage` are subclasses of `Graph[SourceShape[Int],Unit]` which means

that they are already usable in many situations, but do not provide the DSL methods we usually have for other `Source` s. In order to convert this `Graph` to a proper `Source` we need to wrap it using `Source.fromGraph` (see *Modularity, Composition and Hierarchy* for more details about graphs and DSLs). Now we can use the source as any other built-in one:

```
// A GraphStage is a proper Graph, just like what GraphDSL.create would return
val sourceGraph: Graph[SourceShape[Int], NotUsed] = new NumbersSource

// Create a Source from the Graph to access the DSL
val mySource: Source[Int, NotUsed] = Source.fromGraph(sourceGraph)

// Returns 55
val result1: Future[Int] = mySource.take(10).runFold(0)(_ + _)

// The source is reusable. This returns 5050
val result2: Future[Int] = mySource.take(100).runFold(0)(_ + _)
```

Similarly, to create a custom `Sink` one can register a subclass `InHandler` with the stage `Inlet`. The `onPush()` callback is used to signal the handler a new element has been pushed to the stage, and can hence be grabbed and used. `onPush()` can be overridden to provide custom behaviour. Please note, most Sinks would need to request upstream elements as soon as they are created: this can be done by calling `pull(inlet)` in the `preStart()` callback.

```
import akka.stream.SinkShape
import akka.stream.stage.GraphStage
import akka.stream.stage.InHandler

class StdoutSink extends GraphStage[SinkShape[Int]] {
  val in: Inlet[Int] = Inlet("StdoutSink")
  override val shape: SinkShape[Int] = SinkShape(in)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      // This requests one element at the Sink startup.
      override def preStart(): Unit = pull(in)

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          println(grab(in))
          pull(in)
        }
      })
    }
}
```

Port states, InHandler and OutHandler

In order to interact with a port (`Inlet` or `Outlet`) of the stage we need to be able to receive events and generate new events belonging to the port. From the `GraphStageLogic` the following operations are available on an output port:

- `push(out,elem)` pushes an element to the output port. Only possible after the port has been pulled by downstream.
- `complete(out)` closes the output port normally.
- `fail(out,exception)` closes the port with a failure signal.

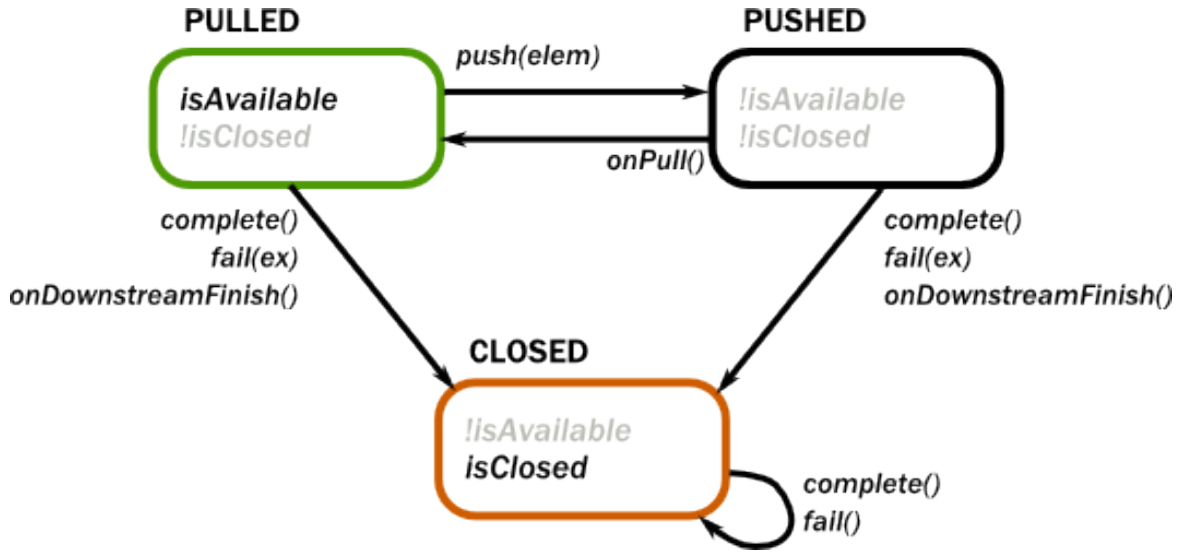
The events corresponding to an *output* port can be received in an `OutHandler` instance registered to the output port using `setHandler(out,handler)`. This handler has two callbacks:

- `onPull()` is called when the output port is ready to emit the next element, `push(out,elem)` is now allowed to be called on this port.
- `onDownstreamFinish()` is called once the downstream has cancelled and no longer allows messages to be pushed to it. No more `onPull()` will arrive after this event. If not overridden this will default to stopping the stage.

Also, there are two query methods available for output ports:

- `isAvailable(out)` returns true if the port can be pushed
- `isClosed(out)` returns true if the port is closed. At this point the port can not be pushed and will not be pulled anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.



The following operations are available for *input* ports:

- `pull(in)` requests a new element from an input port. This is only possible after the port has been pushed by upstream.
- `grab(in)` acquires the element that has been received during an `onPush()`. It cannot be called again until the port is pushed again by the upstream.
- `cancel(in)` closes the input port.

The events corresponding to an *input* port can be received in an `InHandler` instance registered to the input port using `setHandler(in,handler)`. This handler has three callbacks:

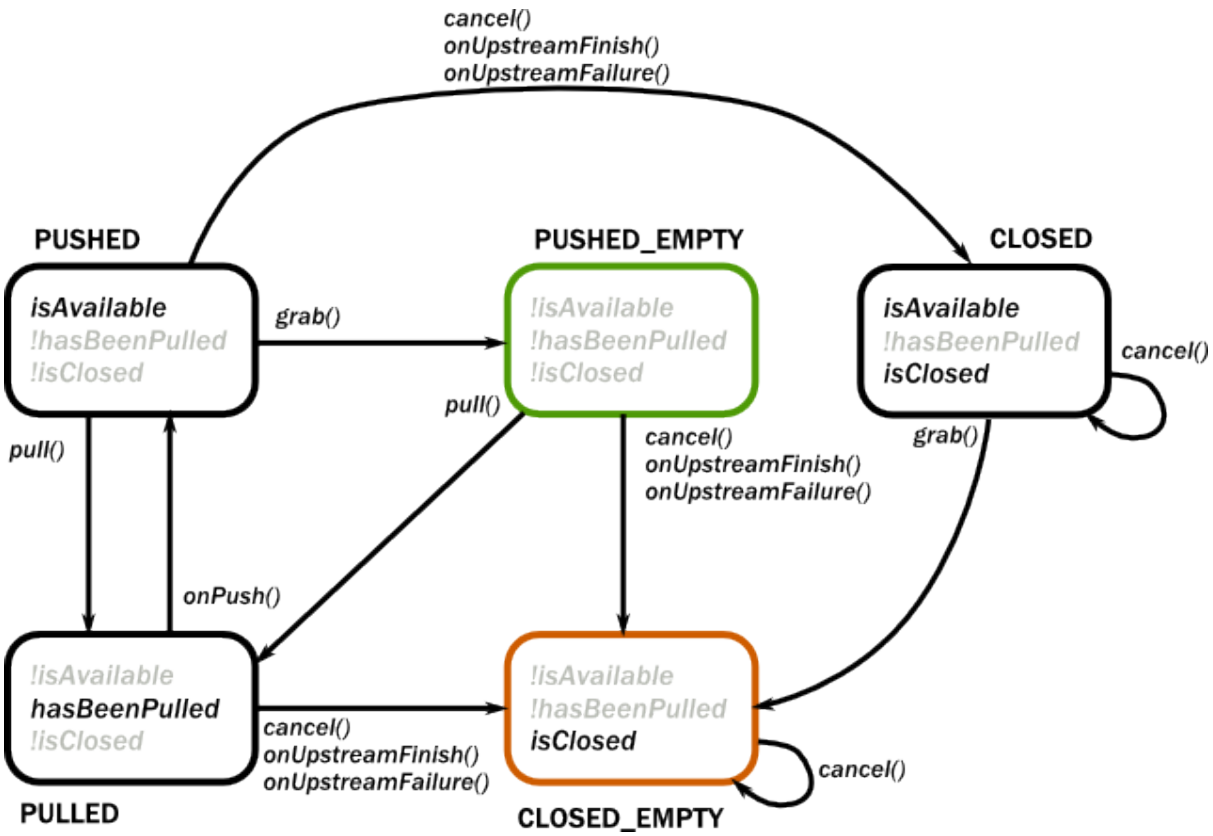
- `onPush()` is called when the input port has now a new element. Now it is possible to acquire this element using `grab(in)` and/or call `pull(in)` on the port to request the next element. It is not mandatory to grab the element, but if it is pulled while the element has not been grabbed it will drop the buffered element.

- `onUpstreamFinish()` is called once the upstream has completed and no longer can be pulled for new elements. No more `onPush()` will arrive after this event. If not overridden this will default to stopping the stage.
- `onUpstreamFailure()` is called if the upstream failed with an exception and no longer can be pulled for new elements. No more `onPush()` will arrive after this event. If not overridden this will default to failing the stage.

Also, there are three query methods available for input ports:

- `isAvailable(in)` returns true if the port can be grabbed.
- `hasBeenPulled(in)` returns true if the port has been already pulled. Calling `pull(in)` in this state is illegal.
- `isClosed(in)` returns true if the port is closed. At this point the port can not be pulled and will not be pushed anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.



Finally, there are two methods available for convenience to complete the stage and all of its ports:

- `completeStage()` is equivalent to closing all output ports and cancelling all input ports.

- `failStage(exception)` is equivalent to failing all output ports and cancelling all input ports.

In some cases it is inconvenient and error prone to react on the regular state machine events with the signal based API described above. For those cases there is an API which allows for a more declarative sequencing of actions which will greatly simplify some use cases at the cost of some extra allocations. The difference between the two APIs could be described as that the first one is signal driven from the outside, while this API is more active and drives its surroundings.

The operations of this part of the :class:`GraphStage` API are:

- `emit(out,elem)` and `emitMultiple(out,Iterable(elem1,elem2))` replaces the `OutHandler` with a handler that emits one or more elements when there is demand, and then reinstalls the current handlers
- `read(in)(andThen)` and `readN(in,n)(andThen)` replaces the `InHandler` with a handler that reads one or more elements as they are pushed and allows the handler to react once the requested number of elements has been read.
- `abortEmitting()` and `abortReading()` which will cancel an ongoing emit or read

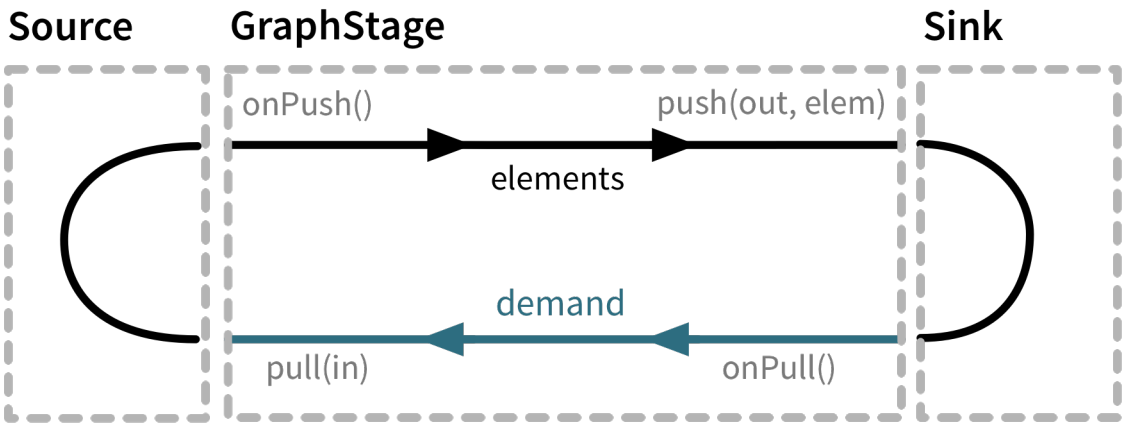
Note that since the above methods are implemented by temporarily replacing the handlers of the stage you should never call `setHandler` while they are running `emit` or `read` as that interferes with how they are implemented. The following methods are safe to call after invoking `emit` and `read` (and will lead to actually running the operation when those are done): `complete(out)`, `completeStage()`, `emit`, `emitMultiple`, `abortEmitting()` and `abortReading()`

An example of how this API simplifies a stage can be found below in the second version of the :class:`Duplicator`.

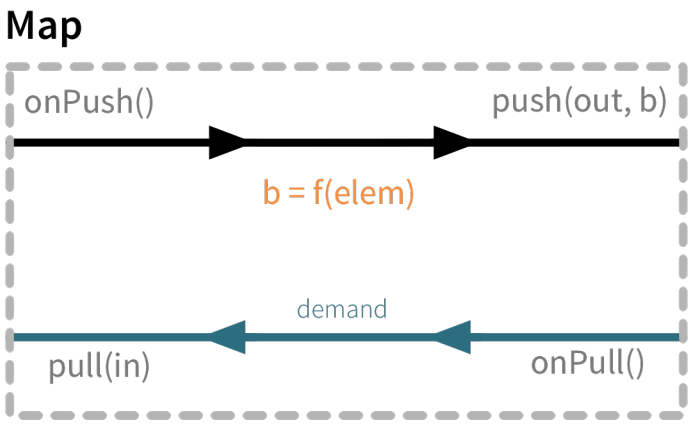
Custom linear processing stages using GraphStage

Graph stages allows for custom linear processing stages through letting them have one input and one output and using `FlowShape` as their shape.

Such a stage can be illustrated as a box with two flows as it is seen in the illustration below. Demand flowing upstream leading to elements flowing downstream.



To illustrate these concepts we create a small `GraphStage` that implements the `map` transformation.

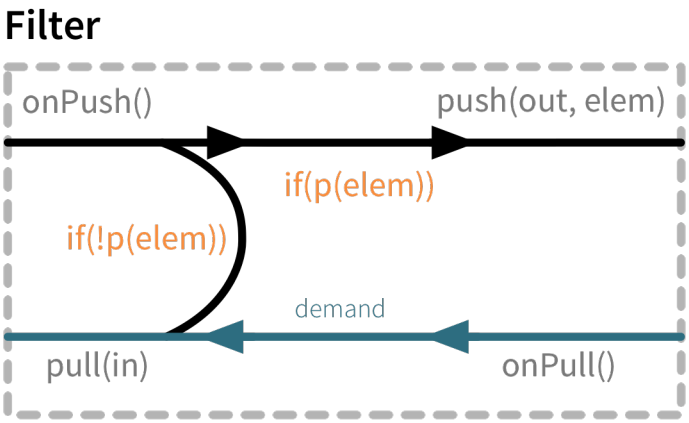


Map calls `push(out)` from the `onPush()` handler and it also calls `pull()` from the `onPull` handler resulting in the conceptual wiring above, and fully expressed in code below:

```
class Map[A, B](f: A => B) extends GraphStage[FlowShape[A, B]] {

  val in = Inlet[A]("Map.in")
  val out = Outlet[B]("Map.out")

  override val shape = FlowShape.of(in, out)

  override def createLogic(attr: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          push(out, f(grab(in)))
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```

Map is a typical example of a one-to-one transformation of a stream where demand is passed along upstream elements passed on downstream.

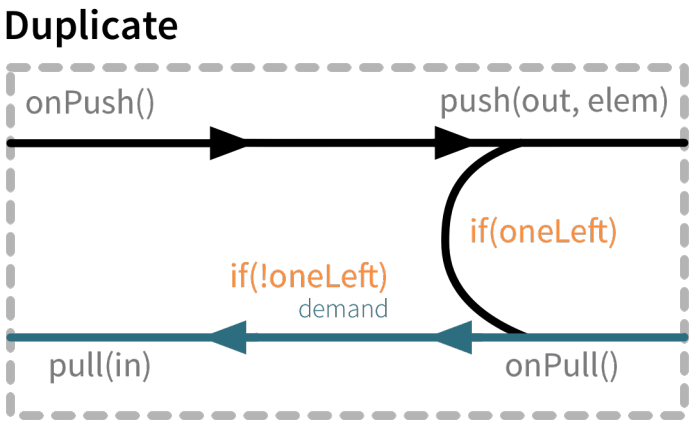
To demonstrate a many-to-one stage we will implement filter. The conceptual wiring of `Filter` looks like this:



As we see above, if the given predicate matches the current element we are propagating it downwards, otherwise we return the "ball" to our upstream so that we get the new element. This is achieved by modifying the map example by adding a conditional in the `onPush` handler and decide between a `pull(in)` or `push(out)` call (and of course not having a mapping `f` function).

```
class Filter[A](p: A => Boolean) extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("Filter.in")
  val out = Outlet[A]("Filter.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          if (p(elem)) push(out, elem)
          else pull(in)
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```

To complete the picture we define a one-to-many transformation as the next step. We chose a straightforward example stage that emits every upstream element twice downstream. The conceptual wiring of this stage looks like this:



This is a stage that has state: an option with the last element it has seen indicating if it has duplicated this last element already or not. We must also make sure to emit the extra element if the upstream completes.

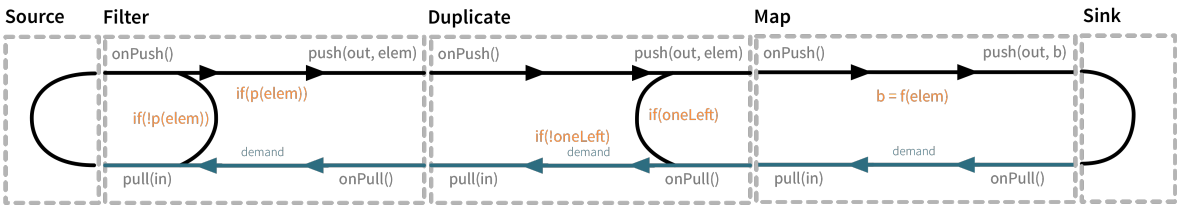
```
class Duplicator[A] extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("Duplicator.in")
  val out = Outlet[A]("Duplicator.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // Again: note that all mutable state
      // MUST be inside the GraphStageLogic
      var lastElem: Option[A] = None

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          lastElem = Some(elem)
          push(out, elem)
        }

        override def onUpstreamFinish(): Unit = {
          if (lastElem.isDefined) emit(out, lastElem.get)
          complete(out)
        }

      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (lastElem.isDefined) {
            push(out, lastElem.get)
            lastElem = None
          } else {
            pull(in)
          }
        }
      })
    }
}
```

In this case a pull from downstream might be consumed by the stage itself rather than passed along upstream as the stage might contain an element it wants to push. Note that we also need to handle the case where the upstream closes while the stage still has elements it wants to push downstream. This is done by overriding *onUpstreamFinish* in the *InHandler* and provide custom logic that should happen when the upstream has been finished.

This example can be simplified by replacing the usage of a mutable state with calls to `emitMultiple` which will replace the handlers, emit each of multiple elements and then reinstate the original handlers:

```
class Duplicator[A] extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("Duplicator.in")
  val out = Outlet[A]("Duplicator.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          // this will temporarily suspend this handler until the two elems
          // are emitted and then reinstates it
          emitMultiple(out, Iterable(elem, elem))
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```

Finally, to demonstrate all of the stages above, we put them together into a processing chain, which conceptually would correspond to the following structure:

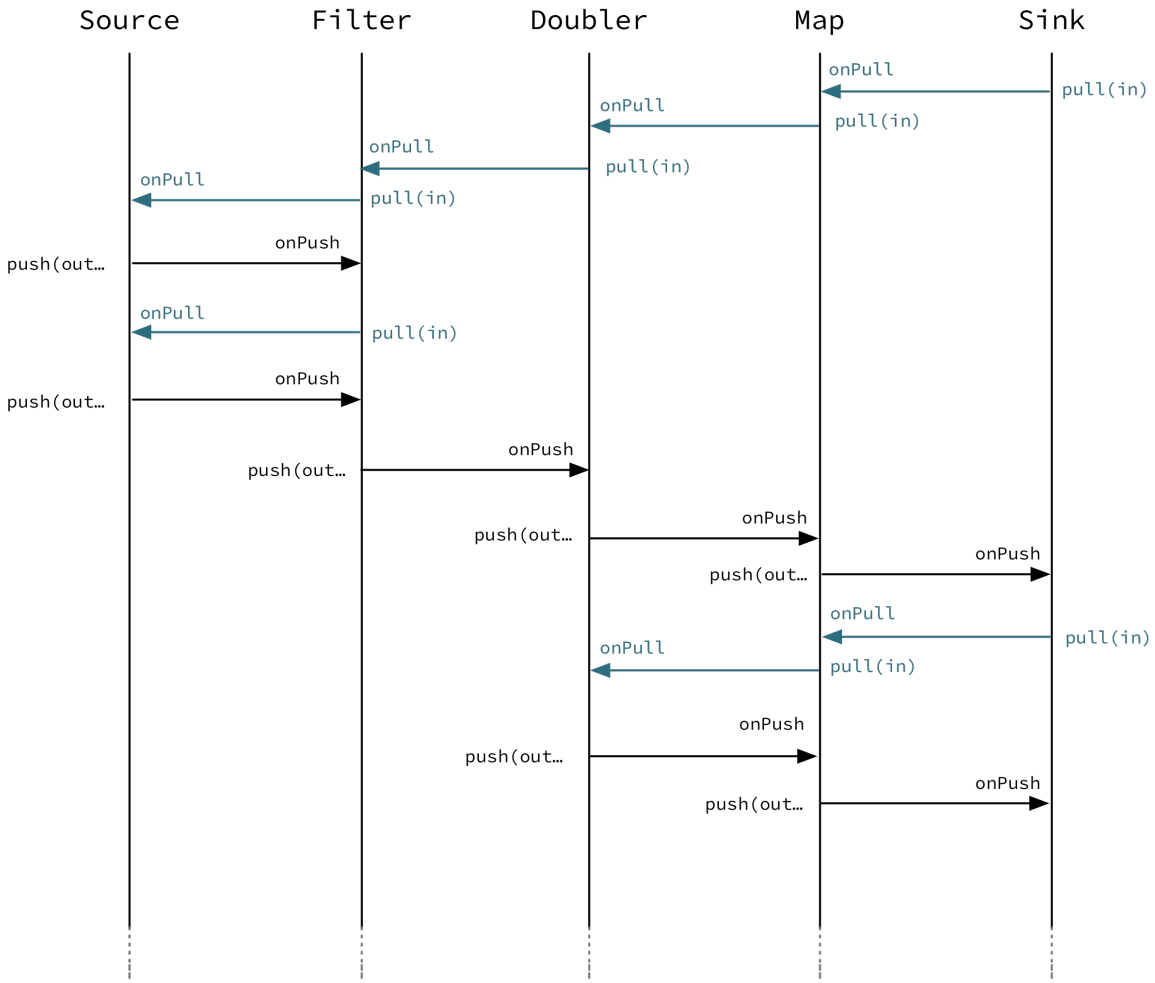


In code this is only a few lines, using the `via` use our custom stages in a stream:

```
val resultFuture = Source(1 to 5)
  .via(new Filter(_ % 2 == 0))
  .via(new Duplicator())
```

```
  .via(new Map(_ / 2))
  .runWith(sink)
```

If we attempt to draw the sequence of events, it shows that there is one "event token" in circulation in a potential chain of stages, just like our conceptual "railroad tracks" representation predicts.



Completion

Completion handling usually (but not exclusively) comes into the picture when processing stages need to emit a few more elements after their upstream source has been completed. We have seen an example of this in our first `Duplicator` implementation where the last element needs to be doubled even after the upstream neighbor stage has been completed. This can be done by overriding the `onUpstreamFinish` method in `InHandler`.

Stages by default automatically stop once all of their ports (input and output) have been closed externally or internally. It is possible to opt out from this behavior by invoking `setKeepGoing(true)` (which is not supported from the stage's constructor and usually done in `preStart`). In this case the stage **must** be explicitly closed by

calling `completeStage()` or `failStage(exception)`. This feature carries the risk of leaking streams and actors, therefore it should be used with care.

Logging inside GraphStages

Logging debug or other important information in your stages is often a very good idea, especially when developing more advances stages which may need to be debugged at some point.

The helper trait `akka.stream.stage.StageLogging` is provided to enable you to easily obtain a `LoggingAdapter` inside of a `GraphStage` as long as the `Materializer` you're using is able to provide you with a logger. In that sense, it serves a very similar purpose as `ActorLogging` does for Actors.

Note: Please note that you can always simply use a logging library directly inside a Stage. Make sure to use an asynchronous appender however, to not accidentally block the stage when writing to files etc. See *Using the SLF4J API directly* for more details on setting up async appenders in SLF4J.

The stage then gets access to the `log` field which it can safely use from any `GraphStage` callbacks:

```
final class RandomLettersSource extends GraphStage[SourceShape[String]] {
  val out = Outlet[String]("RandomLettersSource.out")
  override val shape: SourceShape[String] = SourceShape(out)

  override def createLogic(inheritedAttributes: Attributes) =
    new GraphStageLogic(shape) with StageLogging {
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          val c = nextChar() // ASCII lower case letters

          // `log` is obtained from materializer automatically (via StageLogging)
          log.debug("Randomly generated: [{}]", c)

          push(out, c.toString)
        }
      })
    }

  def nextChar(): Char =
    ThreadLocalRandom.current().nextInt('a', 'z'.toInt + 1).toChar
}
```

Note: **SPI Note:** If you're implementing a Materializer, you can add this ability to your materializer by implementing `MaterializerLoggingProvider` in your `Materializer`.

Using timers

It is possible to use timers in `GraphStages` by using `TimerGraphStageLogic` as the base class for the returned logic. Timers can be scheduled by calling one of `scheduleOnce(key,delay)`, `schedulePeriodically(key,period)` or `schedulePeriodicallyWithInitialDelay(key,delay,period)` and passing an object as a key for that timer (can be any object, for example a `String`). The `onTimer(key)` method needs to be overridden and it will be called once the timer of `key` fires. It is possible to cancel a timer using `cancelTimer(key)` and check the status of a timer with `isTimerActive(key)`. Timers will be automatically cleaned up when the stage completes.

Timers can not be scheduled from the constructor of the logic, but it is possible to schedule them from the `preStart()` lifecycle hook.

In this sample the stage toggles between open and closed, where open means no elements are passed through. The stage starts out as closed but as soon as an element is pushed downstream the gate becomes open for a duration of time during which it will consume and drop upstream messages:

```
// each time an event is pushed through it will trigger a period of silence
class TimedGate[A](silencePeriod: FiniteDuration) extends GraphStage[FlowShape[A,␣
↪A]] {

  val in = Inlet[A]("TimedGate.in")
  val out = Outlet[A]("TimedGate.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new TimerGraphStageLogic(shape) {

      var open = false

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          if (open) pull(in)
          else {
            push(out, elem)
            open = true
            scheduleOnce(None, silencePeriod)
          }
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = { pull(in) }
      })

      override protected def onTimer(timerKey: Any): Unit = {
        open = false
      }
    }
}
```

Using asynchronous side-channels

In order to receive asynchronous events that are not arriving as stream elements (for example a completion of a future or a callback from a 3rd party API) one must acquire a `AsyncCallback` by calling `getAsyncCallback()` from the stage logic. The method `getAsyncCallback` takes as a parameter a callback that will be called once the asynchronous event fires. It is important to **not call the callback directly**, instead, the external API must call the `invoke(event)` method on the returned `AsyncCallback`. The execution engine will take care of calling the provided callback in a thread-safe way. The callback can safely access the state of the `GraphStageLogic` implementation.

Sharing the AsyncCallback from the constructor risks race conditions, therefore it is recommended to use the `preStart()` lifecycle hook instead.

This example shows an asynchronous side channel graph stage that starts dropping elements when a future completes:

```
// will close upstream in all materializations of the graph stage instance
// when the future completes
class KillSwitch[A](switch: Future[Unit]) extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("KillSwitch.in")
  val out = Outlet[A]("KillSwitch.out")
```

```
  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      override def preStart(): Unit = {
        val callback = getAsyncCallback[Unit] { (_) =>
          completeStage()
        }
        switch.foreach(callback.invoke)
      }

      setHandler(in, new InHandler {
        override def onPush(): Unit = { push(out, grab(in)) }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = { pull(in) }
      })
    }
}
```

Integration with actors

This section is a stub and will be extended in the next release This is an experimental feature*

It is possible to acquire an ActorRef that can be addressed from the outside of the stage, similarly how `AsyncCallback` allows injecting asynchronous events into a stage logic. This reference can be obtained by calling `getStageActorRef(receive)` passing in a function that takes a `Pair` of the sender `ActorRef` and the received message. This reference can be used to watch other actors by calling its `watch(ref)` or `unwatch(ref)` methods. The reference can be also watched by external actors. The current limitations of this `ActorRef` are:

- they are not location transparent, they cannot be accessed via remoting.
- they cannot be returned as materialized values.
- they cannot be accessed from the constructor of the `GraphStageLogic`, but they can be accessed from the `preStart()` method.

Custom materialized values

Custom stages can return materialized values instead of `Unit` by inheriting from `GraphStageWithMaterializedValue` instead of the simpler `GraphStage`. The difference is that in this case the method `createLogicAndMaterializedValue(inheritedAttributes)` needs to be overridden, and in addition to the stage logic the materialized value must be provided

> **Warning:** There is no built-in synchronization of accessing this value from both of the thread where the logic runs and the thread that got hold of the materialized value. It is the responsibility of the programmer to add the necessary (non-blocking) synchronization and visibility guarantees to this shared object.

In this sample the materialized value is a future containing the first element to go through the stream:

```
class FirstValue[A] extends GraphStageWithMaterializedValue[FlowShape[A, A],␣
↪Future[A]] {

  val in = Inlet[A]("FirstValue.in")
  val out = Outlet[A]("FirstValue.out")
```

```
  val shape = FlowShape.of(in, out)

  override def createLogicAndMaterializedValue(inheritedAttributes: Attributes):␣
↪(GraphStageLogic, Future[A]) = {
    val promise = Promise[A]()
    val logic = new GraphStageLogic(shape) {

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          promise.success(elem)
          push(out, elem)

          // replace handler with one just forwarding
          setHandler(in, new InHandler {
            override def onPush(): Unit = {
              push(out, grab(in))
            }
          })
        }
      })

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })

    }

    (logic, promise.future)
  }
}
```

Using attributes to affect the behavior of a stage

This section is a stub and will be extended in the next release

Stages can access the `Attributes` object created by the materializer. This contains all the applied (inherited) attributes applying to the stage, ordered from least specific (outermost) towards the most specific (innermost) attribute. It is the responsibility of the stage to decide how to reconcile this inheritance chain to a final effective decision.

See *Modularity, Composition and Hierarchy* for an explanation on how attributes work.

Rate decoupled graph stages

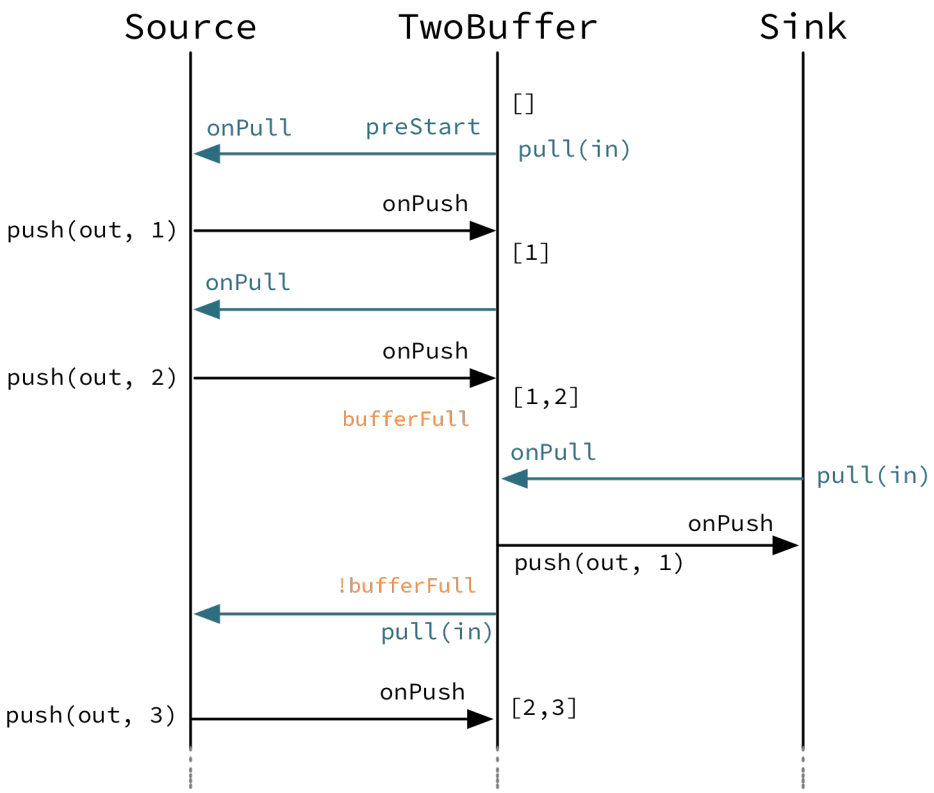
Sometimes it is desirable to *decouple* the rate of the upstream and downstream of a stage, synchronizing only when needed.

This is achieved in the model by representing a `GraphStage` as a *boundary* between two regions where the demand sent upstream is decoupled from the demand that arrives from downstream. One immediate consequence of this difference is that an `onPush` call does not always lead to calling `push` and an `onPull` call does not always lead to calling `pull`.

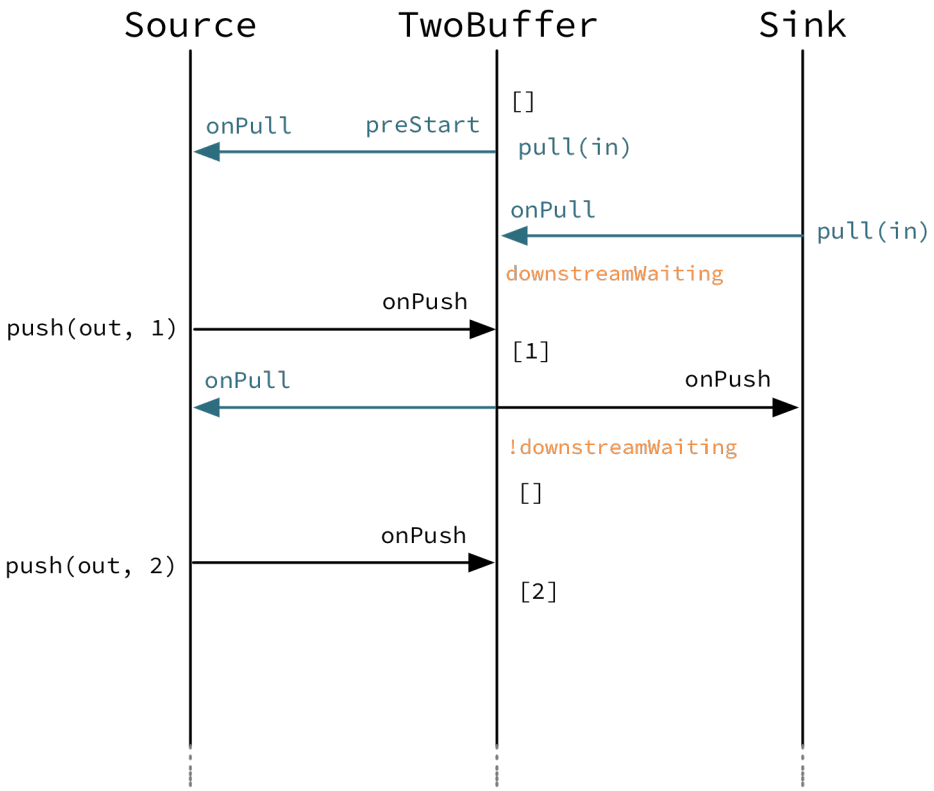
One of the important use-case for this is to build buffer-like entities, that allow independent progress of upstream and downstream stages when the buffer is not full or empty, and slowing down the appropriate side if the buffer becomes empty or full.

The next diagram illustrates the event sequence for a buffer with capacity of two elements in a setting where the downstream demand is slow to start and the buffer will fill up with upstream elements before any demand is seen

from downstream.



Another scenario would be where the demand from downstream starts coming in before any element is pushed into the buffer stage.



The first difference we can notice is that our `Buffer` stage is automatically pulling its upstream on initialization. The buffer has demand for up to two elements without any downstream demand.

The following code example demonstrates a buffer class corresponding to the message sequence chart above.

```
class TwoBuffer[A] extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("TwoBuffer.in")
  val out = Outlet[A]("TwoBuffer.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      val buffer = mutable.Queue[A]()
      def bufferFull = buffer.size == 2
      var downstreamWaiting = false

      override def preStart(): Unit = {
        // a detached stage needs to start upstream demand
        // itself as it is not triggered by downstream demand
        pull(in)
      }

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          buffer.enqueue(elem)
```

```
          if (downstreamWaiting) {
            downstreamWaiting = false
            val bufferedElem = buffer.dequeue()
            push(out, bufferedElem)
          }
          if (!bufferFull) {
            pull(in)
          }
        }

        override def onUpstreamFinish(): Unit = {
          if (buffer.nonEmpty) {
            // emit the rest if possible
            emitMultiple(out, buffer.toIterator)
          }
          completeStage()
        }
      })

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (buffer.isEmpty) {
            downstreamWaiting = true
          } else {
            val elem = buffer.dequeue
            push(out, elem)
          }
          if (!bufferFull && !hasBeenPulled(in)) {
            pull(in)
          }
        }
      })
    }

}
```

7.10.2 Thread safety of custom processing stages

All of the above custom stages (linear or graph) provide a few simple guarantees that implementors can rely on.

- The callbacks exposed by all of these classes are never called concurrently.
- The state encapsulated by these classes can be safely modified from the provided callbacks, without any further synchronization.

In essence, the above guarantees are similar to what `Actor` s provide, if one thinks of the state of a custom stage as state of an actor, and the callbacks as the `receive` block of the actor.

> **Warning:** It is **not safe** to access the state of any custom stage outside of the callbacks that it provides, just like it is unsafe to access the state of an actor from the outside. This means that Future callbacks should **not close over** internal state of custom stages because such access can be concurrent with the provided callbacks, leading to undefined behavior.

7.10.3 Extending Flow Combinators with Custom Operators

The most general way of extending any `Source`, `Flow` or `SubFlow` (e.g. from `groupBy`) is demonstrated above: create a graph of flow-shape like the `Duplicator` example given above and use the `.via(...)` com-

binator to integrate it into your stream topology. This works with all `FlowOps` sub-types, including the ports that you connect with the graph DSL.

Advanced Scala users may wonder whether it is possible to write extension methods that enrich `FlowOps` to allow nicer syntax. The short answer is that Scala 2 does not support this in a fully generic fashion, the problem is that it is impossible to abstract over the kind of stream that is being extended because `Source`, `Flow` and `SubFlow` differ in the number and kind of their type parameters. While it would be possible to write an implicit class that enriches them generically, this class would require explicit instantiation with all type parameters due to SI-2712. For a partial workaround that unifies extensions to `Source` and `Flow` see this sketch by R. Kuhn.

A lot simpler is the task of just adding an extension method to `Source` as shown below:

```
implicit class SourceDuplicator[Out, Mat](s: Source[Out, Mat]) {
  def duplicateElements: Source[Out, Mat] = s.via(new Duplicator)
}

val s = Source(1 to 3).duplicateElements

s.runWith(Sink.seq).futureValue should ===(Seq(1, 1, 2, 2, 3, 3))
```

The analog works for `Flow` as well:

```
implicit class FlowDuplicator[In, Out, Mat](s: Flow[In, Out, Mat]) {
  def duplicateElements: Flow[In, Out, Mat] = s.via(new Duplicator)
}

val f = Flow[Int].duplicateElements

Source(1 to 3).via(f).runWith(Sink.seq).futureValue should ===(Seq(1, 1, 2, 2, 3,␣
↪3))
```

If you try to write this for `SubFlow`, though, you will run into the same issue as when trying to unify the two solutions above, only on a higher level (the type constructors needed for that unification would have rank two, meaning that some of their type arguments are type constructors themselves—when trying to extend the solution shown in the linked sketch the author encountered such a density of compiler StackOverflowErrors and IDE failures that he gave up).

It is interesting to note that a simplified form of this problem has found its way into the dotty test suite. Dotty is the development version of Scala on its way to Scala 3.

7.11 Integration

7.11.1 Integrating with Actors

For piping the elements of a stream as messages to an ordinary actor you can use `ask` in a `mapAsync` or use `Sink.actorRefWithAck`.

Messages can be sent to a stream with `Source.queue` or via the `ActorRef` that is materialized by `Source.actorRef`.

mapAsync + ask

A nice way to delegate some processing of elements in a stream to an actor is to use `ask` in `mapAsync`. The back-pressure of the stream is maintained by the `Future` of the `ask` and the mailbox of the actor will not be filled with more messages than the given `parallelism` of the `mapAsync` stage.

```
import akka.pattern.ask
implicit val askTimeout = Timeout(5.seconds)
val words: Source[String, NotUsed] =
```

```
  Source(List("hello", "hi"))

words
  .mapAsync(parallelism = 5)(elem => (ref ? elem).mapTo[String])
  // continue processing of the replies from the actor
  .map(_.toLowerCase)
  .runWith(Sink.ignore)
```

Note that the messages received in the actor will be in the same order as the stream elements, i.e. the `parallelism` does not change the ordering of the messages. There is a performance advantage of using parallelism > 1 even though the actor will only process one message at a time because then there is already a message in the mailbox when the actor has completed previous message.

The actor must reply to the `sender()` for each message from the stream. That reply will complete the `Future` of the `ask` and it will be the element that is emitted downstreams from `mapAsync`.

```
class Translator extends Actor {
  def receive = {
    case word: String =>
      // ... process message
      val reply = word.toUpperCase
      sender() ! reply // reply to the ask
  }
}
```

The stream can be completed with failure by sending `akka.actor.Status.Failure` as reply from the actor.

If the `ask` fails due to timeout the stream will be completed with `TimeoutException` failure. If that is not desired outcome you can use `recover` on the `ask` `Future`.

If you don't care about the reply values and only use them as back-pressure signals you can use `Sink.ignore` after the `mapAsync` stage and then actor is effectively a sink of the stream.

The same pattern can be used with *Actor routers*. Then you can use `mapAsyncUnordered` for better efficiency if you don't care about the order of the emitted downstream elements (the replies).

Sink.actorRefWithAck

The sink sends the elements of the stream to the given `ActorRef` that sends back back-pressure signal. First element is always *onInitMessage*, then stream is waiting for the given acknowledgement message from the given actor which means that it is ready to process elements. It also requires the given acknowledgement message after each stream element to make back-pressure work.

If the target actor terminates the stream will be cancelled. When the stream is completed successfully the given `onCompleteMessage` will be sent to the destination actor. When the stream is completed with failure a `akka.actor.Status.Failure` message will be sent to the destination actor.

Note: Using `Sink.actorRef` or ordinary `tell` from a `map` or `foreach` stage means that there is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow, unless you use a bounded mailbox with zero *mailbox-push-timeout-time* or use a rate limiting stage in front. It's often better to use `Sink.actorRefWithAck` or `ask` in `mapAsync`, though.

Source.queue

`Source.queue` can be used for emitting elements to a stream from an actor (or from anything running outside the stream). The elements will be buffered until the stream can process them. You can `offer` elements to the queue and they will be emitted to the stream if there is demand from downstream, otherwise they will be buffered until request for demand is received.

Use overflow strategy `akka.stream.OverflowStrategy.backpressure` to avoid dropping of elements if the buffer is full.

`SourceQueue.offer` returns `Future[QueueOfferResult]` which completes with `QueueOfferResult.Enqueued` if element was added to buffer or sent downstream. It completes with `QueueOfferResult.Dropped` if element was dropped. Can also complete with `QueueOfferResult.Failure` - when stream failed or `QueueOfferResult.QueueClosed` when downstream is completed.

When used from an actor you typically `pipe` the result of the `Future` back to the actor to continue processing.

Source.actorRef

Messages sent to the actor that is materialized by `Source.actorRef` will be emitted to the stream if there is demand from downstream, otherwise they will be buffered until request for demand is received.

Depending on the defined `OverflowStrategy` it might drop elements if there is no space available in the buffer. The strategy `OverflowStrategy.backpressure` is not supported for this Source type, i.e. elements will be dropped if the buffer is filled by sending at a rate that is faster than the stream can consume. You should consider using `Source.queue` if you want a backpressured actor interface.

The stream can be completed successfully by sending `akka.actor.PoisonPill` or `akka.actor.Status.Success` to the actor reference.

The stream can be completed with failure by sending `akka.actor.Status.Failure` to the actor reference.

The actor will be stopped when the stream is completed, failed or cancelled from downstream, i.e. you can watch it to get notified when that happens.

7.11.2 Integrating with External Services

Stream transformations and side effects involving external non-stream based services can be performed with `mapAsync` or `mapAsyncUnordered`.

For example, sending emails to the authors of selected tweets using an external email service:

```
def send(email: Email): Future[Unit] = {
  // ...
}
```

We start with the tweet stream of authors:

```
val authors: Source[Author, NotUsed] =
  tweets
    .filter(_.hashtags.contains(akkaTag))
    .map(_.author)
```

Assume that we can lookup their email address using:

```
def lookupEmail(handle: String): Future[Option[String]] =
```

Transforming the stream of authors to a stream of email addresses by using the `lookupEmail` service can be done with `mapAsync`:

```
val emailAddresses: Source[String, NotUsed] =
  authors
    .mapAsync(4)(author => addressSystem.lookupEmail(author.handle))
    .collect { case Some(emailAddress) => emailAddress }
```

Finally, sending the emails:

```
val sendEmails: RunnableGraph[NotUsed] =
  emailAddresses
    .mapAsync(4)(address => {
      emailServer.send(
        Email(to = address, title = "Akka", body = "I like your tweet"))
    })
    .to(Sink.ignore)

sendEmails.run()
```

`mapAsync` is applying the given function that is calling out to the external service to each of the elements as they pass through this processing step. The function returns a `Future` and the value of that future will be emitted downstreams. The number of Futures that shall run in parallel is given as the first argument to `mapAsync`. These Futures may complete in any order, but the elements that are emitted downstream are in the same order as received from upstream.

That means that back-pressure works as expected. For example if the `emailServer.send` is the bottleneck it will limit the rate at which incoming tweets are retrieved and email addresses looked up.

The final piece of this pipeline is to generate the demand that pulls the tweet authors information through the emailing pipeline: we attach a `Sink.ignore` which makes it all run. If our email process would return some interesting data for further transformation then we would of course not ignore it but send that result stream onwards for further processing or storage.

Note that `mapAsync` preserves the order of the stream elements. In this example the order is not important and then we can use the more efficient `mapAsyncUnordered`:

```
val authors: Source[Author, NotUsed] =
  tweets.filter(_.hashtags.contains(akkaTag)).map(_.author)

val emailAddresses: Source[String, NotUsed] =
  authors
    .mapAsyncUnordered(4)(author => addressSystem.lookupEmail(author.handle))
    .collect { case Some(emailAddress) => emailAddress }

val sendEmails: RunnableGraph[NotUsed] =
  emailAddresses
    .mapAsyncUnordered(4)(address => {
      emailServer.send(
        Email(to = address, title = "Akka", body = "I like your tweet"))
    })
    .to(Sink.ignore)

sendEmails.run()
```

In the above example the services conveniently returned a `Future` of the result. If that is not the case you need to wrap the call in a `Future`. If the service call involves blocking you must also make sure that you run it on a dedicated execution context, to avoid starvation and disturbance of other tasks in the system.

```
val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")

val sendTextMessages: RunnableGraph[NotUsed] =
  phoneNumbers
    .mapAsync(4)(phoneNo => {
      Future {
        smsServer.send(
          TextMessage(to = phoneNo, body = "I like your tweet"))
      }(blockingExecutionContext)
    })
    .to(Sink.ignore)

sendTextMessages.run()
```

The configuration of the "blocking-dispatcher" may look something like:

```
blocking-dispatcher {
  executor = "thread-pool-executor"
  thread-pool-executor {
    core-pool-size-min    = 10
    core-pool-size-max    = 10
  }
}
```

An alternative for blocking calls is to perform them in a map operation, still using a dedicated dispatcher for that operation.

```
val send = Flow[String]
  .map { phoneNo =>
    smsServer.send(TextMessage(to = phoneNo, body = "I like your tweet"))
  }
  .withAttributes(ActorAttributes.dispatcher("blocking-dispatcher"))
val sendTextMessages: RunnableGraph[NotUsed] =
  phoneNumbers.via(send).to(Sink.ignore)

sendTextMessages.run()
```

However, that is not exactly the same as mapAsync, since the mapAsync may run several calls concurrently, but map performs them one at a time.

For a service that is exposed as an actor, or if an actor is used as a gateway in front of an external service, you can use ask:

```
import akka.pattern.ask

val akkaTweets: Source[Tweet, NotUsed] = tweets.filter(_.hashtags.
↪contains(akkaTag))

implicit val timeout = Timeout(3.seconds)
val saveTweets: RunnableGraph[NotUsed] =
  akkaTweets
    .mapAsync(4)(tweet => database ? Save(tweet))
    .to(Sink.ignore)
```

Note that if the ask is not completed within the given timeout the stream is completed with failure. If that is not desired outcome you can use recover on the ask Future.

Illustrating ordering and parallelism

Let us look at another example to get a better understanding of the ordering and parallelism characteristics of mapAsync and mapAsyncUnordered.

Several mapAsync and mapAsyncUnordered futures may run concurrently. The number of concurrent futures are limited by the downstream demand. For example, if 5 elements have been requested by downstream there will be at most 5 futures in progress.

mapAsync emits the future results in the same order as the input elements were received. That means that completed results are only emitted downstream when earlier results have been completed and emitted. One slow call will thereby delay the results of all successive calls, even though they are completed before the slow call.

mapAsyncUnordered emits the future results as soon as they are completed, i.e. it is possible that the elements are not emitted downstream in the same order as received from upstream. One slow call will thereby not delay the results of faster successive calls as long as there is downstream demand of several elements.

Here is a fictive service that we can use to illustrate these aspects.

```
class SometimesSlowService(implicit ec: ExecutionContext) {

  private val runningCount = new AtomicInteger

  def convert(s: String): Future[String] = {
    println(s"running: $s (${runningCount.incrementAndGet()})")
    Future {
      if (s.nonEmpty && s.head.isLower)
        Thread.sleep(500)
      else
        Thread.sleep(20)
      println(s"completed: $s (${runningCount.decrementAndGet()})")
      s.toUpperCase
    }
  }
}
```

Elements starting with a lower case character are simulated to take longer time to process.

Here is how we can use it with `mapAsync`:

```
implicit val blockingExecutionContext = system.dispatchers.lookup("blocking-
↪dispatcher")
val service = new SometimesSlowService

implicit val materializer = ActorMaterializer(
  ActorMaterializerSettings(system).withInputBuffer(initialSize = 4, maxSize = 4))

Source(List("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem => { println(s"before: $elem"); elem })
  .mapAsync(4)(service.convert)
  .runForeach(elem => println(s"after: $elem"))
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
before: F
running: D (4)
before: g
before: H
completed: C (3)
completed: B (2)
completed: D (1)
completed: a (0)
after: A
after: B
running: e (1)
after: C
after: D
running: F (2)
before: i
before: J
running: g (3)
running: H (4)
completed: H (2)
completed: F (3)
```

```
completed: e (1)
completed: g (0)
after: E
after: F
running: i (1)
after: G
after: H
running: J (2)
completed: J (1)
completed: i (0)
after: I
after: J
```

Note that `after` lines are in the same order as the `before` lines even though elements are `completed` in a different order. For example `H` is `completed` before `g`, but still emitted afterwards.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorMaterializerSettings`.

Here is how we can use the same service with `mapAsyncUnordered`:

```
implicit val blockingExecutionContext = system.dispatchers.lookup("blocking-
↪dispatcher")
val service = new SometimesSlowService

implicit val materializer = ActorMaterializer(
  ActorMaterializerSettings(system).withInputBuffer(initialSize = 4, maxSize = 4))

Source(List("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem => { println(s"before: $elem"); elem })
  .mapAsyncUnordered(4)(service.convert)
  .runForeach(elem => println(s"after: $elem"))
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
before: F
running: D (4)
before: g
before: H
completed: B (3)
completed: C (1)
completed: D (2)
after: B
after: D
running: e (2)
after: C
running: F (3)
before: i
before: J
completed: F (2)
after: F
running: g (3)
running: H (4)
completed: H (3)
```

```
after: H
completed: a (2)
after: A
running: i (3)
running: J (4)
completed: J (3)
after: J
completed: e (2)
after: E
completed: g (1)
after: G
completed: i (0)
after: I
```

Note that `after` lines are not in the same order as the `before` lines. For example `H` overtakes the slow `G`.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorMaterializerSettings`.

7.11.3 Integrating with Reactive Streams

Reactive Streams defines a standard for asynchronous stream processing with non-blocking back pressure. It makes it possible to plug together stream libraries that adhere to the standard. Akka Streams is one such library.

An incomplete list of other implementations:

- Reactor (1.1+)
- RxJava
- Ratpack
- Slick

The two most important interfaces in Reactive Streams are the `Publisher` and `Subscriber`.

```
import org.reactivestreams.Publisher
import org.reactivestreams.Subscriber
```

Let us assume that a library provides a publisher of tweets:

```
def tweets: Publisher[Tweet]
```

and another library knows how to store author handles in a database:

```
def storage: Subscriber[Author]
```

Using an Akka Streams `Flow` we can transform the stream and connect those:

```
val authors = Flow[Tweet]
  .filter(_.hashtags.contains(akkaTag))
  .map(_.author)

Source.fromPublisher(tweets).via(authors).to(Sink.fromSubscriber(storage)).run()
```

The `Publisher` is used as an input `Source` to the flow and the `Subscriber` is used as an output `Sink`.

A `Flow` can also be also converted to a `RunnableGraph[Processor[In,Out]]` which materializes to a `Processor` when `run()` is called. `run()` itself can be called multiple times, resulting in a new `Processor` instance each time.

```
val processor: Processor[Tweet, Author] = authors.toProcessor.run()

tweets.subscribe(processor)
processor.subscribe(storage)
```

A publisher can be connected to a subscriber with the `subscribe` method.

It is also possible to expose a `Source` as a `Publisher` by using the Publisher-`Sink`:

```
val authorPublisher: Publisher[Author] =
  Source.fromPublisher(tweets).via(authors).runWith(Sink.asPublisher(fanout =␣
↪false))

authorPublisher.subscribe(storage)
```

A publisher that is created with `Sink.asPublisher(fanout = false)` supports only a single subscription. Additional subscription attempts will be rejected with an `IllegalStateException`.

A publisher that supports multiple subscribers using fan-out/broadcasting is created as follows:

```
def storage: Subscriber[Author]
def alert: Subscriber[Author]
```

```
val authorPublisher: Publisher[Author] =
  Source.fromPublisher(tweets).via(authors)
    .runWith(Sink.asPublisher(fanout = true))

authorPublisher.subscribe(storage)
authorPublisher.subscribe(alert)
```

The input buffer size of the stage controls how far apart the slowest subscriber can be from the fastest subscriber before slowing down the stream.

To make the picture complete, it is also possible to expose a `Sink` as a `Subscriber` by using the Subscriber-`Source`:

```
val tweetSubscriber: Subscriber[Tweet] =
  authors.to(Sink.fromSubscriber(storage)).runWith(Source.asSubscriber[Tweet])

tweets.subscribe(tweetSubscriber)
```

It is also possible to use re-wrap `Processor` instances as a `Flow` by passing a factory function that will create the `Processor` instances:

```
// An example Processor factory
def createProcessor: Processor[Int, Int] = Flow[Int].toProcessor.run()

val flow: Flow[Int, Int, NotUsed] = Flow.fromProcessor(() => createProcessor)
```

Please note that a factory is necessary to achieve reusability of the resulting `Flow`.

Implementing Reactive Streams Publisher or Subscriber

As described above any Akka Streams `Source` can be exposed as a Reactive Streams `Publisher` and any `Sink` can be exposed as a Reactive Streams `Subscriber`. Therefore we recommend that you implement Reactive Streams integrations with built-in stages or *custom stages*.

For historical reasons the `ActorPublisher` and `ActorSubscriber` traits are provided to support implementing Reactive Streams `Publisher` and `Subscriber` with an `Actor`.

These can be consumed by other Reactive Stream libraries or used as an Akka Streams `Source` or `Sink`.

Warning: `ActorPublisher` and `ActorSubscriber` will probably be deprecated in future versions of Akka.

Warning: `ActorPublisher` and `ActorSubscriber` cannot be used with remote actors, because if signals of the Reactive Streams protocol (e.g. `request`) are lost the the stream may deadlock.

ActorPublisher

Extend/mixin `akka.stream.actor.ActorPublisher` in your `Actor` to make it a stream publisher that keeps track of the subscription life cycle and requested elements.

Here is an example of such an actor. It dispatches incoming jobs to the attached subscriber:

```
object JobManager {
  def props: Props = Props[JobManager]

  final case class Job(payload: String)
  case object JobAccepted
  case object JobDenied
}

class JobManager extends ActorPublisher[JobManager.Job] {
  import akka.stream.actor.ActorPublisherMessage._
  import JobManager._

  val MaxBufferSize = 100
  var buf = Vector.empty[Job]

  def receive = {
    case job: Job if buf.size == MaxBufferSize =>
      sender() ! JobDenied
    case job: Job =>
      sender() ! JobAccepted
      if (buf.isEmpty && totalDemand > 0)
        onNext(job)
      else {
        buf :+= job
        deliverBuf()
      }
    case Request(_) =>
      deliverBuf()
    case Cancel =>
      context.stop(self)
  }

  @tailrec final def deliverBuf(): Unit =
    if (totalDemand > 0) {
      /*
       * totalDemand is a Long and could be larger than
       * what buf.splitAt can accept
       */
      if (totalDemand <= Int.MaxValue) {
        val (use, keep) = buf.splitAt(totalDemand.toInt)
        buf = keep
        use foreach onNext
      } else {
        val (use, keep) = buf.splitAt(Int.MaxValue)
        buf = keep
```

```
        use foreach onNext
        deliverBuf()
      }
    }
}
```

You send elements to the stream by calling `onNext`. You are allowed to send as many elements as have been requested by the stream subscriber. This amount can be inquired with `totalDemand`. It is only allowed to use `onNext` when `isActive` and `totalDemand>0`, otherwise `onNext` will throw `IllegalStateException`.

When the stream subscriber requests more elements the `ActorPublisherMessage.Request` message is delivered to this actor, and you can act on that event. The `totalDemand` is updated automatically.

When the stream subscriber cancels the subscription the `ActorPublisherMessage.Cancel` message is delivered to this actor. After that subsequent calls to `onNext` will be ignored.

You can complete the stream by calling `onComplete`. After that you are not allowed to call `onNext`, `onError` and `onComplete`.

You can terminate the stream with failure by calling `onError`. After that you are not allowed to call `onNext`, `onError` and `onComplete`.

If you suspect that this `ActorPublisher` may never get subscribed to, you can override the `subscriptionTimeout` method to provide a timeout after which this Publisher should be considered canceled. The actor will be notified when the timeout triggers via an `ActorPublisherMessage.SubscriptionTimeoutExceeded` message and MUST then perform cleanup and stop itself.

If the actor is stopped the stream will be completed, unless it was not already terminated with failure, completed or canceled.

More detailed information can be found in the API documentation.

This is how it can be used as input `Source` to a `Flow`:

```
val jobManagerSource = Source.actorPublisher[JobManager.Job](JobManager.props)
val ref = Flow[JobManager.Job]
  .map(_.payload.toUpperCase)
  .map { elem => println(elem); elem }
  .to(Sink.ignore)
  .runWith(jobManagerSource)

ref ! JobManager.Job("a")
ref ! JobManager.Job("b")
ref ! JobManager.Job("c")
```

A publisher that is created with `Sink.asPublisher` supports a specified number of subscribers. Additional subscription attempts will be rejected with an `IllegalStateException`.

ActorSubscriber

Extend/mixin `akka.stream.actor.ActorSubscriber` in your `Actor` to make it a stream subscriber with full control of stream back pressure. It will receive `ActorSubscriberMessage.OnNext`, `ActorSubscriberMessage.OnComplete` and `ActorSubscriberMessage.OnError` messages from the stream. It can also receive other, non-stream messages, in the same way as any actor.

Here is an example of such an actor. It dispatches incoming jobs to child worker actors:

```
object WorkerPool {
  case class Msg(id: Int, replyTo: ActorRef)
  case class Work(id: Int)
  case class Reply(id: Int)
```

```
  case class Done(id: Int)

  def props: Props = Props(new WorkerPool)
}

class WorkerPool extends ActorSubscriber {
  import WorkerPool._
  import ActorSubscriberMessage._

  val MaxQueueSize = 10
  var queue = Map.empty[Int, ActorRef]

  val router = {
    val routees = Vector.fill(3) {
      ActorRefRoutee(context.actorOf(Props[Worker]))
    }
    Router(RoundRobinRoutingLogic(), routees)
  }

  override val requestStrategy = new MaxInFlightRequestStrategy(max =␣
↪MaxQueueSize) {
    override def inFlightInternally: Int = queue.size
  }

  def receive = {
    case OnNext(Msg(id, replyTo)) =>
      queue += (id -> replyTo)
      assert(queue.size <= MaxQueueSize, s"queued too many: ${queue.size}")
      router.route(Work(id), self)
    case Reply(id) =>
      queue(id) ! Done(id)
      queue -= id
      if (canceled && queue.isEmpty) {
        context.stop(self)
      }
    case OnComplete =>
      if (queue.isEmpty) {
        context.stop(self)
      }
  }
}

class Worker extends Actor {
  import WorkerPool._
  def receive = {
    case Work(id) =>
      // ...
      sender() ! Reply(id)
  }
}
```

Subclass must define the `RequestStrategy` to control stream back pressure. After each incoming message the `ActorSubscriber` will automatically invoke the `RequestStrategy.requestDemand` and propagate the returned demand to the stream.

- The provided `WatermarkRequestStrategy` is a good strategy if the actor performs work itself.
- The provided `MaxInFlightRequestStrategy` is useful if messages are queued internally or delegated to other actors.
- You can also implement a custom `RequestStrategy` or call `request` manually together with `ZeroRequestStrategy` or some other strategy. In that case you must also call `request` when the actor is started or when it is ready, otherwise it will not receive any elements.

More detailed information can be found in the API documentation.

This is how it can be used as output `Sink` to a `Flow`:

```
val N = 117
val worker = Source(1 to N).map(WorkerPool.Msg(_, replyTo))
  .runWith(Sink.actorSubscriber(WorkerPool.props))
```

7.12 Error Handling

Strategies for how to handle exceptions from processing stream elements can be defined when materializing the stream. The error handling strategies are inspired by actor supervision strategies, but the semantics have been adapted to the domain of stream processing.

Warning: *ZipWith*, *GraphStage* junction, *ActorPublisher* source and *ActorSubscriber* sink components do not honour the supervision strategy attribute yet.

7.12.1 Supervision Strategies

There are three ways to handle exceptions from application code:

- `Stop` - The stream is completed with failure.
- `Resume` - The element is dropped and the stream continues.
- `Restart` - The element is dropped and the stream continues after restarting the stage. Restarting a stage means that any accumulated state is cleared. This is typically performed by creating a new instance of the stage.

By default the stopping strategy is used for all exceptions, i.e. the stream will be completed with failure when an exception is thrown.

```
implicit val materializer = ActorMaterializer()
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
// division by zero will fail the stream and the
// result here will be a Future completed with Failure(ArithmeticException)
```

The default supervision strategy for a stream can be defined on the settings of the materializer.

```
val decider: Supervision.Decider = {
  case _: ArithmeticException => Supervision.Resume
  case _                      => Supervision.Stop
}
implicit val materializer = ActorMaterializer(
  ActorMaterializerSettings(system).withSupervisionStrategy(decider))
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
// the element causing division by zero will be dropped
// result here will be a Future completed with Success(228)
```

Here you can see that all `ArithmeticException` will resume the processing, i.e. the elements that cause the division by zero are effectively dropped.

Note: Be aware that dropping elements may result in deadlocks in graphs with cycles, as explained in *Graph cycles, liveness and deadlocks*.

The supervision strategy can also be defined for all operators of a flow.

```
implicit val materializer = ActorMaterializer()
val decider: Supervision.Decider = {
  case _: ArithmeticException => Supervision.Resume
  case _                      => Supervision.Stop
}
val flow = Flow[Int]
  .filter(100 / _ < 50).map(elem => 100 / (5 - elem))
  .withAttributes(ActorAttributes.supervisionStrategy(decider))
val source = Source(0 to 5).via(flow)

val result = source.runWith(Sink.fold(0)(_ + _))
// the elements causing division by zero will be dropped
// result here will be a Future completed with Success(150)
```

`Restart` works in a similar way as `Resume` with the addition that accumulated state, if any, of the failing processing stage will be reset.

```
implicit val materializer = ActorMaterializer()
val decider: Supervision.Decider = {
  case _: IllegalArgumentException => Supervision.Restart
  case _                           => Supervision.Stop
}
val flow = Flow[Int]
  .scan(0) { (acc, elem) =>
    if (elem < 0) throw new IllegalArgumentException("negative not allowed")
    else acc + elem
  }
  .withAttributes(ActorAttributes.supervisionStrategy(decider))
val source = Source(List(1, 3, -1, 5, 7)).via(flow)
val result = source.limit(1000).runWith(Sink.seq)
// the negative element cause the scan stage to be restarted,
// i.e. start from 0 again
// result here will be a Future completed with Success(Vector(0, 1, 4, 0, 5, 12))
```

7.12.2 Errors from mapAsync

Stream supervision can also be applied to the futures of `mapAsync`.

Let's say that we use an external service to lookup email addresses and we would like to discard those that cannot be found.

We start with the tweet stream of authors:

```
val authors: Source[Author, NotUsed] =
  tweets
    .filter(_.hashtags.contains(akkaTag))
    .map(_.author)
```

Assume that we can lookup their email address using:

```
def lookupEmail(handle: String): Future[String] =
```

The `Future` is completed with `Failure` if the email is not found.

Transforming the stream of authors to a stream of email addresses by using the `lookupEmail` service can be done with `mapAsync` and we use `Supervision.resumingDecider` to drop unknown email addresses:

```
import ActorAttributes.supervisionStrategy
import Supervision.resumingDecider
```

```
val emailAddresses: Source[String, NotUsed] =
  authors.via(
    Flow[Author].mapAsync(4)(author => addressSystem.lookupEmail(author.handle))
      .withAttributes(supervisionStrategy(resumingDecider)))
```

If we would not use `Resume` the default stopping strategy would complete the stream with failure on the first `Future` that was completed with `Failure`.

7.13 Working with streaming IO

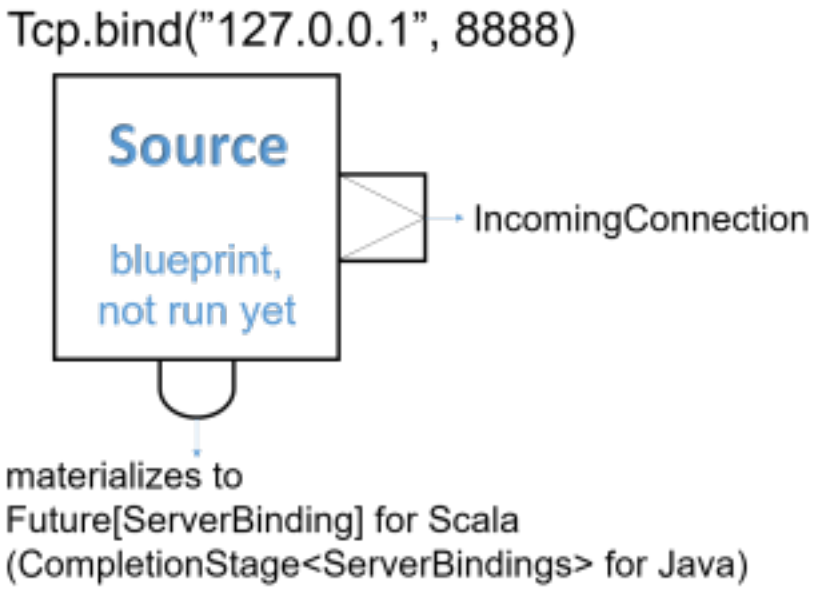
Akka Streams provides a way of handling File IO and TCP connections with Streams. While the general approach is very similar to the *Actor based TCP handling* using Akka IO, by using Akka Streams you are freed of having to manually react to back-pressure signals, as the library does it transparently for you.

7.13.1 Streaming TCP

Accepting connections: Echo Server

In order to implement a simple EchoServer we `bind` to a given address, which returns a `Source[IncomingConnection,Future[ServerBinding]]`, which will emit an `IncomingConnection` element for each new connection that the Server should handle:

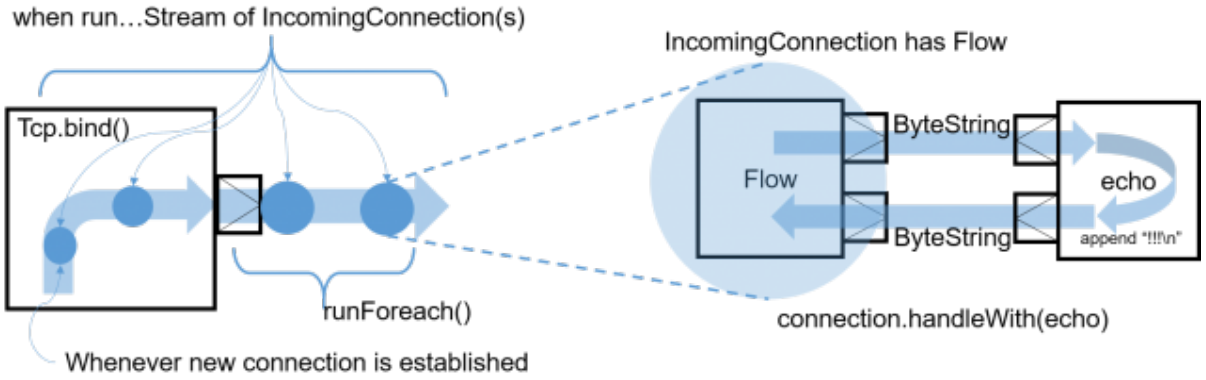
```
val binding: Future[ServerBinding] =
  Tcp().bind("127.0.0.1", 8888).to(Sink.ignore).run()

binding.map { b =>
  b.unbind() onComplete {
    case _ => // ...
  }
}
```



Next, we simply handle *each* incoming connection using a `Flow` which will be used as the processing stage to handle and emit `ByteString` s from and to the TCP Socket. Since one `ByteString` does not have to necessarily correspond to exactly one line of text (the client might be sending the line in chunks) we use the `Framing.delimiter` helper Flow to chunk the inputs up into actual lines of text. The last boolean argument

indicates that we require an explicit line ending even for the last message before the connection is closed. In this example we simply add exclamation marks to each incoming text message and push it through the flow:

```
import akka.stream.scaladsl.Framing

val connections: Source[IncomingConnection, Future[ServerBinding]] =
  Tcp().bind(host, port)
connections runForeach { connection =>
  println(s"New connection from: ${connection.remoteAddress}")

  val echo = Flow[ByteString]
    .via(Framing.delimiter(
      ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true))
    .map(_.utf8String)
    .map(_ + "!!!\n")
    .map(ByteString(_))

  connection.handleWith(echo)
}
```



Notice that while most building blocks in Akka Streams are reusable and freely shareable, this is *not* the case for the incoming connection Flow, since it directly corresponds to an existing, already accepted connection its handling can only ever be materialized *once*.

Closing connections is possible by cancelling the *incoming connection* `Flow` from your server logic (e.g. by connecting its downstream to a `Sink.cancelled` and its upstream to a `Source.empty`). It is also possible to shut down the server's socket by cancelling the `IncomingConnection` source `connections`.

We can then test the TCP server by sending data to the TCP Socket using `netcat`:

```
$ echo -n "Hello World" | netcat 127.0.0.1 8888
Hello World!!!
```

Connecting: REPL Client

In this example we implement a rather naive Read Evaluate Print Loop client over TCP. Let's say we know a server has exposed a simple command line interface over TCP, and would like to interact with it using Akka Streams over TCP. To open an outgoing connection socket we use the `outgoingConnection` method:

```
val connection = Tcp().outgoingConnection("127.0.0.1", 8888)

val replParser =
  Flow[String].takeWhile(_ != "q")
    .concat(Source.single("BYE"))
    .map(elem => ByteString(s"$elem\n"))
```

```
val repl = Flow[ByteString]
  .via(Framing.delimiter(
    ByteString("\n"),
    maximumFrameLength = 256,
    allowTruncation = true))
  .map(_.utf8String)
  .map(text => println("Server: " + text))
  .map(_ => readLine("> "))
  .via(replParser)

connection.join(repl).run()
```

The `repl` flow we use to handle the server interaction first prints the servers response, then awaits on input from the command line (this blocking call is used here just for the sake of simplicity) and converts it to a `ByteString` which is then sent over the wire to the server. Then we simply connect the TCP pipeline to this processing stage–at this point it will be materialized and start processing data once the server responds with an *initial message*.

A resilient REPL client would be more sophisticated than this, for example it should split out the input reading into a separate mapAsync step and have a way to let the server write more data than one ByteString chunk at any given time, these improvements however are left as exercise for the reader.

Avoiding deadlocks and liveness issues in back-pressured cycles

When writing such end-to-end back-pressured systems you may sometimes end up in a situation of a loop, in which *either side is waiting for the other one to start the conversation*. One does not need to look far to find examples of such back-pressure loops. In the two examples shown previously, we always assumed that the side we are connecting to would start the conversation, which effectively means both sides are back-pressured and can not get the conversation started. There are multiple ways of dealing with this which are explained in depth in *Graph cycles, liveness and deadlocks*, however in client-server scenarios it is often the simplest to make either side simply send an initial message.

Note: In case of back-pressured cycles (which can occur even between different systems) sometimes you have to decide which of the sides has start the conversation in order to kick it off. This can be often done by injecting an initial message from one of the sides–a conversation starter.

To break this back-pressure cycle we need to inject some initial message, a "conversation starter". First, we need to decide which side of the connection should remain passive and which active. Thankfully in most situations finding the right spot to start the conversation is rather simple, as it often is inherent to the protocol we are trying to implement using Streams. In chat-like applications, which our examples resemble, it makes sense to make the Server initiate the conversation by emitting a "hello" message:

```
connections.runForeach { connection =>

  // server logic, parses incoming commands
  val commandParser = Flow[String].takeWhile(_ != "BYE").map(_ + "!")

  import connection._
  val welcomeMsg = s"Welcome to: $localAddress, you are: $remoteAddress!"
  val welcome = Source.single(welcomeMsg)

  val serverLogic = Flow[ByteString]
    .via(Framing.delimiter(
      ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true))
    .map(_.utf8String)
    .via(commandParser)
    // merge in the initial banner after parser
```

```
    .merge(welcome)
    .map(_ + "\n")
    .map(ByteString(_))

  connection.handleWith(serverLogic)
}
```

To emit the initial message we merge a `Source` with a single element, after the command processing but before the framing and transformation to `ByteString` s this way we do not have to repeat such logic.

In this example both client and server may need to close the stream based on a parsed command - `BYE` in the case of the server, and `q` in the case of the client. This is implemented by taking from the stream until `q` and and concatenating a `Source` with a single `BYE` element which will then be sent after the original source completed.

7.13.2 Streaming File IO

Akka Streams provide simple Sources and Sinks that can work with `ByteString` instances to perform IO operations on files.

Streaming data from a file is as easy as creating a *FileIO.fromPath* given a target path, and an optional `chunkSize` which determines the buffer size determined as one "element" in such stream:

```
import akka.stream.scaladsl._
val file = Paths.get("example.csv")

val foreach: Future[IOResult] = FileIO.fromPath(file)
  .to(Sink.ignore)
  .run()
```

Please note that these processing stages are backed by Actors and by default are configured to run on a pre-configured threadpool-backed dispatcher dedicated for File IO. This is very important as it isolates the blocking file IO operations from the rest of the ActorSystem allowing each dispatcher to be utilised in the most efficient way. If you want to configure a custom dispatcher for file IO operations globally, you can do so by changing the `akka.stream.blocking-io-dispatcher`, or for a specific stage by specifying a custom Dispatcher in code, like this:

```
FileIO.fromPath(file)
  .withAttributes(ActorAttributes.dispatcher("custom-blocking-io-dispatcher"))
```

7.14 Pipelining and Parallelism

Akka Streams processing stages (be it simple operators on Flows and Sources or graph junctions) are "fused" together and executed sequentially by default. This avoids the overhead of events crossing asynchronous boundaries but limits the flow to execute at most one stage at any given time.

In many cases it is useful to be able to concurrently execute the stages of a flow, this is done by explicitly marking them as asynchronous using the `async` method. Each processing stage marked as asynchronous will run in a dedicated actor internally, while all stages not marked asynchronous will run in one single actor.

We will illustrate through the example of pancake cooking how streams can be used for various processing patterns, exploiting the available parallelism on modern computers. The setting is the following: both Patrik and Roland like to make pancakes, but they need to produce sufficient amount in a cooking session to make all of the children happy. To increase their pancake production throughput they use two frying pans. How they organize their pancake processing is markedly different.

7.14.1 Pipelining

Roland uses the two frying pans in an asymmetric fashion. The first pan is only used to fry one side of the pancake then the half-finished pancake is flipped into the second pan for the finishing fry on the other side. Once the first frying pan becomes available it gets a new scoop of batter. As an effect, most of the time there are two pancakes being cooked at the same time, one being cooked on its first side and the second being cooked to completion. This is how this setup would look like implemented as a stream:

```
// Takes a scoop of batter and creates a pancake with one side cooked
val fryingPan1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] =
  Flow[ScoopOfBatter].map { batter => HalfCookedPancake() }

// Finishes a half-cooked pancake
val fryingPan2: Flow[HalfCookedPancake, Pancake, NotUsed] =
  Flow[HalfCookedPancake].map { halfCooked => Pancake() }

// With the two frying pans we can fully cook pancakes
val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
  Flow[ScoopOfBatter].via(fryingPan1.async).via(fryingPan2.async)
```

The two `map` stages in sequence (encapsulated in the "frying pan" flows) will be executed in a pipelined way, basically doing the same as Roland with his frying pans:

1. A `ScoopOfBatter` enters `fryingPan1`
2. `fryingPan1` emits a HalfCookedPancake once `fryingPan2` becomes available
3. `fryingPan2` takes the HalfCookedPancake
4. at this point fryingPan1 already takes the next scoop, without waiting for fryingPan2 to finish

The benefit of pipelining is that it can be applied to any sequence of processing steps that are otherwise not parallelisable (for example because the result of a processing step depends on all the information from the previous step). One drawback is that if the processing times of the stages are very different then some of the stages will not be able to operate at full throughput because they will wait on a previous or subsequent stage most of the time. In the pancake example frying the second half of the pancake is usually faster than frying the first half, `fryingPan2` will not be able to operate at full capacity [1].

Note: Asynchronous stream processing stages have internal buffers to make communication between them more efficient. For more details about the behavior of these and how to add additional buffers refer to *Buffers and working with rate*.

7.14.2 Parallel processing

Patrik uses the two frying pans symmetrically. He uses both pans to fully fry a pancake on both sides, then puts the results on a shared plate. Whenever a pan becomes empty, he takes the next scoop from the shared bowl of batter. In essence he parallelizes the same process over multiple pans. This is how this setup will look like if implemented using streams:

```
val fryingPan: Flow[ScoopOfBatter, Pancake, NotUsed] =
  Flow[ScoopOfBatter].map { batter => Pancake() }

val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] = Flow.fromGraph(GraphDSL.
↪create() { implicit builder =>
  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergePancakes = builder.add(Merge[Pancake](2))

  // Using two frying pans in parallel, both fully cooking a pancake from the␣
↪batter.
```

[1] Roland's reason for this seemingly suboptimal procedure is that he prefers the temperature of the second pan to be slightly lower than the first in order to achieve a more homogeneous result.

```
  // We always put the next scoop of batter to the first frying pan that becomes␣
↪available.
  dispatchBatter.out(0) ~> fryingPan.async ~> mergePancakes.in(0)
  // Notice that we used the "fryingPan" flow without importing it via builder.
↪add().
  // Flows used this way are auto-imported, which in this case means that the two
  // uses of "fryingPan" mean actually different stages in the graph.
  dispatchBatter.out(1) ~> fryingPan.async ~> mergePancakes.in(1)

  FlowShape(dispatchBatter.in, mergePancakes.out)
})
```

The benefit of parallelizing is that it is easy to scale. In the pancake example it is easy to add a third frying pan with Patrik's method, but Roland cannot add a third frying pan, since that would require a third processing step, which is not practically possible in the case of frying pancakes.

One drawback of the example code above that it does not preserve the ordering of pancakes. This might be a problem if children like to track their "own" pancakes. In those cases the `Balance` and `Merge` stages should be replaced by strict-round robing balancing and merging stages that put in and take out pancakes in a strict order.

A more detailed example of creating a worker pool can be found in the cookbook: *Balancing jobs to a fixed pool of workers*

7.14.3 Combining pipelining and parallel processing

The two concurrency patterns that we demonstrated as means to increase throughput are not exclusive. In fact, it is rather simple to combine the two approaches and streams provide a nice unifying language to express and compose them.

First, let's look at how we can parallelize pipelined processing stages. In the case of pancakes this means that we will employ two chefs, each working using Roland's pipelining method, but we use the two chefs in parallel, just like Patrik used the two frying pans. This is how it looks like if expressed as streams:

```
val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder =>

    val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
    val mergePancakes = builder.add(Merge[Pancake](2))

    // Using two pipelines, having two frying pans each, in total using
    // four frying pans
    dispatchBatter.out(0) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.
↪in(0)
    dispatchBatter.out(1) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.
↪in(1)

    FlowShape(dispatchBatter.in, mergePancakes.out)
  })
```

The above pattern works well if there are many independent jobs that do not depend on the results of each other, but the jobs themselves need multiple processing steps where each step builds on the result of the previous one. In our case individual pancakes do not depend on each other, they can be cooked in parallel, on the other hand it is not possible to fry both sides of the same pancake at the same time, so the two sides have to be fried in sequence.

It is also possible to organize parallelized stages into pipelines. This would mean employing four chefs:

- the first two chefs prepare half-cooked pancakes from batter, in parallel, then putting those on a large enough flat surface.
- the second two chefs take these and fry their other side in their own pans, then they put the pancakes on a shared plate.

This is again straightforward to implement with the streams API:

```
val pancakeChefs1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder =>
    val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
    val mergeHalfPancakes = builder.add(Merge[HalfCookedPancake](2))

    // Two chefs work with one frying pan for each, half-frying the pancakes then␣
↪putting
    // them into a common pool
    dispatchBatter.out(0) ~> fryingPan1.async ~> mergeHalfPancakes.in(0)
    dispatchBatter.out(1) ~> fryingPan1.async ~> mergeHalfPancakes.in(1)

    FlowShape(dispatchBatter.in, mergeHalfPancakes.out)
  })

val pancakeChefs2: Flow[HalfCookedPancake, Pancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder =>
    val dispatchHalfPancakes = builder.add(Balance[HalfCookedPancake](2))
    val mergePancakes = builder.add(Merge[Pancake](2))

    // Two chefs work with one frying pan for each, finishing the pancakes then␣
↪putting
    // them into a common pool
    dispatchHalfPancakes.out(0) ~> fryingPan2.async ~> mergePancakes.in(0)
    dispatchHalfPancakes.out(1) ~> fryingPan2.async ~> mergePancakes.in(1)

    FlowShape(dispatchHalfPancakes.in, mergePancakes.out)
  })

val kitchen: Flow[ScoopOfBatter, Pancake, NotUsed] = pancakeChefs1.
↪via(pancakeChefs2)
```

This usage pattern is less common but might be usable if a certain step in the pipeline might take wildly different times to finish different jobs. The reason is that there are more balance-merge steps in this pattern compared to the parallel pipelines. This pattern rebalances after each step, while the previous pattern only balances at the entry point of the pipeline. This only matters however if the processing time distribution has a large deviation.

7.15 Testing streams

Verifying behaviour of Akka Stream sources, flows and sinks can be done using various code patterns and libraries. Here we will discuss testing these elements using:

- simple sources, sinks and flows;
- sources and sinks in combination with `TestProbe` from the `akka-testkit` module;
- sources and sinks specifically crafted for writing tests from the `akka-stream-testkit` module.

It is important to keep your data processing pipeline as separate sources, flows and sinks. This makes them easily testable by wiring them up to other sources or sinks, or some test harnesses that `akka-testkit` or `akka-stream-testkit` provide.

7.15.1 Built in sources, sinks and combinators

Testing a custom sink can be as simple as attaching a source that emits elements from a predefined collection, running a constructed test flow and asserting on the results that sink produced. Here is an example of a test for a sink:

```
val sinkUnderTest = Flow[Int].map(_ * 2).toMat(Sink.fold(0)(_ + _))(Keep.right)

val future = Source(1 to 4).runWith(sinkUnderTest)
val result = Await.result(future, 3.seconds)
assert(result == 20)
```

The same strategy can be applied for sources as well. In the next example we have a source that produces an infinite stream of elements. Such source can be tested by asserting that first arbitrary number of elements hold some condition. Here the `take` combinator and `Sink.seq` are very useful.

```
import system.dispatcher
import akka.pattern.pipe

val sourceUnderTest = Source.repeat(1).map(_ * 2)

val future = sourceUnderTest.take(10).runWith(Sink.seq)
val result = Await.result(future, 3.seconds)
assert(result == Seq.fill(10)(2))
```

When testing a flow we need to attach a source and a sink. As both stream ends are under our control, we can choose sources that tests various edge cases of the flow and sinks that ease assertions.

```
val flowUnderTest = Flow[Int].takeWhile(_ < 5)

val future = Source(1 to 10).via(flowUnderTest).runWith(Sink.fold(Seq.empty[Int])(_
↪ :+ _))
val result = Await.result(future, 3.seconds)
assert(result == (1 to 4))
```

7.15.2 TestKit

Akka Stream offers integration with Actors out of the box. This support can be used for writing stream tests that use familiar `TestProbe` from the `akka-testkit` API.

One of the more straightforward tests would be to materialize stream to a `Future` and then use `pipe` pattern to pipe the result of that future to the probe.

```
import system.dispatcher
import akka.pattern.pipe

val sourceUnderTest = Source(1 to 4).grouped(2)

val probe = TestProbe()
sourceUnderTest.runWith(Sink.seq).pipeTo(probe.ref)
probe.expectMsg(3.seconds, Seq(Seq(1, 2), Seq(3, 4)))
```

Instead of materializing to a future, we can use a `Sink.actorRef` that sends all incoming elements to the given `ActorRef`. Now we can use assertion methods on `TestProbe` and expect elements one by one as they arrive. We can also assert stream completion by expecting for `onCompleteMessage` which was given to `Sink.actorRef`.

```
case object Tick
val sourceUnderTest = Source.tick(0.seconds, 200.millis, Tick)

val probe = TestProbe()
val cancellable = sourceUnderTest.to(Sink.actorRef(probe.ref, "completed")).run()

probe.expectMsg(1.second, Tick)
probe.expectNoMsg(100.millis)
probe.expectMsg(3.seconds, Tick)
```

```
cancellable.cancel()
probe.expectMsg(3.seconds, "completed")
```

Similarly to `Sink.actorRef` that provides control over received elements, we can use `Source.actorRef` and have full control over elements to be sent.

```
val sinkUnderTest = Flow[Int].map(_.toString).toMat(Sink.fold("")(_ + _))(Keep.
↪right)

val (ref, future) = Source.actorRef(8, OverflowStrategy.fail)
  .toMat(sinkUnderTest)(Keep.both).run()

ref ! 1
ref ! 2
ref ! 3
ref ! akka.actor.Status.Success("done")

val result = Await.result(future, 3.seconds)
assert(result == "123")
```

7.15.3 Streams TestKit

You may have noticed various code patterns that emerge when testing stream pipelines. Akka Stream has a separate `akka-stream-testkit` module that provides tools specifically for writing stream tests. This module comes with two main components that are `TestSource` and `TestSink` which provide sources and sinks that materialize to probes that allow fluent API.

Note: Be sure to add the module `akka-stream-testkit` to your dependencies.

A sink returned by `TestSink.probe` allows manual control over demand and assertions over elements coming downstream.

```
val sourceUnderTest = Source(1 to 4).filter(_ % 2 == 0).map(_ * 2)

sourceUnderTest
  .runWith(TestSink.probe[Int])
  .request(2)
  .expectNext(4, 8)
  .expectComplete()
```

A source returned by `TestSource.probe` can be used for asserting demand or controlling when stream is completed or ended with an error.

```
val sinkUnderTest = Sink.cancelled

TestSource.probe[Int]
  .toMat(sinkUnderTest)(Keep.left)
  .run()
  .expectCancellation()
```

You can also inject exceptions and test sink behaviour on error conditions.

```
val sinkUnderTest = Sink.head[Int]

val (probe, future) = TestSource.probe[Int]
  .toMat(sinkUnderTest)(Keep.both)
  .run()
probe.sendError(new Exception("boom"))
```

```
Await.ready(future, 3.seconds)
val Failure(exception) = future.value.get
assert(exception.getMessage == "boom")
```

Test source and sink can be used together in combination when testing flows.

```
val flowUnderTest = Flow[Int].mapAsyncUnordered(2) { sleep =>
  pattern.after(10.millis * sleep, using = system.scheduler)(Future.
↪successful(sleep))
}

val (pub, sub) = TestSource.probe[Int]
  .via(flowUnderTest)
  .toMat(TestSink.probe[Int])(Keep.both)
  .run()

sub.request(n = 3)
pub.sendNext(3)
pub.sendNext(2)
pub.sendNext(1)
sub.expectNextUnordered(1, 2, 3)

pub.sendError(new Exception("Power surge in the linear subroutine C-47!"))
val ex = sub.expectError()
assert(ex.getMessage.contains("C-47"))
```

7.15.4 Fuzzing Mode

For testing, it is possible to enable a special stream execution mode that exercises concurrent execution paths more aggressively (at the cost of reduced performance) and therefore helps exposing race conditions in tests. To enable this setting add the following line to your configuration:

```
akka.stream.materializer.debug.fuzzing-mode = on
```

Warning: Never use this setting in production or benchmarks. This is a testing tool to provide more coverage of your code during tests, but it reduces the throughput of streams. A warning message will be logged if you have this setting enabled.

7.16 Overview of built-in stages and their semantics

7.16.1 Source stages

These built-in sources are available from `akka.stream.scaladsl.Source`:

fromIterator

Stream the values from an `Iterator`, requesting the next value when there is demand. The iterator will be created anew for each materialization, which is the reason the method takes a function rather than an iterator directly.

If the iterator perform blocking operations, make sure to run it on a separate dispatcher.

emits the next value returned from the iterator

completes when the iterator reaches its end

apply

Stream the values of an `immutable.Seq`.

emits the next value of the seq

completes when the last element of the seq has been emitted

single

Stream a single object

emits the value once

completes when the single value has been emitted

repeat

Stream a single object repeatedly

emits the same value repeatedly when there is demand

completes never

cycle

Stream iterator in cycled manner. Internally new iterator is being created to cycle the one provided via argument meaning when original iterator runs out of elements process will start all over again from the beginning of the iterator provided by the evaluation of provided parameter. If method argument provides empty iterator stream will be terminated with exception.

emits the next value returned from cycled iterator

completes never

tick

A periodical repetition of an arbitrary object. Delay of first tick is specified separately from interval of the following ticks.

emits periodically, if there is downstream backpressure ticks are skipped

completes never

fromFuture

Send the single value of the `Future` when it completes and there is demand. If the future fails the stream is failed with that exception.

emits the future completes

completes after the future has completed

fromCompletionStage

Send the single value of the Java `CompletionStage` when it completes and there is demand. If the future fails the stream is failed with that exception.

emits the future completes

completes after the future has completed

unfold

Stream the result of a function as long as it returns a `Some`, the value inside the option consists of a tuple where the first value is a state passed back into the next call to the function allowing to pass a state. The first invocation of the provided fold function will receive the `zero` state.

Can be used to implement many stateful sources without having to touch the more low level `GraphStage` API.

emits when there is demand and the unfold function over the previous state returns non empty value

completes when the unfold function returns an empty value

unfoldAsync

Just like `unfold` but the fold function returns a `Future` which will cause the source to complete or emit when it completes.

Can be used to implement many stateful sources without having to touch the more low level `GraphStage` API.

emits when there is demand and unfold state returned future completes with some value

completes when the future returned by the unfold function completes with an empty value

empty

Complete right away without ever emitting any elements. Useful when you have to provide a source to an API but there are no elements to emit.

emits never

completes directly

maybe

Materialize a `Promise[Option[T]]` that if completed with a `Some[T]` will emit that *T* and then complete the stream, or if completed with `None` complete the stream right away.

emits when the returned promise is completed with some value

completes after emitting some value, or directly if the promise is completed with no value

failed

Fail directly with a user specified exception.

emits never

completes fails the stream directly with the given exception

actorPublisher

Wrap an actor extending `ActorPublisher` as a source.

emits depends on the actor implementation

completes when the actor stops

actorRef

Materialize an `ActorRef`, sending messages to it will emit them on the stream. The actor contain a buffer but since communication is one way, there is no back pressure. Handling overflow is done by either dropping elements or failing the stream, the strategy is chosen by the user.

emits when there is demand and there are messages in the buffer or a message is sent to the actorref

completes when the actorref is sent `akka.actor.Status.Success` or `PoisonPill`

combine

Combine several sources, using a given strategy such as merge or concat, into one source.

emits when there is demand, but depending on the strategy

completes when all sources has completed

unfoldResource

Wrap any resource that can be opened, queried for next element (in a blocking way) and closed using three distinct functions into a source.

emits when there is demand and read function returns value

completes when read function returns `None`

unfoldResourceAsync

Wrap any resource that can be opened, queried for next element (in a blocking way) and closed using three distinct functions into a source. Functions return `Future` to achieve asynchronous processing

emits when there is demand and `Future` from read function returns value

completes when `Future` from read function returns `None`

queue

Materialize a `SourceQueue` onto which elements can be pushed for emitting from the source. The queue contains a buffer, if elements are pushed onto the queue faster than the source is consumed the overflow will be handled with a strategy specified by the user. Functionality for tracking when an element has been emitted is available through `SourceQueue.offer`.

emits when there is demand and the queue contains elements

completes when downstream completes

asSubscriber

Integration with Reactive Streams, materializes into a `org.reactivestreams.Subscriber`.

fromPublisher

Integration with Reactive Streams, subscribes to a `org.reactivestreams.Publisher`.

zipN

Combine the elements of multiple streams into a stream of sequences.

emits when all of the inputs has an element available

completes when any upstream completes

zipWithN

Combine the elements of multiple streams into a stream of sequences using a combiner function.

emits when all of the inputs has an element available

completes when any upstream completes

7.16.2 Sink stages

These built-in sinks are available from `akka.stream.scaladsl.Sink`:

head

Materializes into a `Future` which completes with the first value arriving, after this the stream is canceled. If no element is emitted, the future is be failed.

cancels after receiving one element

backpressures never

headOption

Materializes into a `Future[Option[T]]` which completes with the first value arriving wrapped in a `Some`, or a `None` if the stream completes without any elements emitted.

cancels after receiving one element

backpressures never

last

Materializes into a `Future` which will complete with the last value emitted when the stream completes. If the stream completes with no elements the future is failed.

cancels never

backpressures never

lastOption

Materialize a `Future[Option[T]]` which completes with the last value emitted wrapped in an `Some` when the stream completes. if the stream completes with no elements the future is completed with `None`.

cancels never

backpressures never

ignore

Consume all elements but discards them. Useful when a stream has to be consumed but there is no use to actually do anything with the elements.

cancels never

backpressures never

cancelled

Immediately cancel the stream

cancels immediately

seq

Collect values emitted from the stream into a collection, the collection is available through a `Future` or which completes when the stream completes. Note that the collection is bounded to `Int.MaxValue`, if more element are emitted the sink will cancel the stream

cancels If too many values are collected

foreach

Invoke a given procedure for each element received. Note that it is not safe to mutate shared state from the procedure.

The sink materializes into a `Future[Option[Done]]` which completes when the stream completes, or fails if the stream fails.

Note that it is not safe to mutate state from the procedure.

cancels never

backpressures when the previous procedure invocation has not yet completed

foreachParallel

Like `foreach` but allows up to `parallellism` procedure calls to happen in parallel.

cancels never

backpressures when the previous parallel procedure invocations has not yet completed

onComplete

Invoke a callback when the stream has completed or failed.

cancels never

backpressures never

lazyInit

Invoke sinkFactory function to create a real sink upon receiving the first element. Internal `Sink` will not be created if there are no elements, because of completion or error. *fallback* will be invoked if there was no elements and completed is received from upstream.

cancels never

backpressures when initialized and when created sink backpressures

queue

Materialize a `SinkQueue` that can be pulled to trigger demand through the sink. The queue contains a buffer in case stream emitting elements faster than queue pulling them.

cancels when `SinkQueue.cancel` is called

backpressures when buffer has some space

fold

Fold over emitted element with a function, where each invocation will get the new element and the result from the previous fold invocation. The first invocation will be provided the `zero` value.

Materializes into a future that will complete with the last state when the stream has completed.

This stage allows combining values into a result without a global mutable state by instead passing the state along between invocations.

cancels never

backpressures when the previous fold function invocation has not yet completed

reduce

Apply a reduction function on the incoming elements and pass the result to the next invocation. The first invocation receives the two first elements of the flow.

Materializes into a future that will be completed by the last result of the reduction function.

cancels never

backpressures when the previous reduction function invocation has not yet completed

combine

Combine several sinks into one using a user specified strategy

cancels depends on the strategy

backpressures depends on the strategy

actorRef

Send the elements from the stream to an `ActorRef`. No backpressure so care must be taken to not overflow the inbox.

cancels when the actor terminates

backpressures never

actorRefWithAck

Send the elements from the stream to an `ActorRef` which must then acknowledge reception after completing a message, to provide back pressure onto the sink.

cancels when the actor terminates

backpressures when the actor acknowledgement has not arrived

actorSubscriber

Create an actor from a `Props` upon materialization, where the actor implements `ActorSubscriber`, which will receive the elements from the stream.

Materializes into an `ActorRef` to the created actor.

cancels when the actor terminates

backpressures depends on the actor implementation

asPublisher

Integration with Reactive Streams, materializes into a `org.reactivestreams.Publisher`.

fromSubscriber

Integration with Reactive Streams, wraps a `org.reactivestreams.Subscriber` as a sink

7.16.3 Additional Sink and Source converters

Sources and sinks for integrating with `java.io.InputStream` and `java.io.OutputStream` can be found on `StreamConverters`. As they are blocking APIs the implementations of these stages are run on a separate dispatcher configured through the `akka.stream.blocking-io-dispatcher`.

fromOutputStream

Create a sink that wraps an `OutputStream`. Takes a function that produces an `OutputStream`, when the sink is materialized the function will be called and bytes sent to the sink will be written to the returned `OutputStream`.

Materializes into a `Future` which will complete with a `IOResult` when the stream completes.

Note that a flow can be materialized multiple times, so the function producing the `OutputStream` must be able to handle multiple invocations.

The `OutputStream` will be closed when the stream that flows into the `Sink` is completed, and the `Sink` will cancel its inflow when the `OutputStream` is no longer writable.

asInputStream

Create a sink which materializes into an `InputStream` that can be read to trigger demand through the sink. Bytes emitted through the stream will be available for reading through the `InputStream`

The `InputStream` will be ended when the stream flowing into this `Sink` completes, and the closing the `InputStream` will cancel the inflow of this `Sink`.

fromInputStream

Create a source that wraps an `InputStream`. Takes a function that produces an `InputStream`, when the source is materialized the function will be called and bytes from the `InputStream` will be emitted into the stream.

Materializes into a `Future` which will complete with a `IOResult` when the stream completes.

Note that a flow can be materialized multiple times, so the function producing the `InputStream` must be able to handle multiple invocations.

The `InputStream` will be closed when the `Source` is canceled from its downstream, and reaching the end of the `InputStream` will complete the `Source`.

asOutputStream

Create a source that materializes into an `OutputStream`. When bytes are written to the `OutputStream` they are emitted from the source.

The `OutputStream` will no longer be writable when the `Source` has been canceled from its downstream, and closing the `OutputStream` will complete the `Source`.

asJavaStream

Create a sink which materializes into Java 8 `Stream` that can be run to trigger demand through the sink. Elements emitted through the stream will be available for reading through the Java 8 `Stream`.

The Java 8 `Stream` will be ended when the stream flowing into this `Sink` completes, and closing the Java `Stream` will cancel the inflow of this `Sink`. Java `Stream` throws exception in case reactive stream failed.

Be aware that Java `Stream` blocks current thread while waiting on next element from downstream.

fromJavaStream

Create a source that wraps a Java 8 `Stream`. `Source` uses a stream iterator to get all its elements and send them downstream on demand.

javaCollector

Create a sink which materializes into a `Future` which will be completed with a result of the Java 8 `Collector` transformation and reduction operations. This allows usage of Java 8 streams transformations for reactive streams. The `Collector` will trigger demand downstream. Elements emitted through the stream will be accumulated into a mutable result container, optionally transformed into a final representation after all input elements have been processed. The `Collector` can also do reduction at the end. Reduction processing is performed sequentially

Note that a flow can be materialized multiple times, so the function producing the `Collector` must be able to handle multiple invocations.

javaCollectorParallelUnordered

Create a sink which materializes into a `Future` which will be completed with a result of the Java 8 `Collector` transformation and reduction operations. This allows usage of Java 8 streams transformations for reactive streams. The `Collector` is triggering demand downstream. Elements emitted through the stream will be accumulated into a mutable result container, optionally transformed into a final representation after all input elements have been processed. The `Collector` can also do reduction at the end. Reduction processing is performed in parallel based on graph `Balance`.

Note that a flow can be materialized multiple times, so the function producing the `Collector` must be able to handle multiple invocations.

7.16.4 File IO Sinks and Sources

Sources and sinks for reading and writing files can be found on `FileIO`.

fromPath

Emit the contents of a file, as `ByteString` s, materializes into a `Future` which will be completed with a `IOResult` upon reaching the end of the file or if there is a failure.

toPath

Create a sink which will write incoming `ByteString` s to a given file path.

7.16.5 Flow stages

All flows by default backpressure if the computation they encapsulate is not fast enough to keep up with the rate of incoming elements from the preceding stage. There are differences though how the different stages handle when some of their downstream stages backpressure them.

Most stages stop and propagate the failure downstream as soon as any of their upstreams emit a failure. This happens to ensure reliable teardown of streams and cleanup when failures happen. Failures are meant to be to model unrecoverable conditions, therefore they are always eagerly propagated. For in-band error handling of normal errors (dropping elements if a map fails for example) you should use the supervision support, or explicitly wrap your element types in a proper container that can express error or success states (for example `Try` in Scala).

7.16.6 Simple processing stages

These stages can transform the rate of incoming elements since there are stages that emit multiple elements for a single input (e.g. *mapConcat') or consume multiple elements before emitting one output (e.g. ''filter'*). However, these rate transformations are data-driven, i.e. it is the incoming elements that define how the rate is affected. This is in contrast with *Backpressure aware stages* which can change their processing behavior depending on being backpressured by downstream or not.

map

Transform each element in the stream by calling a mapping function with it and passing the returned value downstream.

emits when the mapping function returns an element

backpressures when downstream backpressures

completes when upstream completes

mapConcat

Transform each element into zero or more elements that are individually passed downstream.

emits when the mapping function returns an element or there are still remaining elements from the previously calculated collection

backpressures when downstream backpressures or there are still available elements from the previously calculated collection

completes when upstream completes and all remaining elements has been emitted

statefulMapConcat

Transform each element into zero or more elements that are individually passed downstream. The difference to `mapConcat` is that the transformation function is created from a factory for every materialization of the flow.

emits when the mapping function returns an element or there are still remaining elements from the previously calculated collection

backpressures when downstream backpressures or there are still available elements from the previously calculated collection

completes when upstream completes and all remaining elements has been emitted

filter

Filter the incoming elements using a predicate. If the predicate returns true the element is passed downstream, if it returns false the element is discarded.

emits when the given predicate returns true for the element

backpressures when the given predicate returns true for the element and downstream backpressures

completes when upstream completes

filterNot

Filter the incoming elements using a predicate. If the predicate returns false the element is passed downstream, if it returns true the element is discarded.

emits when the given predicate returns false for the element

backpressures when the given predicate returns false for the element and downstream backpressures

completes when upstream completes

collect

Apply a partial function to each incoming element, if the partial function is defined for a value the returned value is passed downstream. Can often replace `filter` followed by `map` to achieve the same in one single stage.

emits when the provided partial function is defined for the element

backpressures the partial function is defined for the element and downstream backpressures

completes when upstream completes

grouped

Accumulate incoming events until the specified number of elements have been accumulated and then pass the collection of elements downstream.

emits when the specified number of elements has been accumulated or upstream completed

backpressures when a group has been assembled and downstream backpressures

completes when upstream completes

sliding

Provide a sliding window over the incoming stream and pass the windows as groups of elements downstream.

Note: the last window might be smaller than the requested size due to end of stream.

emits the specified number of elements has been accumulated or upstream completed

backpressures when a group has been assembled and downstream backpressures

completes when upstream completes

scan

Emit its current value which starts at `zero` and then applies the current and next value to the given function emitting the next current value.

Note that this means that scan emits one element downstream before and upstream elements will not be requested until the second element is required from downstream.

emits when the function scanning the element returns a new element

backpressures when downstream backpressures

completes when upstream completes

scanAsync

Just like `scan` but receiving a function that results in a `Future` to the next value.

emits when the `Future` resulting from the function scanning the element resolves to the next value

backpressures when downstream backpressures

completes when upstream completes and the last `Future` is resolved

fold

Start with current value `zero` and then apply the current and next value to the given function, when upstream complete the current value is emitted downstream.

emits when upstream completes

backpressures when downstream backpressures

completes when upstream completes

foldAsync

Just like `fold` but receiving a function that results in a `Future` to the next value.

emits when upstream completes and the last `Future` is resolved

backpressures when downstream backpressures

completes when upstream completes and the last `Future` is resolved

reduce

Start with first element and then apply the current and next value to the given function, when upstream complete the current value is emitted downstream. Similar to `fold`.

emits when upstream completes

backpressures when downstream backpressures

completes when upstream completes

drop

Drop `n` elements and then pass any subsequent element downstream.

emits when the specified number of elements has been dropped already

backpressures when the specified number of elements has been dropped and downstream backpressures

completes when upstream completes

take

Pass `n` incoming elements downstream and then complete

emits while the specified number of elements to take has not yet been reached

backpressures when downstream backpressures

completes when the defined number of elements has been taken or upstream completes

takeWhile

Pass elements downstream as long as a predicate function return true for the element include the element when the predicate first return false and then complete.

emits while the predicate is true and until the first false result

backpressures when downstream backpressures

completes when predicate returned false or upstream completes

dropWhile

Drop elements as long as a predicate function return true for the element

emits when the predicate returned false and for all following stream elements

backpressures predicate returned false and downstream backpressures

completes when upstream completes

recover

Allow sending of one last element downstream when a failure has happened upstream.

emits when the element is available from the upstream or upstream is failed and pf returns an element

backpressures when downstream backpressures, not when failure happened

completes when upstream completes or upstream failed with exception pf can handle

recoverWith

Allow switching to alternative Source when a failure has happened upstream.

emits the element is available from the upstream or upstream is failed and pf returns alternative Source

backpressures downstream backpressures, after failure happened it backprssures to alternative Source

completes upstream completes or upstream failed with exception pf can handle

detach

Detach upstream demand from downstream demand without detaching the stream rates.

emits when the upstream stage has emitted and there is demand

backpressures when downstream backpressures

completes when upstream completes

throttle

Limit the throughput to a specific number of elements per time unit, or a specific total cost per time unit, where a function has to be provided to calculate the individual cost of each element.

emits when upstream emits an element and configured time per each element elapsed

backpressures when downstream backpressures

completes when upstream completes

intersperse

Intersperse stream with provided element similar to `List.mkString`. It can inject start and end marker elements to stream.

emits when upstream emits an element or before with the *start* element if provided

backpressures when downstream backpressures

completes when upstream completes

limit

Limit number of element from upstream to given `max` number.

emits when upstream emits and the number of emitted elements has not reached max

backpressures when downstream backpressures

completes when upstream completes and the number of emitted elements has not reached max

limitWeighted

Ensure stream boundedness by evaluating the cost of incoming elements using a cost function. Evaluated cost of each element defines how many elements will be allowed to travel downstream.

emits when upstream emits and the number of emitted elements has not reached max

backpressures when downstream backpressures

completes when upstream completes and the number of emitted elements has not reached max

log

Log elements flowing through the stream as well as completion and erroring. By default element and completion signals are logged on debug level, and errors are logged on Error level. This can be changed by calling `Attributes.logLevels(...)` on the given Flow.

emits when upstream emits

backpressures when downstream backpressures

completes when upstream completes

recoverWithRetries

Switch to alternative Source on flow failure. It stays in effect after a failure has been recovered up to `attempts` number of times. Each time a failure is fed into the partial function and a new Source may be materialized.

emits when element is available from the upstream or upstream is failed and element is available from alternative Source

backpressures when downstream backpressures

completes when upstream completes or upstream failed with exception provided partial function can handle

7.16.7 Asynchronous processing stages

These stages encapsulate an asynchronous computation, properly handling backpressure while taking care of the asynchronous operation at the same time (usually handling the completion of a Future).

mapAsync

Pass incoming elements to a function that return a `Future` result. When the future arrives the result is passed downstream. Up to `n` elements can be processed concurrently, but regardless of their completion time the incoming order will be kept when results complete. For use cases where order does not mather `mapAsyncUnordered` can be used.

If a Future fails, the stream also fails (unless a different supervision strategy is applied)

emits when the Future returned by the provided function finishes for the next element in sequence

backpressures when the number of futures reaches the configured parallelism and the downstream backpressures

completes when upstream completes and all futures has been completed and all elements has been emitted

mapAsyncUnordered

Like `mapAsync` but `Future` results are passed downstream as they arrive regardless of the order of the elements that triggered them.

If a Future fails, the stream also fails (unless a different supervision strategy is applied)

emits any of the Futures returned by the provided function complete

backpressures when the number of futures reaches the configured parallelism and the downstream backpressures

completes upstream completes and all futures has been completed and all elements has been emitted

7.16.8 Timer driven stages

These stages process elements using timers, delaying, dropping or grouping elements for certain time durations.

takeWithin

Pass elements downstream within a timeout and then complete.

emits when an upstream element arrives

backpressures downstream backpressures

completes upstream completes or timer fires

dropWithin

Drop elements until a timeout has fired

emits after the timer fired and a new upstream element arrives

backpressures when downstream backpressures

completes upstream completes

groupedWithin

Chunk up the stream into groups of elements received within a time window, or limited by the given number of elements, whichever happens first.

emits when the configured time elapses since the last group has been emitted

backpressures when the group has been assembled (the duration elapsed) and downstream backpressures

completes when upstream completes

initialDelay

Delay the initial element by a user specified duration from stream materialization.

emits upstream emits an element if the initial delay already elapsed

backpressures downstream backpressures or initial delay not yet elapsed

completes when upstream completes

delay

Delay every element passed through with a specific duration.

emits there is a pending element in the buffer and configured time for this element elapsed

backpressures differs, depends on `OverflowStrategy` set

completes when upstream completes and buffered elements has been drained

7.16.9 Backpressure aware stages

These stages are aware of the backpressure provided by their downstreams and able to adapt their behavior to that signal.

conflate

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure. The summary value must be of the same type as the incoming elements, for example the sum or average of incoming numbers, if aggregation should lead to a different type `conflateWithSeed` can be used:

emits when downstream stops backpressuring and there is a conflated element available

backpressures when the aggregate function cannot keep up with incoming elements

completes when upstream completes

conflateWithSeed

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure. When backpressure starts or there is no backpressure element is passed into a `seed` function to transform it to the summary type.

emits when downstream stops backpressuring and there is a conflated element available

backpressures when the aggregate or seed functions cannot keep up with incoming elements

completes when upstream completes

batch

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure and a maximum number of batched elements is not yet reached. When the maximum number is reached and downstream still backpressures batch will also backpressure.

When backpressure starts or there is no backpressure element is passed into a `seed` function to transform it to the summary type.

Will eagerly pull elements, this behavior may result in a single pending (i.e. buffered) element which cannot be aggregated to the batched value.

emits when downstream stops backpressuring and there is a batched element available

backpressures when batched elements reached the max limit of allowed batched elements & downstream backpressures

completes when upstream completes and a "possibly pending" element was drained

batchWeighted

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure and a maximum weight batched elements is not yet reached. The weight of each element is determined by applying `costFn`. When the maximum total weight is reached and downstream still backpressures batch will also backpressure.

Will eagerly pull elements, this behavior may result in a single pending (i.e. buffered) element which cannot be aggregated to the batched value.

emits downstream stops backpressuring and there is a batched element available

backpressures batched elements reached the max weight limit of allowed batched elements & downstream backpressures

completes upstream completes and a "possibly pending" element was drained

expand

Allow for a faster downstream by expanding the last incoming element to an `Iterator`. For example `Iterator.continually(element)` to keep repating the last incoming element.

emits when downstream stops backpressuring

backpressures when downstream backpressures

completes when upstream completes

buffer (Backpressure)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full backpressure is applied.

emits when downstream stops backpressuring and there is a pending element in the buffer

backpressures when buffer is full

completes when upstream completes and buffered elements has been drained

buffer (Drop)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full elements are dropped according to the specified `OverflowStrategy`:

- `dropHead` drops the oldest element in the buffer to make space for the new element
- `dropTail` drops the youngest element in the buffer to make space for the new element
- `dropBuffer` drops the entire buffer and buffers the new element
- `dropNew` drops the new element

emits when downstream stops backpressuring and there is a pending element in the buffer

backpressures never (when dropping cannot keep up with incoming elements)

completes upstream completes and buffered elements has been drained

buffer (Fail)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full the stage fails the flow with a `BufferOverflowException`.

emits when downstream stops backpressuring and there is a pending element in the buffer

backpressures never, fails the stream instead of backpressuring when buffer is full

completes when upstream completes and buffered elements has been drained

7.16.10 Nesting and flattening stages

These stages either take a stream and turn it into a stream of streams (nesting) or they take a stream that contains nested streams and turn them into a stream of elements instead (flattening).

prefixAndTail

Take up to *n* elements from the stream (less than *n* only if the upstream completes before emitting *n* elements) and returns a pair containing a strict sequence of the taken element and a stream representing the remaining elements.

emits when the configured number of prefix elements are available. Emits this prefix, and the rest as a substream

backpressures when downstream backpressures or substream backpressures

completes when prefix elements has been consumed and substream has been consumed

groupBy

Demultiplex the incoming stream into separate output streams.

emits an element for which the grouping function returns a group that has not yet been created. Emits the new group there is an element pending for a group whose substream backpressures

completes when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

splitWhen

Split off elements into a new substream whenever a predicate function return `true`.

emits an element for which the provided predicate is true, opening and emitting a new substream for subsequent elements

backpressures when there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures

completes when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

splitAfter

End the current substream whenever a predicate returns `true`, starting a new substream for the next element.

emits when an element passes through. When the provided predicate is true it emits the element * and opens a new substream for subsequent element

backpressures when there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures

completes when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

flatMapConcat

Transform each input element into a `Source` whose elements are then flattened into the output stream through concatenation. This means each source is fully consumed before consumption of the next source starts.

emits when the current consumed substream has an element available

backpressures when downstream backpressures

completes when upstream completes and all consumed substreams complete

flatMapMerge

Transform each input element into a `Source` whose elements are then flattened into the output stream through merging. The maximum number of merged sources has to be specified.

emits when one of the currently consumed substreams has an element available

backpressures when downstream backpressures

completes when upstream completes and all consumed substreams complete

7.16.11 Time aware stages

Those stages operate taking time into consideration.

initialTimeout

If the first element has not passed through this stage before the provided timeout, the stream is failed with a `TimeoutException`.

emits when upstream emits an element

backpressures when downstream backpressures

completes when upstream completes or fails if timeout elapses before first element arrives

cancels when downstream cancels

completionTimeout

If the completion of the stream does not happen until the provided timeout, the stream is failed with a `TimeoutException`.

emits when upstream emits an element

backpressures when downstream backpressures

completes when upstream completes or fails if timeout elapses before upstream completes

cancels when downstream cancels

idleTimeout

If the time between two processed elements exceeds the provided timeout, the stream is failed with a `TimeoutException`. The timeout is checked periodically, so the resolution of the check is one period (equals to timeout value).

emits when upstream emits an element

backpressures when downstream backpressures

completes when upstream completes or fails if timeout elapses between two emitted elements

cancels when downstream cancels

backpressureTimeout

If the time between the emission of an element and the following downstream demand exceeds the provided timeout, the stream is failed with a `TimeoutException`. The timeout is checked periodically, so the resolution of the check is one period (equals to timeout value).

emits when upstream emits an element

backpressures when downstream backpressures

completes when upstream completes or fails if timeout elapses between element emission and downstream demand.

cancels when downstream cancels

keepAlive

Injects additional (configured) elements if upstream does not emit for a configured amount of time.

emits when upstream emits an element or if the upstream was idle for the configured period

backpressures when downstream backpressures

completes when upstream completes

cancels when downstream cancels

initialDelay

Delays the initial element by the specified duration.

emits when upstream emits an element if the initial delay is already elapsed

backpressures when downstream backpressures or initial delay is not yet elapsed

completes when upstream completes

cancels when downstream cancels

7.16.12 Fan-in stages

These stages take multiple streams as their input and provide a single output combining the elements from all of the inputs in different ways.

merge

Merge multiple sources. Picks elements randomly if all sources has elements ready.

emits when one of the inputs has an element available

backpressures when downstream backpressures

completes when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

mergeSorted

Merge multiple sources. Waits for one element to be ready from each input stream and emits the smallest element.

emits when all of the inputs have an element available

backpressures when downstream backpressures

completes when all upstreams complete

mergePreferred

Merge multiple sources. Prefer one source if all sources has elements ready.

emits when one of the inputs has an element available, preferring a defined input if multiple have elements available

backpressures when downstream backpressures

completes when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

zip

Combines elements from each of multiple sources into tuples and passes the tuples downstream.

emits when all of the inputs have an element available

backpressures when downstream backpressures

completes when any upstream completes

zipWith

Combines elements from multiple sources through a `combine` function and passes the returned value downstream.

emits when all of the inputs have an element available

backpressures when downstream backpressures

completes when any upstream completes

zipWithIndex

Zips elements of current flow with its indices.

emits upstream emits an element and is paired with their index

backpressures when downstream backpressures

completes when upstream completes

concat

After completion of the original upstream the elements of the given source will be emitted.

emits when the current stream has an element available; if the current input completes, it tries the next one

backpressures when downstream backpressures

completes when all upstreams complete

++

Just a shorthand for concat

emits when the current stream has an element available; if the current input completes, it tries the next one

backpressures when downstream backpressures

completes when all upstreams complete

prepend

Prepends the given source to the flow, consuming it until completion before the original source is consumed.

If materialized values needs to be collected `prependMat` is available.

emits when the given stream has an element available; if the given input completes, it tries the current one

backpressures when downstream backpressures

completes when all upstreams complete

orElse

If the primary source completes without emitting any elements, the elements from the secondary source are emitted. If the primary source emits any elements the secondary source is cancelled.

Note that both sources are materialized directly and the secondary source is backpressured until it becomes the source of elements or is cancelled.

Signal errors downstream, regardless which of the two sources emitted the error.

emits when an element is available from first stream or first stream closed without emitting any elements and an element is available from the second stream

backpressures when downstream backpressures

completes the primary stream completes after emitting at least one element, when the primary stream completes without emitting and the secondary stream already has completed or when the secondary stream completes

interleave

Emits a specifiable number of elements from the original source, then from the provided source and repeats. If one source completes the rest of the other stream will be emitted.

emits when element is available from the currently consumed upstream

backpressures when upstream backpressures

completes when both upstreams have completed

7.16.13 Fan-out stages

These have one input and multiple outputs. They might route the elements between different outputs, or emit elements on multiple outputs at the same time.

unzip

Takes a stream of two element tuples and unzips the two elements ino two different downstreams.

emits when all of the outputs stops backpressuring and there is an input element available

backpressures when any of the outputs backpressures

completes when upstream completes

unzipWith

Splits each element of input into multiple downstreams using a function

emits when all of the outputs stops backpressuring and there is an input element available

backpressures when any of the outputs backpressures

completes when upstream completes

broadcast

Emit each incoming element each of `n` outputs.

emits when all of the outputs stops backpressuring and there is an input element available

backpressures when any of the outputs backpressures

completes when upstream completes

balance

Fan-out the stream to several streams. Each upstream element is emitted to the first available downstream consumer.

emits when any of the outputs stops backpressuring; emits the element to the first available output

backpressures when all of the outputs backpressure

completes when upstream completes

partition

Fan-out the stream to several streams. Each upstream element is emitted to one downstream consumer according to the partitioner function applied to the element.

emits when the chosen output stops backpressuring and there is an input element available

backpressures when the chosen output backpressures

completes when upstream completes and no output is pending

7.16.14 Watching status stages

watchTermination

Materializes to a `Future` that will be completed with Done or failed depending whether the upstream of the stage has been completed or failed. The stage otherwise passes through elements unchanged.

emits when input has an element available

backpressures when output backpressures

completes when upstream completes

monitor

Materializes to a `FlowMonitor` that monitors messages flowing through or completion of the stage. The stage otherwise passes through elements unchanged. Note that the `FlowMonitor` inserts a memory barrier every time it processes an event, and may therefore affect performance.

emits when upstream emits an element

backpressures when downstream **backpressures**

completes when upstream completes

7.17 Streams Cookbook

7.17.1 Introduction

This is a collection of patterns to demonstrate various usage of the Akka Streams API by solving small targeted problems in the format of "recipes". The purpose of this page is to give inspiration and ideas how to approach various small tasks involving streams. The recipes in this page can be used directly as-is, but they are most powerful as starting points: customization of the code snippets is warmly encouraged.

This part also serves as supplementary material for the main body of documentation. It is a good idea to have this page open while reading the manual and look for examples demonstrating various streaming concepts as they appear in the main body of documentation.

If you need a quick reference of the available processing stages used in the recipes see *Overview of built-in stages and their semantics*.

7.17.2 Working with Flows

In this collection we show simple recipes that involve linear flows. The recipes in this section are rather general, more targeted recipes are available as separate sections (*Buffers and working with rate*, *Working with streaming IO*).

Logging elements of a stream

Situation: During development it is sometimes helpful to see what happens in a particular section of a stream.

The simplest solution is to simply use a `map` operation and use `println` to print the elements received to the console. While this recipe is rather simplistic, it is often suitable for a quick debug session.

```
val loggedSource = mySource.map { elem => println(elem); elem }
```

Another approach to logging is to use `log()` operation which allows configuring logging for elements flowing through the stream as well as completion and erroring.

```
// customise log levels
mySource.log("before-map")
  .withAttributes(Attributes.logLevels(onElement = Logging.WarningLevel))
  .map(analyse)

// or provide custom logging adapter
implicit val adapter = Logging(system, "customLogger")
mySource.log("custom")
```

Flattening a stream of sequences

Situation: A stream is given as a stream of sequence of elements, but a stream of elements needed instead, streaming all the nested elements inside the sequences separately.

The `mapConcat` operation can be used to implement a one-to-many transformation of elements using a mapper function in the form of `In => immutable.Seq[Out]`. In this case we want to map a `Seq` of elements to the elements in the collection itself, so we can just call `mapConcat(identity)`.

```
val myData: Source[List[Message], NotUsed] = someDataSource
val flattened: Source[Message, NotUsed] = myData.mapConcat(identity)
```

Draining a stream to a strict collection

Situation: A possibly unbounded sequence of elements is given as a stream, which needs to be collected into a Scala collection while ensuring boundedness

A common situation when working with streams is one where we need to collect incoming elements into a Scala collection. This operation is supported via `Sink.seq` which materializes into a `Future[Seq[T]]`.

The function `limit` or `take` should always be used in conjunction in order to guarantee stream boundedness, thus preventing the program from running out of memory.

For example, this is best avoided:

```
// Dangerous: might produce a collection with 2 billion elements!
val f: Future[Seq[String]] = mySource.runWith(Sink.seq)
```

Rather, use `limit` or `take` to ensure that the resulting `Seq` will contain only up to `max` elements:

```
val MAX_ALLOWED_SIZE = 100

// OK. Future will fail with a `StreamLimitReachedException`
// if the number of incoming elements is larger than max
val limited: Future[Seq[String]] =
  mySource.limit(MAX_ALLOWED_SIZE).runWith(Sink.seq)

// OK. Collect up until max-th elements only, then cancel upstream
val ignoreOverflow: Future[Seq[String]] =
  mySource.take(MAX_ALLOWED_SIZE).runWith(Sink.seq)
```

Calculating the digest of a ByteString stream

Situation: A stream of bytes is given as a stream of `ByteString` s and we want to calculate the cryptographic digest of the stream.

This recipe uses a `GraphStage` to host a mutable `MessageDigest` class (part of the Java Cryptography API) and update it with the bytes arriving from the stream. When the stream starts, the `onPull` handler of the stage is called, which just bubbles up the `pull` event to its upstream. As a response to this pull, a ByteString chunk will arrive (`onPush`) which we use to update the digest, then it will pull for the next chunk.

Eventually the stream of `ByteString` s depletes and we get a notification about this event via `onUpstreamFinish`. At this point we want to emit the digest value, but we cannot do it with `push` in this handler directly since there may be no downstream demand. Instead we call `emit` which will temporarily replace the handlers, emit the provided value when demand comes in and then reset the stage state. It will then complete the stage.

```
import akka.stream.stage._
class DigestCalculator(algorithm: String) extends GraphStage[FlowShape[ByteString,␣
↪ByteString]] {
  val in = Inlet[ByteString]("DigestCalculator.in")
  val out = Outlet[ByteString]("DigestCalculator.out")
  override val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new␣
↪GraphStageLogic(shape) {
    val digest = MessageDigest.getInstance(algorithm)

    setHandler(out, new OutHandler {
```

```
      override def onPull(): Trigger = {
        pull(in)
      }
    })

    setHandler(in, new InHandler {
      override def onPush(): Trigger = {
        val chunk = grab(in)
        digest.update(chunk.toArray)
        pull(in)
      }

      override def onUpstreamFinish(): Unit = {
        emit(out, ByteString(digest.digest()))
        completeStage()
      }
    })

  }
}
val digest: Source[ByteString, NotUsed] = data.via(new DigestCalculator("SHA-256"))
```

Parsing lines from a stream of ByteStrings

Situation: A stream of bytes is given as a stream of `ByteString` s containing lines terminated by line ending characters (or, alternatively, containing binary frames delimited by a special delimiter byte sequence) which needs to be parsed.

The `Framing` helper object contains a convenience method to parse messages from a stream of `ByteString` s:

```
import akka.stream.scaladsl.Framing
val linesStream = rawData.via(Framing.delimiter(
  ByteString("\r\n"), maximumFrameLength = 100, allowTruncation = true))
  .map(_.utf8String)
```

Dealing with compressed data streams

Situation: A gzipped stream of bytes is given as a stream of `ByteString` s, for example from a `FileIO` source.

The `Compression` helper object contains convenience methods for decompressing data streams compressed with Gzip or Deflate.

```
import akka.stream.scaladsl.Compression
val uncompressed = compressed.via(Compression.gunzip())
  .map(_.utf8String)
```

Implementing reduce-by-key

Situation: Given a stream of elements, we want to calculate some aggregated value on different subgroups of the elements.

The "hello world" of reduce-by-key style operations is *wordcount* which we demonstrate below. Given a stream of words we first create a new stream that groups the words according to the `identity` function, i.e. now we have a stream of streams, where every substream will serve identical words.

To count the words, we need to process the stream of streams (the actual groups containing identical words). `groupBy` returns a `SubFlow`, which means that we transform the resulting substreams directly. In this case we use the `reduce` combinator to aggregate the word itself and the number of its occurrences within a tuple

`(String,Integer)`. Each substream will then emit one final value—precisely such a pair—when the overall input completes. As a last step we merge back these values from the substreams into one single output stream.

One noteworthy detail pertains to the `MaximumDistinctWords` parameter: this defines the breadth of the groupBy and merge operations. Akka Streams is focused on bounded resource consumption and the number of concurrently open inputs to the merge operator describes the amount of resources needed by the merge itself. Therefore only a finite number of substreams can be active at any given time. If the `groupBy` operator encounters more keys than this number then the stream cannot continue without violating its resource bound, in this case `groupBy` will terminate with a failure.

```
val counts: Source[(String, Int), NotUsed] = words
  // split the words into separate streams first
  .groupBy(MaximumDistinctWords, identity)
  //transform each element to pair with number of words in it
  .map(_ -> 1)
  // add counting logic to the streams
  .reduce((l, r) => (l._1, l._2 + r._2))
  // get a stream of word counts
  .mergeSubstreams
```

By extracting the parts specific to *wordcount* into

- a `groupKey` function that defines the groups
- a `map` map each element to value that is used by the reduce on the substream
- a `reduce` function that does the actual reduction

we get a generalized version below:

```
def reduceByKey[In, K, Out](
  maximumGroupSize: Int,
  groupKey:         (In) => K,
  map:              (In) => Out)(reduce: (Out, Out) => Out): Flow[In, (K, Out),␣
↪NotUsed] = {

  Flow[In]
    .groupBy[K](maximumGroupSize, groupKey)
    .map(e => groupKey(e) -> map(e))
    .reduce((l, r) => l._1 -> reduce(l._2, r._2))
    .mergeSubstreams
}

val wordCounts = words.via(
  reduceByKey(
    MaximumDistinctWords,
    groupKey = (word: String) => word,
    map = (word: String) => 1)((left: Int, right: Int) => left + right))
```

Note: Please note that the reduce-by-key version we discussed above is sequential in reading the overall input stream, in other words it is **NOT** a parallelization pattern like MapReduce and similar frameworks.

Sorting elements to multiple groups with groupBy

Situation: The `groupBy` operation strictly partitions incoming elements, each element belongs to exactly one group. Sometimes we want to map elements into multiple groups simultaneously.

To achieve the desired result, we attack the problem in two steps:

- first, using a function `topicMapper` that gives a list of topics (groups) a message belongs to, we transform our stream of `Message` to a stream of `(Message,Topic)` where for each topic the message belongs to a separate pair will be emitted. This is achieved by using `mapConcat`

- Then we take this new stream of message topic pairs (containing a separate pair for each topic a given message belongs to) and feed it into groupBy, using the topic as the group key.

```
val topicMapper: (Message) => immutable.Seq[Topic] = extractTopics

val messageAndTopic: Source[(Message, Topic), NotUsed] = elems.mapConcat { msg:␣
↪Message =>
  val topicsForMessage = topicMapper(msg)
  // Create a (Msg, Topic) pair for each of the topics
  // the message belongs to
  topicsForMessage.map(msg -> _)
}

val multiGroups = messageAndTopic
  .groupBy(2, _._2).map {
    case (msg, topic) =>
      // do what needs to be done
  }
```

7.17.3 Working with Graphs

In this collection we show recipes that use stream graph elements to achieve various goals.

Triggering the flow of elements programmatically

Situation: Given a stream of elements we want to control the emission of those elements according to a trigger signal. In other words, even if the stream would be able to flow (not being backpressured) we want to hold back elements until a trigger signal arrives.

This recipe solves the problem by simply zipping the stream of `Message` elements with the stream of `Trigger` signals. Since `Zip` produces pairs, we simply map the output stream selecting the first element of the pair.

```
val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._
  val zip = builder.add(Zip[Message, Trigger]())
  elements ~> zip.in0
  triggerSource ~> zip.in1
  zip.out ~> Flow[(Message, Trigger)].map { case (msg, trigger) => msg } ~> sink
  ClosedShape
})
```

Alternatively, instead of using a `Zip`, and then using `map` to get the first element of the pairs, we can avoid creating the pairs in the first place by using `ZipWith` which takes a two argument function to produce the output element. If this function would return a pair of the two argument it would be exactly the behavior of `Zip` so `ZipWith` is a generalization of zipping.

```
val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._
  val zip = builder.add(ZipWith((msg: Message, trigger: Trigger) => msg))

  elements ~> zip.in0
  triggerSource ~> zip.in1
  zip.out ~> sink
  ClosedShape
})
```

Balancing jobs to a fixed pool of workers

Situation: Given a stream of jobs and a worker process expressed as a `Flow` create a pool of workers that automatically balances incoming jobs to available workers, then merges the results.

We will express our solution as a function that takes a worker flow and the number of workers to be allocated and gives a flow that internally contains a pool of these workers. To achieve the desired result we will create a `Flow` from a graph.

The graph consists of a `Balance` node which is a special fan-out operation that tries to route elements to available downstream consumers. In a `for` loop we wire all of our desired workers as outputs of this balancer element, then we wire the outputs of these workers to a `Merge` element that will collect the results from the workers.

To make the worker stages run in parallel we mark them as asynchronous with *async*.

```
def balancer[In, Out](worker: Flow[In, Out, Any], workerCount: Int): Flow[In, Out,␣
↪NotUsed] = {
  import GraphDSL.Implicits._

  Flow.fromGraph(GraphDSL.create() { implicit b =>
    val balancer = b.add(Balance[In](workerCount, waitForAllDownstreams = true))
    val merge = b.add(Merge[Out](workerCount))

    for (_ <- 1 to workerCount) {
      // for each worker, add an edge from the balancer to the worker, then wire
      // it to the merge element
      balancer ~> worker.async ~> merge
    }

    FlowShape(balancer.in, merge.out)
  })
}

val processedJobs: Source[Result, NotUsed] = myJobs.via(balancer(worker, 3))
```

7.17.4 Working with rate

This collection of recipes demonstrate various patterns where rate differences between upstream and downstream needs to be handled by other strategies than simple backpressure.

Dropping elements

Situation: Given a fast producer and a slow consumer, we want to drop elements if necessary to not slow down the producer too much.

This can be solved by using a versatile rate-transforming operation, `conflate`. Conflate can be thought as a special `reduce` operation that collapses multiple upstream elements into one aggregate element if needed to keep the speed of the upstream unaffected by the downstream.

When the upstream is faster, the reducing process of the `conflate` starts. Our reducer function simply takes the freshest element. This in a simple dropping operation.

```
val droppyStream: Flow[Message, Message, NotUsed] =
  Flow[Message].conflate((lastMessage, newMessage) => newMessage)
```

There is a more general version of `conflate` named `conflateWithSeed` that allows to express more complex aggregations, more similar to a `fold`.

Dropping broadcast

Situation: The default `Broadcast` graph element is properly backpressured, but that means that a slow downstream consumer can hold back the other downstream consumers resulting in lowered throughput. In other words the rate of `Broadcast` is the rate of its slowest downstream consumer. In certain cases it is desirable to allow faster consumers to progress independently of their slower siblings by dropping elements if necessary.

One solution to this problem is to append a `buffer` element in front of all of the downstream consumers defining a dropping strategy instead of the default `Backpressure`. This allows small temporary rate differences between the different consumers (the buffer smooths out small rate variances), but also allows faster consumers to progress by dropping from the buffer of the slow consumers if necessary.

```
val graph = RunnableGraph.fromGraph(GraphDSL.create(mySink1, mySink2, mySink3)((_,␣
↪_, _)) { implicit b => (sink1, sink2, sink3) =>
  import GraphDSL.Implicits._

  val bcast = b.add(Broadcast[Int](3))
  myElements ~> bcast

  bcast.buffer(10, OverflowStrategy.dropHead) ~> sink1
  bcast.buffer(10, OverflowStrategy.dropHead) ~> sink2
  bcast.buffer(10, OverflowStrategy.dropHead) ~> sink3
  ClosedShape
})
```

Collecting missed ticks

Situation: Given a regular (stream) source of ticks, instead of trying to backpressure the producer of the ticks we want to keep a counter of the missed ticks instead and pass it down when possible.

We will use `conflateWithSeed` to solve the problem. The seed version of conflate takes two functions:

- A seed function that produces the zero element for the folding process that happens when the upstream is faster than the downstream. In our case the seed function is a constant function that returns 0 since there were no missed ticks at that point.
- A fold function that is invoked when multiple upstream messages needs to be collapsed to an aggregate value due to the insufficient processing rate of the downstream. Our folding function simply increments the currently stored count of the missed ticks so far.

As a result, we have a flow of `Int` where the number represents the missed ticks. A number 0 means that we were able to consume the tick fast enough (i.e. zero means: 1 non-missed tick + 0 missed ticks)

```
val missedTicks: Flow[Tick, Int, NotUsed] =
  Flow[Tick].conflateWithSeed(seed = (_) => 0)(
    (missedTicks, tick) => missedTicks + 1)
```

Create a stream processor that repeats the last element seen

Situation: Given a producer and consumer, where the rate of neither is known in advance, we want to ensure that none of them is slowing down the other by dropping earlier unconsumed elements from the upstream if necessary, and repeating the last value for the downstream if necessary.

We have two options to implement this feature. In both cases we will use `GraphStage` to build our custom element. In the first version we will use a provided initial value `initial` that will be used to feed the downstream if no upstream element is ready yet. In the `onPush()` handler we just overwrite the `currentValue` variable and immediately relieve the upstream by calling `pull()`. The downstream `onPull` handler is very similar, we immediately relieve the downstream by emitting `currentValue`.

```
import akka.stream._
import akka.stream.stage._
final class HoldWithInitial[T](initial: T) extends GraphStage[FlowShape[T, T]] {
  val in = Inlet[T]("HoldWithInitial.in")
  val out = Outlet[T]("HoldWithInitial.out")

  override val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new␣
↪GraphStageLogic(shape) {
    private var currentValue: T = initial

    setHandlers(in, out, new InHandler with OutHandler {
      override def onPush(): Unit = {
        currentValue = grab(in)
        pull(in)
      }

      override def onPull(): Unit = {
        push(out, currentValue)
      }
    })

    override def preStart(): Unit = {
      pull(in)
    }
  }

}
```

While it is relatively simple, the drawback of the first version is that it needs an arbitrary initial element which is not always possible to provide. Hence, we create a second version where the downstream might need to wait in one single case: if the very first element is not yet available.

We introduce a boolean variable `waitingFirstValue` to denote whether the first element has been provided or not (alternatively an `Option` can be used for `currentValue` or if the element type is a subclass of AnyRef a null can be used with the same purpose). In the downstream `onPull()` handler the difference from the previous version is that we check if we have received the first value and only emit if we have. This leads to that when the first element comes in we must check if there possibly already was demand from downstream so that we in that case can push the element directly.

```
import akka.stream._
import akka.stream.stage._
final class HoldWithWait[T] extends GraphStage[FlowShape[T, T]] {
  val in = Inlet[T]("HoldWithWait.in")
  val out = Outlet[T]("HoldWithWait.out")

  override val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new␣
↪GraphStageLogic(shape) {
    private var currentValue: T = _
    private var waitingFirstValue = true

    setHandlers(in, out, new InHandler with OutHandler {
      override def onPush(): Unit = {
        currentValue = grab(in)
        if (waitingFirstValue) {
          waitingFirstValue = false
          if (isAvailable(out)) push(out, currentValue)
        }
        pull(in)
```

```
      }

      override def onPull(): Unit = {
        if (!waitingFirstValue) push(out, currentValue)
      }
    })

    override def preStart(): Unit = {
      pull(in)
    }
  }
}
```

Globally limiting the rate of a set of streams

Situation: Given a set of independent streams that we cannot merge, we want to globally limit the aggregate throughput of the set of streams.

One possible solution uses a shared actor as the global limiter combined with mapAsync to create a reusable `Flow` that can be plugged into a stream to limit its rate.

As the first step we define an actor that will do the accounting for the global rate limit. The actor maintains a timer, a counter for pending permit tokens and a queue for possibly waiting participants. The actor has an `open` and `closed` state. The actor is in the `open` state while it has still pending permits. Whenever a request for permit arrives as a `WantToPass` message to the actor the number of available permits is decremented and we notify the sender that it can pass by answering with a `MayPass` message. If the amount of permits reaches zero, the actor transitions to the `closed` state. In this state requests are not immediately answered, instead the reference of the sender is added to a queue. Once the timer for replenishing the pending permits fires by sending a `ReplenishTokens` message, we increment the pending permits counter and send a reply to each of the waiting senders. If there are more waiting senders than permits available we will stay in the `closed` state.

```
object Limiter {
  case object WantToPass
  case object MayPass

  case object ReplenishTokens

  def props(maxAvailableTokens: Int, tokenRefreshPeriod: FiniteDuration,
            tokenRefreshAmount: Int): Props =
    Props(new Limiter(maxAvailableTokens, tokenRefreshPeriod, tokenRefreshAmount))
}

class Limiter(
  val maxAvailableTokens: Int,
  val tokenRefreshPeriod: FiniteDuration,
  val tokenRefreshAmount: Int) extends Actor {
  import Limiter._
  import context.dispatcher
  import akka.actor.Status

  private var waitQueue = immutable.Queue.empty[ActorRef]
  private var permitTokens = maxAvailableTokens
  private val replenishTimer = system.scheduler.schedule(
    initialDelay = tokenRefreshPeriod,
    interval = tokenRefreshPeriod,
    receiver = self,
    ReplenishTokens)

  override def receive: Receive = open

  val open: Receive = {
```

```
    case ReplenishTokens =>
      permitTokens = math.min(permitTokens + tokenRefreshAmount,␣
↪maxAvailableTokens)
    case WantToPass =>
      permitTokens -= 1
      sender() ! MayPass
      if (permitTokens == 0) context.become(closed)
  }

  val closed: Receive = {
    case ReplenishTokens =>
      permitTokens = math.min(permitTokens + tokenRefreshAmount,␣
↪maxAvailableTokens)
      releaseWaiting()
    case WantToPass =>
      waitQueue = waitQueue.enqueue(sender())
  }

  private def releaseWaiting(): Unit = {
    val (toBeReleased, remainingQueue) = waitQueue.splitAt(permitTokens)
    waitQueue = remainingQueue
    permitTokens -= toBeReleased.size
    toBeReleased foreach (_ ! MayPass)
    if (permitTokens > 0) context.become(open)
  }

  override def postStop(): Unit = {
    replenishTimer.cancel()
    waitQueue foreach (_ ! Status.Failure(new IllegalStateException("limiter␣
↪stopped")))
  }
}
```

To create a Flow that uses this global limiter actor we use the `mapAsync` function with the combination of the `ask` pattern. We also define a timeout, so if a reply is not received during the configured maximum wait period the returned future from `ask` will fail, which will fail the corresponding stream as well.

```
def limitGlobal[T](limiter: ActorRef, maxAllowedWait: FiniteDuration): Flow[T, T,␣
↪NotUsed] = {
  import akka.pattern.ask
  import akka.util.Timeout
  Flow[T].mapAsync(4)((element: T) => {
    import system.dispatcher
    implicit val triggerTimeout = Timeout(maxAllowedWait)
    val limiterTriggerFuture = limiter ? Limiter.WantToPass
    limiterTriggerFuture.map((_) => element)
  })

}
```

Note: The global actor used for limiting introduces a global bottleneck. You might want to assign a dedicated dispatcher for this actor.

7.17.5 Working with IO

Chunking up a stream of ByteStrings into limited size ByteStrings

Situation: Given a stream of `ByteString` s we want to produce a stream of `ByteString` s containing the same bytes in the same sequence, but capping the size of `ByteString` s. In other words we want to slice up `ByteString` s into smaller chunks if they exceed a size threshold.

This can be achieved with a single `GraphStage`. The main logic of our stage is in `emitChunk()` which implements the following logic:

- if the buffer is empty, and upstream is not closed we pull for more bytes, if it is closed we complete
- if the buffer is nonEmpty, we split it according to the `chunkSize`. This will give a next chunk that we will emit, and an empty or nonempty remaining buffer.

Both `onPush()` and `onPull()` calls `emitChunk()` the only difference is that the push handler also stores the incoming chunk by appending to the end of the buffer.

```
import akka.stream.stage._

class Chunker(val chunkSize: Int) extends GraphStage[FlowShape[ByteString,␣
↪ByteString]] {
  val in = Inlet[ByteString]("Chunker.in")
  val out = Outlet[ByteString]("Chunker.out")
  override val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new␣
↪GraphStageLogic(shape) {
    private var buffer = ByteString.empty

    setHandler(out, new OutHandler {
      override def onPull(): Unit = {
        if (isClosed(in)) emitChunk()
        else pull(in)
      }
    })
    setHandler(in, new InHandler {
      override def onPush(): Unit = {
        val elem = grab(in)
        buffer ++= elem
        emitChunk()
      }

      override def onUpstreamFinish(): Unit = {
        if (buffer.isEmpty) completeStage()
        else {
          // There are elements left in buffer, so
          // we keep accepting downstream pulls and push from buffer until emptied.
          //
          // It might be though, that the upstream finished while it was pulled,␣
↪in which
          // case we will not get an onPull from the downstream, because we␣
↪already had one.
          // In that case we need to emit from the buffer.
          if (isAvailable(out)) emitChunk()
        }
      }
    })

    private def emitChunk(): Unit = {
      if (buffer.isEmpty) {
        if (isClosed(in)) completeStage()
        else pull(in)
      } else {
        val (chunk, nextBuffer) = buffer.splitAt(chunkSize)
```

```
        buffer = nextBuffer
        push(out, chunk)
      }
    }

  }
}

val chunksStream = rawBytes.via(new Chunker(ChunkLimit))
```

Limit the number of bytes passing through a stream of ByteStrings

Situation: Given a stream of `ByteString` s we want to fail the stream if more than a given maximum of bytes has been consumed.

This recipe uses a `GraphStage` to implement the desired feature. In the only handler we override, `onPush()` we just update a counter and see if it gets larger than `maximumBytes`. If a violation happens we signal failure, otherwise we forward the chunk we have received.

```
import akka.stream.stage._
class ByteLimiter(val maximumBytes: Long) extends GraphStage[FlowShape[ByteString,␣
↪ByteString]] {
  val in = Inlet[ByteString]("ByteLimiter.in")
  val out = Outlet[ByteString]("ByteLimiter.out")
  override val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = new␣
↪GraphStageLogic(shape) {
    private var count = 0

    setHandlers(in, out, new InHandler with OutHandler {

      override def onPull(): Unit = {
        pull(in)
      }

      override def onPush(): Unit = {
        val chunk = grab(in)
        count += chunk.size
        if (count > maximumBytes) failStage(new IllegalStateException("Too much␣
↪bytes"))
        else push(out, chunk)
      }
    })
  }
}

val limiter = Flow[ByteString].via(new ByteLimiter(SizeLimit))
```

Compact ByteStrings in a stream of ByteStrings

Situation: After a long stream of transformations, due to their immutable, structural sharing nature `ByteString` s may refer to multiple original ByteString instances unnecessarily retaining memory. As the final step of a transformation chain we want to have clean copies that are no longer referencing the original `ByteString` s.

The recipe is a simple use of map, calling the `compact()` method of the `ByteString` elements. This does copying of the underlying arrays, so this should be the last element of a long chain if used.

```
val compacted: Source[ByteString, NotUsed] = data.map(_.compact)
```

Injecting keep-alive messages into a stream of ByteStrings

Situation: Given a communication channel expressed as a stream of `ByteString` s we want to inject keep-alive messages but only if this does not interfere with normal traffic.

There is a built-in operation that allows to do this directly:

```
import scala.concurrent.duration._
val injectKeepAlive: Flow[ByteString, ByteString, NotUsed] =
  Flow[ByteString].keepAlive(1.second, () => keepaliveMessage)
```

7.18 Configuration

```
####################################
# Akka Stream Reference Config File #
####################################

akka {
  stream {

    # Default flow materializer settings
    materializer {

      # Initial size of buffers used in stream elements
      initial-input-buffer-size = 4
      # Maximum size of buffers used in stream elements
      max-input-buffer-size = 16

      # Fully qualified config path which holds the dispatcher configuration
      # to be used by FlowMaterialiser when creating Actors.
      # When this value is left empty, the default-dispatcher will be used.
      dispatcher = ""

      # Cleanup leaked publishers and subscribers when they are not used within a␣
↪given
      # deadline
      subscription-timeout {
        # when the subscription timeout is reached one of the following strategies␣
↪on
        # the "stale" publisher:
        # cancel - cancel it (via `onError` or subscribing to the publisher and
        #          `cancel()`ing the subscription right away
        # warn   - log a warning statement about the stale element (then drop the
        #          reference to it)
        # noop   - do nothing (not recommended)
        mode = cancel

        # time after which a subscriber / publisher is considered stale and␣
↪eligible
        # for cancelation (see `akka.stream.subscription-timeout.mode`)
        timeout = 5s
      }

      # Enable additional troubleshooting logging at DEBUG log level
      debug-logging = off

      # Maximum number of elements emitted in batch if downstream signals large␣
↪demand
      output-burst-limit = 1000
```

```
      # Enable automatic fusing of all graphs that are run. For short-lived streams
      # this may cause an initial runtime overhead, but most of the time fusing is
      # desirable since it reduces the number of Actors that are created.
      auto-fusing = on

      # Those stream elements which have explicit buffers (like mapAsync,␣
↪mapAsyncUnordered,
      # buffer, flatMapMerge, Source.actorRef, Source.queue, etc.) will␣
↪preallocate a fixed
      # buffer upon stream materialization if the requested buffer size is less␣
↪than this
      # configuration parameter. The default is very high because failing early is␣
↪better
      # than failing under load.
      #
      # Buffers sized larger than this will dynamically grow/shrink and consume␣
↪more memory
      # per element than the fixed size buffers.
      max-fixed-buffer-size = 1000000000

      # Maximum number of sync messages that actor can process for stream to␣
↪substream communication.
      # Parameter allows to interrupt synchronous processing to get upsteam/
↪downstream messages.
      # Allows to accelerate message processing that happening withing same actor␣
↪but keep system responsive.
      sync-processing-limit = 1000

      debug {
        # Enables the fuzzing mode which increases the chance of race conditions
        # by aggressively reordering events and making certain operations more
        # concurrent than usual.
        # This setting is for testing purposes, NEVER enable this in a production
        # environment!
        # To get the best results, try combining this setting with a throughput
        # of 1 on the corresponding dispatchers.
        fuzzing-mode = off
      }
    }

    # Fully qualified config path which holds the dispatcher configuration
    # to be used by FlowMaterialiser when creating Actors for IO operations,
    # such as FileSource, FileSink and others.
    blocking-io-dispatcher = "akka.stream.default-blocking-io-dispatcher"

    default-blocking-io-dispatcher {
      type = "Dispatcher"
      executor = "thread-pool-executor"
      throughput = 1

      thread-pool-executor {
        core-pool-size-min = 2
        core-pool-size-factor = 2.0
        core-pool-size-max = 16
      }
    }
  }

  # configure overrides to ssl-configuration here (to be used by akka-streams, and␣
↪akka-http - i.e. when serving https connections)
  ssl-config {
    protocol = "TLSv1.2"
  }
```

```
}

# ssl configuration
# folded in from former ssl-config-akka module
ssl-config {
  logger = "com.typesafe.sslconfig.akka.util.AkkaLoggerBridge"
}
```

7.19 Migration Guide 1.0 to 2.x

For this migration guide see the documentation for Akka Streams 2.0.

7.20 Migration Guide 2.0.x to 2.4.x

7.20.1 General notes

akka.Done and akka.NotUsed replacing Unit and BoxedUnit

To provide more clear signatures and have a unified API for both Java and Scala two new types have been introduced:

`akka.NotUsed` is meant to be used instead of `Unit` in Scala and `BoxedUnit` in Java to signify that the type parameter is required but not actually used. This is commonly the case with `Source`, `Flow` and `Sink` that do not materialize into any value.

`akka.Done` is added for the use case where it is boxed inside another object to signify completion but there is no actual value attached to the completion. It is used to replace occurrences of `Future<BoxedUnit>` with `Future<Done>` in Java and `Future[Unit]` with `Future[Done]` in Scala.

All previous usage of `Unit` and `BoxedUnit` for these two cases in the Akka Streams APIs has been updated.

This means that Scala code like this:

```
Source[Int, Unit] source = Source.from(1 to 5)
Sink[Int, Future[Unit]] sink = Sink.ignore()
```

needs to be changed into:

```
Source[Int, NotUsed] source = Source.from(1 to 5)
Sink[Int, Future[Done]] sink = Sink.ignore()
```

These changes apply to all the places where streams are used, which means that signatures in the persistent query APIs also are affected.

7.20.2 Removed ImplicitMaterializer

The helper trait `ImplicitMaterializer` has been removed as it was hard to find and the feature was not worth the extra trait. Defining an implicit materializer inside an enclosing actor can be done this way:

```
final implicit val materializer: ActorMaterializer =␣
↪ActorMaterializer(ActorMaterializerSettings(context.system))
```

7.20.3 Changed Operators

`expand()` is now based on an Iterator

Previously the `expand` combinator required two functions as input: the first one lifted incoming values into an extrapolation state and the second one extracted values from that, possibly evolving that state. This has been simplified into a single function that turns the incoming element into an Iterator.

The most prominent use-case previously was to just repeat the previously received value:

```
Flow[Int].expand(identity)(s => (s, s)) // This no longer works!
```

In Akka 2.4.x this is simplified to:

```
Flow[Int].expand(Iterator.continually(_))
```

If state needs to be be kept during the expansion process then this state will need to be managed by the Iterator. The example of counting the number of expansions might previously have looked like:

```
// This no longer works!
Flow[Int].expand((_, 0)){ case (in, count) => (in, count) -> (in, count + 1) }
```

In Akka 2.4.x this is formulated like so:

```
Flow[Int].expand(i => {
  var state = 0
  Iterator.continually({
    state += 1
    (i, state)
  })
})
```

`conflate` has been renamed to `conflateWithSeed()`

The new `conflate` operator is a special case of the original behavior (renamed to `conflateWithSeed`) that does not change the type of the stream. The usage of the new operator is as simple as:

```
Flow[Int].conflate(_ + _) // Add numbers while downstream is not ready
```

Which is the same as using `conflateWithSeed` with an identity function

```
Flow[Int].conflateWithSeed(identity)(_ + _) // Add numbers while downstream is not ready
```

`viaAsync` and `viaAsyncMat` has been replaced with `async`

`async` is available from `Sink`, `Source`, `Flow` and the sub flows. It provides a shortcut for setting the attribute `Attributes.asyncBoundary` on a flow. The existing methods `Flow.viaAsync` and `Flow.viaAsyncMat` has been removed to make marking out asynchronous boundaries more consistent:

```
// This no longer works
source.viaAsync(flow)
```

In Akka 2.4.x this will instead look lile this:

```
val flow = Flow[Int].map(_ + 1)
Source(1 to 10).via(flow.async)
```

7.20.4 Changes in Akka HTTP

Routing settings parameter name

`RoutingSettings` were previously the only setting available on `RequestContext`, and were accessible via `settings`. We now made it possible to configure the parsers settings as well, so `RoutingSettings` is now `routingSettings` and `ParserSettings` is now accessible via `parserSettings`.

Client / server behaviour on cancelled entity

Previously if request or response were cancelled or consumed only partially (e.g. by using `take` combinator) the remaining data was silently drained to prevent stalling the connection, since there could still be more requests / responses incoming. Now the default behaviour is to close the connection in order to prevent using excessive resource usage in case of huge entities.

The old behaviour can be achieved by explicitly draining the entity:

```
response.entity.dataBytes.runWith(Sink.ignore)
```

7.20.5 Changed Sources / Sinks

IO Sources / Sinks materialize IOResult

Materialized values of the following sources and sinks:

- `FileIO.fromPath`
- `FileIO.toPath`
- `StreamConverters.fromInputStream`
- `StreamConverters.fromOutputStream`

have been changed from `Long` to `akka.stream.io.IOResult`. This allows to signal more complicated completion scenarios. For example, on failure it is now possible to return the exception and the number of bytes written until that exception occured.

7.20.6 PushStage, PushPullStage and DetachedStage have been deprecated in favor of GraphStage

The `PushStage` `PushPullStage` and `DetachedStage` classes have been deprecated and should be replaced by `GraphStage` (*Custom processing with GraphStage*) which is now a single powerful API for custom stream processing.

Update procedure

Please consult the `GraphStage` documentation (*Custom processing with GraphStage*) and the previous migration guide on migrating from `AsyncStage` to `GraphStage`.

Websocket now consistently named WebSocket

Previously we had a mix of methods and classes called `websocket` or `Websocket`, which was in contradiction with how the word is spelled in the spec and some other places of Akka HTTP.

Methods and classes using the word WebSocket now consistently use it as `WebSocket`, so updating is as simple as find-and-replacing the lower-case `s` to an upper-case `S` wherever the word WebSocket appeared.

Java DSL for Http binding and connections changed

In order to minimise the number of needed overloads for each method defined on the `Http` extension a new mini-DSL has been introduced for connecting to hosts given a hostname, port and optional `ConnectionContext`.

The availability of the connection context (if it's set to `HttpsConnectionContext`) makes the server be bound as an HTTPS server, and for outgoing connections those settings are used instead of the default ones if provided.

Was:

```
http.cachedHostConnectionPool(toHost("akka.io"), materializer());
http.cachedHostConnectionPool("akka.io", 80, httpsConnectionContext,␣
↪materializer()); // does not work anymore
```

Replace with:

```
http.cachedHostConnectionPool(toHostHttps("akka.io", 8081), materializer());
http.cachedHostConnectionPool(toHostHttps("akka.io", 8081).
↪withCustomHttpsContext(httpsContext), materializer());
```

SslTls has been renamed to TLS and moved

The DSL to access a TLS (or SSL) `BidiFlow` have now split between the `javadsl` and `scaladsl` packages and have been renamed to `TLS`. Common option types (closing modes, authentication modes, etc.) have been moved to the top level `stream` package, and the common message types are accessible in the class `akka.stream.TLSProtocol`

Framing moved to akka.stream.[javadsl/scaladsl]

The `Framing` object which can be used to chunk up `ByteString` streams into framing dependent chunks (such as lines) has moved to `akka.stream.scaladsl.Framing`, and has gotten a Java DSL equivalent type in `akka.stream.javadsl.Framing`.

CHAPTER

EIGHT

AKKA HTTP DOCUMENTATION (SCALA) MOVED!

Akka HTTP has been released as independent stable module (from Akka HTTP 3.x onwards). The documentation is available under doc.akka.io/akka-http/current/.

CHAPTER

NINE

HOWTO: COMMON PATTERNS

This section lists common actor patterns which have been found to be useful, elegant or instructive. Anything is welcome, example topics being message routing strategies, supervision patterns, restart handling, etc. As a special bonus, additions to this section are marked with the contributor's name, and it would be nice if every Akka user who finds a recurring pattern in his or her code could share it for the profit of all. Where applicable it might also make sense to add to the `akka.pattern` package for creating an OTP-like library.

9.1 Throttling Messages

Contributed by: Kaspar Fischer

"A message throttler that ensures that messages are not sent out at too high a rate."

The pattern is described in Throttling Messages in Akka 2.

9.2 Balancing Workload Across Nodes

Contributed by: Derek Wyatt

"Often times, people want the functionality of the BalancingDispatcher with the stipulation that the Actors doing the work have distinct Mailboxes on remote nodes. In this post we'll explore the implementation of such a concept."

The pattern is described Balancing Workload across Nodes with Akka 2.

9.3 Work Pulling Pattern to throttle and distribute work, and prevent mailbox overflow

Contributed by: Michael Pollmeier

"This pattern ensures that your mailboxes don't overflow if creating work is fast than actually doing it – which can lead to out of memory errors when the mailboxes eventually become too full. It also let's you distribute work around your cluster, scale dynamically scale and is completely non-blocking. This pattern is a specialisation of the above 'Balancing Workload Pattern'."

The pattern is described Work Pulling Pattern to prevent mailbox overflow, throttle and distribute work.

9.4 Ordered Termination

Contributed by: Derek Wyatt

"When an Actor stops, its children stop in an undefined order. Child termination is asynchronous and thus non-deterministic.

If an Actor has children that have order dependencies, then you might need to ensure a particular shutdown order of those children so that their postStop() methods get called in the right order."

The pattern is described An Akka 2 Terminator.

9.5 Akka AMQP Proxies

Contributed by: Fabrice Drouin

""AMQP proxies" is a simple way of integrating AMQP with Akka to distribute jobs across a network of computing nodes. You still write "local" code, have very little to configure, and end up with a distributed, elastic, fault-tolerant grid where computing nodes can be written in nearly every programming language."

The pattern is described Akka AMQP Proxies.

9.6 Shutdown Patterns in Akka 2

Contributed by: Derek Wyatt

"How do you tell Akka to shut down the ActorSystem when everything's finished? It turns out that there's no magical flag for this, no configuration setting, no special callback you can register for, and neither will the illustrious shutdown fairy grace your application with her glorious presence at that perfect moment. She's just plain mean.

In this post, we'll discuss why this is the case and provide you with a simple option for shutting down "at the right time", as well as a not-so-simple-option for doing the exact same thing."

The pattern is described Shutdown Patterns in Akka 2.

9.7 Distributed (in-memory) graph processing with Akka

Contributed by: Adelbert Chang

"Graphs have always been an interesting structure to study in both mathematics and computer science (among other fields), and have become even more interesting in the context of online social networks such as Facebook and Twitter, whose underlying network structures are nicely represented by graphs."

The pattern is described Distributed In-Memory Graph Processing with Akka.

9.8 Case Study: An Auto-Updating Cache Using Actors

Contributed by: Eric Pederson

"We recently needed to build a caching system in front of a slow backend system with the following requirements:

The data in the backend system is constantly being updated so the caches need to be updated every N minutes. Requests to the backend system need to be throttled. The caching system we built used Akka actors and Scala's support for functions as first class objects."

The pattern is described Case Study: An Auto-Updating Cache using Actors.

9.9 Discovering message flows in actor systems with the Spider Pattern

Contributed by: Raymond Roestenburg

"Building actor systems is fun but debugging them can be difficult, you mostly end up browsing through many log files on several machines to find out what's going on. I'm sure you have browsed through logs and thought, "Hey, where did that message go?", "Why did this message cause that effect" or "Why did this actor never get a message?"

This is where the Spider pattern comes in."

The pattern is described Discovering Message Flows in Actor System with the Spider Pattern.

9.10 Scheduling Periodic Messages

This pattern describes how to schedule periodic messages to yourself in two different ways.

The first way is to set up periodic message scheduling in the constructor of the actor, and cancel that scheduled sending in `postStop` or else we might have multiple registered message sends to the same actor.

Note: With this approach the scheduled periodic message send will be restarted with the actor on restarts. This also means that the time period that elapses between two tick messages during a restart may drift off based on when you restart the scheduled message sends relative to the time that the last message was sent, and how long the initial delay is. Worst case scenario is `interval` plus `initialDelay`.

```
class ScheduleInConstructor extends Actor {
  import context.dispatcher
  val tick =
    context.system.scheduler.schedule(500 millis, 1000 millis, self, "tick")

  override def postStop() = tick.cancel()

  def receive = {
    case "tick" =>
      // do something useful here
  }
}
```

The second variant sets up an initial one shot message send in the `preStart` method of the actor, and the then the actor when it receives this message sets up a new one shot message send. You also have to override `postRestart` so we don't call `preStart` and schedule the initial message send again.

Note: With this approach we won't fill up the mailbox with tick messages if the actor is under pressure, but only schedule a new tick message when we have seen the previous one.

```
class ScheduleInReceive extends Actor {
  import context._

  override def preStart() =
    system.scheduler.scheduleOnce(500 millis, self, "tick")

  // override postRestart so we don't call preStart and schedule a new message
  override def postRestart(reason: Throwable) = {}

  def receive = {
    case "tick" =>
      // send another periodic tick after the specified delay
      system.scheduler.scheduleOnce(1000 millis, self, "tick")
      // do something useful here
  }
}
```

CHAPTER

TEN

EXPERIMENTAL MODULES

The following modules of Akka are marked as experimental, which means that they are in early access mode, which also means that they are not covered by commercial support. The purpose of releasing them early, as experimental, is to make them easily available and improve based on feedback, or even discover that the module wasn't useful.

An experimental module doesn't have to obey the rule of staying binary compatible between micro releases. Breaking API changes may be introduced in minor releases without notice as we refine and simplify based on your feedback. An experimental module may be dropped in minor releases without prior deprecation.

10.1 Multi Node Testing

10.1.1 Multi Node Testing Concepts

When we talk about multi node testing in Akka we mean the process of running coordinated tests on multiple actor systems in different JVMs. The multi node testing kit consist of three main parts.

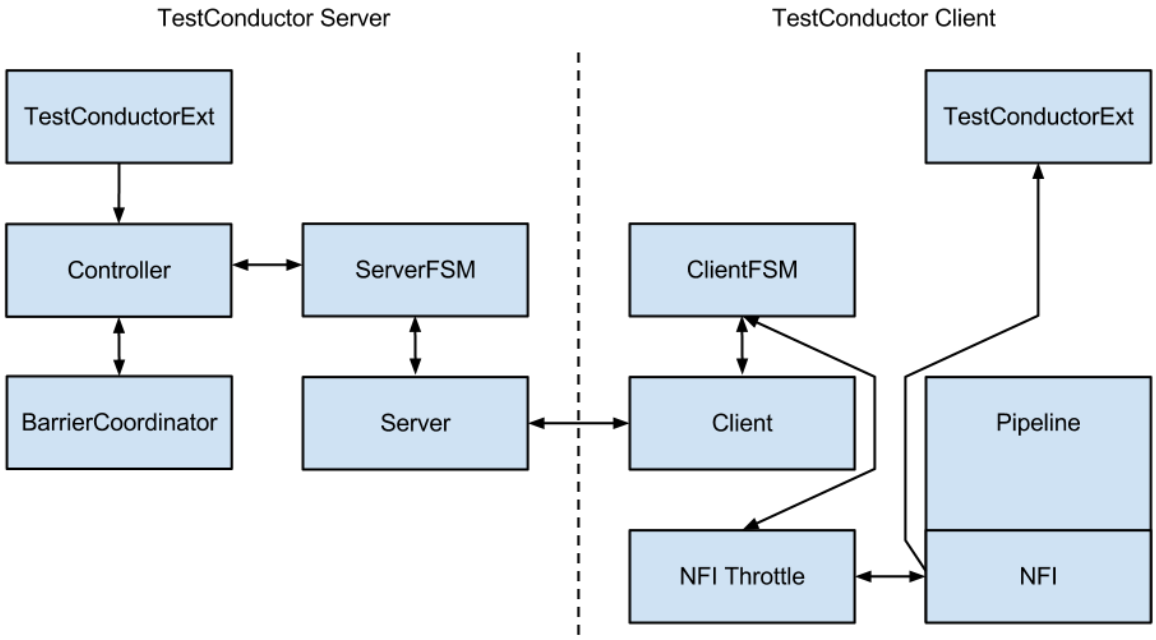
- *The Test Conductor*. that coordinates and controls the nodes under test.
- *The Multi Node Spec*. that is a convenience wrapper for starting the `TestConductor` and letting all nodes connect to it.
- *The SbtMultiJvm Plugin*. that starts tests in multiple JVMs possibly on multiple machines.

10.1.2 The Test Conductor

The basis for the multi node testing is the `TestConductor`. It is an Akka Extension that plugs in to the network stack and it is used to coordinate the nodes participating in the test and provides several features including:

- Node Address Lookup: Finding out the full path to another test node (No need to share configuration between test nodes)
- Node Barrier Coordination: Waiting for other nodes at named barriers.
- Network Failure Injection: Throttling traffic, dropping packets, unplugging and plugging nodes back in.

This is a schematic overview of the test conductor.



The test conductor server is responsible for coordinating barriers and sending commands to the test conductor clients that act upon them, e.g. throttling network traffic to/from another client. More information on the possible operations is available in the `akka.remote.testconductor.Conductor` API documentation.

10.1.3 The Multi Node Spec

The Multi Node Spec consists of two parts. The `MultiNodeConfig` that is responsible for common configuration and enumerating and naming the nodes under test. The `MultiNodeSpec` that contains a number of convenience functions for making the test nodes interact with each other. More information on the possible operations is available in the `akka.remote.testkit.MultiNodeSpec` API documentation.

The setup of the `MultiNodeSpec` is configured through java system properties that you set on all JVMs that's going to run a node under test. These can easily be set on the JVM command line with `-Dproperty=value`.

These are the available properties:

- `multinode.max-nodes`

  The maximum number of nodes that a test can have.

- `multinode.host`

  The host name or IP for this node. Must be resolvable using InetAddress.getByName.

- `multinode.port`

  The port number for this node. Defaults to 0 which will use a random port.

- `multinode.server-host`

  The host name or IP for the server node. Must be resolvable using InetAddress.getByName.

- `multinode.server-port`

  The port number for the server node. Defaults to 4711.

- `multinode.index`

  The index of this node in the sequence of roles defined for the test. The index 0 is special and that machine will be the server. All failure injection and throttling must be done from this node.

10.1.4 The SbtMultiJvm Plugin

The *SbtMultiJvm Plugin* has been updated to be able to run multi node tests, by automatically generating the relevant `multinode.*` properties. This means that you can easily run multi node tests on a single machine without any special configuration by just running them as normal multi-jvm tests. These tests can then be run distributed over multiple machines without any changes simply by using the multi-node additions to the plugin.

Multi Node Specific Additions

The plugin also has a number of new `multi-node-*` sbt tasks and settings to support running tests on multiple machines. The necessary test classes and dependencies are packaged for distribution to other machines with SbtAssembly into a jar file with a name on the format `<projectName>_<scalaVersion>-<projectVersion>-multi-jvm-assembly.jar`

Note: To be able to distribute and kick off the tests on multiple machines, it is assumed that both host and target systems are POSIX like systems with `ssh` and `rsync` available.

These are the available sbt multi-node settings:

- `multiNodeHosts`

  A sequence of hosts to use for running the test, on the form `user@host:java` where host is the only required part. Will override settings from file.

- `multiNodeHostsFileName`

  A file to use for reading in the hosts to use for running the test. One per line on the same format as above. Defaults to `multi-node-test.hosts` in the base project directory.

- `multiNodeTargetDirName`

  A name for the directory on the target machine, where to copy the jar file. Defaults to `multi-node-test` in the base directory of the ssh user used to rsync the jar file.

- `multiNodeJavaName`

  The name of the default Java executable on the target machines. Defaults to `java`.

Here are some examples of how you define hosts:

- `localhost`

  The current user on localhost using the default java.

- `user1@host1`

  User `user1` on host `host1` with the default java.

- `user2@host2:/usr/lib/jvm/java-7-openjdk-amd64/bin/java`

  User `user2` on host `host2` using java 7.

- `host3:/usr/lib/jvm/java-6-openjdk-amd64/bin/java`

  The current user on host `host3` using java 6.

Running the Multi Node Tests

To run all the multi node test in multi-node mode (i.e. distributing the jar files and kicking off the tests remotely) from inside sbt, use the `multi-node-test` task:

```
multi-node-test
```

To run all of them in multi-jvm mode (i.e. all JVMs on the local machine) do:

```
multi-jvm:test
```

To run individual tests use the `multi-node-test-only` task:

```
multi-node-test-only your.MultiNodeTest
```

To run individual tests in the multi-jvm mode do:

```
multi-jvm:test-only your.MultiNodeTest
```

More than one test name can be listed to run multiple specific tests. Tab completion in sbt makes it easy to complete the test names.

10.1.5 Preparing Your Project for Multi Node Testing

The multi node testing kit is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-multi-node-testkit" % "2.4.14"
```

If you are using the latest nightly build you should pick a timestamped Akka version from http://repo.akka.io/snapshots/com/typesafe/akka/akka-multi-node-testkit_2.11/. We recommend against using `SNAPSHOT` in order to obtain stable builds.

10.1.6 A Multi Node Testing Example

First we need some scaffolding to hook up the `MultiNodeSpec` with your favorite test framework. Lets define a trait `STMultiNodeSpec` that uses ScalaTest to start and stop `MultiNodeSpec`.

```
package sample.multinode

import org.scalatest.{ BeforeAndAfterAll, WordSpecLike }
import org.scalatest.Matchers
import akka.remote.testkit.MultiNodeSpecCallbacks

/**
 * Hooks up MultiNodeSpec with ScalaTest
 */
trait STMultiNodeSpec extends MultiNodeSpecCallbacks
  with WordSpecLike with Matchers with BeforeAndAfterAll {

  override def beforeAll() = multiNodeSpecBeforeAll()

  override def afterAll() = multiNodeSpecAfterAll()
}
```

Then we need to define a configuration. Lets use two nodes `"node1` and `"node2"` and call it `MultiNodeSampleConfig`.

```
package sample.multinode
import akka.remote.testkit.MultiNodeConfig

object MultiNodeSampleConfig extends MultiNodeConfig {
  val node1 = role("node1")
  val node2 = role("node2")
}
```

And then finally to the node test code. That starts the two nodes, and demonstrates a barrier, and a remote actor message send/receive.

```
package sample.multinode
import akka.remote.testkit.MultiNodeSpec
import akka.testkit.ImplicitSender
import akka.actor.{ Props, Actor }

class MultiNodeSampleSpecMultiJvmNode1 extends MultiNodeSample
class MultiNodeSampleSpecMultiJvmNode2 extends MultiNodeSample

object MultiNodeSample {
  class Ponger extends Actor {
    def receive = {
      case "ping" => sender() ! "pong"
    }
  }
}

class MultiNodeSample extends MultiNodeSpec(MultiNodeSampleConfig)
  with STMultiNodeSpec with ImplicitSender {

  import MultiNodeSampleConfig._
  import MultiNodeSample._

  def initialParticipants = roles.size

  "A MultiNodeSample" must {

    "wait for all nodes to enter a barrier" in {
      enterBarrier("startup")
    }

    "send to and receive from a remote node" in {
      runOn(node1) {
        enterBarrier("deployed")
        val ponger = system.actorSelection(node(node2) / "user" / "ponger")
        ponger ! "ping"
        import scala.concurrent.duration._
        expectMsg(10.seconds, "pong")
      }

      runOn(node2) {
        system.actorOf(Props[Ponger], "ponger")
        enterBarrier("deployed")
      }

      enterBarrier("finished")
    }
  }
}
```

The easiest way to run this example yourself is to download Lightbend Activator and open the tutorial named Akka Multi-Node Testing Sample with Scala.

10.1.7 Things to Keep in Mind

There are a couple of things to keep in mind when writing multi node tests or else your tests might behave in surprising ways.

- Don't issue a shutdown of the first node. The first node is the controller and if it shuts down your test will break.
- To be able to use `blackhole`, `passThrough`, and `throttle` you must activate the failure injector and throttler transport adapters by specifying `testTransport(on = true)` in your MultiNodeConfig.

- Throttling, shutdown and other failure injections can only be done from the first node, which again is the controller.
- Don't ask for the address of a node using `node(address)` after the node has been shut down. Grab the address before shutting down the node.
- Don't use MultiNodeSpec methods like address lookup, barrier entry et.c. from other threads than the main test thread. This also means that you shouldn't use them from inside an actor, a future, or a scheduled task.

10.1.8 Configuration

There are several configuration properties for the Multi-Node Testing module, please refer to the *reference configuration*.

10.2 Actors (Java with Lambda Support)

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

The API of Akka's Actors is similar to Scala Actors which has borrowed some of its syntax from Erlang.

Warning: The Java with lambda support part of Akka is marked as **"experimental"** as of its introduction in Akka 2.3.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum, but the binary compatibility guarantee for maintenance releases does not apply to the `akka.actor.AbstractActor`, related classes and the `akka.japi.pf` package.

10.2.1 Creating Actors

Note: Since Akka enforces parental supervision every actor is supervised and (potentially) the supervisor of its children, it is advisable that you familiarize yourself with *Actor Systems* and *Supervision and Monitoring* and it may also help to read *Actor References, Paths and Addresses*.

Defining an Actor class

Actor classes are implemented by extending the `AbstractActor` class and setting the "initial behavior" in the constructor by calling the `receive` method in the `AbstractActor`.

The argument to the `receive` method is a `PartialFunction<Object,BoxedUnit>` that defines which messages your Actor can handle, along with the implementation of how the messages should be processed.

Don't let the type signature scare you. To allow you to easily build up a partial function there is a builder named `ReceiveBuilder` that you can use.

Here is an example:

```
import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;
```

```
public class MyActor extends AbstractActor {
  private final LoggingAdapter log = Logging.getLogger(context().system(), this);

  public MyActor() {
    receive(ReceiveBuilder.
      match(String.class, s -> {
        log.info("Received String message: {}", s);
      }).
      matchAny(o -> log.info("received unknown message")).build()
    );
  }
}
```

In case you want to provide many `match` cases but want to avoid creating a long call trail, you can split the creation of the builder into multiple statements as in the example:

```
import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;
import akka.japi.pf.UnitPFBuilder;

public class GraduallyBuiltActor extends AbstractActor {
  private final LoggingAdapter log = Logging.getLogger(context().system(), this);

  public GraduallyBuiltActor() {
    UnitPFBuilder<Object> builder = ReceiveBuilder.create();
    builder.match(String.class, s -> {
      log.info("Received String message: {}", s);
    });
    // do some other stuff in between
    builder.matchAny(o -> log.info("received unknown message"));
    receive(builder.build());
  }
}
```

Please note that the Akka Actor `receive` message loop is exhaustive, which is different compared to Erlang and the late Scala Actors. This means that you need to provide a pattern match for all messages that it can accept and if you want to be able to handle unknown messages then you need to have a default case as in the example above. Otherwise an `akka.actor.UnhandledMessage(message,sender,recipient)` will be published to the `ActorSystem`'s `EventStream`.

Note further that the return type of the behavior defined above is `Unit`; if the actor shall reply to the received message then this must be done explicitly as explained below.

The argument to the `receive` method is a partial function object, which is stored within the actor as its "initial behavior", see *Become/Unbecome* for further information on changing the behavior of an actor after its construction.

Props

`Props` is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information (e.g. which dispatcher to use, see more below). Here are some examples of how to create a `Props` instance.

```
import akka.actor.Props;
```

```
Props props1 = Props.create(MyActor.class);
Props props2 = Props.create(ActorWithArgs.class,
  () -> new ActorWithArgs("arg")); // careful, see below
Props props3 = Props.create(ActorWithArgs.class, "arg");
```

The second variant shows how to pass constructor arguments to the `Actor` being created, but it should only be used outside of actors as explained below.

The last line shows a possibility to pass constructor arguments regardless of the context it is being used in. The presence of a matching constructor is verified during construction of the `Props` object, resulting in an `IllegalArgumentException` if no or multiple matching constructors are found.

Dangerous Variants

```
// NOT RECOMMENDED within another actor:
// encourages to close over enclosing class
Props props7 = Props.create(ActorWithArgs.class,
  () -> new ActorWithArgs("arg"));
```

This method is not recommended to be used within another actor because it encourages to close over the enclosing scope, resulting in non-serializable `Props` and possibly race conditions (breaking the actor encapsulation). On the other hand using this variant in a `Props` factory in the actor's companion object as documented under "Recommended Practices" below is completely fine.

There were two use-cases for these methods: passing constructor arguments to the actor—which is solved by the newly introduced `Props.create(clazz, args)` method above or the recommended practice below—and creating actors "on the spot" as anonymous classes. The latter should be solved by making these actors named classes instead (if they are not declared within a top-level `object` then the enclosing instance's `this` reference needs to be passed as the first argument).

Warning: Declaring one actor within another is very dangerous and breaks actor encapsulation. Never pass an actor's `this` reference into `Props`!

Recommended Practices

It is a good idea to provide factory methods on the companion object of each `Actor` which help keeping the creation of suitable `Props` as close to the actor definition as possible. This also avoids the pitfalls associated with using the `Props.create(...)` method which takes a by-name argument, since within a companion object the given code block will not retain a reference to its enclosing scope:

```
public class DemoActor extends AbstractActor {
  /**
   * Create Props for an actor of this type.
   * @param magicNumber The magic number to be passed to this actor's constructor.
   * @return a Props for creating this actor, which can then be further configured
   *         (e.g. calling `.withDispatcher()` on it)
   */
  static Props props(Integer magicNumber) {
    // You need to specify the actual type of the returned actor
    // since Java 8 lambdas have some runtime type information erased
    return Props.create(DemoActor.class, () -> new DemoActor(magicNumber));
  }

  private final Integer magicNumber;

  DemoActor(Integer magicNumber) {
    this.magicNumber = magicNumber;
    receive(ReceiveBuilder.
      match(Integer.class, i -> {
        sender().tell(i + magicNumber, self());
      }).build()
    );
  }
```

```
}

public class SomeOtherActor extends AbstractActor {
  // Props(new DemoActor(42)) would not be safe
  ActorRef demoActor = context().actorOf(DemoActor.props(42), "demo");
  // ...
}
```

Another good practice is to declare what messages an Actor can receive as close to the actor definition as possible (e.g. as static classes inside the Actor or using other suitable class), which makes it easier to know what it can receive.

```
public class DemoMessagesActor extends AbstractLoggingActor {

  static public class Greeting {
    private final String from;

    public Greeting(String from) {
      this.from = from;
    }

    public String getGreeter() {
      return from;
    }
  }

  DemoMessagesActor() {
    receive(ReceiveBuilder.
      match(Greeting.class, g -> {
        log().info("I was greeted by {}", g.getGreeter());
      }).build()
    );
  };
}
```

Creating Actors with Props

Actors are created by passing a `Props` instance into the `actorOf` factory method which is available on `ActorSystem` and `ActorContext`.

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
```

```
// ActorSystem is a heavy object: create only one per application
final ActorSystem system = ActorSystem.create("MySystem", config);
final ActorRef myActor = system.actorOf(Props.create(MyActor.class), "myactor");
```

Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

```
public class FirstActor extends AbstractActor {
  final ActorRef child = context().actorOf(Props.create(MyActor.class), "myChild");
  // plus some behavior ...
}
```

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see *Actor Systems*.

The call to `actorOf` returns an instance of `ActorRef`. This is a handle to the actor instance and the only way to interact with it. The `ActorRef` is immutable and has a one to one relationship with the Actor it represents. The

`ActorRef` is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same Actor on the original node, across the network.

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with `$`, but it may contain URL encoded characters (eg. `%20` for a blank space). If the given name is already in use by another child to the same parent an *InvalidActorNameException* is thrown.

Actors are automatically started asynchronously when created.

Dependency Injection

If your UntypedActor has a constructor that takes parameters then those need to be part of the `Props` as well, as described *above*. But there are cases when a factory method must be used, for example when the actual constructor arguments are determined by a dependency injection framework.

```
import akka.actor.Actor;
import akka.actor.IndirectActorProducer;
```

```
class DependencyInjector implements IndirectActorProducer {
  final Object applicationContext;
  final String beanName;

  public DependencyInjector(Object applicationContext, String beanName) {
    this.applicationContext = applicationContext;
    this.beanName = beanName;
  }

  @Override
  public Class<? extends Actor> actorClass() {
    return MyActor.class;
  }

  @Override
  public MyActor produce() {
    MyActor result;
    // obtain fresh Actor instance from DI framework ...
    return result;
  }
}

  final ActorRef myActor = getContext().actorOf(
    Props.create(DependencyInjector.class, applicationContext, "MyActor"),
      "myactor3");
```

Warning: You might be tempted at times to offer an `IndirectActorProducer` which always returns the same instance, e.g. by using a static field. This is not supported, as it goes against the meaning of an actor restart, which is described here: *What Restarting Means*.

When using a dependency injection framework, actor beans *MUST NOT* have singleton scope.

Techniques for dependency injection and integration with dependency injection frameworks are described in more depth in the Using Akka with Dependency Injection guideline and the Akka Java Spring tutorial in Lightbend Activator.

The Inbox

When writing code outside of actors which shall communicate with actors, the `ask` pattern can be a solution (see below), but there are two things it cannot do: receiving multiple replies (e.g. by subscribing an `ActorRef` to a

notification service) and watching other actors' lifecycle. For these purposes there is the `Inbox` class:

```
final Inbox inbox = Inbox.create(system);
inbox.send(target, "hello");
try {
  assert inbox.receive(Duration.create(1, TimeUnit.SECONDS)).equals("world");
} catch (java.util.concurrent.TimeoutException e) {
  // timeout
}
```

The `send` method wraps a normal `tell` and supplies the internal actor's reference as the sender. This allows the reply to be received on the last line. Watching an actor is quite simple as well:

```
final Inbox inbox = Inbox.create(system);
inbox.watch(target);
target.tell(PoisonPill.getInstance(), ActorRef.noSender());
try {
  assert inbox.receive(Duration.create(1, TimeUnit.SECONDS)) instanceof Terminated;
} catch (java.util.concurrent.TimeoutException e) {
  // timeout
}
```

10.2.2 Actor API

The `AbstractActor` class defines a method called `receive`, that is used to set the "initial behavior" of the actor.

If the current actor behavior does not match a received message, `unhandled` is called, which by default publishes an `akka.actor.UnhandledMessage(message,sender,recipient)` on the actor system's event stream (set configuration item `akka.actor.debug.unhandled` to `on` to have them converted into actual Debug messages).

In addition, it offers:

- `self` reference to the `ActorRef` of the actor
- `sender` reference sender Actor of the last received message, typically used as described in *Reply to messages*
- `supervisorStrategy` user overridable definition the strategy to use for supervising child actors

  This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the `sender` reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).
- `context` exposes contextual information for the actor and the current message, such as:
  - factory methods to create child actors (`actorOf`)
  - system that the actor belongs to
  - parent supervisor
  - supervised children
  - lifecycle monitoring
  - hotswap behavior stack as described in *Become/Unbecome*

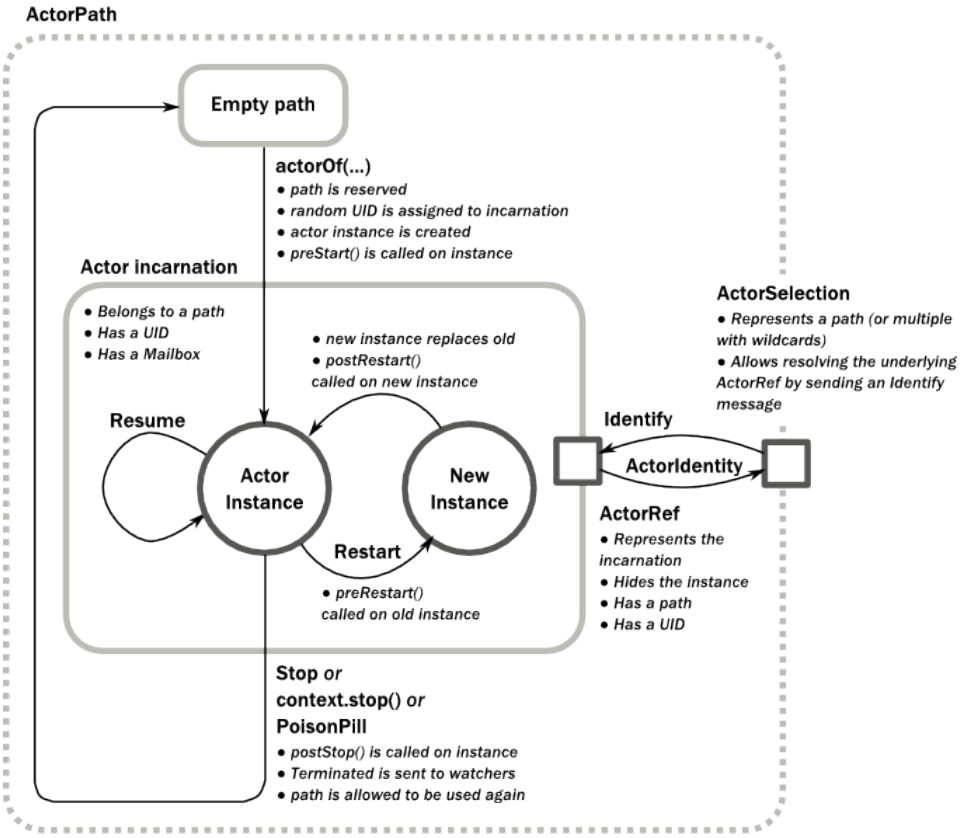
The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

```
public void preStart() {
}
```

```
public void preRestart(Throwable reason, scala.Option<Object> message) {
  for (ActorRef each : getContext().getChildren()) {
    getContext().unwatch(each);
    getContext().stop(each);
  }
  postStop();
}

public void postRestart(Throwable reason) {
  preStart();
}

public void postStop() {
}
```

The implementations shown above are the defaults provided by the `AbstractActor` class.

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When `actorOf()` is called it assigns an *incarnation* of the actor described by the passed `Props` to the given path. An actor incarnation is identified by the path *and a UID*. A restart only swaps the `Actor` instance defined by the `Props` but the incarnation and hence the UID remains the same.

The lifecycle of an incarnation ends when the actor is stopped. At that point the appropriate lifecycle events are called and watching actors are notified of the termination. After the incarnation is stopped, the path can be reused again by creating an actor with `actorOf()`. In this case the name of the new incarnation will be the same as the previous one but the UIDs will differ.

Note: It is important to note that Actors do not stop automatically when no longer referenced, every Actor that is created must also explicitly be destroyed. The only simplification is that stopping a parent Actor will also recursively stop all the child Actors that this parent has created.

An `ActorRef` always represents an incarnation (path and UID) not just a given path. Therefore if an actor is stopped and a new one with the same name is created an `ActorRef` of the old incarnation will not point to the new one.

`ActorSelection` on the other hand points to the path (or multiple paths if wildcards are used) and is completely oblivious to which incarnation is currently occupying it. `ActorSelection` cannot be watched for this reason. It is possible to resolve the current incarnation's `ActorRef` living under the path by sending an `Identify` message to the `ActorSelection` which will be replied to with an `ActorIdentity` containing the correct reference (see *Identifying Actors via Actor Selection*). This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the `Terminated` message dispatched by the other actor upon termination (see *Stopping Actors*). This service is provided by the `DeathWatch` component of the actor system.

Registering a monitor is easy:

```
public class WatchActor extends AbstractActor {
  private final ActorRef child = context().actorOf(Props.empty(), "target");
  private ActorRef lastSender = system.deadLetters();

  public WatchActor() {
    context().watch(child); // <-- this is the only call needed for registration

    receive(ReceiveBuilder.
      matchEquals("kill", s -> {
        context().stop(child);
        lastSender = sender();
      }).
      match(Terminated.class, t -> t.actor().equals(child), t -> {
        lastSender.tell("finished", self());
      }).build()
    );
  }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. In particular, the watching actor will receive a `Terminated` message even if the watched actor has already been terminated at the time of registration.

Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the `Terminated` message.

It is also possible to deregister from watching another actor's liveliness using `context.unwatch(target)`. This works even if the `Terminated` message has already been enqueued in the mailbox; after calling `unwatch` no `Terminated` message for that actor will be processed anymore.

Start Hook

Right after starting the actor, its `preStart` method is invoked.

```
@Override
public void preStart() {
  target = context().actorOf(Props.create(MyActor.class, "target"));
}
```

This method is called when the actor is first created. During restarts it is called by the default implementation of `postRestart`, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the actor's constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see *Supervision and Monitoring*). This restart involves the hooks mentioned above:

1. The old actor is informed by calling `preRestart` with the exception which caused the restart and the message which triggered that exception; the latter may be `None` if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling `sender`).

   This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc. By default it stops all children and calls `postStop`.

2. The initial factory from the `actorOf` call is used to produce the fresh instance.

3. The new actor's `postRestart` method is invoked with the exception which caused the restart. By default the `preStart` is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the `postRestart` hook returns. The message that triggered the exception will not be received again. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

> **Warning:** Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See *Discussion: Message Ordering* for details.

Stop Hook

After stopping an actor, its `postStop` hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. messages sent to a stopped actor will be redirected to the `deadLetters` of the `ActorSystem`.

10.2.3 Identifying Actors via Actor Selection

As described in *Actor References, Paths and Addresses*, each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an `ActorSelection` with the result:

```
// will look up this absolute path
context().actorSelection("/user/serviceA/actor");
// will look up sibling beneath same supervisor
context().actorSelection("../joe");
```

Note: It is always preferable to communicate with other Actors using their ActorRef instead of relying upon ActorSelection. Exceptions are

- sending messages using the at-least-once-delivery-java-lambda facility
- initiating first contact with a remote system

In all other cases ActorRefs can be provided during Actor creation or initialization, passing them from parent to child or introducing Actors by sending their ActorRefs to other Actors within messages.

The supplied path is parsed as a `java.net.URI`, which basically means that it is split on `/` into path elements. If the path starts with `/`, it is absolute and the look-up starts at the root guardian (which is the parent of `"/user"`); otherwise it starts at the current actor. If a path element equals `..`, the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the `..` in actor paths here always means the logical structure, i.e. the supervisor.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

```
// will look all children to serviceB with names starting with worker
context().actorSelection("/user/serviceB/worker*");
// will look up all siblings beneath same supervisor
context().actorSelection("../*");
```

Messages can be sent via the `ActorSelection` and the path of the `ActorSelection` is looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender()` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

```
import akka.actor.ActorIdentity;
import akka.actor.ActorSelection;
import akka.actor.Identify;
```

```
public class Follower extends AbstractActor {
  final Integer identifyId = 1;

  public Follower(){
    ActorSelection selection = context().actorSelection("/user/another");
    selection.tell(new Identify(identifyId), self());

    receive(ReceiveBuilder.
      match(ActorIdentity.class, id -> id.getRef() != null, id -> {
        ActorRef ref = id.getRef();
        context().watch(ref);
        context().become(active(ref));
      }).
      match(ActorIdentity.class, id -> id.getRef() == null, id -> {
        context().stop(self());
      }).build()
    );
  }
```

```
  final PartialFunction<Object, BoxedUnit> active(final ActorRef another) {
    return ReceiveBuilder.
      match(Terminated.class, t -> t.actor().equals(another), t -> {
        context().stop(self());
      }).build();
  }
}
```

You can also acquire an `ActorRef` for an `ActorSelection` with the `resolveOne` method of the `ActorSelection`. It returns a `Future` of the matching `ActorRef` if such an actor exists (see also actor-java-lambda for Java compatibility). It is completed with failure [[akka.actor.ActorNotFound]] if no such actor exists or the identification didn't complete within the supplied *timeout*.

Remote actor addresses may also be looked up, if remoting is enabled:

```
context().actorSelection("akka.tcp://app@otherhost:1234/user/serviceB");
```

An example demonstrating actor look-up is given in remote-sample-java.

10.2.4 Messages and immutability

IMPORTANT: Messages can be any kind of object but have to be immutable. Akka can't enforce immutability (yet) so this has to be by convention.

Here is an example of an immutable message:

```
public class ImmutableMessage {
  private final int sequenceNumber;
  private final List<String> values;

  public ImmutableMessage(int sequenceNumber, List<String> values) {
    this.sequenceNumber = sequenceNumber;
    this.values = Collections.unmodifiableList(new ArrayList<String>(values));
  }

  public int getSequenceNumber() {
    return sequenceNumber;
  }

  public List<String> getValues() {
    return values;
  }
}
```

10.2.5 Send messages

Messages are sent to an Actor through one of the following methods.

- `tell` means "fire-and-forget", e.g. send a message asynchronously and return immediately.
- `ask` sends a message asynchronously and returns a `Future` representing a possible reply.

Message ordering is guaranteed on a per-sender basis.

Note: There are performance implications of using `ask` since something needs to keep track of when it times out, there needs to be something that bridges a `Promise` into an `ActorRef` and it also needs to be reachable through remoting. So always prefer `tell` for performance, and only `ask` if you must.

In all these methods you have the option of passing along your own `ActorRef`. Make it a practice of doing so because it will allow the receiver actors to be able to respond to your message, since the sender reference is sent along with the message.

Tell: Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

```
// don't forget to think about who is the sender (2nd argument)
target.tell(message, self());
```

The sender reference is passed along with the message and available within the receiving actor via its `sender` method while processing this message. Inside of an actor it is usually `self` who shall be the sender, but there can be cases where replies shall be routed to some other actor—e.g. the parent—in which the second argument to `tell` would be a different one. Outside of an actor and if no reply is needed the second argument can be `null`; if a reply is needed outside of an actor you can use the ask-pattern described next..

Ask: Send-And-Receive-Future

The `ask` pattern involves actors as well as futures, hence it is offered as a use pattern rather than a method on `ActorRef`:

```
import static akka.pattern.Patterns.ask;
import static akka.pattern.Patterns.pipe;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.dispatch.Futures;
import akka.dispatch.Mapper;
import akka.util.Timeout;
```

```
final Timeout t = new Timeout(Duration.create(5, TimeUnit.SECONDS));

final ArrayList<Future<Object>> futures = new ArrayList<Future<Object>>();
futures.add(ask(actorA, "request", 1000)); // using 1000ms timeout
futures.add(ask(actorB, "another request", t)); // using timeout from
                                                // above

final Future<Iterable<Object>> aggregate = Futures.sequence(futures,
    system.dispatcher());

final Future<Result> transformed = aggregate.map(
    new Mapper<Iterable<Object>, Result>() {
      public Result apply(Iterable<Object> coll) {
        final Iterator<Object> it = coll.iterator();
        final String x = (String) it.next();
        final String s = (String) it.next();
        return new Result(x, s);
      }
    }, system.dispatcher());

pipe(transformed, system.dispatcher()).to(actorC);
```

This example demonstrates `ask` together with the `pipe` pattern on futures, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: `ask` produces a `Future`, two of which are composed into a new future using the `Futures.sequence` and `map` methods and then `pipe` installs an `onComplete`-handler on the future to effect the submission of the aggregated `Result` to another actor.

Using `ask` will send a message to the receiving Actor as with `tell`, and the receiving actor must reply with `sender().tell(reply,self())` in order to complete the returned `Future` with a value. The `ask` operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

Warning: To complete the future with an exception you need send a Failure message to the sender. This is *not done automatically* when an actor throws an exception while processing a message.

```
try {
  String result = operation();
  sender().tell(result, self());
} catch (Exception e) {
  sender().tell(new akka.actor.Status.Failure(e), self());
  throw e;
}
```

If the actor does not complete the future, it will expire after the timeout period, specified as parameter to the `ask` method; this will complete the `Future` with an `AskTimeoutException`.

See futures-java for more information on how to await or query a future.

The `onComplete`, `onSuccess`, or `onFailure` methods of the `Future` can be used to register a callback to get a notification when the Future completes. Gives you a way to avoid blocking.

Warning: When using future callbacks, inside actors you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time. See also: *Actors and shared mutable state*

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a 'mediator'. This can be useful when writing actors that work as routers, load-balancers, replicators etc.

```
target.forward(result, context());
```

10.2.6 Receive messages

An Actor either has to set its initial receive behavior in the constructor by calling the `receive` method in the `AbstractActor`:

```
public SomeActor() {
  receive(ReceiveBuilder.
    // and some behavior ...
  build());
}
```

or by implementing the `receive` method in the `Actor` interface:

```
public abstract PartialFunction<Object, BoxedUnit> receive();
```

Both the argument to the `AbstractActor` `receive` method and the return type of the `Actor` `receive` method is a `PartialFunction<Object,BoxedUnit>` that defines which messages your Actor can handle, along with the implementation of how the messages should be processed.

Don't let the type signature scare you. To allow you to easily build up a partial function there is a builder named `ReceiveBuilder` that you can use.

Here is an example:

```
import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;

public class MyActor extends AbstractActor {
  private final LoggingAdapter log = Logging.getLogger(context().system(), this);

  public MyActor() {
    receive(ReceiveBuilder.
      match(String.class, s -> {
        log.info("Received String message: {}", s);
      }).
      matchAny(o -> log.info("received unknown message")).build()
    );
  }
}
```

10.2.7 Reply to messages

If you want to have a handle for replying to a message, you can use `sender()`, which gives you an ActorRef. You can reply by sending to that ActorRef with `sender().tell(replyMsg, self())`. You can also store the ActorRef for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or future context) then the sender defaults to a 'dead-letter' actor ref.

```
sender().tell(s, self());
```

10.2.8 Receive timeout

The *ActorContext* `setReceiveTimeout` defines the inactivity timeout after which the sending of a *ReceiveTimeout* message is triggered. When specified, the receive function should be able to handle an *akka.actor.ReceiveTimeout* message. 1 millisecond is the minimum supported timeout.

Please note that the receive timeout might fire and enqueue the *ReceiveTimeout* message right after another message was enqueued; hence it is **not guaranteed** that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in *Duration.Undefined* to switch off this feature.

```
public class ReceiveTimeoutActor extends AbstractActor {
  public ReceiveTimeoutActor() {
    // To set an initial delay
    context().setReceiveTimeout(Duration.create("10 seconds"));

    receive(ReceiveBuilder.
      matchEquals("Hello", s -> {
        // To set in a response to a message
        context().setReceiveTimeout(Duration.create("1 second"));
      }).
      match(ReceiveTimeout.class, r -> {
        // To turn it off
        context().setReceiveTimeout(Duration.Undefined());
      }).build()
    );
```

```
  }
}
```

Messages marked with `NotInfluenceReceiveTimeout` will not reset the timer. This can be useful when `ReceiveTimeout` should be fired by external inactivity but not influenced by internal activity, e.g. scheduled tick messages.

10.2.9 Stopping actors

Actors are stopped by invoking the `stop` method of a `ActorRefFactory`, i.e. `ActorContext` or `ActorSystem`. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. `stop` may return before the actor is stopped.

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the `deadLetters` of the `ActorSystem`, but that depends on the mailbox implementation.

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the internal termination notifications from its children until the last one is gone, finally terminating itself (invoking `postStop`, dumping mailbox, publishing `Terminated` on the *DeathWatch*, telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon `ActorSystem.terminate`, the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

The `postStop` hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

```
@Override
public void postStop() {
  // clean up some resources ...
}
```

Note: Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an `InvalidActorNameException`. Instead, `watch` the terminating actor and create its replacement in response to the `Terminated` message which will eventually arrive.

PoisonPill

You can also send an actor the `akka.actor.PoisonPill` message, which will stop the actor when the message is processed. `PoisonPill` is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Graceful Stop

`gracefulStop` is useful if you need to wait for termination or compose ordered termination of several actors:

```
import static akka.pattern.Patterns.gracefulStop;
import scala.concurrent.Await;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.pattern.AskTimeoutException;
```

```
try {
  Future<Boolean> stopped =
    gracefulStop(actorRef, Duration.create(5, TimeUnit.SECONDS), Manager.SHUTDOWN);
  Await.result(stopped, Duration.create(6, TimeUnit.SECONDS));
  // the actor has been stopped
} catch (AskTimeoutException e) {
  // the actor wasn't stopped within 5 seconds
}
```

```
public class Manager extends AbstractActor {
  private static enum Shutdown {
    Shutdown
  }
  public static final Shutdown SHUTDOWN = Shutdown.Shutdown;

  private ActorRef worker =
  context().watch(context().actorOf(Props.create(Cruncher.class), "worker"));

  public Manager() {
    receive(ReceiveBuilder.
      matchEquals("job", s -> {
        worker.tell("crunch", self());
      }).
      matchEquals(SHUTDOWN, x -> {
        worker.tell(PoisonPill.getInstance(), self());
        context().become(shuttingDown);
      }).build()
    );
  }

  public PartialFunction<Object, BoxedUnit> shuttingDown =
    ReceiveBuilder.
      matchEquals("job", s -> {
        sender().tell("service unavailable, shutting down", self());
      }).
      match(Terminated.class, t -> t.actor().equals(worker), t -> {
        context().stop(self());
      }).build();
}
```

When `gracefulStop()` returns successfully, the actor's `postStop()` hook will have been executed: there exists a happens-before edge between the end of `postStop()` and the return of `gracefulStop()`.

In the above example a custom `Manager.Shutdown` message is sent to the target actor to initiate the process of stopping the actor. You can use `PoisonPill` for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in `postStop`.

Warning: Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after `gracefulStop()` returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a `Terminated` message, i.e. not for top-level actors.

10.2.10 Become/Unbecome

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime: invoke the `context.become` method from within the Actor. `become` takes a

`PartialFunction<Object,BoxedUnit>` that implements the new message handler. The hotswapped code is kept in a Stack which can be pushed and popped.

> **Warning:** Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor behavior using `become`:

```
public class HotSwapActor extends AbstractActor {
  private PartialFunction<Object, BoxedUnit> angry;
  private PartialFunction<Object, BoxedUnit> happy;

  public HotSwapActor() {
    angry =
      ReceiveBuilder.
        matchEquals("foo", s -> {
          sender().tell("I am already angry?", self());
        }).
        matchEquals("bar", s -> {
          context().become(happy);
        }).build();

    happy = ReceiveBuilder.
      matchEquals("bar", s -> {
        sender().tell("I am already happy :-)", self());
      }).
      matchEquals("foo", s -> {
        context().become(angry);
      }).build();

    receive(ReceiveBuilder.
      matchEquals("foo", s -> {
        context().become(angry);
      }).
      matchEquals("bar", s -> {
        context().become(happy);
      }).build()
    );
  }
}
```

This variant of the `become` method is useful for many different things, such as to implement a Finite State Machine (FSM, for an example see Dining Hakkers). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use `unbecome`, instead always the next behavior is explicitly installed.

The other way of using `become` does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of "pop" operations (i.e. `unbecome`) matches the number of "push" ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

```
public class Swapper extends AbstractLoggingActor {
  public Swapper() {
    receive(ReceiveBuilder.
      matchEquals(Swap, s -> {
          log().info("Hi");
          context().become(ReceiveBuilder.
                  matchEquals(Swap, x -> {
                    log().info("Ho");
                    context().unbecome(); // resets the latest 'become' (just for␣
↪fun)
                  }).build(), false); // push on top instead of replace
      }).build()
```

```
    );
  }
}

public class SwapperApp {
  public static void main(String[] args) {
    ActorSystem system = ActorSystem.create("SwapperSystem");
    ActorRef swapper = system.actorOf(Props.create(Swapper.class), "swapper");
    swapper.tell(Swap, ActorRef.noSender()); // logs Hi
    swapper.tell(Swap, ActorRef.noSender()); // logs Ho
    swapper.tell(Swap, ActorRef.noSender()); // logs Hi
    swapper.tell(Swap, ActorRef.noSender()); // logs Ho
    swapper.tell(Swap, ActorRef.noSender()); // logs Hi
    swapper.tell(Swap, ActorRef.noSender()); // logs Ho
    system.terminate();
  }
}
```

10.2.11 Stash

The `AbstractActorWithStash` class enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking `context().become()` or `context().unbecome()`, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally. An actor that extends `AbstractActorWithStash` will automatically get a deque-based mailbox.

Note: The abstract class `AbstractActorWithStash` implements the marker interface `RequiresMessageQueue<DequeBasedMessageQueueSemantics>` which requests the system to automatically choose a deque based mailbox implementation for the actor. If you want more control over the mailbox, see the documentation on mailboxes: mailboxes-java.

Here is an example of the `AbstractActorWithStash` class in action:

```
public class ActorWithProtocol extends AbstractActorWithStash {
  public ActorWithProtocol() {
    receive(ReceiveBuilder.
      matchEquals("open", s -> {
        context().become(ReceiveBuilder.
          matchEquals("write", ws -> { /* do writing */ }).
          matchEquals("close", cs -> {
            unstashAll();
            context().unbecome();
          }).
          matchAny(msg -> stash()).build(), false);
      }).
      matchAny(msg -> stash()).build()
    );
  }
}
```

Invoking `stash()` adds the current message (the message that the actor received last) to the actor's stash. It is typically invoked when handling the default case in the actor's message handler to stash messages that aren't handled by the other cases. It is illegal to stash the same message twice; to do so results in an `IllegalStateException` being thrown. The stash may also be bounded in which case invoking `stash()` may lead to a capacity violation, which results in a `StashOverflowException`. The capacity of the stash can be configured using the `stash-capacity` setting (an `Int`) of the mailbox's configuration.

Invoking `unstashAll()` enqueues messages from the stash to the actor's mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a `MessageQueueAppendFailedException` is thrown. The stash is guaranteed to be empty after calling `unstashAll()`.

The stash is backed by a `scala.collection.immutable.Vector`. As a result, even a very large number of messages may be stashed without a major impact on performance.

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor's state which have the same property. The `AbstractActorWithStash` implementation of `preRestart` will call `unstashAll()`, which is usually the desired behavior.

Note: If you want to enforce that your actor can only work with an unbounded stash, then you should use the `AbstractActorWithUnboundedStash` class instead.

10.2.12 Killing an Actor

You can kill an actor by sending a `Kill` message. This will cause the actor to throw a `ActorKilledException`, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See *What Supervision Means* for more information.

Use `Kill` like this:

```
victim.tell(akka.actor.Kill.getInstance(), ActorRef.noSender());
```

10.2.13 Actors and exceptions

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

What happens to the Message

If an exception is thrown while a message is being processed (i.e. taken out of its mailbox and handed over to the current behavior), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don't want a system to livelock (so consuming a lot of cpu cycles without making progress). Another possibility would be to have a look at the *PeekMailbox pattern*.

What happens to the mailbox

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox will be there as well.

What happens to the actor

If code within an actor throws an exception, that actor is suspended and the supervision process is started (see *Supervision and Monitoring*). Depending on the supervisor's decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.

10.2.14 Initialization patterns

The rich lifecycle hooks of Actors provide a useful toolkit to implement various initialization patterns. During the lifetime of an `ActorRef`, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the `ActorRef`.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the `ActorRef` is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use `val` fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via preStart

The method `preStart()` of an actor is only called once directly during the initialization of the first instance, that is, at creation of its `ActorRef`. In the case of restarts, `preStart()` is called from `postRestart()`, therefore if not overridden, `preStart()` is called on every incarnation. However, overriding `postRestart()` one can disable this behavior, and ensure that there is only one call to `preStart()`.

One useful usage of this pattern is to disable creation of new `ActorRefs` for children during restarts. This can be achieved by overriding `preRestart()`:

```
@Override
public void preStart() {
  // Initialize children here
}

// Overriding postRestart to disable the call to preStart()
// after restarts
@Override
public void postRestart(Throwable reason) {
}

// The default implementation of preRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override preRestart()
@Override
public void preRestart(Throwable reason, Option<Object> message)
  throws Exception {
  // Keep the call to postStop(), but no stopping of children
  postStop();
}
```

Please note, that the child actors are *still restarted*, but no new `ActorRef` is created. One can recursively apply the same principles for the children, ensuring that their `preStart()` method is called only at the creation of their refs.

For more information see *What Restarting Means*.

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization

message, and use `become()` or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

```
receive(ReceiveBuilder.
    matchEquals("init", m1 -> {
      initializeMe = "Up and running";
      context().become(ReceiveBuilder.
        matchEquals("U OK?", m2 -> {
          sender().tell(initializeMe, self());
        }).build());

    }).build()
```

If the actor may receive messages before it has been initialized, a useful tool can be the `Stash` to save messages until the initialization finishes, and replaying them after the actor became initialized.

Warning: This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an `ActorRef` in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

10.2.15 Lambdas and Performance

There is one big difference between the optimized partial functions created by the Scala compiler and the ones created by the `ReceiveBuilder`. The partial functions created by the `ReceiveBuilder` consist of multiple lambda expressions for every match statement, where each lambda is an object referencing the code to be run. This is something that the JVM can have problems optimizing and the resulting code might not be as performant as the Scala equivalent or the corresponding untyped actor version.

10.3 FSM (Java with Lambda Support)

10.3.1 Overview

The FSM (Finite State Machine) is available as an abstract base class that implements an Akka Actor and is best described in the Erlang design principles

A FSM can be described as a set of relations of the form:

> **State(S) x Event(E) -> Actions (A), State(S')**

These relations are interpreted as meaning:

> *If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.*

Warning: The Java with lambda support part of Akka is marked as **"experimental"** as of its introduction in Akka 2.3.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum, but the binary compatibility guarantee for maintenance releases does not apply to the `akka.actor.AbstractFSM`, related classes and the `akka.japi.pf` package.

10.3.2 A Simple Example

To demonstrate most of the features of the `AbstractFSM` class, consider an actor which shall receive and queue messages while they arrive in a burst and send them on after the burst ended or a flush request is received.

First, consider all of the below to use these import statements:

```
import akka.actor.AbstractFSM;
import akka.actor.ActorRef;
import akka.japi.pf.UnitMatch;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import scala.concurrent.duration.Duration;
```

The contract of our "Buncher" actor is that it accepts or produces the following messages:

```
public final class SetTarget {
  private final ActorRef ref;

  public SetTarget(ActorRef ref) {
    this.ref = ref;
  }

  public ActorRef getRef() {
    return ref;
  }
  // boilerplate ...
}

public final class Queue {
  private final Object obj;

  public Queue(Object obj) {
    this.obj = obj;
  }

  public Object getObj() {
    return obj;
  }
  // boilerplate ...
}

public final class Batch {
  private final List<Object> list;

  public Batch(List<Object> list) {
    this.list = list;
  }

  public List<Object> getList() {
    return list;
  }
  // boilerplate ...
}

public enum Flush {
  Flush
}
```

`SetTarget` is needed for starting it up, setting the destination for the `Batches` to be passed on; `Queue` will add to the internal queue while `Flush` will mark the end of a burst.

The actor can be in two states: no message queued (aka `Idle`) or some message queued (aka `Active`). The states and the state data is defined like this:

```
// states
enum State {
```

```
  Idle, Active
}

// state data
interface Data {
}

enum Uninitialized implements Data {
  Uninitialized
}

final class Todo implements Data {
  private final ActorRef target;
  private final List<Object> queue;

  public Todo(ActorRef target, List<Object> queue) {
    this.target = target;
    this.queue = queue;
  }

  public ActorRef getTarget() {
    return target;
  }

  public List<Object> getQueue() {
    return queue;
  }
  // boilerplate ...
}
```

The actor starts out in the idle state. Once a message arrives it will go to the active state and stay there as long as messages keep arriving and no flush is requested. The internal state data of the actor is made up of the target actor reference to send the batches to and the actual queue of messages.

Now let's take a look at the skeleton for our FSM actor:

```
public class Buncher extends AbstractFSM<State, Data> {
  {
    startWith(Idle, Uninitialized);

    when(Idle,
      matchEvent(SetTarget.class, Uninitialized.class,
        (setTarget, uninitialized) ->
          stay().using(new Todo(setTarget.getRef(), new LinkedList<>()))));

    // transition elided ...

    when(Active, Duration.create(1, "second"),
      matchEvent(Arrays.asList(Flush.class, StateTimeout()), Todo.class,
        (event, todo) -> goTo(Idle).using(todo.copy(new LinkedList<>()))));

    // unhandled elided ...

    initialize();
  }
}
```

The basic strategy is to declare the actor, by inheriting the `AbstractFSM` class and specifying the possible states and data values as type parameters. Within the body of the actor a DSL is used for declaring the state machine:

- `startWith` defines the initial state and initial data
- then there is one `when(<state>) { ... }` declaration per state to be handled (could potentially be

multiple ones, the passed `PartialFunction` will be concatenated using `orElse`)

- finally starting it up using `initialize`, which performs the transition into the initial state and sets up timers (if required).

In this case, we start out in the `Idle` and `Uninitialized` state, where only the `SetTarget()` message is handled; `stay` prepares to end this event's processing for not leaving the current state, while the `using` modifier makes the FSM replace the internal state (which is `Uninitialized` at this point) with a fresh `Todo()` object containing the target actor reference. The `Active` state has a state timeout declared, which means that if no message is received for 1 second, a `FSM.StateTimeout` message will be generated. This has the same effect as receiving the `Flush` command in this case, namely to transition back into the `Idle` state and resetting the internal queue to the empty vector. But how do messages get queued? Since this shall work identically in both states, we make use of the fact that any event which is not handled by the `when()` block is passed to the `whenUnhandled()` block:

```
whenUnhandled(
  matchEvent(Queue.class, Todo.class,
    (queue, todo) -> goTo(Active).using(todo.addElement(queue.getObj()))).
    anyEvent((event, state) -> {
      log().warning("received unhandled request {} in state {}/{}",
        event, stateName(), state);
      return stay();
    }));
```

The first case handled here is adding `Queue()` requests to the internal queue and going to the `Active` state (this does the obvious thing of staying in the `Active` state if already there), but only if the FSM data are not `Uninitialized` when the `Queue()` event is received. Otherwise—and in all other non-handled cases—the second case just logs a warning and does not change the internal state.

The only missing piece is where the `Batches` are actually sent to the target, for which we use the `onTransition` mechanism: you can declare multiple such blocks and all of them will be tried for matching behavior in case a state transition occurs (i.e. only when the state actually changes).

```
onTransition(
  matchState(Active, Idle, () -> {
    // reuse this matcher
    final UnitMatch<Data> m = UnitMatch.create(
      matchData(Todo.class,
        todo -> todo.getTarget().tell(new Batch(todo.getQueue()), self())));
    m.match(stateData());
  }).
  state(Idle, Active, () -> {/* Do something here */}));
```

The transition callback is a partial function which takes as input a pair of states—the current and the next state. During the state change, the old state data is available via `stateData` as shown, and the new state data would be available as `nextStateData`.

To verify that this buncher actually works, it is quite easy to write a test using the *Testing Actor Systems*, here using JUnit as an example:

```
public class BuncherTest extends AbstractJavaTest {

  static ActorSystem system;

  @BeforeClass
  public static void setup() {
    system = ActorSystem.create("BuncherTest");
  }

  @AfterClass
  public static void tearDown() {
    JavaTestKit.shutdownActorSystem(system);
    system = null;
```

```
  }

  @Test
  public void testBuncherActorBatchesCorrectly() {
    new JavaTestKit(system) {{
      final ActorRef buncher =
        system.actorOf(Props.create(Buncher.class));
      final ActorRef probe = getRef();

      buncher.tell(new SetTarget(probe), probe);
      buncher.tell(new Queue(42), probe);
      buncher.tell(new Queue(43), probe);
      LinkedList<Object> list1 = new LinkedList<>();
      list1.add(42);
      list1.add(43);
      expectMsgEquals(new Batch(list1));
      buncher.tell(new Queue(44), probe);
      buncher.tell(Flush, probe);
      buncher.tell(new Queue(45), probe);
      LinkedList<Object> list2 = new LinkedList<>();
      list2.add(44);
      expectMsgEquals(new Batch(list2));
      LinkedList<Object> list3 = new LinkedList<>();
      list3.add(45);
      expectMsgEquals(new Batch(list3));
      system.stop(buncher);
    }};
  }

  @Test
  public void testBuncherActorDoesntBatchUninitialized() {
    new JavaTestKit(system) {{
      final ActorRef buncher =
        system.actorOf(Props.create(Buncher.class));
      final ActorRef probe = getRef();

      buncher.tell(new Queue(42), probe);
      expectNoMsg();
      system.stop(buncher);
    }};
  }
}
```

10.3.3 Reference

The AbstractFSM Class

The `AbstractFSM` abstract class is the base class used to implement an FSM. It implements Actor since an Actor is created to drive the FSM.

```
public class Buncher extends AbstractFSM<State, Data> {
  {
    // fsm body ...
  }
}
```

Note: The AbstractFSM class defines a `receive` method which handles internal messages and passes everything else through to the FSM logic (according to the current state). When overriding the `receive` method, keep in

mind that e.g. state timeout handling depends on actually passing the messages through the FSM logic.

The `AbstractFSM` class takes two type parameters:

1. the supertype of all state names, usually an enum,
2. the type of the state data which are tracked by the `AbstractFSM` module itself.

Note: The state data together with the state name describe the internal state of the state machine; if you stick to this scheme and do not add mutable fields to the FSM class you have the advantage of making all changes of the internal state explicit in a few well-known places.

Defining States

A state is defined by one or more invocations of the method

```
when(<name>[, stateTimeout = <timeout>])(stateFunction).
```

The given name must be an object which is type-compatible with the first type parameter given to the `AbstractFSM` class. This object is used as a hash key, so you must ensure that it properly implements `equals` and `hashCode`; in particular it must not be mutable. The easiest fit for these requirements are case objects.

If the `stateTimeout` parameter is given, then all transitions into this state, including staying, receive this timeout by default. Initiating the transition with an explicit timeout may be used to override this default, see *Initiating Transitions* for more information. The state timeout of any state may be changed during action processing with `setStateTimeout(state, duration)`. This enables runtime configuration e.g. via external message.

The `stateFunction` argument is a `PartialFunction[Event, State]`, which is conveniently given using the state function builder syntax as demonstrated below:

```
when(Idle,
  matchEvent(SetTarget.class, Uninitialized.class,
    (setTarget, uninitialized) ->
      stay().using(new Todo(setTarget.getRef(), new LinkedList<>()))));
```

Warning: It is required that you define handlers for each of the possible FSM states, otherwise there will be failures when trying to switch to undeclared states.

It is recommended practice to declare the states as an enum and then verify that there is a `when` clause for each of the states. If you want to leave the handling of a state "unhandled" (more below), it still needs to be declared like this:

```
when(SomeState, AbstractFSM.NullFunction());
```

Defining the Initial State

Each FSM needs a starting point, which is declared using

```
startWith(state, data[, timeout])
```

The optionally given timeout argument overrides any specification given for the desired initial state. If you want to cancel a default timeout, use `Duration.Inf`.

Unhandled Events

If a state doesn't handle a received event a warning is logged. If you want to do something else in this case you can specify that with `whenUnhandled(stateFunction)`:

```
whenUnhandled(
  matchEvent(X.class, (x, data) -> {
    log().info("Received unhandled event: " + x);
    return stay();
  }).
  anyEvent((event, data) -> {
    log().warning("Received unknown event: " + event);
    return goTo(Error);
  }));
}
```

Within this handler the state of the FSM may be queried using the `stateName` method.

IMPORTANT: This handler is not stacked, meaning that each invocation of `whenUnhandled` replaces the previously installed handler.

Initiating Transitions

The result of any `stateFunction` must be a definition of the next state unless terminating the FSM, which is described in *Termination from Inside*. The state definition can either be the current state, as described by the `stay` directive, or it is a different state as given by `goto(state)`. The resulting object allows further qualification by way of the modifiers described in the following:

- `forMax(duration)`

  This modifier sets a state timeout on the next state. This means that a timer is started which upon expiry sends a `StateTimeout` message to the FSM. This timer is canceled upon reception of any other message in the meantime; you can rely on the fact that the `StateTimeout` message will not be processed after an intervening message.

  This modifier can also be used to override any default timeout which is specified for the target state. If you want to cancel the default timeout, use `Duration.Inf`.

- `using(data)`

  This modifier replaces the old state data with the new data given. If you follow the advice *above*, this is the only place where internal state data are ever modified.

- `replying(msg)`

  This modifier sends a reply to the currently processed message and otherwise does not modify the state transition.

All modifiers can be chained to achieve a nice and concise description:

```
when(SomeState, matchAnyEvent((msg, data) -> {
    return goTo(Processing).using(newData).
      forMax(Duration.create(5, SECONDS)).replying(WillDo);
}));
```

The parentheses are not actually needed in all cases, but they visually distinguish between modifiers and their arguments and therefore make the code even more pleasant to read for foreigners.

Note: Please note that the `return` statement may not be used in `when` blocks or similar; this is a Scala restriction. Either refactor your code using `if () ... else ...` or move it into a method definition.

Monitoring Transitions

Transitions occur "between states" conceptually, which means after any actions you have put into the event handling block; this is obvious since the next state is only defined by the value returned by the event handling logic.

You do not need to worry about the exact order with respect to setting the internal state variable, as everything within the FSM actor is running single-threaded anyway.

Internal Monitoring

Up to this point, the FSM DSL has been centered on states and events. The dual view is to describe it as a series of transitions. This is enabled by the method

```
onTransition(handler)
```

which associates actions with a transition instead of with a state and event. The handler is a partial function which takes a pair of states as input; no resulting state is needed as it is not possible to modify the transition in progress.

```
onTransition(
  matchState(Active, Idle, () -> setTimer("timeout",
    Tick, Duration.create(1, SECONDS), true)).
  state(Active, null, () -> cancelTimer("timeout")).
  state(null, Idle, (f, t) -> log().info("entering Idle from " + f)));
```

It is also possible to pass a function object accepting two states to `onTransition`, in case your transition handling logic is implemented as a method:

```
public void handler(StateType from, StateType to) {
  // handle transition here
}

onTransition(this::handler);
```

The handlers registered with this method are stacked, so you can intersperse `onTransition` blocks with `when` blocks as suits your design. It should be noted, however, that *all handlers will be invoked for each transition*, not only the first matching one. This is designed specifically so you can put all transition handling for a certain aspect into one place without having to worry about earlier declarations shadowing later ones; the actions are still executed in declaration order, though.

Note: This kind of internal monitoring may be used to structure your FSM according to transitions, so that for example the cancellation of a timer upon leaving a certain state cannot be forgot when adding new target states.

External Monitoring

External actors may be registered to be notified of state transitions by sending a message `SubscribeTransitionCallBack(actorRef)`. The named actor will be sent a `CurrentState(self, stateName)` message immediately and will receive `Transition(actorRef, oldState, newState)` messages whenever a new state is reached. External monitors may be unregistered by sending `UnsubscribeTransitionCallBack(actorRef)` to the FSM actor.

Stopping a listener without unregistering will not remove the listener from the subscription list; use `UnsubscribeTransitionCallback` before stopping the listener.

Timers

Besides state timeouts, FSM manages timers identified by `String` names. You may set a timer using

```
setTimer(name, msg, interval, repeat)
```

where `msg` is the message object which will be sent after the duration `interval` has elapsed. If `repeat` is `true`, then the timer is scheduled at fixed rate given by the `interval` parameter. Any existing timer with the same name will automatically be canceled before adding the new timer.

Timers may be canceled using

```
cancelTimer(name)
```

which is guaranteed to work immediately, meaning that the scheduled message will not be processed after this call even if the timer already fired and queued it. The status of any timer may be inquired with

```
isTimerActive(name)
```

These named timers complement state timeouts because they are not affected by intervening reception of other messages.

Termination from Inside

The FSM is stopped by specifying the result state as

```
stop([reason[, data]])
```

The reason must be one of `Normal` (which is the default), `Shutdown` or `Failure(reason)`, and the second argument may be given to change the state data which is available during termination handling.

Note: It should be noted that `stop` does not abort the actions and stop the FSM immediately. The stop action must be returned from the event handler in the same way as a state transition (but note that the `return` statement may not be used within a `when` block).

```
when(Error, matchEventEquals("stop", (event, data) -> {
  // do cleanup ...
  return stop();
}));
```

You can use `onTermination(handler)` to specify custom code that is executed when the FSM is stopped. The handler is a partial function which takes a `StopEvent(reason, stateName, stateData)` as argument:

```
onTermination(
  matchStop(Normal(),
    (state, data) -> {/* Do something here */}).
    stop(Shutdown(),
      (state, data) -> {/* Do something here */}).
    stop(Failure.class,
      (reason, state, data) -> {/* Do something here */}));
```

As for the `whenUnhandled` case, this handler is not stacked, so each invocation of `onTermination` replaces the previously installed handler.

Termination from Outside

When an `ActorRef` associated to a FSM is stopped using the `stop` method, its `postStop` hook will be executed. The default implementation by the `AbstractFSM` class is to execute the `onTermination` handler if that is prepared to handle a `StopEvent(Shutdown, ...)`.

> **Warning:** In case you override `postStop` and want to have your `onTermination` handler called, do not forget to call `super.postStop`.

10.3.4 Testing and Debugging Finite State Machines

During development and for trouble shooting FSMs need care just as any other actor. There are specialized tools available as described in *Testing Finite State Machines* and in the following.

Event Tracing

The setting `akka.actor.debug.fsm` in *Configuration* enables logging of an event trace by `LoggingFSM` instances:

```
public class MyFSM extends AbstractLoggingFSM<StateType, Data> {
  // body elided ...
}
```

This FSM will log at DEBUG level:

- all processed events, including `StateTimeout` and scheduled timer messages
- every setting and cancellation of named timers
- all state transitions

Life cycle changes and special messages can be logged as described for *Actors*.

Rolling Event Log

The `AbstractLoggingFSM` class adds one more feature to the FSM: a rolling event log which may be used during debugging (for tracing how the FSM entered a certain failure state) or for other creative uses:

```
public class MyFSM extends AbstractLoggingFSM<StateType, Data> {
  @Override
  public int logDepth() { return 12; }
  {
    onTermination(
      matchStop(Failure.class, (reason, state, data) -> {
        String lastEvents = getLog().mkString("\n\t");
        log().warning("Failure in state " + state + " with data " + data + "\n" +
          "Events leading up to this point:\n\t" + lastEvents);
      })
    );
    //...
  }
}
```

The `logDepth` defaults to zero, which turns off the event log.

> **Warning:** The log buffer is allocated during actor creation, which is why the configuration is done using a virtual method call. If you want to override with a `val`, make sure that its initialization happens before the initializer of `LoggingFSM` runs, and do not change the value returned by `logDepth` after the buffer has been allocated.

The contents of the event log are available using method `getLog`, which returns an `IndexedSeq[LogEntry]` where the oldest entry is at index zero.

10.3.5 Examples

A bigger FSM example contrasted with Actor's `become`/`unbecome` can be found in the Lightbend Activator template named Akka FSM in Scala

Another reason for marking a module as experimental is that it's too early to tell if the module has a maintainer that can take the responsibility of the module over time. These modules live in the `akka-contrib` subproject:

10.4 External Contributions

This subproject provides a home to modules contributed by external developers which may or may not move into the officially supported code base over time. The conditions under which this transition can occur include:

- there must be enough interest in the module to warrant inclusion in the standard distribution,
- the module must be actively maintained and
- code quality must be good enough to allow efficient maintenance by the Akka core development team

If a contributions turns out to not "take off" it may be removed again at a later time.

10.4.1 Caveat Emptor

A module in this subproject doesn't have to obey the rule of staying binary compatible between minor releases. Breaking API changes may be introduced in minor releases without notice as we refine and simplify based on your feedback. A module may be dropped in any release without prior deprecation. The Lightbend subscription does not cover support for these modules.

10.4.2 The Current List of Modules

Reliable Proxy Pattern

Looking at *Message Delivery Reliability* one might come to the conclusion that Akka actors are made for blue-sky scenarios: sending messages is the only way for actors to communicate, and then that is not even guaranteed to work. Is the whole paradigm built on sand? Of course the answer is an emphatic "No!".

A local message send—within the same JVM instance—is not likely to fail, and if it does the reason was one of

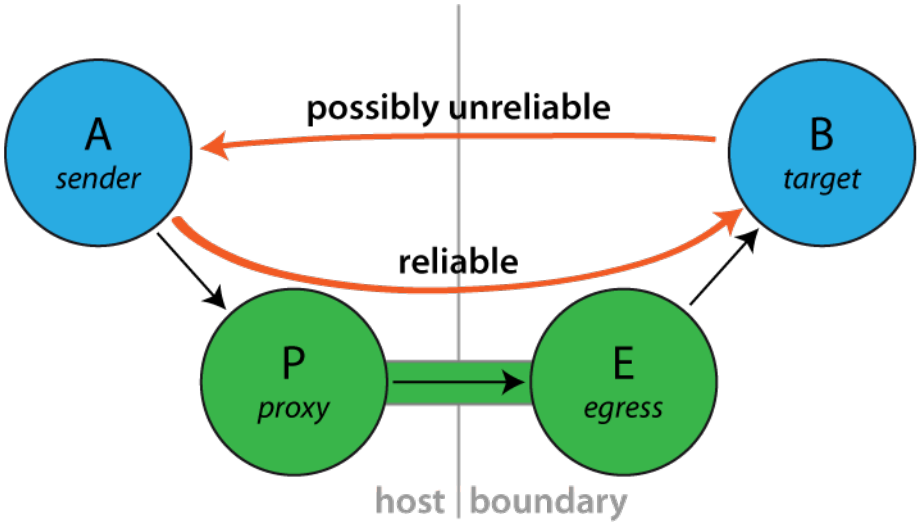
- it was meant to fail (due to consciously choosing a bounded mailbox, which upon overflow will have to drop messages)
- or it failed due to a catastrophic VM error, e.g. an `OutOfMemoryError`, a memory access violation ("segmentation fault", GPF, etc.), JVM bug—or someone calling `System.exit()`.

In all of these cases, the actor was very likely not in a position to process the message anyway, so this part of the non-guarantee is not problematic.

It is a lot more likely for an unintended message delivery failure to occur when a message send crosses JVM boundaries, i.e. an intermediate unreliable network is involved. If someone unplugs an ethernet cable, or a power failure shuts down a router, messages will be lost while the actors would be able to process them just fine.

Note: This does not mean that message send semantics are different between local and remote operations, it just means that in practice there is a difference between how good the "best effort" is.

Introducing the Reliable Proxy



To bridge the disparity between "local" and "remote" sends is the goal of this pattern. When sending from A to B must be as reliable as in-JVM, regardless of the deployment, then you can interject a reliable tunnel and send through that instead. The tunnel consists of two end-points, where the ingress point P (the "proxy") is a child of A and the egress point E is a child of P, deployed onto the same network node where B lives. Messages sent to P will be wrapped in an envelope, tagged with a sequence number and sent to E, who verifies that the received envelope has the right sequence number (the next expected one) and forwards the contained message to B. When B receives this message, the `sender()` will be a reference to the sender() of the original message to P. Reliability is added by E replying to orderly received messages with an ACK, so that P can tick those messages off its resend list. If ACKs do not come in a timely fashion, P will try to resend until successful.

Exactly what does it guarantee?

Sending via a `ReliableProxy` makes the message send exactly as reliable as if the represented target were to live within the same JVM, provided that the remote actor system does not terminate. In effect, both ends (i.e. JVM and actor system) must be considered as one when evaluating the reliability of this communication channel. The benefit is that the network in-between is taken out of that equation.

Connecting to the target

The `proxy` tries to connect to the `target` using the mechanism outlined in *Identifying Actors via Actor Selection*. Once connected, if the `tunnel` terminates the `proxy` will optionally try to reconnect to the target using using the same process.

Note that during the reconnection process there is a possibility that a message could be delivered to the `target` more than once. Consider the case where a message is delivered to the `target` and the target system crashes before the ACK is sent to the `sender`. After the `proxy` reconnects to the `target` it will start resending all of the messages that it has not received an ACK for, and the message that it never got an ACK for will be redelivered. Either this possibility should be considered in the design of the `target` or reconnection should be disabled.

How to use it

Since this implementation does not offer much in the way of configuration, simply instantiate a proxy wrapping a target `ActorPath`. From Java it looks like this:

```
import akka.contrib.pattern.ReliableProxy;


  public class ProxyParent extends UntypedActor {
    private final ActorRef proxy;

    public ProxyParent(ActorPath targetPath) {
      proxy = getContext().actorOf(
          ReliableProxy.props(targetPath,
              Duration.create(100, TimeUnit.MILLISECONDS)));
    }

    public void onReceive(Object msg) {
      if ("hello".equals(msg)) {
        proxy.tell("world!", getSelf());
      }
    }
  }
```

And from Scala like this:

```
import akka.contrib.pattern.ReliableProxy

class ProxyParent(targetPath: ActorPath) extends Actor {
  val proxy = context.actorOf(ReliableProxy.props(targetPath, 100.millis))

  def receive = {
    case "hello" ⇒ proxy ! "world!"
  }
}
```

Since the `ReliableProxy` actor is an *FSM*, it also offers the capability to subscribe to state transitions. If you need to know when all enqueued messages have been received by the remote end-point (and consequently been forwarded to the target), you can subscribe to the FSM notifications and observe a transition from state `ReliableProxy.Active` to state `ReliableProxy.Idle`.

```
public class ProxyTransitionParent extends UntypedActor {
  private final ActorRef proxy;
  private ActorRef client = null;

  public ProxyTransitionParent(ActorPath targetPath) {
    proxy = getContext().actorOf(
        ReliableProxy.props(targetPath,
            Duration.create(100, TimeUnit.MILLISECONDS)));
    proxy.tell(new FSM.SubscribeTransitionCallBack(getSelf()), getSelf());
  }

  public void onReceive(Object msg) {
    if ("hello".equals(msg)) {
      proxy.tell("world!", getSelf());
      client = getSender();
    } else if (msg instanceof FSM.CurrentState<?>) {
      // get initial state
    } else if (msg instanceof FSM.Transition<?>) {
      @SuppressWarnings("unchecked")
      final FSM.Transition<ReliableProxy.State> transition =
        (FSM.Transition<ReliableProxy.State>) msg;
      assert transition.fsmRef().equals(proxy);
      if (transition.from().equals(ReliableProxy.active()) &&
              transition.to().equals(ReliableProxy.idle())) {
        client.tell("done", getSelf());
      }
```

```
    }
  }
}
```

From Scala it would look like so:

```
class ProxyTransitionParent(targetPath: ActorPath) extends Actor {
  val proxy = context.actorOf(ReliableProxy.props(targetPath, 100.millis))
  proxy ! FSM.SubscribeTransitionCallBack(self)

  var client: ActorRef = _

  def receive = {
    case "go" ⇒
      proxy ! 42
      client = sender()
    case FSM.CurrentState(`proxy`, initial) ⇒
    case FSM.Transition(`proxy`, from, to) ⇒
      if (to == ReliableProxy.Idle)
        client ! "done"
  }
}
```

Configuration

- Set `akka.reliable-proxy.debug` to `on` to turn on extra debug logging for your `ReliableProxy` actors.
- `akka.reliable-proxy.default-connect-interval` is used only if you create a `ReliableProxy` with no reconnections (that is, `reconnectAfter == None`). The default value is the value of the configuration property `akka.remote.retry-gate-closed-for`. For example, if `akka.remote.retry-gate-closed-for` is `5 s` case the `ReliableProxy` will send an `Identify` message to the *target* every 5 seconds to try to resolve the `ActorPath` to an `ActorRef` so that messages can be sent to the *target*.

The Actor Contract

Message it Processes

- `FSM.SubscribeTransitionCallBack` and `FSM.UnsubscribeTransitionCallBack`, see *FSM*
- `ReliableProxy.Unsent`, see the API documentation for details.
- any other message is transferred through the reliable tunnel and forwarded to the designated target actor

Messages it Sends

- `FSM.CurrentState` and `FSM.Transition`, see *FSM*
- `ReliableProxy.TargetChanged` is sent to the FSM transition subscribers if the proxy reconnects to a new target.
- `ReliableProxy.ProxyTerminated` is sent to the FSM transition subscribers if the proxy is stopped.

Exceptions it Escalates

- no specific exception types
- any exception encountered by either the local or remote end-point are escalated (only fatal VM errors)

Arguments it Takes

- *target* is the `ActorPath` to the actor to which the tunnel shall reliably deliver messages, `B` in the above illustration.
- *retryAfter* is the timeout for receiving ACK messages from the remote end-point; once it fires, all outstanding message sends will be retried.
- *reconnectAfter* is an optional interval between connection attempts. It is also used as the interval between receiving a `Terminated` for the tunnel and attempting to reconnect to the target actor.
- *maxConnectAttempts* is an optional maximum number of attempts to connect to the target while in the `Connecting` state.

Throttling Actor Messages

Introduction

Suppose you are writing an application that makes HTTP requests to an external web service and that this web service has a restriction in place: you may not make more than 10 requests in 1 minute. You will get blocked or need to pay if you don't stay under this limit. In such a scenario you will want to employ a *message throttler*.

This extension module provides a simple implementation of a throttling actor, the `TimerBasedThrottler`.

How to use it

You can use a `TimerBasedThrottler` as follows. From Java it looks like this:

```
  // A simple actor that prints whatever it receives
  ActorRef printer = system.actorOf(Props.create(Printer.class));
  // The throttler for this example, setting the rate
  ActorRef throttler = system.actorOf(Props.create(TimerBasedThrottler.class,
      new Throttler.Rate(3, Duration.create(1, TimeUnit.SECONDS))));
  // Set the target
  throttler.tell(new Throttler.SetTarget(printer), null);
  // These three messages will be sent to the target immediately
  throttler.tell("1", null);
  throttler.tell("2", null);
  throttler.tell("3", null);
  // These two will wait until a second has passed
  throttler.tell("4", null);
  throttler.tell("5", null);

//A simple actor that prints whatever it receives
public class Printer extends UntypedActor {
  @Override
  public void onReceive(Object msg) {
    System.out.println(msg);
  }
}
```

And from Scala like this:

```
// A simple actor that prints whatever it receives
class PrintActor extends Actor {
  def receive = {
    case x ⇒ println(x)
  }
}

    val printer = system.actorOf(Props[PrintActor])
    // The throttler for this example, setting the rate
    val throttler = system.actorOf(Props(
      classOf[TimerBasedThrottler],
      3 msgsPer 1.second))
    // Set the target
    throttler ! SetTarget(Some(printer))
    // These three messages will be sent to the target immediately
    throttler ! "1"
    throttler ! "2"
    throttler ! "3"
    // These two will wait until a second has passed
    throttler ! "4"
    throttler ! "5"
```

Please refer to the JavaDoc/ScalaDoc documentation for the details.

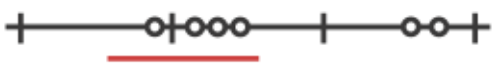
The guarantees

`TimerBasedThrottler` uses a timer internally. When the throttler's rate is 3 msg/s, for example, the throttler will start a timer that triggers every second and each time will give the throttler exactly three "vouchers"; each voucher gives the throttler a right to deliver a message. In this way, at most 3 messages will be sent out by the throttler in each interval.

It should be noted that such timer-based throttlers provide relatively **weak guarantees**:

- Only *start times* are taken into account. This may be a problem if, for example, the throttler is used to throttle requests to an external web service. If a web request takes very long on the server then the rate *observed on the server* may be higher.
- A timer-based throttler only makes guarantees for the intervals of its own timer. In our example, no more than 3 messages are delivered within such intervals. Other intervals on the timeline, however, may contain more calls.

The two cases are illustrated in the two figures below, each showing a timeline and three intervals of the timer. The message delivery times chosen by the throttler are indicated by dots, and as you can see, each interval contains at most 3 point, so the throttler works correctly. Still, there is in each example an interval (the red one) that is problematic. In the first scenario, this is because the delivery times are merely the start times of longer requests (indicated by the four bars above the timeline that start at the dots), so that the server observes four requests during the red interval. In the second scenario, the messages are centered around one of the points in time where the timer triggers, causing the red interval to contain too many messages.



For some application scenarios, the guarantees provided by a timer-based throttler might be too weak. Charles Cordingley's blog post discusses a throttler with stronger guarantees (it solves problem 2 from above). Future versions of this module may feature throttlers with better guarantees.

Java Logging (JUL)

This extension module provides a logging backend which uses the *java.util.logging* (j.u.l) API to do the endpoint logging for *akka.event.Logging*.

Provided with this module is an implementation of *akka.event.LoggingAdapter* which is independent of any *ActorSystem* being in place. This means that j.u.l can be used as the backend, via the Akka Logging API, for both Actor and non-Actor codebases.

To enable j.u.l as the *akka.event.Logging* backend, use the following Akka config:

loggers = ["akka.contrib.jul.JavaLogger"]

To access the *akka.event.Logging* API from non-Actor code, mix in *akka.contrib.jul.JavaLogging*.

This module is preferred over SLF4J with its JDK14 backend, due to integration issues resulting in the incorrect handling of *threadId*, *className* and *methodName*.

This extension module was contributed by Sam Halliday.

Mailbox with Explicit Acknowledgement

When an Akka actor is processing a message and an exception occurs, the normal behavior is for the actor to drop that message, and then continue with the next message after it has been restarted. This is in some cases not the desired solution, e.g. when using failure and supervision to manage a connection to an unreliable resource; the actor could after the restart go into a buffering mode (i.e. change its behavior) and retry the real processing later, when the unreliable resource is back online.

One way to do this is by sending all messages through the supervisor and buffering them there, acknowledging successful processing in the child; another way is to build an explicit acknowledgement mechanism into the mailbox. The idea with the latter is that a message is reprocessed in case of failure until the mailbox is told that processing was successful.

The pattern is implemented here. A demonstration of how to use it (although for brevity not a perfect example) is the following:

```
class MyActor extends Actor {
  def receive = {
    case msg ⇒
      println(msg)
      doStuff(msg) // may fail
      PeekMailboxExtension.ack()
  }

  // business logic elided ...
}

object MyApp extends App {
  val system = ActorSystem("MySystem", ConfigFactory.parseString("""
    peek-dispatcher {
      mailbox-type = "akka.contrib.mailbox.PeekMailboxType"
      max-retries = 2
    }
    """))

  val myActor = system.actorOf(
    Props[MyActor].withDispatcher("peek-dispatcher"),
    name = "myActor")
```

```
  myActor ! "Hello"
  myActor ! "World"
  myActor ! PoisonPill
}
```

Running this application (try it in the Akka sources by saying `sbt akka-contrib/test:run`) may produce the following output (note the processing of "World" on lines 2 and 16):

```
Hello
World
[ERROR] [12/17/2012 16:28:36.581] [MySystem-peek-dispatcher-5] [akka://MySystem/
↪user/myActor] DONTWANNA
java.lang.Exception: DONTWANNA
     at akka.contrib.mailbox.MyActor.doStuff(PeekMailbox.scala:105)
     at akka.contrib.mailbox.MyActor$$anonfun$receive$1.applyOrElse(PeekMailbox.
↪scala:98)
     at akka.actor.ActorCell.receiveMessage(ActorCell.scala:425)
     at akka.actor.ActorCell.invoke(ActorCell.scala:386)
     at akka.dispatch.Mailbox.processMailbox(Mailbox.scala:230)
     at akka.dispatch.Mailbox.run(Mailbox.scala:212)
     at akka.dispatch.ForkJoinExecutorConfigurator$MailboxExecutionTask.
↪exec(AbstractDispatcher.scala:502)
     at scala.concurrent.forkjoin.ForkJoinTask.doExec(ForkJoinTask.java:262)
     at scala.concurrent.forkjoin.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:
↪975)
     at scala.concurrent.forkjoin.ForkJoinPool.runWorker(ForkJoinPool.java:1478)
     at scala.concurrent.forkjoin.ForkJoinWorkerThread.run(ForkJoinWorkerThread.
↪java:104)
World
```

Normally one would want to make processing idempotent (i.e. it does not matter if a message is processed twice) or `context.become` a different behavior upon restart; the above example included the `println(msg)` call just to demonstrate the re-processing.

Aggregator Pattern

The aggregator pattern supports writing actors that aggregate data from multiple other actors and updates its state based on those responses. It is even harder to optionally aggregate more data based on the runtime state of the actor or take certain actions (sending another message and get another response) based on two or more previous responses.

A common thought is to use the ask pattern to request information from other actors. However, ask creates another actor specifically for the ask. We cannot use a callback from the future to update the state as the thread executing the callback is not defined. This will likely close-over the current actor.

The aggregator pattern solves such scenarios. It makes sure we're acting from the same actor in the scope of the actor receive.

Introduction

The aggregator pattern allows match patterns to be dynamically added to and removed from an actor from inside the message handling logic. All match patterns are called from the receive loop and run in the thread handling the incoming message. These dynamically added patterns and logic can safely read and/or modify this actor's mutable state without risking integrity or concurrency issues.

Usage

To use the aggregator pattern, you need to extend the `Aggregator` trait. The trait takes care of `receive` and actors extending this trait should not override `receive`. The trait provides the `expect`, `expectOnce`, and `unexpect` calls. The `expect` and `expectOnce` calls return a handle that can be used for later de-registration by passing the handle to `unexpect`.

`expect` is often used for standing matches such as catching error messages or timeouts.

```
expect {
  case TimedOut ⇒ collectBalances(force = true)
}
```

`expectOnce` is used for matching the initial message as well as other requested messages

```
expectOnce {
  case GetCustomerAccountBalances(id, types) ⇒
    new AccountAggregator(sender(), id, types)
  case _ ⇒
    sender() ! CantUnderstand
    context.stop(self)
}
```

```
def fetchCheckingAccountsBalance() {
  context.actorOf(Props[CheckingAccountProxy]) ! GetAccountBalances(id)
  expectOnce {
    case CheckingAccountBalances(balances) ⇒
      results += (Checking → balances)
      collectBalances()
  }
}
```

`unexpect` can be used for expecting multiple responses until a timeout or when the logic dictates such an `expect` no longer applies.

```
val handle = expect {
  case Response(name, value) ⇒
    values += value
    if (values.size > 3) processList()
  case TimedOut ⇒ processList()
}

def processList() {
  unexpect(handle)

  if (values.size > 0) {
    context.actorSelection("/user/evaluator") ! values.toList
    expectOnce {
      case EvaluationResults(name, eval) ⇒ processFinal(eval)
    }
  } else processFinal(List.empty[Int])
}
```

As the name eludes, `expect` keeps the partial function matching any received messages until `unexpect` is called or the actor terminates, whichever comes first. On the other hand, `expectOnce` removes the partial function once a match has been established.

It is a common pattern to register the initial expectOnce from the construction of the actor to accept the initial message. Once that message is received, the actor starts doing all aggregations and sends the response back to the original requester. The aggregator should terminate after the response is sent (or timed out). A different original request should use a different actor instance.

As you can see, aggregator actors are generally stateful, short lived actors.

Sample Use Case - AccountBalanceRetriever

This example below shows a typical and intended use of the aggregator pattern.

```
final case class InitialRequest(name: String)
final case class Request(name: String)
final case class Response(name: String, value: String)
final case class EvaluationResults(name: String, eval: List[Int])
final case class FinalResponse(qualifiedValues: List[String])

/**
 * An actor sample demonstrating use of unexpect and chaining.
 * This is just an example and not a complete test case.
 */
class ChainingSample extends Actor with Aggregator {

  expectOnce {
    case InitialRequest(name) ⇒ new MultipleResponseHandler(sender(), name)
  }

  class MultipleResponseHandler(originalSender: ActorRef, propName: String) {

    import context.dispatcher
    import collection.mutable.ArrayBuffer

    val values = ArrayBuffer.empty[String]

    context.actorSelection("/user/request_proxies") ! Request(propName)
    context.system.scheduler.scheduleOnce(50.milliseconds, self, TimedOut)

    val handle = expect {
      case Response(name, value) ⇒
        values += value
        if (values.size > 3) processList()
      case TimedOut ⇒ processList()
    }

    def processList() {
      unexpect(handle)

      if (values.size > 0) {
        context.actorSelection("/user/evaluator") ! values.toList
        expectOnce {
          case EvaluationResults(name, eval) ⇒ processFinal(eval)
        }
      } else processFinal(List.empty[Int])
    }

    def processFinal(eval: List[Int]) {
      // Select only the entries coming back from eval
      originalSender ! FinalResponse(eval map values)
      context.stop(self)
    }
  }
}

class AggregatorSpec extends TestKit(ActorSystem("AggregatorSpec")) with␣
↪ImplicitSender with FunSuiteLike with Matchers with BeforeAndAfterAll {

  override def afterAll(): Unit = {
    shutdown()
  }
```

```
  test("Test request 1 account type") {
    system.actorOf(Props[AccountBalanceRetriever]) ! GetCustomerAccountBalances(1,␣
↪Set(Savings))
    receiveOne(10.seconds) match {
      case result: List[_] ⇒
        result should have size 1
      case result ⇒
        assert(false, s"Expect List, got ${result.getClass}")
    }
  }

  test("Test request 3 account types") {
    system.actorOf(Props[AccountBalanceRetriever]) !
      GetCustomerAccountBalances(1, Set(Checking, Savings, MoneyMarket))
    receiveOne(10.seconds) match {
      case result: List[_] ⇒
        result should have size 3
      case result ⇒
        assert(false, s"Expect List, got ${result.getClass}")
    }
  }
}

final case class TestEntry(id: Int)

class WorkListSpec extends FunSuiteLike {

  val workList = WorkList.empty[TestEntry]
  var entry2: TestEntry = null
  var entry4: TestEntry = null

  test("Processing empty WorkList") {
    // ProcessAndRemove something in the middle
    val processed = workList process {
      case TestEntry(9) ⇒ true
      case _            ⇒ false
    }
    assert(!processed)
  }

  test("Insert temp entries") {
    assert(workList.head === workList.tail)

    val entry0 = TestEntry(0)
    workList.add(entry0, permanent = false)

    assert(workList.head.next != null)
    assert(workList.tail === workList.head.next)
    assert(workList.tail.ref.get === entry0)

    val entry1 = TestEntry(1)
    workList.add(entry1, permanent = false)

    assert(workList.head.next != workList.tail)
    assert(workList.head.next.ref.get === entry0)
    assert(workList.tail.ref.get === entry1)

    entry2 = TestEntry(2)
    workList.add(entry2, permanent = false)

    assert(workList.tail.ref.get === entry2)

    val entry3 = TestEntry(3)
```

```
    workList.add(entry3, permanent = false)

    assert(workList.tail.ref.get === entry3)
  }

  test("Process temp entries") {

    // ProcessAndRemove something in the middle
    assert(workList process {
      case TestEntry(2) ⇒ true
      case _            ⇒ false
    })

    // ProcessAndRemove the head
    assert(workList process {
      case TestEntry(0) ⇒ true
      case _            ⇒ false
    })

    // ProcessAndRemove the tail
    assert(workList process {
      case TestEntry(3) ⇒ true
      case _            ⇒ false
    })
  }

  test("Re-insert permanent entry") {
    entry4 = TestEntry(4)
    workList.add(entry4, permanent = true)

    assert(workList.tail.ref.get === entry4)
  }

  test("Process permanent entry") {
    assert(workList process {
      case TestEntry(4) ⇒ true
      case _            ⇒ false
    })
  }

  test("Remove permanent entry") {
    val removed = workList remove entry4
    assert(removed)
  }

  test("Remove temp entry already processed") {
    val removed = workList remove entry2
    assert(!removed)
  }

  test("Process non-matching entries") {

    val processed =
      workList process {
        case TestEntry(2) ⇒ true
        case _            ⇒ false
      }

    assert(!processed)

    val processed2 =
      workList process {
        case TestEntry(5) ⇒ true
```

```
      case _            ⇒ false
    }

  assert(!processed2)

}

test("Append two lists") {
  workList.removeAll()
  0 to 4 foreach { id ⇒ workList.add(TestEntry(id), permanent = false) }

  val l2 = new WorkList[TestEntry]
  5 to 9 foreach { id ⇒ l2.add(TestEntry(id), permanent = true) }

  workList addAll l2

  @tailrec
  def checkEntries(id: Int, entry: WorkList.Entry[TestEntry]): Int = {
    if (entry == null) id
    else {
      assert(entry.ref.get.id === id)
      checkEntries(id + 1, entry.next)
    }
  }

  assert(checkEntries(0, workList.head.next) === 10)
}

test("Clear list") {
  workList.removeAll()
  assert(workList.head.next === null)
  assert(workList.tail === workList.head)
}

val workList2 = WorkList.empty[PartialFunction[Any, Unit]]

val fn1: PartialFunction[Any, Unit] = {
  case s: String ⇒
    val result1 = workList2 remove fn1
    assert(result1 === true, "First remove must return true")
    val result2 = workList2 remove fn1
    assert(result2 === false, "Second remove must return false")
}

val fn2: PartialFunction[Any, Unit] = {
  case s: String ⇒
    workList2.add(fn1, permanent = true)
}

test("Reentrant insert") {
  workList2.add(fn2, permanent = false)
  assert(workList2.head.next != null)
  assert(workList2.tail == workList2.head.next)

  // Processing inserted fn1, reentrant adding fn2
  workList2 process { fn ⇒
    var processed = true
    fn.applyOrElse("Foo", (_: Any) ⇒ processed = false)
    processed
  }
}

test("Reentrant delete") {
```

```
    // Processing inserted fn2, should delete itself
    workList2 process { fn ⇒
      var processed = true
      fn.applyOrElse("Foo", (_: Any) ⇒ processed = false)
      processed
    }
  }
}
```

Sample Use Case - Multiple Response Aggregation and Chaining

A shorter example showing aggregating responses and chaining further requests.

```
final case class InitialRequest(name: String)
final case class Request(name: String)
final case class Response(name: String, value: String)
final case class EvaluationResults(name: String, eval: List[Int])
final case class FinalResponse(qualifiedValues: List[String])

/**
 * An actor sample demonstrating use of unexpect and chaining.
 * This is just an example and not a complete test case.
 */
class ChainingSample extends Actor with Aggregator {

  expectOnce {
    case InitialRequest(name) ⇒ new MultipleResponseHandler(sender(), name)
  }

  class MultipleResponseHandler(originalSender: ActorRef, propName: String) {

    import context.dispatcher
    import collection.mutable.ArrayBuffer

    val values = ArrayBuffer.empty[String]

    context.actorSelection("/user/request_proxies") ! Request(propName)
    context.system.scheduler.scheduleOnce(50.milliseconds, self, TimedOut)

    val handle = expect {
      case Response(name, value) ⇒
        values += value
        if (values.size > 3) processList()
      case TimedOut ⇒ processList()
    }

    def processList() {
      unexpect(handle)

      if (values.size > 0) {
        context.actorSelection("/user/evaluator") ! values.toList
        expectOnce {
          case EvaluationResults(name, eval) ⇒ processFinal(eval)
        }
      } else processFinal(List.empty[Int])
    }

    def processFinal(eval: List[Int]) {
      // Select only the entries coming back from eval
      originalSender ! FinalResponse(eval map values)
      context.stop(self)
```

```
    }
  }
}
```

Pitfalls

- The current implementation does not match the sender of the message. This is designed to work with `ActorSelection` as well as `ActorRef`. Without the sender(), there is a chance a received message can be matched by more than one partial function. The partial function that was registered via `expect` or `expectOnce` first (chronologically) and is not yet de-registered by `unexpect` takes precedence in this case. Developers should make sure the messages can be uniquely matched or the wrong logic can be executed for a certain message.
- The `sender` referenced in any `expect` or `expectOnce` logic refers to the sender() of that particular message and not the sender() of the original message. The original sender() still needs to be saved so a final response can be sent back.
- `context.become` is not supported when extending the `Aggregator` trait.
- We strongly recommend against overriding `receive`. If your use case really dictates, you may do so with extreme caution. Always provide a pattern match handling aggregator messages among your `receive` pattern matches, as follows:

```
case msg if handleMessage(msg) ⇒ // noop
// side effects of handleMessage does the actual match
```

Sorry, there is not yet a Java implementation of the aggregator pattern available.

Receive Pipeline Pattern

The Receive Pipeline Pattern lets you define general interceptors for your messages and plug an arbitrary amount of them into your Actors. It's useful for defining cross aspects to be applied to all or many of your Actors.

Some Possible Use Cases

- Measure the time spent for processing the messages
- Audit messages with an associated author
- Log all important messages
- Secure restricted messages
- Text internationalization

Interceptors

Multiple interceptors can be added to actors that mixin the `ReceivePipeline` trait. These interceptors internally define layers of decorators around the actor's behavior. The first interceptor defines an outer decorator which delegates to a decorator corresponding to the second interceptor and so on, until the last interceptor which defines a decorator for the actor's `Receive`.

The first or outermost interceptor receives messages sent to the actor.

For each message received by an interceptor, the interceptor will typically perform some processing based on the message and decide whether or not to pass the received message (or some other message) to the next interceptor.

An `Interceptor` is a type alias for `PartialFunction[Any,Delegation]`. The `Any` input is the message it receives from the previous interceptor (or, in the case of the first interceptor, the message that was sent

to the actor). The `Delegation` return type is used to control what (if any) message is passed on to the next interceptor.

A simple example

To pass a transformed message to the actor (or next inner interceptor) an interceptor can return `Inner(newMsg)` where `newMsg` is the transformed message.

The following interceptor shows this. It intercepts `Int` messages, adds one to them and passes on the incremented value to the next interceptor.

```
val incrementInterceptor: Interceptor = {
  case i: Int ⇒ Inner(i + 1)
}
```

Building the Pipeline

To give your Actor the ability to pipeline received messages in this way, you'll need to mixin with the `ReceivePipeline` trait. It has two methods for controlling the pipeline, `pipelineOuter` and `pipelineInner`, both receiving an `Interceptor`. The first one adds the interceptor at the beginning of the pipeline and the second one adds it at the end, just before the current Actor's behavior.

In this example we mixin our Actor with the `ReceivePipeline` trait and we add `Increment` and `Double` interceptors with `pipelineInner`. They will be applied in this very order.

```
class PipelinedActor extends Actor with ReceivePipeline {

  // Increment
  pipelineInner { case i: Int ⇒ Inner(i + 1) }
  // Double
  pipelineInner { case i: Int ⇒ Inner(i * 2) }

  def receive: Receive = { case any ⇒ println(any) }
}

actor ! 5 // prints 12 = (5 + 1) * 2
```

If we add `Double` with `pipelineOuter` it will be applied before `Increment` so the output is 11

```
// Increment
pipelineInner { case i: Int ⇒ Inner(i + 1) }
// Double
pipelineOuter { case i: Int ⇒ Inner(i * 2) }

// prints 11 = (5 * 2) + 1
```

Interceptors Mixin

Defining all the pipeline inside the Actor implementation is good for showing up the pattern, but it isn't very practical. The real flexibility of this pattern comes when you define every interceptor in its own trait and then you mixin any of them into your Actors.

Let's see it in an example. We have the following model:

```
val texts = Map(
  "that.rug_EN" → "That rug really tied the room together.",
  "your.opinion_EN" → "Yeah, well, you know, that's just, like, your opinion, man.
↪",
```

```
  "that.rug_ES" → "Esa alfombra realmente completaba la sala.",
  "your.opinion_ES" → "Sí, bueno, ya sabes, eso es solo, como, tu opinion, amigo.")

case class I18nText(locale: String, key: String)
case class Message(author: Option[String], text: Any)
```

and these two interceptors defined, each one in its own trait:

```
trait I18nInterceptor {
  this: ReceivePipeline ⇒

  pipelineInner {
    case m @ Message(_, I18nText(loc, key)) ⇒
      Inner(m.copy(text = texts(s"${key}_$loc")))
  }
}

trait AuditInterceptor {
  this: ReceivePipeline ⇒

  pipelineOuter {
    case m @ Message(Some(author), text) ⇒
      println(s"$author is about to say: $text")
      Inner(m)
  }
}
```

The first one intercepts any messages having an internationalized text and replaces it with the resolved text before resuming with the chain. The second one intercepts any message with an author defined and prints it before resuming the chain with the message unchanged. But since `I18n` adds the interceptor with `pipelineInner` and `Audit` adds it with `pipelineOuter`, the audit will happen before the internationalization.

So if we mixin both interceptors in our Actor, we will see the following output for these example messages:

```
class PrinterActor extends Actor with ReceivePipeline
  with I18nInterceptor with AuditInterceptor {

  override def receive: Receive = {
    case Message(author, text) ⇒
      println(s"${author.getOrElse("Unknown")} says '$text'")
  }
}

printerActor ! Message(Some("The Dude"), I18nText("EN", "that.rug"))
// The Dude is about to say: I18nText(EN,that.rug)
// The Dude says 'That rug really tied the room together.'

printerActor ! Message(Some("The Dude"), I18nText("EN", "your.opinion"))
// The Dude is about to say: I18nText(EN,your.opinion)
// The Dude says 'Yeah, well, you know, that's just, like, your opinion, man.'
```

Unhandled Messages

With all that behaviors chaining occurring, what happens to unhandled messages? Let me explain it with a simple rule.

Note: Every message not handled by an interceptor will be passed to the next one in the chain. If none of the interceptors handles a message, the current Actor's behavior will receive it, and if the behavior doesn't handle it either, it will be treated as usual with the unhandled method.

But sometimes it is desired for interceptors to break the chain. You can do it by explicitly indicating that the message has been completely handled by the interceptor by returning `HandledCompletely`.

```
case class PrivateMessage(userId: Option[Long], msg: Any)

trait PrivateInterceptor {
  this: ReceivePipeline ⇒

  pipelineInner {
    case PrivateMessage(Some(userId), msg) ⇒
      if (isGranted(userId))
        Inner(msg)
      else
        HandledCompletely
  }
}
```

Processing after delegation

But what if you want to perform some action after the actor has processed the message (for example to measure the message processing time)?

In order to support such use cases, the `Inner` return type has a method `andAfter` which accepts a code block that can perform some action after the message has been processed by subsequent inner interceptors.

The following is a simple interceptor that times message processing:

```
trait TimerInterceptor extends ActorLogging {
  this: ReceivePipeline ⇒

  def logTimeTaken(time: Long) = log.debug(s"Time taken: $time ns")

  pipelineOuter {
    case e ⇒
      val start = System.nanoTime
      Inner(e).andAfter {
        val end = System.nanoTime
        logTimeTaken(end - start)
      }
  }
}
```

Note: The `andAfter` code blocks are run on return from handling the message with the next inner handler and on the same thread. It is therefore safe for the `andAfter` logic to close over the interceptor's state.

Using Receive Pipelines with Persistence

When using `ReceivePipeline` together with *PersistentActor* make sure to mix-in the traits in the following order for them to properly co-operate:

```
class ExampleActor extends PersistentActor with ReceivePipeline {
  /* ... */
}
```

The order is important here because of how both traits use internal "around" methods to implement their features, and if mixed-in the other way around it would not work as expected. If you want to learn more about how exactly this works, you can read up on Scala's type linearization mechanism;

Circuit-Breaker Actor

This is an alternative implementation of the *Akka Circuit Breaker Pattern*. The main difference is that it is intended to be used only for request-reply interactions with an actor using the Circuit-Breaker as a proxy of the target one in order to provide the same failfast functionalities and a protocol similar to the circuit-breaker implementation in Akka.

Usage

Let's assume we have an actor wrapping a back-end service and able to respond to `Request` calls with a `Response` object containing an `Either[String,String]` to map successful and failed responses. The service is also potentially slowing down because of the workload.

A simple implementation can be given by this class

```
object SimpleService {
  case class Request(content: String)
  case class Response(content: Either[String, String])
  case object ResetCount
}

/**
 * This is a simple actor simulating a service
 * - Becoming slower with the increase of frequency of input requests
 * - Failing around 30% of the requests
 */
class SimpleService extends Actor with ActorLogging {
  import SimpleService._

  var messageCount = 0

  import context.dispatcher

  context.system.scheduler.schedule(1.second, 1.second, self, ResetCount)

  override def receive = {
    case ResetCount ⇒
      messageCount = 0

    case Request(content) ⇒
      messageCount += 1
      // simulate workload
      Thread.sleep(100 * messageCount)
      // Fails around 30% of the times
      if (Random.nextInt(100) < 70) {
        sender ! Response(Right(s"Successfully processed $content"))
      } else {
        sender ! Response(Left(s"Failure processing $content"))
      }

  }
}
```

If we want to interface with this service using the Circuit Breaker we can use two approaches:

Using a non-conversational approach:

```
class CircuitBreaker(potentiallyFailingService: ActorRef) extends Actor with␣
↪ActorLogging {
  import SimpleService._

  val serviceCircuitBreaker =
    context.actorOf(
      CircuitBreakerPropsBuilder(maxFailures = 3, callTimeout = 2.seconds,␣
↪resetTimeout = 30.seconds)
```

```
        .copy(
          failureDetector = {
          _ match {
            case Response(Left(_)) ⇒ true
            case _                 ⇒ false
          }
        })
        .props(potentiallyFailingService),
      "serviceCircuitBreaker")

  override def receive: Receive = {
    case AskFor(requestToForward) ⇒
      serviceCircuitBreaker ! Request(requestToForward)

    case Right(Response(content)) ⇒
      //handle response
      log.info("Got successful response {}", content)

    case Response(Right(content)) ⇒
      //handle response
      log.info("Got successful response {}", content)

    case Response(Left(content)) ⇒
      //handle response
      log.info("Got failed response {}", content)

    case CircuitOpenFailure(failedMsg) ⇒
      log.warning("Unable to send message {}", failedMsg)
  }
}
```

Using the `ask` pattern, in this case it is useful to be able to map circuit open failures to the same type of failures returned by the service (a `Left[String]` in our case):

```
class CircuitBreakerAsk(potentiallyFailingService: ActorRef) extends Actor with␣
↪ActorLogging {
  import SimpleService._
  import akka.pattern._

  implicit val askTimeout: Timeout = 2.seconds

  val serviceCircuitBreaker =
    context.actorOf(
      CircuitBreakerPropsBuilder(maxFailures = 3, callTimeout = askTimeout,␣
↪resetTimeout = 30.seconds)
        .copy(
          failureDetector = {
          _ match {
            case Response(Left(_)) ⇒ true
            case _                 ⇒ false
          }
        })
        .copy(
          openCircuitFailureConverter = { failure ⇒
          Left(s"Circuit open when processing ${failure.failedMsg}")
        })
        .props(potentiallyFailingService),
      "serviceCircuitBreaker")

  import context.dispatcher

  override def receive: Receive = {
```

```
    case AskFor(requestToForward) ⇒
      (serviceCircuitBreaker ? Request(requestToForward)).mapTo[Either[String,␣
↪String]].onComplete {
        case Success(Right(successResponse)) ⇒
          //handle response
          log.info("Got successful response {}", successResponse)

        case Success(Left(failureResponse)) ⇒
          //handle response
          log.info("Got successful response {}", failureResponse)

        case Failure(exception) ⇒
          //handle response
          log.info("Got successful response {}", exception)

      }
  }
}
```

If it is not possible to define define a specific error response, you can map the Open Circuit notification to a failure. That also means that your `CircuitBreakerActor` will be useful to protect you from time out for extra workload or temporary failures in the target actor. You can decide to do that in two ways:

The first is to use the `askWithCircuitBreaker` method on the `ActorRef` or `ActorSelection` instance pointing to your circuit breaker proxy (enabled by importing `import akka.contrib.circuitbreaker.Implicits.askWithCircuitBreaker`)

```
class CircuitBreakerAskWithCircuitBreaker(potentiallyFailingService: ActorRef)␣
↪extends Actor with ActorLogging {
  import SimpleService._
  import akka.contrib.circuitbreaker.Implicits.askWithCircuitBreaker

  implicit val askTimeout: Timeout = 2.seconds

  val serviceCircuitBreaker =
    context.actorOf(
      CircuitBreakerPropsBuilder(maxFailures = 3, callTimeout = askTimeout,␣
↪resetTimeout = 30.seconds)
        .props(target = potentiallyFailingService),
      "serviceCircuitBreaker")

  import context.dispatcher

  override def receive: Receive = {
    case AskFor(requestToForward) ⇒
      serviceCircuitBreaker.askWithCircuitBreaker(Request(requestToForward)).
↪mapTo[String].onComplete {
        case Success(successResponse) ⇒
          //handle response
          log.info("Got successful response {}", successResponse)

        case Failure(exception) ⇒
          //handle response
          log.info("Got successful response {}", exception)

      }
  }
}
```

The second is to map the future response of your `ask` pattern application with the `failForOpenCircuit` enabled by importing `import akka.contrib.circuitbreaker.Implicits.futureExtensions`

```
class CircuitBreakerAskWithFailure(potentiallyFailingService: ActorRef) extends␣
↪Actor with ActorLogging {
  import SimpleService._
  import akka.pattern._
  import akka.contrib.circuitbreaker.Implicits.futureExtensions

  implicit val askTimeout: Timeout = 2.seconds

  val serviceCircuitBreaker =
    context.actorOf(
      CircuitBreakerPropsBuilder(maxFailures = 3, callTimeout = askTimeout,␣
↪resetTimeout = 30.seconds)
        .props(target = potentiallyFailingService),
      "serviceCircuitBreaker")

  import context.dispatcher

  override def receive: Receive = {
    case AskFor(requestToForward) ⇒
      (serviceCircuitBreaker ? Request(requestToForward)).failForOpenCircuit.
↪mapTo[String].onComplete {
        case Success(successResponse) ⇒
          //handle response
          log.info("Got successful response {}", successResponse)

        case Failure(exception) ⇒
          //handle response
          log.info("Got successful response {}", exception)

      }
  }
}
```

Direct Communication With The Target Actor

To send messages to the *target* actor without expecting any response you can wrap your message in a `TellOnly` or a `Passthrough` envelope. The difference between the two is that `TellOnly` will forward the message only when in closed mode and `Passthrough` will do it in any state. You can for example use the `Passthrough` envelope to wrap a `PoisonPill` message to terminate the target actor. That will cause the circuit breaker proxy to be terminated too

10.4.3 Suggested Way of Using these Contributions

Since the Akka team does not restrict updates to this subproject even during otherwise binary compatible releases, and modules may be removed without deprecation, it is suggested to copy the source files into your own code base, changing the package name. This way you can choose when to update or which fixes to include (to keep binary compatibility if needed) and later releases of Akka do not potentially break your application.

10.4.4 Suggested Format of Contributions

Each contribution should be a self-contained unit, consisting of one source file or one exclusively used package, without dependencies to other modules in this subproject; it may depend on everything else in the Akka distribution, though. This ensures that contributions may be moved into the standard distribution individually. The module shall be within a subpackage of `akka.contrib`.

Each module must be accompanied by a test suite which verifies that the provided features work, possibly complemented by integration and unit tests. The tests should follow the *Developer Guidelines* and go into the `src/test/scala` or `src/test/java` directories (with package name matching the module which is being tested). As an example, if the module were called `akka.contrib.pattern.ReliableProxy`, then the test suite should be called `akka.contrib.pattern.ReliableProxySpec`.

Each module must also have proper documentation in reStructured Text format. The documentation should be a single `<module>.rst` file in the `akka-contrib/docs` directory, including a link from `index.rst` (this file).

CHAPTER

ELEVEN

INFORMATION FOR AKKA DEVELOPERS

11.1 Building Akka

This page describes how to build and run Akka from the latest source code.

11.1.1 Get the Source Code

Akka uses Git and is hosted at Github.

You first need Git installed on your machine. You can then clone the source repository from http://github.com/akka/akka.

For example:

```
git clone git://github.com/akka/akka.git
```

If you have already cloned the repository previously then you can update the code with `git pull`:

```
git pull origin master
```

11.1.2 sbt

Akka is using the excellent sbt build system. So the first thing you have to do is to download and install sbt. You can read more about how to do that in the sbt setup documentation.

The sbt commands that you'll need to build Akka are all included below. If you want to find out more about sbt and using it for your own projects do read the sbt documentation.

The main Akka sbt build file is `project/AkkaBuild.scala`, with a *build.sbt* in each subproject's directory. It is advisable to allocate at least 2GB of heap size to the JVM that runs sbt, otherwise you may experience some spurious failures when running the tests.

11.1.3 Building Akka

First make sure that you are in the akka code directory:

```
cd akka
```

Building

To compile all the Akka core modules use the `compile` command:

```
sbt compile
```

You can run all tests with the `test` command:

```
sbt test
```

If compiling and testing are successful then you have everything working for the latest Akka development version.

Parallel Execution

By default the tests are executed sequentially. They can be executed in parallel to reduce build times, if hardware can handle the increased memory and cpu usage. Add the following system property to sbt launch script to activate parallel execution:

```
-Dakka.parallelExecution=true
```

Long Running and Time Sensitive Tests

By default are the long running tests (mainly cluster tests) and time sensitive tests (dependent on the performance of the machine it is running on) disabled. You can enable them by adding one of the flags:

```
-Dakka.test.tags.include=long-running
-Dakka.test.tags.include=timing
```

Or if you need to enable them both:

```
-Dakka.test.tags.include=long-running,timing
```

Publish to Local Ivy Repository

If you want to deploy the artifacts to your local Ivy repository (for example, to use from an sbt project) use the `publish-local` command:

```
sbt publish-local
```

sbt Interactive Mode

Note that in the examples above we are calling `sbt compile` and `sbt test` and so on, but sbt also has an interactive mode. If you just run `sbt` you enter the interactive sbt prompt and can enter the commands directly. This saves starting up a new JVM instance for each command and can be much faster and more convenient.

For example, building Akka as above is more commonly done like this:

```
% sbt
[info] Set current project to default (in build file:/.../akka/project/plugins/)
[info] Set current project to akka (in build file:/.../akka/)
> compile
...
> test
...
```

sbt Batch Mode

It's also possible to combine commands in a single call. For example, testing, and publishing Akka to the local Ivy repository can be done with:

```
sbt test publish-local
```

11.1.4 Dependencies

You can look at the Ivy dependency resolution information that is created on `sbt update` and found in `~/.ivy2/cache`. For example, the `~/.ivy2/cache/com.typesafe.akka-akka-remote-compile.xml` file contains the resolution information for the akka-remote module compile dependencies. If you open this file in a web browser you will get an easy to navigate view of dependencies.

11.1.5 Scaladoc Dependencies

Akka generates class diagrams for the API documentation using ScalaDoc. This needs the `dot` command from the Graphviz software package to be installed to avoid errors. You can disable the diagram generation by adding the flag `-Dakka.scaladoc.diagrams=false`. After installing Graphviz, make sure you add the toolset to the PATH (definitely on Windows).

11.2 Multi JVM Testing

Supports running applications (objects with main methods) and ScalaTest tests in multiple JVMs at the same time. Useful for integration testing where multiple systems communicate with each other.

11.2.1 Setup

The multi-JVM testing is an sbt plugin that you can find at http://github.com/typesafehub/sbt-multi-jvm.

You can add it as a plugin by adding the following to your project/plugins.sbt:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-multi-jvm" % "0.3.8")
```

You can then add multi-JVM testing to `build.sbt` or `project/Build.scala` by including the `MultiJvm` settings and config. Please note that MultiJvm test sources are located in `src/multi-jvm/...`, and not in `src/test/...`.

Here is an example `build.sbt` file for sbt 0.13 that uses the MultiJvm plugin:

```
import com.typesafe.sbt.SbtMultiJvm
import com.typesafe.sbt.SbtMultiJvm.MultiJvmKeys.MultiJvm

val akkaVersion = "2.4.14"

val project = Project(
    id = "akka-sample-multi-node-scala",
    base = file(".")
  )
  .settings(SbtMultiJvm.multiJvmSettings: _*)
  .settings(
    name := "akka-sample-multi-node-scala",
    version := "2.4.14",
    scalaVersion := "2.11.7",
    libraryDependencies ++= Seq(
      "com.typesafe.akka" %% "akka-actor" % akkaVersion,
      "com.typesafe.akka" %% "akka-remote" % akkaVersion,
      "com.typesafe.akka" %% "akka-multi-node-testkit" % akkaVersion,
      "org.scalatest" %% "scalatest" % "2.2.1" % "test"),
    // make sure that MultiJvm test are compiled by the default test compilation
```

```
    compile in MultiJvm <<= (compile in MultiJvm) triggeredBy (compile in Test),
    // disable parallel tests
    parallelExecution in Test := false,
    // make sure that MultiJvm tests are executed by the default test target,
    // and combine the results from ordinary test and multi-jvm tests
    executeTests in Test <<= (executeTests in Test, executeTests in MultiJvm) map {
      case (testResults, multiNodeResults)  =>
        val overall =
          if (testResults.overall.id < multiNodeResults.overall.id)
            multiNodeResults.overall
          else
            testResults.overall
        Tests.Output(overall,
          testResults.events ++ multiNodeResults.events,
          testResults.summaries ++ multiNodeResults.summaries)
    },
    licenses := Seq(("CC0", url("http://creativecommons.org/publicdomain/zero/1.0
↪")))
  )
  .configs (MultiJvm)
```

You can specify JVM options for the forked JVMs:

```
jvmOptions in MultiJvm := Seq("-Xmx256M")
```

11.2.2 Running tests

The multi-JVM tasks are similar to the normal tasks: `test`, `test-only`, and `run`, but are under the `multi-jvm` configuration.

So in Akka, to run all the multi-JVM tests in the akka-remote project use (at the sbt prompt):

```
akka-remote-tests/multi-jvm:test
```

Or one can change to the `akka-remote-tests` project first, and then run the tests:

```
project akka-remote-tests
multi-jvm:test
```

To run individual tests use `test-only`:

```
multi-jvm:test-only akka.remote.RandomRoutedRemoteActor
```

More than one test name can be listed to run multiple specific tests. Tab-completion in sbt makes it easy to complete the test names.

It's also possible to specify JVM options with `test-only` by including those options after the test names and `--`. For example:

```
multi-jvm:test-only akka.remote.RandomRoutedRemoteActor -- -Dsome.option=something
```

11.2.3 Creating application tests

The tests are discovered, and combined, through a naming convention. MultiJvm test sources are located in `src/multi-jvm/...`. A test is named with the following pattern:

```
{TestName}MultiJvm{NodeName}
```

That is, each test has `MultiJvm` in the middle of its name. The part before it groups together tests/applications under a single `TestName` that will run together. The part after, the `NodeName`, is a distinguishing name for each forked JVM.

So to create a 3-node test called `Sample`, you can create three applications like the following:

```
package sample

object SampleMultiJvmNode1 {
  def main(args: Array[String]) {
    println("Hello from node 1")
  }
}

object SampleMultiJvmNode2 {
  def main(args: Array[String]) {
    println("Hello from node 2")
  }
}

object SampleMultiJvmNode3 {
  def main(args: Array[String]) {
    println("Hello from node 3")
  }
}
```

When you call `multi-jvm:run sample.Sample` at the sbt prompt, three JVMs will be spawned, one for each node. It will look like this:

```
> multi-jvm:run sample.Sample
...
[info] * sample.Sample
[JVM-1] Hello from node 1
[JVM-2] Hello from node 2
[JVM-3] Hello from node 3
[success] Total time: ...
```

11.2.4 Changing Defaults

You can change the name of the multi-JVM test source directory by adding the following configuration to your project:

```
unmanagedSourceDirectories in MultiJvm <<=
   Seq(baseDirectory(_ / "src/some_directory_here")).join
```

You can change what the `MultiJvm` identifier is. For example, to change it to `ClusterTest` use the `multiJvmMarker` setting:

```
multiJvmMarker in MultiJvm := "ClusterTest"
```

Your tests should now be named `{TestName}ClusterTest{NodeName}`.

11.2.5 Configuration of the JVM instances

You can define specific JVM options for each of the spawned JVMs. You do that by creating a file named after the node in the test with suffix `.opts` and put them in the same directory as the test.

For example, to feed the JVM options `-Dakka.remote.port=9991` and `-Xmx256m` to the `SampleMultiJvmNode1` let's create three `*.opts` files and add the options to them. Separate multiple options with space.

SampleMultiJvmNode1.opts:

```
-Dakka.remote.port=9991 -Xmx256m
```

SampleMultiJvmNode2.opts:

```
-Dakka.remote.port=9992 -Xmx256m
```

SampleMultiJvmNode3.opts:

```
-Dakka.remote.port=9993 -Xmx256m
```

11.2.6 ScalaTest

There is also support for creating ScalaTest tests rather than applications. To do this use the same naming convention as above, but create ScalaTest suites rather than objects with main methods. You need to have ScalaTest on the classpath. Here is a similar example to the one above but using ScalaTest:

```
package sample

import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers

class SpecMultiJvmNode1 extends WordSpec with MustMatchers {
  "A node" should {
    "be able to say hello" in {
      val message = "Hello from node 1"
      message must be("Hello from node 1")
    }
  }
}

class SpecMultiJvmNode2 extends WordSpec with MustMatchers {
  "A node" should {
    "be able to say hello" in {
      val message = "Hello from node 2"
      message must be("Hello from node 2")
    }
  }
}
```

To run just these tests you would call `multi-jvm:test-only sample.Spec` at the sbt prompt.

11.2.7 Multi Node Additions

There has also been some additions made to the `SbtMultiJvm` plugin to accommodate the *experimental* module *multi node testing*, described in that section.

11.3 I/O Layer Design

The `akka.io` package has been developed in collaboration between the Akka and spray.io teams. Its design incorporates the experiences with the `spray-io` module along with improvements that were jointly developed for more general consumption as an actor-based service.

11.3.1 Requirements

In order to form a general and extensible IO layer basis for a wide range of applications, with Akka remoting and spray HTTP being the initial ones, the following requirements were established as key drivers for the design:

- scalability to millions of concurrent connections
- lowest possible latency in getting data from an input channel into the target actor's mailbox
- maximal throughput
- optional back-pressure in both directions (i.e. throttling local senders as well as allowing local readers to throttle remote senders, where allowed by the protocol)
- a purely actor-based API with immutable data representation
- extensibility for integrating new transports by way of a very lean SPI; the goal is to not force I/O mechanisms into a lowest common denominator but instead allow completely protocol-specific user-level APIs.

11.3.2 Basic Architecture

Each transport implementation will be made available as a separate Akka extension, offering an `ActorRef` representing the initial point of contact for client code. This "manager" accepts requests for establishing a communications channel (e.g. connect or listen on a TCP socket). Each communications channel is represented by one dedicated actor, which is exposed to client code for all interaction with this channel over its entire lifetime.

The central element of the implementation is the transport-specific "selector" actor; in the case of TCP this would wrap a `java.nio.channels.Selector`. The channel actors register their interest in readability or writability of their channel by sending corresponding messages to their assigned selector actor. However, the actual channel reading and writing is performed by the channel actors themselves, which frees the selector actors from time-consuming tasks and thereby ensures low latency. The selector actor's only responsibility is the management of the underlying selector's key set and the actual select operation, which is the only operation to typically block.

The assignment of channels to selectors is performed by the manager actor and remains unchanged for the entire lifetime of a channel. Thereby the management actor "stripes" new channels across one or more selector actors based on some implementation-specific distribution logic. This logic may be delegated (in part) to the selectors actors, which could, for example, choose to reject the assignment of a new channel when they consider themselves to be at capacity.

The manager actor creates (and therefore supervises) the selector actors, which in turn create and supervise their channel actors. The actor hierarchy of one single transport implementation therefore consists of three distinct actor levels, with the management actor at the top-, the channel actors at the leaf- and the selector actors at the mid-level.

Back-pressure for output is enabled by allowing the user to specify within its `Write` messages whether it wants to receive an acknowledgement for enqueuing that write to the O/S kernel. Back-pressure for input is enabled by sending the channel actor a message which temporarily disables read interest for the channel until reading is re-enabled with a corresponding resume command. In the case of transports with flow control—like TCP—the act of not consuming data at the receiving end (thereby causing them to remain in the kernels read buffers) is propagated back to the sender, linking these two mechanisms across the network.

11.3.3 Design Benefits

Staying within the actor model for the whole implementation allows us to remove the need for explicit thread handling logic, and it also means that there are no locks involved (besides those which are part of the underlying transport library). Writing only actor code results in a cleaner implementation, while Akka's efficient actor messaging does not impose a high tax for this benefit. In fact the event-based nature of I/O maps so well to the actor model that we expect clear performance and especially scalability benefits over traditional solutions with explicit thread management and synchronization.

Another benefit of supervision hierarchies is that clean-up of resources comes naturally: shutting down a selector actor will automatically clean up all channel actors, allowing proper closing of the channels and sending the

appropriate messages to user-level client actors. DeathWatch allows the channel actors to notice the demise of their user-level handler actors and terminate in an orderly fashion in that case as well; this naturally reduces the chances of leaking open channels.

The choice of using `ActorRef` for exposing all functionality entails that these references can be distributed or delegated freely and in general handled as the user sees fit, including the use of remoting and life-cycle monitoring (just to name two).

11.3.4 How to go about Adding a New Transport

The best start is to study the TCP reference implementation to get a good grip on the basic working principle and then design an implementation, which is similar in spirit, but adapted to the new protocol in question. There are vast differences between I/O mechanisms (e.g. compare file I/O to a message broker) and the goal of this I/O layer is explicitly **not** to shoehorn all of them into a uniform API, which is why only the basic architecture ideas are documented here.

11.4 Developer Guidelines

Note: First read The Akka Contributor Guidelines.

11.4.1 Code Style

The Akka code style follows the Scala Style Guide . The only exception is the style of block comments:

```
/**
  * Style mandated by "Scala Style Guide"
  */

/**
 * Style adopted in the Akka codebase
 */
```

Akka is using `Scalariform` to format the source code as part of the build. So just hack away and then run `sbt compile` and it will reformat the code according to Akka standards.

11.4.2 Process

The full process is described in The Akka Contributor Guidelines. In summary:

- Make sure you have signed the Akka CLA, if not, sign it online.
- Pick a ticket, if there is no ticket for your work then create one first.
- Fork akka/akka. Start working in a feature branch.
- When you are done, create a GitHub Pull-Request towards the targeted branch.
- When there's consensus on the review, someone from the Akka Core Team will merge it.

11.4.3 Commit messages

Please follow the conventions described in The Akka Contributor Guidelines when creating public commits and writing commit messages.

11.4.4 Testing

All code that is checked in **should** have tests. All testing is done with `ScalaTest` and `ScalaCheck`.

- Name tests as **Test.scala** if they do not depend on any external stuff. That keeps surefire happy.
- Name tests as **Spec.scala** if they have external dependencies.

Actor TestKit

There is a useful test kit for testing actors: akka.util.TestKit. It enables assertions concerning replies received and their timing, there is more documentation in the *Testing Actor Systems* module.

Multi-JVM Testing

Included in the example is an sbt trait for multi-JVM testing which will fork JVMs for multi-node testing. There is support for running applications (objects with main methods) and running ScalaTest tests.

NetworkFailureTest

You can use the 'NetworkFailureTest' trait to test network failure.

11.5 Documentation Guidelines

The Akka documentation uses reStructuredText as its markup language and is built using Sphinx.

11.5.1 Sphinx

For more details see The Sphinx Documentation

11.5.2 reStructuredText

For more details see The reST Quickref

Sections

Section headings are very flexible in reST. We use the following convention in the Akka documentation:

- # (over and under) for module headings
- = for sections
- − for subsections
- ^ for subsubsections
- ~ for subsubsubsections

Cross-referencing

Sections that may be cross-referenced across the documentation should be marked with a reference. To mark a section use `.. _ref-name:` before the section heading. The section can then be linked with `:ref:`ref-name``. These are unique references across the entire documentation.

For example:

```
.. _akka-module:

#############
 Akka Module
#############

This is the module documentation.

.. _akka-section:

Akka Section
============

Akka Subsection
---------------

Here is a reference to "akka section": :ref:`akka-section` which will have the
name "Akka Section".
```

11.5.3 Build the documentation

First install Sphinx. See below.

Building

For the html version of the docs:

```
sbt sphinx:generateHtml

open <project-dir>/akka-docs/target/sphinx/html/index.html
```

For the pdf version of the docs:

```
sbt sphinx:generatePdf

open <project-dir>/akka-docs/target/sphinx/latex/AkkaJava.pdf
or
open <project-dir>/akka-docs/target/sphinx/latex/AkkaScala.pdf
```

Installing Sphinx on OS X

Install Homebrew

Install Python with Homebrew:

```
brew install python
```

Homebrew will automatically add Python executable to your $PATH and pip is a part of the default Python installation with Homebrew.

More information in case of trouble: https://github.com/mxcl/homebrew/wiki/Homebrew-and-Python

Install sphinx:

```
pip install sphinx
```

Install BasicTeX package from: http://www.tug.org/mactex/morepackages.html

Add texlive bin to $PATH:

```
export TEXLIVE_PATH=/usr/local/texlive/2016basic/bin/universal-darwin
export PATH=$TEXLIVE_PATH:$PATH
```

Add missing tex packages:

```
sudo tlmgr update --self
sudo tlmgr install titlesec
sudo tlmgr install framed
sudo tlmgr install threeparttable
sudo tlmgr install wrapfig
sudo tlmgr install helvetic
sudo tlmgr install courier
sudo tlmgr install multirow
sudo tlmgr install capt-of
sudo tlmgr install needspace
sudo tlmgr install eqparbox
sudo tlmgr install environ
sudo tlmgr install trimspaces
```

If you get the error "unknown locale: UTF-8" when generating the documentation the solution is to define the following environment variables:

```
export LANG=en_US.UTF-8
export LC_ALL=en_US.UTF-8
```

CHAPTER

TWELVE

PROJECT INFORMATION

12.1 Migration Guides

12.1.1 Migration Guide 1.3.x to 2.0.x

Migration from 1.3.x to 2.0.x is described in the documentation of 2.0.

12.1.2 Migration Guide 2.0.x to 2.1.x

Migration from 2.0.x to 2.1.x is described in the documentation of 2.1.

12.1.3 Migration Guide 2.1.x to 2.2.x

Migration from 2.1.x to 2.2.x is described in the documentation of 2.2.

12.1.4 Migration Guide 2.2.x to 2.3.x

Migration from 2.2.x to 2.3.x is described in the documentation of 2.3.

12.1.5 Migration Guide Akka Persistence (experimental) 2.3.3 to 2.3.4 (and 2.4.x)

Akka Persistence is an **experimental module**, which means that neither Binary Compatibility nor API stability is provided for Persistence while under the *experimental* flag. The goal of this phase is to gather user feedback before we freeze the APIs in a major release.

defer renamed to deferAsync

The `defer` method in `PersistentActor` was renamed to `deferAsync` as it matches the semantics of `persistAsync` more closely than `persist`, which was causing confusion for users.

Its semantics remain unchanged.

Renamed EventsourcedProcessor to PersistentActor

`EventsourcedProcessor` is now deprecated and replaced by `PersistentActor` which provides the same (and more) API. Migrating to `2.4.x` is as simple as changing all your classes to extending `PersistentActor`.

Replace all classes like:

```
class DeprecatedProcessor extends EventsourcedProcessor {
  def processorId = "id"
  /*...*/
}
```

To extend `PersistentActor`:

```
class NewPersistentProcessor extends PersistentActor {
  def persistenceId = "id"
  /*...*/
}
```

Read more about the persistent actor in the *documentation for Scala* and documentation for Java.

Changed processorId to (abstract) persistenceId

In Akka Persistence `2.3.3` and previously, the main building block of applications were Processors. Persistent messages, as well as processors implemented the `processorId` method to identify which persistent entity a message belonged to.

This concept remains the same in Akka `2.3.4`, yet we rename `processorId` to `persistenceId` because Processors will be removed, and persistent messages can be used from different classes not only `PersistentActor` (Views, directly from Journals etc).

Please note that `persistenceId` is **abstract** in the new API classes (`PersistentActor` and `PersistentView`), and we do **not** provide a default (actor-path derived) value for it like we did for `processorId`. The rationale behind this change being stricter de-coupling of your Actor hierarchy and the logical "which persistent entity this actor represents". A longer discussion on this subject can be found on issue #15436 on github.

In case you want to preserve the old behavior of providing the actor's path as the default `persistenceId`, you can easily implement it yourself either as a helper trait or simply by overriding `persistenceId` as follows:

```
override def persistenceId = self.path.toStringWithoutAddress
```

We provided the renamed method also on already deprecated classes (Channels), so you can simply apply a global rename of `processorId` to `persistenceId`.

Removed Processor in favour of extending PersistentActor with persistAsync

The `Processor` is now deprecated since `2.3.4` and will be removed in `2.4.x`. It's semantics replicated in `PersistentActor` in the form of an additional `persist` method: `persistAsync`.

In essence, the difference between `persist` and `persistAsync` is that the former will stash all incoming commands until all persist callbacks have been processed, whereas the latter does not stash any commands. The new `persistAsync` should be used in cases of low consistency yet high responsiveness requirements, the Actor can keep processing incoming commands, even though not all previous events have been handled.

When these `persist` and `persistAsync` are used together in the same `PersistentActor`, the `persist` logic will win over the async version so that all guarantees concerning persist still hold. This will however lower the throughput

Now deprecated code using Processor:

```
class OldProcessor extends Processor {
  override def processorId = "user-wallet-1337"

  def receive = {
    case Persistent(cmd) => sender() ! cmd
  }
}
```

Replacement code, with the same semantics, using PersistentActor:

```
class NewProcessor extends PersistentActor {
  override def persistenceId = "user-wallet-1337"

  def receiveCommand = {
    case cmd =>
      persistAsync(cmd) { e => sender() ! e }
  }

  def receiveRecover = {
    case _ => // logic for handling replay
  }
}
```

It is worth pointing out that using `sender()` inside the persistAsync callback block is **valid**, and does *not* suffer any of the problems Futures have when closing over the sender reference.

Using the `PersistentActor` instead of `Processor` also shifts the responsibility of deciding if a message should be persisted to the receiver instead of the sender of the message. Previously, using `Processor`, clients would have to wrap messages as `Persistent(cmd)` manually, as well as have to be aware of the receiver being a `Processor`, which didn't play well with transparency of the ActorRefs in general.

How to migrate data from Processor to PersistentActor

The recommended approach for migrating persisted messages from a `Processor` to events that can be replayed by a `PersistentActor` is to write a custom migration tool with a `PersistentView` and a `PersistentActor`. Connect the `PersistentView` to the `persistenceId` of the old `Processor` to replay the stored persistent messages. Send the messages from the view to a `PersistentActor` with another `persistenceId`. There you can transform the old messages to domain events that the real `PersistentActor` will be able to understand. Store the events with `persistAsync`.

Note that you can implement back-pressure between the writing `PersistentActor` and the reading `PersistentView` by turning off auto-update in the view and send custom `Update` messages to the view with a limited *replayMax* value.

Removed deleteMessage

`deleteMessage` is deprecated and will be removed. When using command sourced `Processor` the command was stored before it was received and could be validated and then there was a reason to remove faulty commands to avoid repeating the error during replay. When using `PersistentActor` you can always validate the command before persisting and therefore the stored event (or command) should always be valid for replay.

`deleteMessages` can still be used for pruning of all messages up to a sequence number.

Renamed View to PersistentView, which receives plain messages (Persistent() wrapper is gone)

Views used to receive messages wrapped as `Persistent(payload,seqNr)`, this is no longer the case and views receive the `payload` as message from the `Journal` directly. The rationale here is that the wrapper approach was inconsistent with the other Akka Persistence APIs and also is not easily "discoverable" (you have to *know* you will be getting this Persistent wrapper).

Instead, since `2.3.4`, views get plain messages, and can use additional methods provided by the `View` to identify if a message was sent from the Journal (had been played back to the view). So if you had code like this:

```
class AverageView extends View {
  override def processorId = "average-view"

  def receive = {
```

```
    case Persistent(msg, seqNr) =>
      // from Journal

    case msg =>
      // from user-land
}
```

You should update it to extend `PersistentView` instead:

```
class AverageView extends PersistentView {
  override def persistenceId = "persistence-sample"
  override def viewId = "persistence-sample-average"

  def receive = {
    case msg if isPersistent =>
      // from Journal
      val seqNr = lastSequenceNr // in case you require the sequence number

    case msg =>
      // from user-land
  }
}
```

In case you need to obtain the current sequence number the view is looking at, you can use the `lastSequenceNr` method. It is equivalent to "current sequence number", when `isPersistent` returns true, otherwise it yields the sequence number of the last persistent message that this view was updated with.

Removed Channel and PersistentChannel in favour of AtLeastOnceDelivery trait

One of the primary tasks of a `Channel` was to de-duplicate messages that were sent from a `Processor` during recovery. Performing external side effects during recovery is not encouraged with event sourcing and therefore the `Channel` is not needed for this purpose.

The `Channel` and `PersistentChannel` also performed at-least-once delivery of messages, but it did not free a sending actor from implementing retransmission or timeouts, since the acknowledgement from the channel is needed to guarantee safe hand-off. Therefore at-least-once delivery is provided in a new `AtLeastOnceDelivery` trait that is mixed-in to the persistent actor on the sending side.

Read more about at-least-once delivery in the *documentation for Scala* and documentation for Java.

Default persistence plugins

Previously default `akka.persistence.journal.plugin` was set to the LevelDB journal `akka.persistence.journal.leveldb` and default `akka.persistence.snapshot-store.plugin` was set to the local file-system snapshot `akka.persistence.snapshot-store.local`. Now default journal and default snapshot-store plugins are set to empty "" in the persistence extension `reference.conf`, and require explicit user configuration via override in the user `application.conf`. This change was needed to decouple persistence extension from the LevelDB dependency, and to support multiple plugin configurations. Please see persistence extension `reference.conf` for details.

Converted LevelDB to an optional dependency

Persistence extension uses LevelDB based plugins for own development and keeps related code in the published jar. However previously LevelDB was a `compile` scope dependency, and now it is an `optional;provided` dependency. To continue using LevelDB based persistence plugins it is now required for related user projects to include an additional explicit dependency declaration for the LevelDB artifacts. This change allows production Akka

deployments to avoid need for the LevelDB provisioning. Please see persistence extension `reference.conf` for details.

12.1.6 Migration Guide Eventsourced to Akka Persistence 2.3.x

General notes

Eventsourced and Akka *Persistence* share many high-level concepts but strongly differ on design, implementation and usage level. This migration guide is more a higher-level comparison of Eventsourced and Akka Persistence rather than a sequence of low-level instructions how to transform Eventsourced application code into Akka Persistence application code. This should provide a good starting point for a migration effort. Development teams should consult the user documentation of both projects for further details.

Scope of this migration guide is code migration, not journal migration. Journals written by Eventsourced can neither be used directly Akka Persistence nor migrated to Akka Persistence compatible journals. Journal migration tools may be provided in the future but do not exist at the moment.

Extensions

Eventsourced and Akka Persistence are both *Akka Extensions*.

Eventsourced: `EventsourcingExtension`

- Must be explicitly created with an actor system and an application-defined journal actor as arguments. (see example usage).
- Processors and Channels must be created with the factory methods `processorOf` and `channelOf` defined on `EventsourcingExtension`.
- Is a central registry of created processors and channels.

Akka Persistence: `Persistence` extension

- Must **not** be explicitly created by an application. A `Persistence` extension is implicitly created upon first *PersistentActor'* creation. Journal actors are automatically created from a journal plugin configuration (see *Journal plugin API*).
- `PersistentActor` can be created like any other actor with `actorOf` without using the `Persistence` extension.
- Is **not** a central registry of persistent actors.

Processors / PersistentActor

Eventsourced: `Eventsourced`

- Stackable trait that adds journaling (write-ahead-logging) to actors (see processor definition and creation). Name `Eventsourced` caused some confusion in the past as many examples used `Eventsourced` processors for *command sourcing*. See also this FAQ entry for clarification.
- Must be explicitly recovered using recovery methods on `EventsourcingExtension`. Applications are responsible for avoiding an interference of replayed messages and new messages i.e. applications have to explicitly wait for recovery to complete. Recovery on processor re-start is not supported out-of-the box.
- Journaling-preserving behavior changes are only possible with special-purpose methods `become` and `unbecome`, in addition to non-journaling-preserving behavior changes with default methods `context.become` and `context.unbecome`.
- Writes messages of type `Message` to the journal (see processor usage). Sender references are not stored in the journal i.e. the sender reference of a replayed message is always `system.deadLetters`.
- Supports snapshots.

- Identifiers are of type `Int` and must be application-defined.
- Does not support batch-writes of messages to the journal.
- Does not support stashing of messages.

Akka Persistence: `PersistentActor`

- Trait that adds journaling to actors (see *Event sourcing*) and used by applications for *event sourcing* or *command sourcing*. Corresponds to `Eventsourced` processors in Eventsourced but is not a stackable trait.
- Automatically recovers on start and re-start, by default. *Recovery* can be customized or turned off by overriding actor life cycle hooks `preStart` and `preRestart`. `Processor` takes care that new messages never interfere with replayed messages. New messages are internally buffered until recovery completes.
- No special-purpose behavior change methods. Default behavior change methods `context.become` and `context.unbecome` can be used and are journaling-preserving.
- Sender references are written to the journal. Sender references of type `PromiseActorRef` are not journaled, they are `system.deadLetters` on replay.
- Supports *Snapshots*.
- *Identifiers* are of type `String`, have a default value and can be overridden by applications.
- Supports *Batch writes*.
- Supports stashing of messages.

Channels

Eventsourced: `DefaultChannel`

- Prevents redundant delivery of messages to a destination (see usage example and default channel).
- Is associated with a single destination actor reference. A new incarnation of the destination is not automatically resolved by the channel. In this case a new channel must be created.
- Must be explicitly activated using methods `deliver` or `recover` defined on `EventsourcingExtension`.
- Must be activated **after** all processors have been activated. Depending on the recovery method, this is either done automatically or must be followed by the application (see non-blocking recovery). This is necessary for a network of processors and channels to recover consistently.
- Does not redeliver messages on missing or negative delivery confirmation.
- Cannot be used standalone.

Eventsourced: `ReliableChannel`

- Provides `DefaultChannel` functionality plus persistence and recovery from sender JVM crashes (see ReliableChannel). Also provides message redelivery in case of missing or negative delivery confirmations.
- Delivers next message to a destination only if previous message has been successfully delivered (flow control is done by destination).
- Stops itself when the maximum number of redelivery attempts has been reached.
- Cannot reply on persistence.
- Can be used standalone.

Akka Persistence: `AtLeastOnceDelivery`

- `AtLeastOnceDelivery` trait is mixed in to a `PersistentActor`
- Does not prevent redundant delivery of messages to a destination entirely, but won't re-send messages whose delivery is confirmed during recovery.

- Is not associated with a single destination. A destination can be specified with each `deliver` request and is referred to by an actor path. A destination path is resolved to the current destination incarnation during delivery (via `actorSelection`).
- Redelivers messages on missing delivery confirmation. In contrast to Eventsourced, Akka Persistence doesn't distinguish between missing and negative confirmations. It only has a notion of missing confirmations using timeouts (which are closely related to negative confirmations as both trigger message redelivery).

Views

Eventsourced:

- No direct support for views. Only way to maintain a view is to use a channel and a processor as destination.

Akka Persistence: `View`

- Receives the message stream written by a `PersistentActor` by reading it directly from the journal (see *Persistent Views*). Alternative to using channels. Useful in situations where actors shall receive a persistent message stream in correct order without duplicates.
- Supports *Snapshots*.

Serializers

Eventsourced:

- Uses a protobuf serializer for serializing `Message` objects.
- Delegates to a configurable Akka serializer for serializing `Message` payloads.
- Delegates to a configurable, proprietary (stream) serializer for serializing snapshots.
- See Serialization.

Akka Persistence:

- Uses a protobuf serializer for serializing `Persistent` objects.
- Delegates to a configurable Akka serializer for serializing `Persistent` payloads.
- Delegates to a configurable Akka serializer for serializing snapshots.
- See *Custom serialization*.

Sequence numbers

Eventsourced:

- Generated on a per-journal basis.

Akka Persistence:

- Generated on a per persistent actor basis.

Storage plugins

Eventsourced:

- Plugin API: SynchronousWriteReplaySupport and AsynchronousWriteReplaySupport
- No separation between journal and snapshot storage plugins.
- All plugins pre-packaged with project (see journals and snapshot configuration)

Akka Persistence:

- Plugin API: `SyncWriteJournal`, `AsyncWriteJournal`, `AsyncRecovery`, `SnapshotStore` (see *Storage plugins*).
- Clear separation between journal and snapshot storage plugins.
- Limited number of *Pre-packaged plugins* (LevelDB journal and local snapshot store).
- Impressive list of community plugins.

12.1.7 Migration Guide 2.3.x to 2.4.x

The 2.4 release contains some structural changes that require some simple, mechanical source-level changes in client code.

When migrating from earlier versions you should first follow the instructions for migrating *1.3.x to 2.0.x* and then *2.0.x to 2.1.x* and then *2.1.x to 2.2.x* and then *2.2.x to 2.3.x*.

Binary Compatibility

Akka 2.4.x is backwards binary compatible with previous 2.3.x versions apart from the following exceptions. This means that the new JARs are a drop-in replacement for the old one (but not the other way around) as long as your build does not enable the inliner (Scala-only restriction).

The following parts are not binary compatible with 2.3.x:

- akka-testkit and akka-remote-testkit
- experimental modules, such as akka-persistence and akka-contrib
- features, classes, methods that were deprecated in 2.3.x and removed in 2.4.x

The dependency to **Netty** has been updated from version 3.8.0.Final to 3.10.3.Final. The changes in those versions might not be fully binary compatible, but we believe that it will not be a problem in practice. No changes were needed to the Akka source code for this update. Users of libraries that depend on 3.8.0.Final that break with 3.10.3.Final should be able to manually downgrade the dependency to 3.8.0.Final and Akka will still work with that version.

Advanced Notice: TypedActors will go away

While technically not yet deprecated, the current `akka.actor.TypedActor` support will be superseded by the *Akka Typed* project that is currently being developed in open preview mode. If you are using TypedActors in your projects you are advised to look into this, as it is superior to the Active Object pattern expressed in TypedActors. The generic ActorRefs in Akka Typed allow the same type-safety that is afforded by TypedActors while retaining all the other benefits of an explicit actor model (including the ability to change behaviors etc.).

It is likely that TypedActors will be officially deprecated in the next major update of Akka and subsequently removed.

Removed Deprecated Features

The following, previously deprecated, features have been removed:

- akka-dataflow
- akka-transactor
- durable mailboxes (akka-mailboxes-common, akka-file-mailbox)
- Cluster.publishCurrentClusterState
- akka.cluster.auto-down, replaced by akka.cluster.auto-down-unreachable-after in Akka 2.3

- Old routers and configuration.

  Note that in router configuration you must now specify if it is a `pool` or a `group` in the way that was introduced in Akka 2.3.

- Timeout constructor without unit
- JavaLoggingEventHandler, replaced by JavaLogger
- UntypedActorFactory
- Java API TestKit.dilated, moved to JavaTestKit.dilated

Protobuf Dependency

The transitive dependency to Protobuf has been removed to make it possible to use any version of Protobuf for the application messages. If you use Protobuf in your application you need to add the following dependency with desired version number:

```
"com.google.protobuf" % "protobuf-java" % "2.5.0"
```

Internally Akka is using an embedded version of protobuf that corresponds to `com.google.protobuf/protobuf-java` version 2.5.0. The package name of the embedded classes has been changed to `akka.protobuf`.

Added parameter validation to RootActorPath

Previously `akka.actor.RootActorPath` allowed passing in arbitrary strings into its name parameter, which is meant to be the *name* of the root Actor. Subsequently, if constructed with an invalid name such as a full path for example (`/user/Full/Path`) some features using this path may transparently fail - such as using `actorSelection` on such invalid path.

In Akka 2.4.x the `RootActorPath` validates the input and may throw an `IllegalArgumentException` if the passed in name string is illegal (contains `/` elsewhere than in the begining of the string or contains `#`).

TestKit.remaining throws AssertionError

In earlier versions of Akka *TestKit.remaining* returned the default timeout configurable under "akka.test.single-expect-default". This was a bit confusing and thus it has been changed to throw an AssertionError if called outside of within. The old behavior however can still be achieved by calling *TestKit.remainingOrDefault* instead.

EventStream and ManagedActorClassification EventBus now require an ActorSystem

Both the `EventStream` (*Scala*, Java) and the `ManagedActorClassification`, `ManagedActorEventBus` (*Scala*, Java) now require an `ActorSystem` to properly operate. The reason for that is moving away from stateful internal lifecycle checks to a fully reactive model for unsubscribing actors that have `Terminated`. Therefore the `ActorClassification` and `ActorEventBus` was deprecated and replaced by `ManagedActorClassification` and `ManagedActorEventBus`

If you have implemented a custom event bus, you will need to pass in the actor system through the constructor now:

```
import akka.event.ActorEventBus
import akka.event.ManagedActorClassification
import akka.event.ActorClassifier

final case class Notification(ref: ActorRef, id: Int)

class ActorBusImpl(val system: ActorSystem) extends ActorEventBus with␣
↪ActorClassifier with ManagedActorClassification {
```

```
  type Event = Notification

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): ActorRef = event.ref

  // determines the initial size of the index data structure
  // used internally (i.e. the expected number of different classifiers)
  override protected def mapSize: Int = 128
}
```

If you have been creating EventStreams manually, you now have to provide an actor system and *start the unsubscriber*:

```
val bus = new EventStream(system, true)
bus.startUnsubscriber()
```

Please note that this change affects you only if you have implemented your own buses, Akka's own `context.eventStream` is still there and does not require any attention from you concerning this change.

FSM notifies on same state transitions

When changing states in an Finite-State-Machine Actor (`FSM`), state transition events are emitted and can be handled by the user either by registering `onTransition` handlers or by subscribing to these events by sending it an `SubscribeTransitionCallBack` message.

Previously in `2.3.x` when an `FSM` was in state `A` and performed a `goto(A)` transition, no state transition notification would be sent. This is because it would effectively stay in the same state, and was deemed to be semantically equivalent to calling `stay()`.

In `2.4.x` when an `FSM` performs an any `goto(X)` transition, it will always trigger state transition events. Which turns out to be useful in many systems where same-state transitions actually should have an effect.

In case you do *not* want to trigger a state transition event when effectively performing an `X->X` transition, use `stay()` instead.

Circuit Breaker Timeout Change

In `2.3.x` calls protected by the `CircuitBreaker` were allowed to run indefinitely and the check to see if the timeout had been exceeded was done after the call had returned.

In `2.4.x` the failureCount of the Breaker will be increased as soon as the timeout is reached and a `Failure[TimeoutException]` will be returned immediately for asynchronous calls. Synchronous calls will now throw a `TimeoutException` after the call is finished.

Slf4j logging filter

If you use `Slf4jLogger` you should add the following configuration:

```
akka.logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"
```

It will filter the log events using the backend configuration (e.g. logback.xml) before they are published to the event bus.

Inbox.receive Java API

`Inbox.receive` now throws a checked `java.util.concurrent.TimeoutException` exception if the receive timeout is reached.

Pool routers nrOfInstances method now takes ActorSystem

In order to make cluster routers smarter about when they can start local routees, `nrOfInstances` defined on `Pool` now takes `ActorSystem` as an argument. In case you have implemented a custom Pool you will have to update the method's signature, however the implementation can remain the same if you don't need to rely on an ActorSystem in your logic.

Group routers paths method now takes ActorSystem

In order to make cluster routers smarter about when they can start local routees, `paths` defined on `Group` now takes `ActorSystem` as an argument. In case you have implemented a custom Group you will have to update the method's signature, however the implementation can remain the same if you don't need to rely on an ActorSystem in your logic.

Cluster aware router max-total-nr-of-instances

In 2.3.x the deployment configuration property `nr-of-instances` was used for cluster aware routers to specify total number of routees in the cluster. This was confusing, especially since the default value is 1.

In 2.4.x there is a new deployement property `cluster.max-total-nr-of-instances` that defines total number of routees in the cluster. By default `max-total-nr-of-instances` is set to a high value (10000) that will result in new routees added to the router when nodes join the cluster. Set it to a lower value if you want to limit total number of routees.

For backwards compatibility reasons `nr-of-instances` is still used if defined by user, i.e. if defined it takes precedence over `max-total-nr-of-instances`.

Logger names use full class name

Previously, few places in Akka used "simple" logger names, such as `Cluster` or `Remoting`. Now they use full class names, such as `akka.cluster.Cluster` or `akka.remote.Remoting`, in order to allow package level log level definitions and ease source code lookup. In case you used specific "simple" logger name based rules in your `logback.xml` configurations, please change them to reflect appropriate package name, such as `<logger name='akka.cluster' level='warn' />` or `<logger name='akka.remote' level='error' />`

Default interval for TestKit.awaitAssert changed to 100 ms

Default check interval changed from 800 ms to 100 ms. You can define the interval explicitly if you need a longer interval.

Secure Cookies

Secure cookies feature was deprecated.

AES128CounterInetRNG and AES256CounterInetRNG are Deprecated

Use `AES128CounterSecureRNG` or `AES256CounterSecureRNG` as `akka.remote.netty.ssl.security.random-number-generator`.

Microkernel is Deprecated

Akka Microkernel is deprecated and will be removed. It is replaced by using an ordinary user defined main class and packaging with sbt-native-packager or Lightbend ConductR. Please see *Use-case and Deployment Scenarios* for more information.

New Cluster Metrics Extension

Previously, cluster metrics functionality was located in the `akka-cluster` jar. Now it is split out and moved into a separate Akka module: `akka-cluster-metrics` jar. The module comes with few enhancements, such as use of Kamon sigar-loader for native library provisioning as well as use of statistical averaging of metrics data. Note that both old and new metrics configuration entries in the `reference.conf` are still in the same name space `akka.cluster.metrics` but are not compatible. Make sure to disable legacy metrics in akka-cluster: `akka.cluster.metrics.enabled=off`, since it is still enabled in akka-cluster by default (for compatibility with past releases). Router configuration entries have also changed for the module, they use prefix `cluster-metrics-`: `cluster-metrics-adaptive-pool` and `cluster-metrics-adaptive-group` Metrics extension classes and objects are located in the new package `akka.cluster.metrics`. Please see *Scala*, Java for more information.

Cluster tools moved to separate module

The Cluster Singleton, Distributed Pub-Sub, and Cluster Client previously located in the `akka-contrib` jar is now moved to a separate module named `akka-cluster-tools`. You need to replace this dependency if you use any of these tools.

The classes changed package name from `akka.contrib.pattern` to `akka.cluster.singleton`, `akka.cluster.pubsub` and `akka.cluster.client`.

The configuration properties changed name to `akka.cluster.pub-sub` and `akka.cluster.client`.

Cluster sharding moved to separate module

The Cluster Sharding previously located in the `akka-contrib` jar is now moved to a separate module named `akka-cluster-sharding`. You need to replace this dependency if you use Cluster Sharding.

The classes changed package name from `akka.contrib.pattern` to `akka.cluster.sharding`.

The configuration properties changed name to `akka.cluster.sharding`.

ClusterSharding construction

Several parameters of the `start` method of the `ClusterSharding` extension are now defined in a settings object `ClusterShardingSettings`. It can be created from system configuration properties and also amended with API. These settings can be defined differently per entry type if needed.

Starting the `ShardRegion` in proxy mode is now done with the `startProxy` method of the `ClusterSharding` extension instead of the optional `entryProps` parameter.

Entry was renamed to Entity, for example in the `MessagesExtractor` in the Java API and the `EntityId` type in the Scala API.

`idExtractor` function was renamed to `extractEntityId`. `shardResolver` function was renamed to `extractShardId`.

Cluster Sharding Entry Path Change

Previously in `2.3.x` entries were direct children of the local `ShardRegion`. In examples the `persistenceId` of entries included `self.path.parent.name` to include the cluster type name.

In `2.4.x` entries are now children of a `Shard`, which in turn is a child of the local `ShardRegion`. To include the shard type in the `persistenceId` it is now accessed by `self.path.parent.parent.name` from each entry.

Asynchronous ShardAllocationStrategy

The methods of the `ShardAllocationStrategy` and `AbstractShardAllocationStrategy` in Cluster Sharding have changed return type to a `Future` to support asynchronous decision. For example you can ask an actor external actor of how to allocate shards or rebalance shards.

For the synchronous case you can return the result via `scala.concurrent.Future.successful` in Scala or `akka.dispatch.Futures.successful` in Java.

Cluster Sharding internal data

The Cluster Sharding coordinator stores the locations of the shards using Akka Persistence. This data can safely be removed when restarting the whole Akka Cluster.

The serialization format of the internal persistent events stored by the Cluster Sharding coordinator has been changed and it cannot load old data from 2.3.x or some 2.4 milestone.

The `persistenceId` of the Cluster Sharding coordinator has been changed since 2.3.x so it should not load such old data, but it can be a problem if you have used a 2.4 milestone release. In that case you should remove the persistent data that the Cluster Sharding coordinator stored. Note that this is not application data.

You can use the *RemoveInternalClusterShardingData* utility program to remove this data.

The new `persistenceId` is `s"/sharding/${typeName}Coordinator"`. The old `persistenceId` is `s"/user/sharding/${typeName}Coordinator/singleton/coordinator"`.

ClusterSingletonManager and ClusterSingletonProxy construction

Parameters to the `Props` factory methods have been moved to settings object `ClusterSingletonManagerSettings` and `ClusterSingletonProxySettings`. These can be created from system configuration properties and also amended with API as needed.

The buffer size of the `ClusterSingletonProxy` can be defined in the `ClusterSingletonProxySettings` instead of defining `stash-capacity` of the mailbox. Buffering can be disabled by using a buffer size of 0.

The `singletonPath` parameter of `ClusterSingletonProxy.props` has changed. It is now named `singletonManagerPath` and is the logical path of the singleton manager, e.g. `/user/singletonManager`, which ends with the name you defined in `actorOf` when creating the `ClusterSingletonManager`. In 2.3.x it was the path to singleton instance, which was error-prone because one had to provide both the name of the singleton manager and the singleton actor.

DistributedPubSub construction

Normally, the `DistributedPubSubMediator` actor is started by the `DistributedPubSubExtension`. This extension has been renamed to `DistributedPubSub`. It is also possible to start it as an ordinary actor if you need multiple instances of it with different settings. The parameters of the `Props` factory methods in the `DistributedPubSubMediator` companion has been moved to settings object `DistributedPubSubSettings`. This can be created from system configuration properties and also amended with API as needed.

ClusterClient construction

The parameters of the `Props` factory methods in the `ClusterClient` companion has been moved to settings object `ClusterClientSettings`. This can be created from system configuration properties and also amended with API as needed.

The buffer size of the `ClusterClient` can be defined in the `ClusterClientSettings` instead of defining `stash-capacity` of the mailbox. Buffering can be disabled by using a buffer size of 0.

Normally, the `ClusterReceptionist` actor is started by the `ClusterReceptionistExtension`. This extension has been renamed to `ClusterClientReceptionist`. It is also possible to start it as an ordinary actor if you need multiple instances of it with different settings. The parameters of the `Props` factory methods in the `ClusterReceptionist` companion has been moved to settings object `ClusterReceptionistSettings`. This can be created from system configuration properties and also amended with API as needed.

The `ClusterReceptionist` actor that is started by the `ClusterReceptionistExtension` is now started as a `system` actor instead of a `user` actor, i.e. the default path for the `ClusterClient` initial contacts has changed to `"akka.tcp://system@hostname:port/system/receptionist"`.

ClusterClient sender

In 2.3 the `sender()` of the response messages, as seen by the client, was the actor in cluster.

In 2.4 the `sender()` of the response messages, as seen by the client, is `deadLetters` since the client should normally send subsequent messages via the `ClusterClient`. It is possible to pass the original sender inside the reply messages if the client is supposed to communicate directly to the actor in the cluster.

Akka Persistence

Experimental removed

The artifact name has changed from `akka-persistence-experimental` to `akka-persistence`.

New sbt dependency:

```
"com.typesafe.akka" %% "akka-persistence" % "2.4.14"
```

New Maven dependency:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-persistence_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

The artefact name of the Persistent TCK has changed from `akka-persistence-tck-experimental` (`akka-persistence-experimental-tck`) to `akka-persistence-tck`.

Mandatory persistenceId

It is now mandatory to define the `persistenceId` in subclasses of `PersistentActor`, `UntypedPersistentActor` and `AbstractPersistentId`.

The rationale behind this change being stricter de-coupling of your Actor hierarchy and the logical "which persistent entity this actor represents".

In case you want to preserve the old behavior of providing the actor's path as the default `persistenceId`, you can easily implement it yourself either as a helper trait or simply by overriding `persistenceId` as follows:

```
override def persistenceId = self.path.toStringWithoutAddress
```

Failures

Backend journal failures during recovery and persist are treated differently than in 2.3.x. The `PersistenceFailure` message is removed and the actor is unconditionally stopped. The new behavior and reasons for it is explained in *Failures*.

Persist sequence of events

The `persist` method that takes a `Seq` (Scala) or `Iterable` (Java) of events parameter was deprecated and renamed to `persistAll` to avoid mistakes of persisting other collection types as one single event by calling the overloaded `persist(event)` method.

non-permanent deletion

The `permanent` flag in `deleteMessages` was removed. non-permanent deletes are not supported any more. Events that were deleted with `permanent=false` with older version will still not be replayed in this version.

Recover message is gone, replaced by Recovery config

Previously the way to cause recover in PersistentActors was sending them a `Recover()` message. Most of the time it was the actor itself sending such message to `self` in its `preStart` method, however it was possible to send this message from an external source to any `PersistentActor` or `PresistentView` to make it start recovering.

This style of starting recovery does not fit well with usual Actor best practices: an Actor should be independent and know about its internal state, and also about its recovery or lack thereof. In order to guide users towards more independent Actors, the `Recovery()` object is now not used as a message, but as configuration option used by the Actor when it starts. In order to migrate previous code which customised its recovery mode use this example as reference:

```
// previously
class OldCookieMonster extends PersistentActor {
  def preStart() = self ! Recover(toSequenceNr = 42L)
  // ...
}
// now:
class NewCookieMonster extends PersistentActor {
  override def recovery = Recovery(toSequenceNr = 42L)
  // ...
}
```

Sender reference of replayed events is deadLetters

While undocumented, previously the `sender()` of the replayed messages would be the same sender that originally had sent the message. Since sender is an `ActorRef` and those events are often replayed in different incarnations of actor systems and during the entire lifetime of the app, relying on the existence of this reference is most likely not going to succeed. In order to avoid bugs in the style of "it worked last week", the `sender()` reference is now not stored, in order to avoid potential bugs which this could have provoked.

The previous behaviour was never documented explicitly (nor was it a design goal), so it is unlikely that applications have explicitly relied on this behaviour, however if you find yourself with an application that did exploit this you should rewrite it to explicitly store the `ActorPath` of where such replies during replay may have to be sent to, instead of relying on the sender reference during replay.

max-message-batch-size config

Configuration property `akka.persistence.journal.max-message-batch-size` has been moved into the plugin configuration section, to allow different values for different journal plugins. See `reference.conf`.

akka.persistence.snapshot-store.plugin config

The configuration property `akka.persistence.snapshot-store.plugin` now by default is empty. To restore the previous setting add `akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"` to your application.conf. See `reference.conf`.

PersistentView is deprecated

`PersistentView` is deprecated. Use *Persistence Query* instead. The corresponding query type is `EventsByPersistenceId`. There are several alternatives for connecting the `Source` to an actor corresponding to a previous `PersistentView` actor:

- Sink.actorRef is simple, but has the disadvantage that there is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow
- mapAsync combined with *Ask: Send-And-Receive-Future* is almost as simple with the advantage of back-pressure being propagated all the way
- ActorSubscriber in case you need more fine grained control

The consuming actor may be a plain `Actor` or a `PersistentActor` if it needs to store its own state (e.g. fromSequenceNr offset).

Persistence Plugin APIs

SyncWriteJournal removed

`SyncWriteJournal` removed in favor of using `AsyncWriteJournal`.

If the storage backend API only supports synchronous, blocking writes, the methods can still be implemented in terms of the asynchronous API. Example of how to do that is in included in the See *Journal plugin API for Scala* or Journal plugin API for Java.

SnapshotStore: Snapshots can now be deleted asynchronously (and report failures)

Previously the `SnapshotStore` plugin SPI did not allow for asynchronous deletion of snapshots, and failures of deleting a snapshot may have been even silently ignored.

Now `SnapshotStore` must return a `Future` representing the deletion of the snapshot. If this future completes successfully the `PersistentActor` which initiated the snapshotting will be notified via an `DeleteSnapshotSuccess` message. If the deletion fails for some reason a `DeleteSnapshotFailure` will be sent to the actor instead.

For `criteria` based deletion of snapshots (`def deleteSnapshots(criteria: SnapshotSelectionCriteria)`) equivalent `DeleteSnapshotsSuccess` and `DeleteSnapshotsFailure` messages are sent, which contain the specified criteria, instead of `SnapshotMetadata` as is the case with the single snapshot deletion messages.

SnapshotStore: Removed 'saved' callback

Snapshot Stores previously were required to implement a `def saved(meta: SnapshotMetadata): Unit` method which would be called upon successful completion of a `saveAsync` (`doSaveAsync` in Java API) snapshot write.

Currently all journals and snapshot stores perform asynchronous writes and deletes, thus all could potentially benefit from such callback methods. The only gain these callback give over composing an `onComplete` over `Future` returned by the journal or snapshot store is that it is executed in the Actors context, thus it can safely (without additional synchronization modify its internal state - for example a "pending writes" counter).

However, this feature was not used by many plugins, and expanding the API to accomodate all callbacks would have grown the API a lot. Instead, Akka Persistence 2.4.x introduces an additional (optionally overrideable) `receivePluginInternal:Actor.Receive` method in the plugin API, which can be used for handling those as well as any custom messages that are sent to the plugin Actor (imagine use cases like "wake up and continue reading" or custom protocols which your specialised journal can implement).

Implementations using the previous feature should adjust their code as follows:

```
// previously
class MySnapshots extends SnapshotStore {
  // old API:
  // def saved(meta: SnapshotMetadata): Unit = doThings()

  // new API:
  def saveAsync(metadata: SnapshotMetadata, snapshot: Any): Future[Unit] = {
    // completion or failure of the returned future triggers internal messages in␣
↪receivePluginInternal
    val f: Future[Unit] = ???

    // custom messages can be piped to self in order to be received in␣
↪receivePluginInternal
    f.map(MyCustomMessage(_)) pipeTo self

    f
  }

  def receivePluginInternal = {
    case SaveSnapshotSuccess(metadata) => doThings()
    case MyCustomMessage(data)         => doOtherThings()
  }

  // ...
}
```

SnapshotStore: Java 8 Optional used in Java plugin APIs

In places where previously `akka.japi.Option` was used in Java APIs, including the return type of `doLoadAsync`, the Java 8 provided `Optional` type is used now.

Please remember that when creating an `java.util.Optional` instance from a (possibly) `null` value you will want to use the non-throwing `Optional.fromNullable` method, which converts a `null` into a `None` value - which is slightly different than its Scala counterpart (where `Option.apply(null)` returns `None`).

Atomic writes

`asyncWriteMessages` takes a `immutable.Seq[AtomicWrite]` parameter instead of `immutable.Seq[PersistentRepr]`.

Each *AtomicWrite* message contains the single `PersistentRepr` that corresponds to the event that was passed to the `persist` method of the `PersistentActor`, or it contains several `PersistentRepr` that corresponds to the events that were passed to the `persistAll` method of the `PersistentActor`. All `PersistentRepr` of the *AtomicWrite* must be written to the data store atomically, i.e. all or none must be stored.

If the journal (data store) cannot support atomic writes of multiple events it should reject such writes with a `Try Failure` with an `UnsupportedOperationException` describing the issue. This limitation should also be documented by the journal plugin.

Rejecting writes

`asyncWriteMessages` returns a `Future[immutable.Seq[Try[Unit]]]`.

The journal can signal that it rejects individual messages (`AtomicWrite`) by the returned *immutable.Seq[Try[Unit]]*. The returned `Seq` must have as many elements as the input `messages Seq`. Each `Try` element signals if the corresponding `AtomicWrite` is rejected or not, with an exception describing the problem. Rejecting a message means it was not stored, i.e. it must not be included in a later replay. Rejecting a message is typically done before attempting to store it, e.g. because of serialization error.

Read the *API documentation* of this method for more information about the semantics of rejections and failures.

asyncReplayMessages Java API

The signature of *asyncReplayMessages* in the Java API changed from `akka.japi.Procedure` to `java.util.function.Consumer`.

asyncDeleteMessagesTo

The `permanent` deletion flag was removed. Support for non-permanent deletions was removed. Events that were deleted with `permanent=false` with older version will still not be replayed in this version.

References to "replay" in names

Previously a number of classes and methods used the word "replay" interchangeably with the word "recover". This lead to slight inconsistencies in APIs, where a method would be called `recovery`, yet the signal for a completed recovery was named `ReplayMessagesSuccess`.

This is now fixed, and all methods use the same "recovery" wording consistently across the entire API. The old `ReplayMessagesSuccess` is now called `RecoverySuccess`, and an additional method called `onRecoveryFailure` has been introduced.

AtLeastOnceDelivery deliver signature

The signature of `deliver` changed slightly in order to allow both `ActorSelection` and `ActorPath` to be used with it.

Previously:

> def deliver(destination: ActorPath, deliveryIdToMessage: Long ⇒ Any): Unit

Now:

> def deliver(destination: ActorSelection)(deliveryIdToMessage: Long ⇒ Any): Unit def deliver(destination: ActorPath)(deliveryIdToMessage: Long ⇒ Any): Unit

The Java API remains unchanged and has simply gained the 2nd overload which allows `ActorSelection` to be passed in directly (without converting to `ActorPath`).

Actor system shutdown

`ActorSystem.shutdown`, `ActorSystem.awaitTermination` and `ActorSystem.isTerminated` has been deprecated in favor of `ActorSystem.terminate` and `ActorSystem.whenTerminated`. Both returns a `Future[Terminated]` value that will complete when the actor system has terminated.

To get the same behavior as `ActorSystem.awaitTermination` block and wait for `Future[Terminated]` value with `Await.result` from the Scala standard library.

To trigger a termination and wait for it to complete:

import scala.concurrent.duration._ Await.result(system.terminate(), 10.seconds)

Be careful to not do any operations on the `Future[Terminated]` using the `system.dispatcher` as `ExecutionContext` as it will be shut down with the `ActorSystem`, instead use for example the Scala standard library context from `scala.concurrent.ExecutionContext.global`.

```
// import system.dispatcher <- this would not work
import scala.concurrent.ExecutionContext.Implicits.global

system.terminate().foreach { _ =>
  println("Actor system was shut down")
}
```

12.1.8 Upcoming Migration Guide 2.4.x to 2.5.x

Akka Persistence

Persistence Plugin Proxy

A new *persistence plugin proxy* was added, that allows sharing of an otherwise non-sharable journal or snapshot store. The proxy is available by setting `akka.persistence.journal.plugin` or `akka.persistence.snapshot-store.plugin` to `akka.persistence.journal.proxy` or `akka.persistence.snapshot-store.proxy`, respectively. The proxy supplants the *Shared LevelDB journal*.

12.2 Issue Tracking

Akka is using GitHub Issues as its issue tracking system.

12.2.1 Browsing

Tickets

Before filing a ticket, please check the existing Akka tickets for earlier reports of the same problem. You are very welcome to comment on existing tickets, especially if you have reproducible test cases that you can share.

Roadmaps

Short and long-term plans are published in the akka/akka-meta repository.

12.2.2 Creating tickets

Please include the versions of Scala and Akka and relevant configuration files.

You can create a new ticket if you have registered a GitHub user account.

Thanks a lot for reporting bugs and suggesting features!

12.2.3 Submitting Pull Requests

Note: *A pull request is worth a thousand +1's.* – Old Klangian Proverb

Pull Requests fixing issues or adding functionality are very welcome. Please read CONTRIBUTING.md for more information about contributing to Akka.

12.3 Licenses

12.3.1 Akka License

```
This software is licensed under the Apache 2 license, quoted below.

Copyright 2009-2015 Lightbend Inc. <http://www.lightbend.com>

Licensed under the Apache License, Version 2.0 (the "License"); you may not
use this file except in compliance with the License. You may obtain a copy of
the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
License for the specific language governing permissions and limitations under
the License.
```

12.3.2 Akka Committer License Agreement

All committers have signed this CLA. It can be signed online.

12.3.3 Licenses for Dependency Libraries

Each dependency and its license can be seen in the project build file (the comment on the side of each dependency): AkkaBuild.scala

12.4 Sponsors

12.4.1 Lightbend

Lightbend is the company behind the Akka Project, Scala Programming Language, Play Web Framework, Scala IDE, sbt and many other open source projects. It also provides the Lightbend Stack, a full-featured development stack consisting of AKka, Play and Scala. Learn more at lightbend.com.

12.4.2 YourKit

YourKit is kindly supporting open source projects with its full-featured Java Profiler.

YourKit, LLC is the creator of innovative and intelligent tools for profiling Java and .NET applications. Take a look at YourKit's leading software products: YourKit Java Profiler and YourKit .NET Profiler

12.5 Project

12.5.1 Commercial Support

Commercial support is provided by Lightbend. Akka is part of the Lightbend Reactive Platform.

12.5.2 Mailing List

Akka User Google Group

Akka Developer Google Group

12.5.3 Downloads

http://akka.io/downloads

12.5.4 Source Code

Akka uses Git and is hosted at Github.

- Akka: clone the Akka repository from http://github.com/akka/akka

12.5.5 Releases Repository

All Akka releases are published via Sonatype to Maven Central, see search.maven.org or search.maven.org (Akka versions before 2.4.3)

12.5.6 Snapshots Repository

Nightly builds are available in http://repo.akka.io/snapshots/ as both `SNAPSHOT` and timestamped versions.

For timestamped versions, pick a timestamp from http://repo.akka.io/snapshots/com/lightbend/akka/akka-actor_2.11/. All Akka modules that belong to the same build have the same timestamp.

sbt definition of snapshot repository

Make sure that you add the repository to the sbt resolvers:

```
resolvers += "Lightbend Snapshots" at "http://repo.akka.io/snapshots/"
```

Define the library dependencies with the timestamp as version. For example:

```
libraryDependencies += "com.typesafe.akka" % "akka-remote_2.11" %
  "2.1-20121016-001042"
```

maven definition of snapshot repository

Make sure that you add the repository to the maven repositories in pom.xml:

```
<repositories>
  <repository>
    <id>akka-snapshots</id>
    <name>Akka Snapshots</name>
    <url>http://repo.akka.io/snapshots/</url>
    <layout>default</layout>
  </repository>
</repositories>
```

Define the library dependencies with the timestamp as version. For example:

```
<dependencies>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_2.11</artifactId>
    <version>2.1-20121016-001042</version>
  </dependency>
</dependencies>
```

CHAPTER

THIRTEEN

ADDITIONAL INFORMATION

13.1 Binary Compatibility Rules

Akka maintains and verifies *backwards binary compatibility* across versions of modules.

In the rest of this document whenever *binary compatibility* is mentioned "*backwards binary compatibility*" is meant (as opposed to forward compatibility).

This means that the new JARs are a drop-in replacement for the old one (but not the other way around) as long as your build does not enable the inliner (Scala-only restriction).

13.1.1 Binary compatibility rules explained

Binary compatibility is maintained between:

- **minor** and **patch** versions - please note that the meaning of "minor" has shifted to be more restrictive with Akka `2.4.0`, read *Change in versioning scheme, stronger compatibility since 2.4* for details.

Binary compatibility is **NOT** maintained between:

- **major** versions
- any versions of **experimental** modules – read *The meaning of "experimental"* for details
- a few notable exclusions explained below

Specific examples (please read *Change in versioning scheme, stronger compatibility since 2.4* to understand the difference in "before 2.4 era" and "after 2.4 era"):

```
# [epoch.major.minor] era
OK:  2.2.0 --> 2.2.1 --> ... --> 2.2.x
NO:  2.2.y --x 2.3.y
OK:  2.3.0 --> 2.3.1 --> ... --> 2.3.x
OK:  2.3.x --> 2.4.x (special case, migration to new versioning scheme)
# [major.minor.path] era
OK:  2.4.0 --> 2.5.x
OK:  2.5.0 --> 2.6.x
NO:  2.x.y --x 3.x.y
OK:  3.0.0 --> 3.0.1 --> ... --> 3.0.n
OK:  3.0.n --> 3.1.0 --> ... --> 3.1.n
OK:  3.1.n --> 3.2.0 ...
     ...
```

Cases where binary compatibility is not retained

Some modules are excluded from the binary compatibility guarantees, such as:

- `*-testkit` modules - since these are to be used only in tests, which usually are re-compiled and run on demand

- **`*-tck` modules - since they may want to add new tests (or force configuring something), in order to discover possible** failures in an existing implementation that the TCK is supposed to be testing. Compatibility here is not *guaranteed*, however it is attempted to make the upgrade prosess as smooth as possible.
- all *experimental* modules - which by definition are subject to rapid iteration and change. Read more about them in *The meaning of "experimental"*

13.1.2 Change in versioning scheme, stronger compatibility since 2.4

Since the release of Akka `2.4.0` a new versioning scheme is in effect.

Historically, Akka has been following the Java or Scala style of versioning where as the first number would mean "**epoch**", the second one would mean **major**, and third be the **minor**, thus: `epoch.major.minor` (versioning scheme followed until and during `2.3.x`).

Currently, since Akka `2.4.0`, the new versioning applies which is closer to semantic versioning many have come to expect, in which the version number is deciphered as `major.minor.patch`.

In addition to that, Akka `2.4.x` has been made binary compatible with the `2.3.x` series, so there is no reason to remain on Akka 2.3.x, since upgrading is completely compatible (and many issues have been fixed ever since).

13.1.3 Mixed versioning is not allowed

Modules that are released together under the Akka project are intended to be upgraded together. For example, it is not legal to mix Akka Actor `2.4.2` with Akka Cluster `2.4.5` even though "Akka `2.4.2`" and "Akka `2.4.5`" *are* binary compatible.

This is because modules may assume internals changes across module boundaries, for example some feature in Clustering may have required an internals change in Actor, however it is not public API, thus such change is considered safe.

Note: We recommend keeping an `akkaVersion` variable in your build file, and re-use it for all included modules, so when you upgrade you can simply change it in this one place.

13.1.4 The meaning of "experimental"

Experimental is a keyword used in module descriptions as well as their artifact names, in order to signify that the API that they contain is subject to change without any prior warning.

Experimental modules are are not covered by Lightbend's Commercial Support, unless specifically stated otherwise. The purpose of releasing them early, as experimental, is to make them easily available and improve based on feedback, or even discover that the module wasn't useful.

An experimental module doesn't have to obey the rule of staying binary compatible between micro releases. Breaking API changes may be introduced in minor releases without notice as we refine and simplify based on your feedback. An experimental module may be dropped in minor releases without prior deprecation.

Best effort migration guides may be provided, but this is decided on a case-by-case basis for **experimental** modules.

13.1.5 The meaning of INTERNAL API

When browsing the source code and/or looking for methods available to be called, especially from Java which does not have as rich of an access protection system as Scala has, you may sometimes find methods or classes annotated with the `/** INTERNAL API */` comment.

No compatibility guarantees are given about these classes, they may change or even disapear in minor versions, and user code is not supposed to be calling (or even touching) them.

Side-note on JVM representation details of the Scala `private[akka]` pattern that Akka is using extensively in it's internals: Such methods or classes, which act as "accessible only from the given package" in Scala, are compiled down to `public` (!) in raw Java bytecode, and the access restriction, that Scala understands is carried along as metadata stored in the classfile. Thus, such methods are safely guarded from being accessed from Scala, however Java users will not be warned about this fact by the `javac` compiler. Please be aware of this and do not call into Internal APIs, as they are subject to change without any warning.

13.1.6 Binary Compatibility Checking Toolchain

Akka uses the Lightbend maintained Migration Manager, called `MiMa` for short, for enforcing binary compatibility is kept where it was promised.

All Pull Requests must pass MiMa validation (which happens automatically), and if failures are detected, manual exception overrides may be put in place if the change happened to be in an Internal API for example.

13.2 Frequently Asked Questions

13.2.1 Akka Project

Where does the name Akka come from?

It is the name of a beautiful Swedish mountain up in the northern part of Sweden called Laponia. The mountain is also sometimes called 'The Queen of Laponia'.

Akka is also the name of a goddess in the Sámi (the native Swedish population) mythology. She is the goddess that stands for all the beauty and good in the world. The mountain can be seen as the symbol of this goddess.

Also, the name AKKA is the a palindrome of letters A and K as in Actor Kernel.

Akka is also:

- the name of the goose that Nils traveled across Sweden on in The Wonderful Adventures of Nils by the Swedish writer Selma Lagerlöf.
- the Finnish word for 'nasty elderly woman' and the word for 'elder sister' in the Indian languages Tamil, Telugu, Kannada and Marathi.
- a font
- a town in Morocco
- a near-earth asteroid

13.2.2 Resources with Explicit Lifecycle

Actors, ActorSystems, ActorMaterializers (for streams), all these types of objects bind resources that must be released explicitly. The reason is that Actors are meant to have a life of their own, existing independently of whether messages are currently en route to them. Therefore you should always make sure that for every creation of such an object you have a matching `stop`, `terminate`, or `shutdown` call implemented.

In particular you typically want to bind such values to immutable references, i.e. `final ActorSystem system` in Java or `val system: ActorSystem` in Scala.

JVM application or Scala REPL "hanging"

Due to an ActorSystem's explicit lifecycle the JVM will not exit until it is stopped. Therefore it is necessary to shutdown all ActorSystems within a running application or Scala REPL session in order to allow these processes to terminate.

Shutting down an ActorSystem will properly terminate all Actors and ActorMaterializers that were created within it.

13.2.3 Actors in General

sender()/getSender() disappears when I use Future in my Actor, why?

When using future callbacks, inside actors you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This breaks the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time.

Read more about it in the docs for *Actors and shared mutable state*.

Why OutOfMemoryError?

It can be many reasons for OutOfMemoryError. For example, in a pure push based system with message consumers that are potentially slower than corresponding message producers you must add some kind of message flow control. Otherwise messages will be queued in the consumers' mailboxes and thereby filling up the heap memory.

Some articles for inspiration:

- Balancing Workload across Nodes with Akka 2.
- Work Pulling Pattern to prevent mailbox overflow, throttle and distribute work

13.2.4 Actors Scala API

How can I get compile time errors for missing messages in *receive*?

One solution to help you get a compile time warning for not handling a message that you should be handling is to define your actors input and output messages implementing base traits, and then do a match that the will be checked for exhaustiveness.

Here is an example where the compiler will warn you that the match in receive isn't exhaustive:

```
object MyActor {
  // these are the messages we accept
  sealed abstract trait Message
  final case class FooMessage(foo: String) extends Message
  final case class BarMessage(bar: Int) extends Message

  // these are the replies we send
  sealed abstract trait Reply
  final case class BazMessage(baz: String) extends Reply
}

class MyActor extends Actor {
  import MyActor._
  def receive = {
    case message: Message => message match {
      case BarMessage(bar) => sender() ! BazMessage("Got " + bar)
      // warning here:
      // "match may not be exhaustive. It would fail on the following input:␣
↪FooMessage(_)"
    }
  }
}
```

13.2.5 Remoting

I want to send to a remote system but it does not do anything

Make sure that you have remoting enabled on both ends: client and server. Both do need hostname and port configured, and you will need to know the port of the server; the client can use an automatic port in most cases (i.e. configure port zero). If both systems are running on the same network host, their ports must be different

If you still do not see anything, look at what the logging of remote life-cycle events tells you (normally logged at INFO level) or switch on logging-remote-java to see all sent and received messages (logged at DEBUG level).

Which options shall I enable when debugging remoting issues?

Have a look at the remote-configuration-java, the typical candidates are:

- *akka.remote.log-sent-messages*
- *akka.remote.log-received-messages*
- *akka.remote.log-remote-lifecycle-events* (this also includes deserialization errors)

What is the name of a remote actor?

When you want to send messages to an actor on a remote host, you need to know its *full path*, which is of the form:

```
akka.protocol://system@host:1234/user/my/actor/hierarchy/path
```

Observe all the parts you need here:

- **`protocol` is the protocol to be used to communicate with the remote system.** Most of the cases this is *tcp*.
- `system` is the remote system's name (must match exactly, case-sensitive!)
- `host` is the remote system's IP address or DNS name, and it must match that system's configuration (i.e. *akka.remote.netty.tcp.hostname*)
- `1234` is the port number on which the remote system is listening for connections and receiving messages
- `/user/my/actor/hierarchy/path` is the absolute path of the remote actor in the remote system's supervision hierarchy, including the system's guardian (i.e. `/user`, there are others e.g. `/system` which hosts loggers, `/temp` which keeps temporary actor refs used with `ask`, `/remote` which enables remote deployment, etc.); this matches how the actor prints its own `self` reference on the remote host, e.g. in log output.

Why are replies not received from a remote actor?

The most common reason is that the local system's name (i.e. the `system@host:1234` part in the answer above) is not reachable from the remote system's network location, e.g. because `host` was configured to be `0.0.0.0`, `localhost` or a NAT'ed IP address.

If you are running an ActorSystem under a NAT or inside a docker container, make sure to set *akka.remote.netty.tcp.hostname* and *akka.remote.netty.tcp.port* to the address it is reachable at from other ActorSystems. If you need to bind your network interface to a different address - use *akka.remote.netty.tcp.bind-hostname* and *akka.remote.netty.tcp.bind-port* settings. Also make sure your network is configured to translate from the address your ActorSystem is reachable at to the address your ActorSystem network interface is bound to.

How reliable is the message delivery?

The general rule is **at-most-once delivery**, i.e. no guaranteed delivery. Stronger reliability can be built on top, and Akka provides tools to do so.

Read more in *Message Delivery Reliability*.

13.2.6 Debugging

How do I turn on debug logging?

To turn on debug logging in your actor system add the following to your configuration:

```
akka.loglevel = DEBUG
```

To enable different types of debug logging add the following to your configuration:

- `akka.actor.debug.receive` will log all messages sent to an actor if that actors *receive* method is a `LoggingReceive`
- `akka.actor.debug.autoreceive` will log all *special* messages like `Kill`, `PoisonPill` e.t.c. sent to all actors
- `akka.actor.debug.lifecycle` will log all actor lifecycle events of all actors

Read more about it in the docs for logging-java and *Tracing Actor Invocations*.

13.3 Books

- Learning Akka, by Jason Goodwin, PACKT Publishing, ISBN: 9781784393007, December 2015
- Akka in Action, by Raymond Roestenburg and Rob Bakker, Manning Publications Co., ISBN: 9781617291012, estimated in 2016
- Reactive Messaging Patterns with the Actor Model, by Vaughn Vernon, Addison-Wesley Professional, ISBN: 0133846830, August 2015
- Developing an Akka Edge, by Thomas Lockney and Raymond Tay, Bleeding Edge Press, ISBN: 9781939902054, April 2014
- Effective Akka, by Jamie Allen, O'Reilly Media, ISBN: 1449360076, August 2013
- Akka Concurrency, by Derek Wyatt, artima developer, ISBN: 0981531660, May 2013
- Akka Essentials, by Munish K. Gupta, PACKT Publishing, ISBN: 1849518289, October 2012

13.4 Videos

- Learning Akka Videos, by Salma Khater, PACKT Publishing, ISBN: 9781784391836, January 2016

13.5 Akka in OSGi

13.5.1 Background

OSGi is a mature packaging and deployment standard for component-based systems. It has similar capabilities as Project Jigsaw (originally scheduled for JDK 1.8), but has far stronger facilities to support legacy Java code. This is to say that while Jigsaw-ready modules require significant changes to most source files and on occasion to the

structure of the overall application, OSGi can be used to modularize almost any Java code as far back as JDK 1.2, usually with no changes at all to the binaries.

These legacy capabilities are OSGi's major strength and its major weakness. The creators of OSGi realized early on that implementors would be unlikely to rush to support OSGi metadata in existing JARs. There were already a handful of new concepts to learn in the JRE and the added value to teams that were managing well with straight J2EE was not obvious. Facilities emerged to "wrap" binary JARs so they could be used as bundles, but this functionality was only used in limited situations. An application of the "80/20 Rule" here would have that "80% of the complexity is with 20% of the configuration", but it was enough to give OSGi a reputation that has stuck with it to this day.

This document aims to the productivity basics folks need to use it with Akka, the 20% that users need to get 80% of what they want. For more information than is provided here, OSGi In Action is worth exploring.

13.5.2 Core Components and Structure of OSGi Applications

The fundamental unit of deployment in OSGi is the `Bundle`. A bundle is a Java JAR with *additional entries <https://www.osgi.org/bundle-headers-reference/>* in `MANIFEST.MF` that minimally expose the name and version of the bundle and packages for import and export. Since these manifest entries are ignored outside OSGi deployments, a bundle can interchangeably be used as a JAR in the JRE.

When a bundle is loaded, a specialized implementation of the Java `ClassLoader` is instantiated for each bundle. Each classloader reads the manifest entries and publishes both capabilities (in the form of the `Bundle-Exports`) and requirements (as `Bundle-Imports`) in a container singleton for discovery by other bundles. The process of matching imports to exports across bundles through these classloaders is the process of resolution, one of six discrete steps in the lifecycle FSM of a bundle in an OSGi container:

1. INSTALLED: A bundle that is installed has been loaded from disk and a classloader instantiated with its capabilities. Bundles are iteratively installed manually or through container-specific descriptors. For those familiar with legacy packging such as EJB, the modular nature of OSGi means that bundles may be used by multiple applications with overlapping dependencies. By resolving them individually from repositories, these overlaps can be de-duplicated across multiple deployemnts to the same container.
2. RESOLVED: A bundle that has been resolved is one that has had its requirements (imports) satisfied. Resolution does mean that a bundle can be started.
3. STARTING: A bundle that is started can be used by other bundles. For an otherwise complete application closure of resolved bundles, the implication here is they must be started in the order directed by a depth-first search for all to be started. When a bundle is starting, any exposed lifecycle interfaces in the bundle are called, giving the bundle the opportunity to start its own service endpoints and threads.
4. ACTIVE: Once a bundle's lifecycle interfaces return without error, a bundle is marked as active.
5. STOPPING: A bundle that is stopping is in the process of calling the bundle's stop lifecycle and transitions back to the RESOLVED state when complete. Any long running services or threads that were created while STARTING should be shut down when the bundle's stop lifecycle is called.
6. UNINSTALLED: A bundle can only transition to this state from the INSTALLED state, meaning it cannot be uninstalled before it is stopped.

Note the dependency in this FSM on lifecycle interfaces. While there is no requirement that a bundle publishes these interfaces or accepts such callbacks, the lifecycle interfaces provide the semantics of a `main()` method and allow the bundle to start and stop long-running services such as REST web services, ActorSystems, Clusters, etc.

Secondly, note when considering requirements and capabilities, it's a common misconception to equate these with repository dependencies as might be found in Maven or Ivy. While they provide similar practical functionality, OSGi has several parallel type of dependency (such as Blueprint Services) that cannot be easily mapped to repository capabilities. In fact, the core specification leaves these facilities up to the container in use. In turn, some containers have tooling to generate application load descriptors from repository metadata.

13.5.3 Notable Behavior Changes

Combined with understanding the bundle lifecycle, the OSGi developer must pay attention to sometimes unexpected behaviors that are introduced. These are generally within the JVM specification, but are unexpected and can lead to frustration.

- Bundles should not export overlapping package spaces. It is not uncommon for legacy JVM frameworks to expect plugins in an application composed of multiple JARs to reside under a single package name. For example, a frontend application might scan all classes from `com.example.plugins` for specific service implementations with that package existing in several contributed JARs.

  While it is possible to support overlapping packages with complex manifest headers, it's much better to use non-overlapping package spaces and facilities such as Akka Cluster for service discovery. Stylistically, many organizations opt to use the root package path as the name of the bundle distribution file.

- Resources are not shared across bundles unless they are explicitly exported, as with classes. The common case of this is expecting that `getClass().getClassLoader().getResources("foo")` will return all files on the classpath named `foo`. The `getResources()` method only returns resources from the current classloader, and since there are separate classloaders for every bundle, resource files such as configurations are no longer searchable in this manner.

13.5.4 Configuring the OSGi Framework

To use Akka in an OSGi environment, the container must be configured such that the `org.osgi.framework.bootdelegation` property delegates the `sun.misc` package to the boot classloader instead of resolving it through the normal OSGi class space.

13.5.5 Intended Use

Akka only supports the usage of an ActorSystem strictly confined to a single OSGi bundle, where that bundle contains or imports all of the actor system's requirements. This means that the approach of offering an ActorSystem as a service to which Actors can be deployed dynamically via other bundles is not recommended — an ActorSystem and its contained actors are not meant to be dynamic in this way. ActorRefs may safely be exposed to other bundles.

13.5.6 Activator

To bootstrap Akka inside an OSGi environment, you can use the `akka.osgi.ActorSystemActivator` class to conveniently set up the ActorSystem.

```
import akka.actor.{ Props, ActorSystem }
import org.osgi.framework.BundleContext
import akka.osgi.ActorSystemActivator

class Activator extends ActorSystemActivator {

  def configure(context: BundleContext, system: ActorSystem) {
    // optionally register the ActorSystem in the OSGi Service Registry
    registerService(context, system)

    val someActor = system.actorOf(Props[SomeActor], name = "someName")
    someActor ! SomeMessage
  }

}
```

The goal here is to map the OSGi lifecycle more directly to the Akka lifecycle. The `ActorSystemActivator` creates the actor system with a class loader that finds resources (`application.conf` and `reference.conf` files) and classes from the application bundle and all transitive dependencies.

The `ActorSystemActivator` class is included in the `akka-osgi` artifact:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-osgi_2.11</artifactId>
  <version>2.4.14</version>
</dependency>
```

13.5.7 Sample

A complete sample project is provided in akka-sample-osgi-dining-hakkers