CONTINUUITY

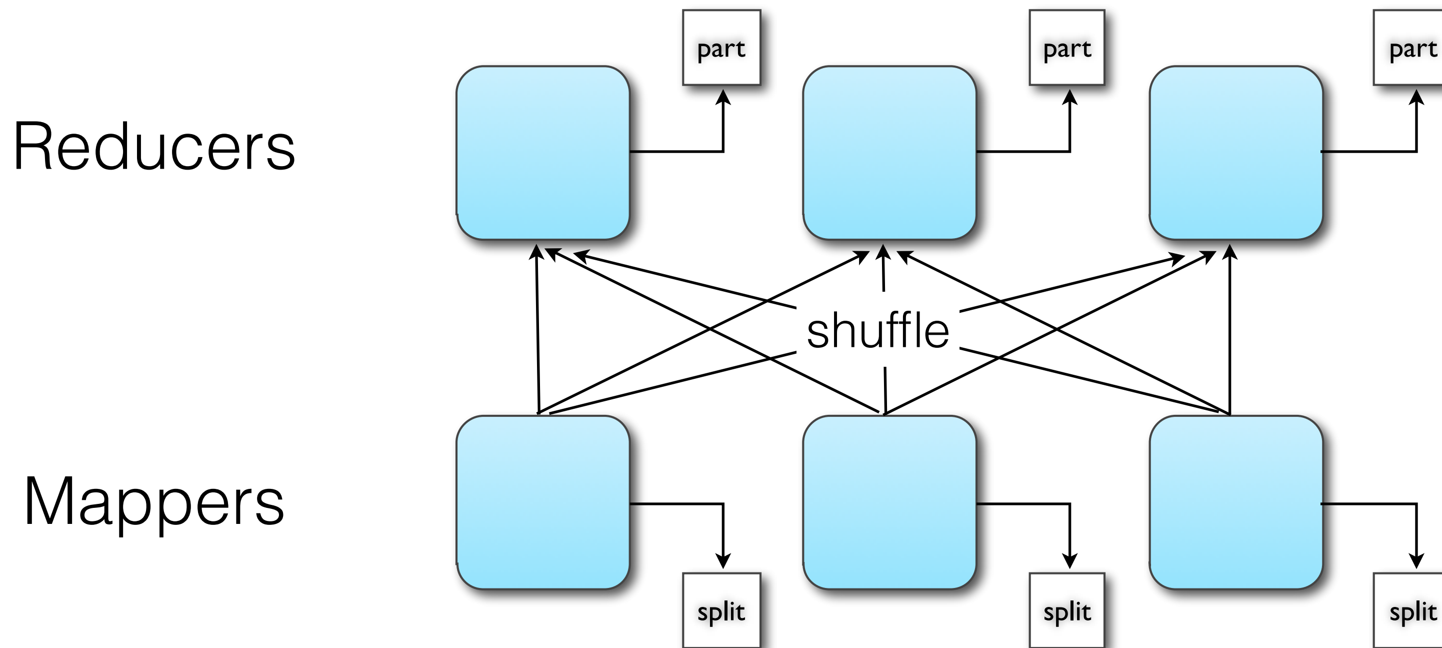
Harnessing the Power of YARN with Apache Twill

Terence Yim

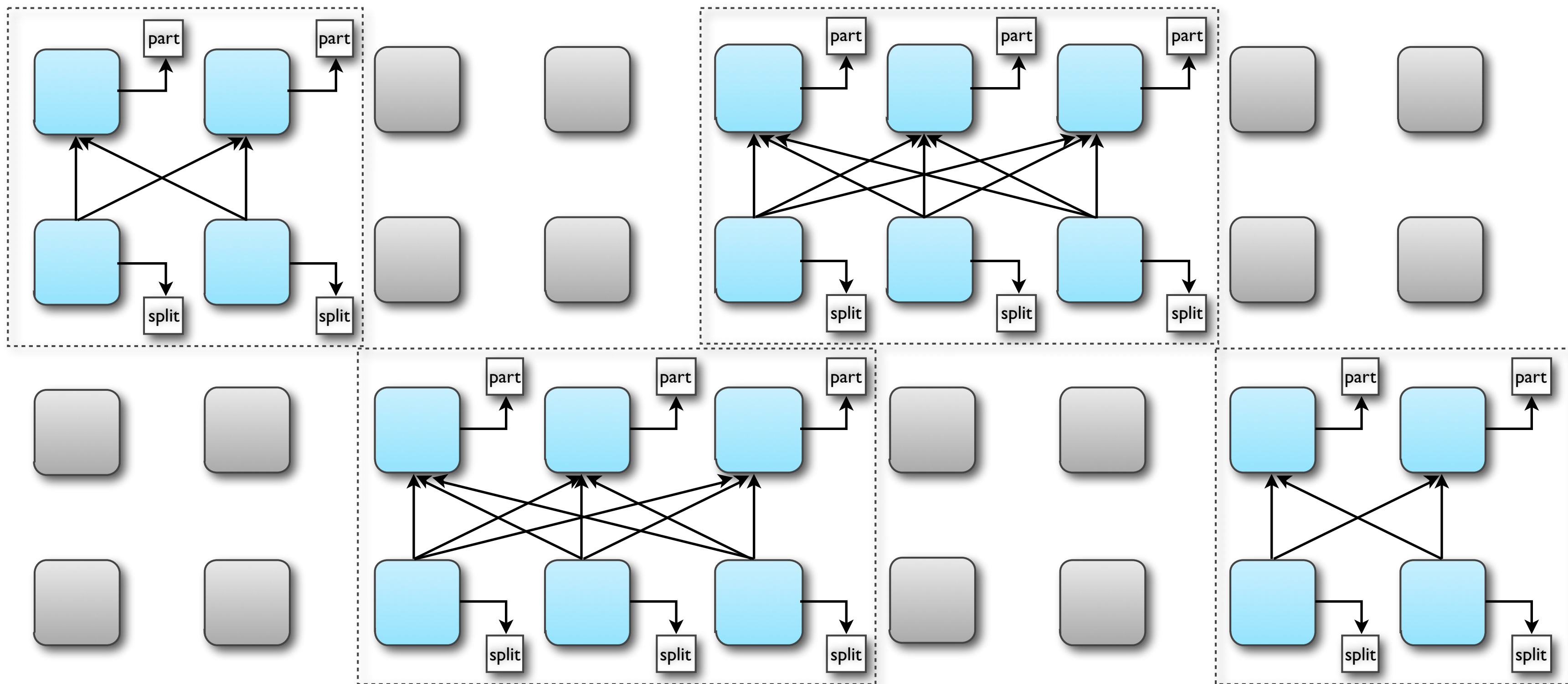
terence@continuuity.com

@chtyim

A Distributed App

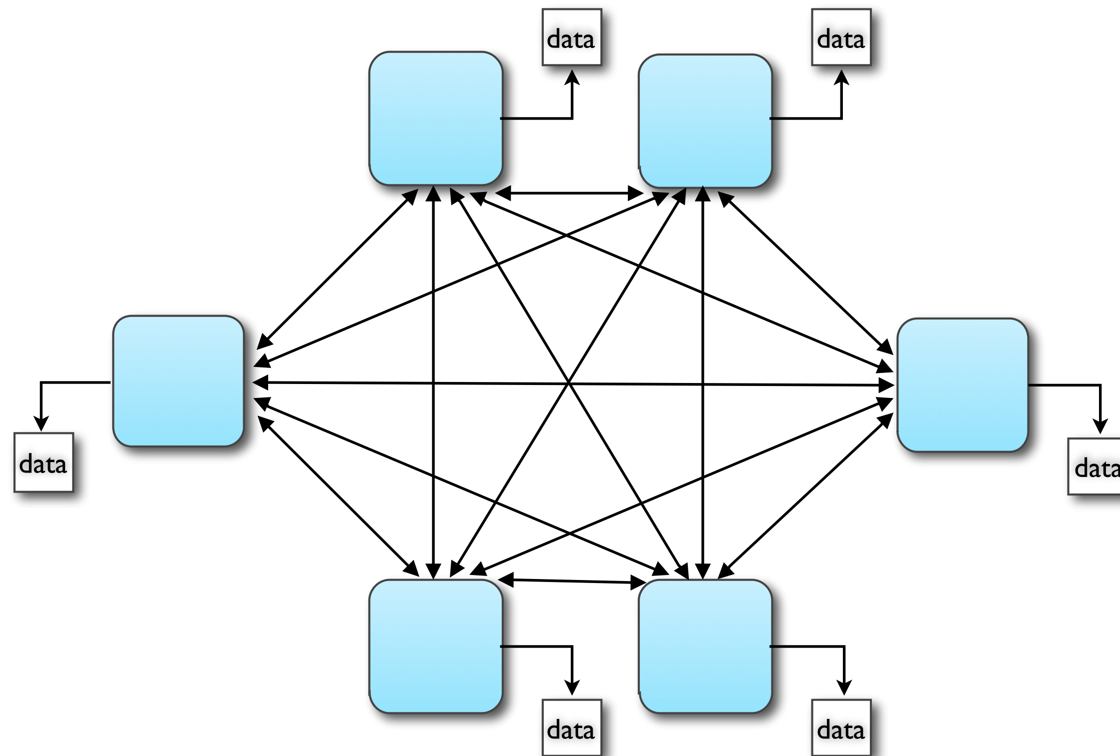


A Map/Reduce Cluster

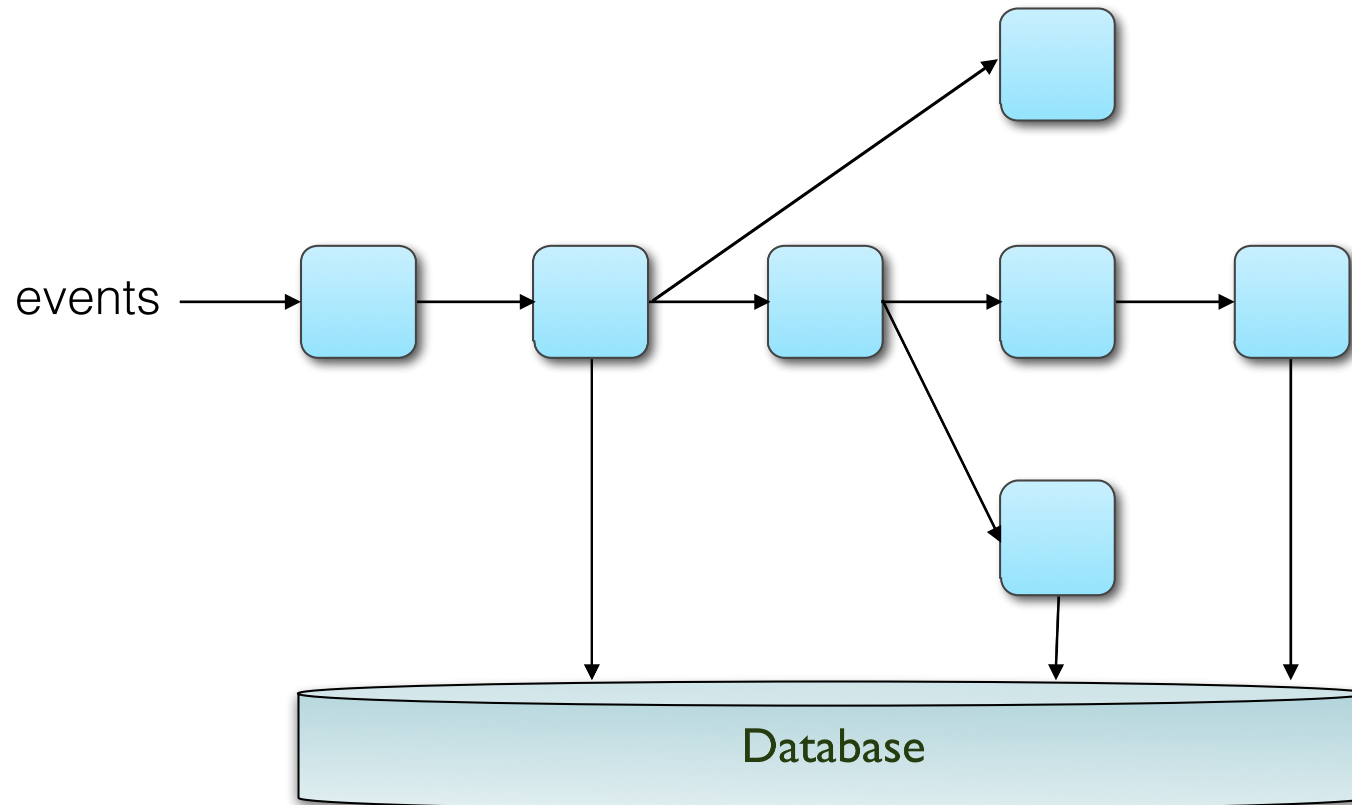


What Next?

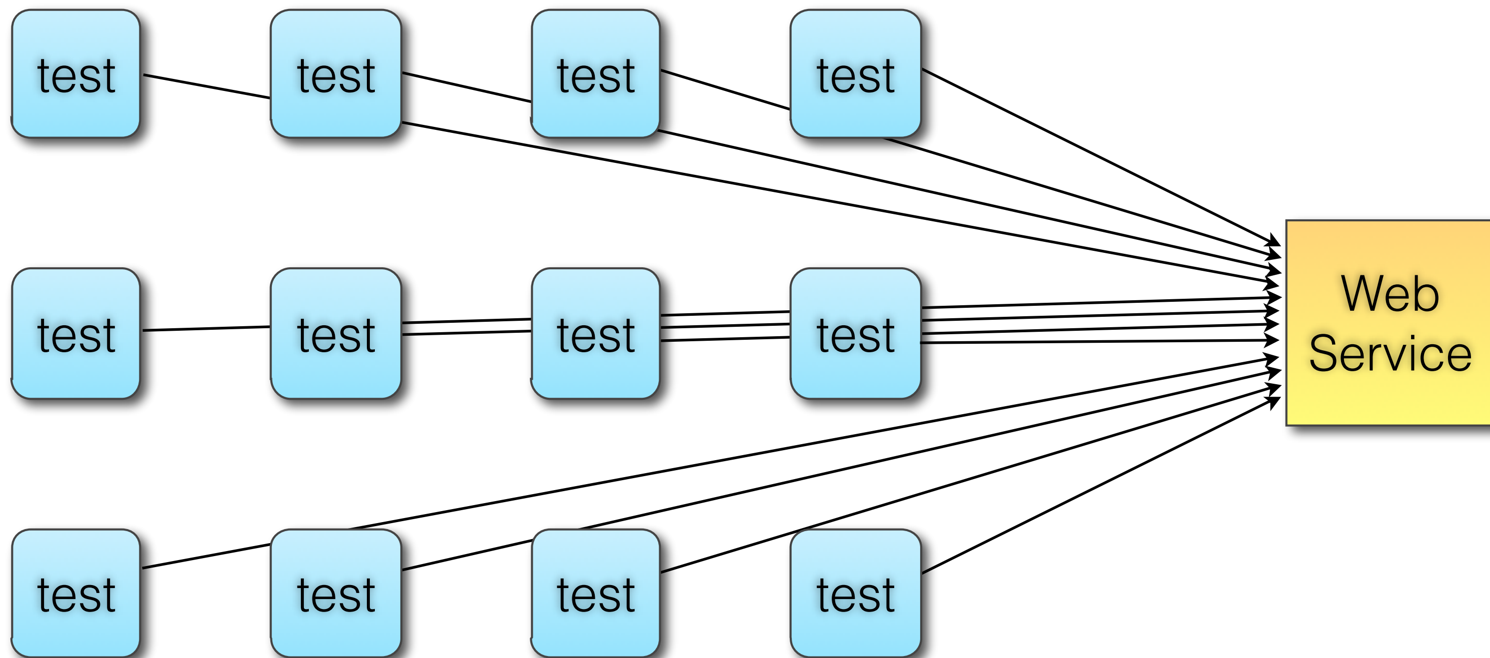
A Message Passing (MPI) App



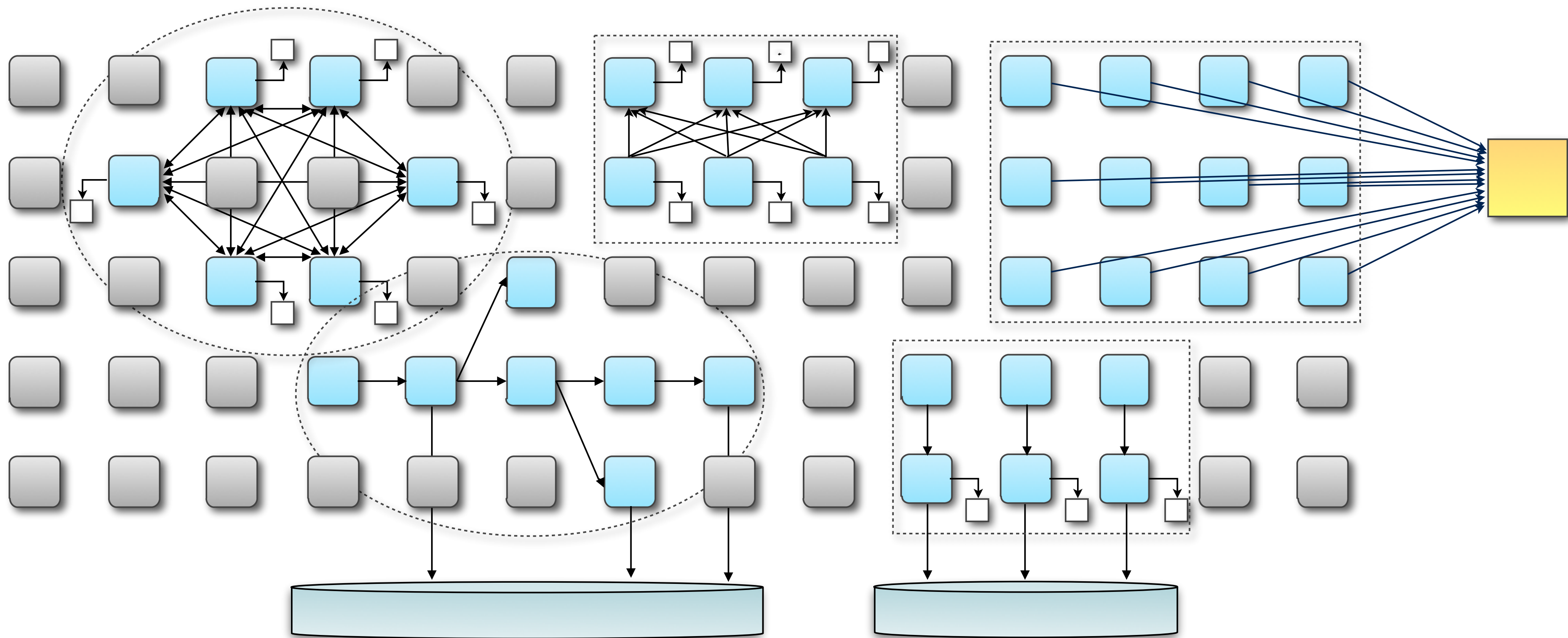
A Stream Processing App



A Distributed Load Test



A Multi-Purpose Cluster



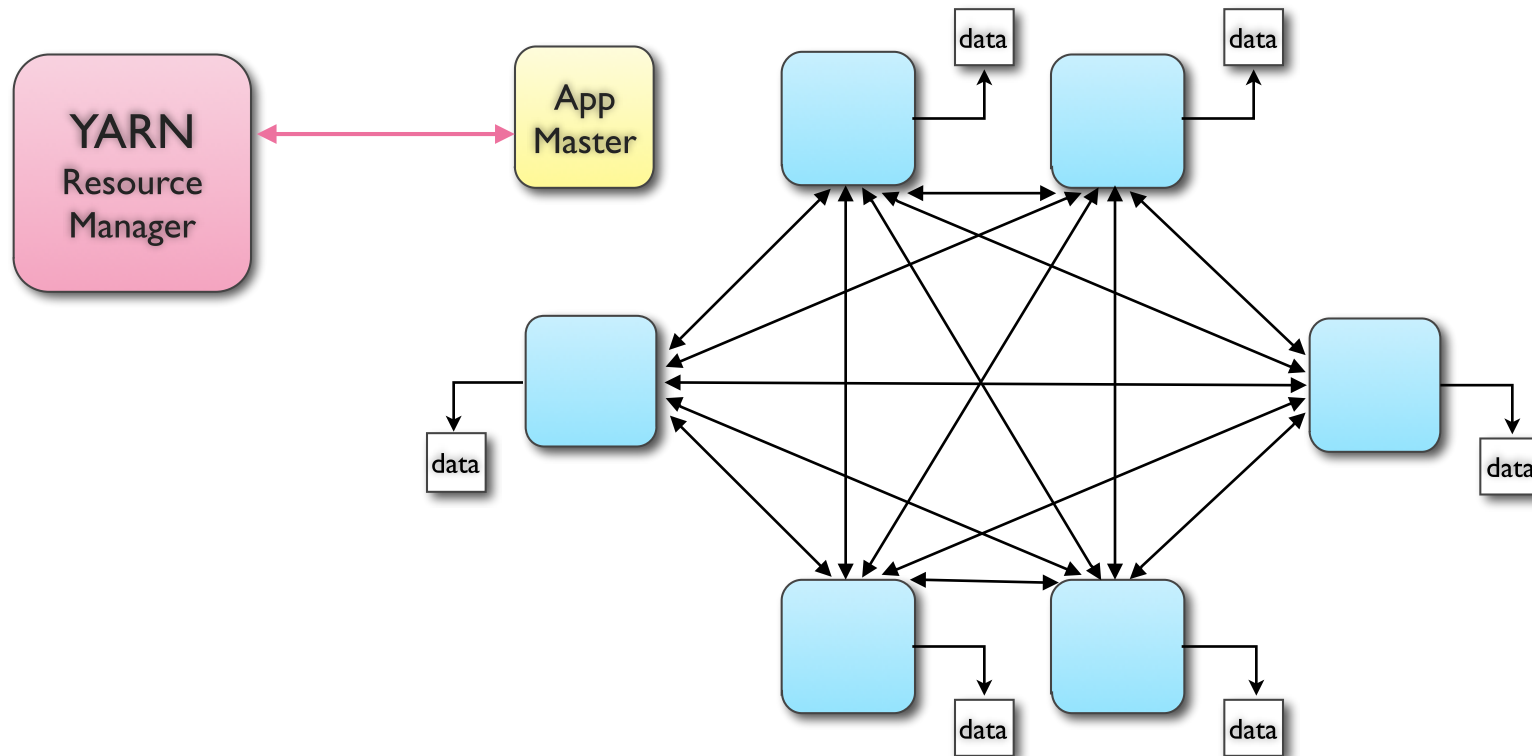
Continuuity Reactor

- Developer-Centric Big Data Application Platform
- Many different types of jobs in Hadoop cluster
  - Real-time stream processing
  - Ad-hoc queries
  - Map/Reduce
  - Web Apps

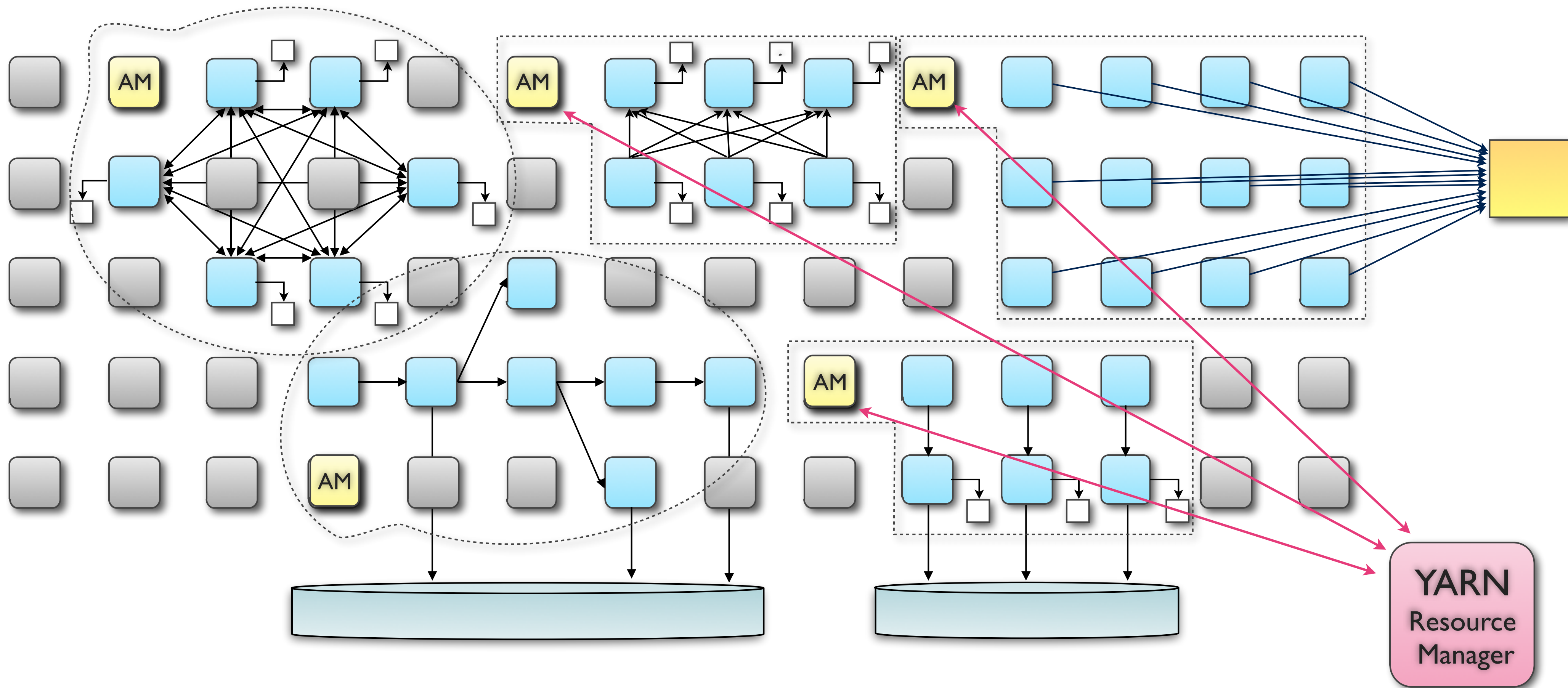
The Answer is YARN

- Resource Manager of Hadoop 2.0
- Separates
  - Resource Management
  - Programming Paradigm
- (Almost) any distributed app in Hadoop cluster
  - Application Master to negotiate resources

A YARN Application

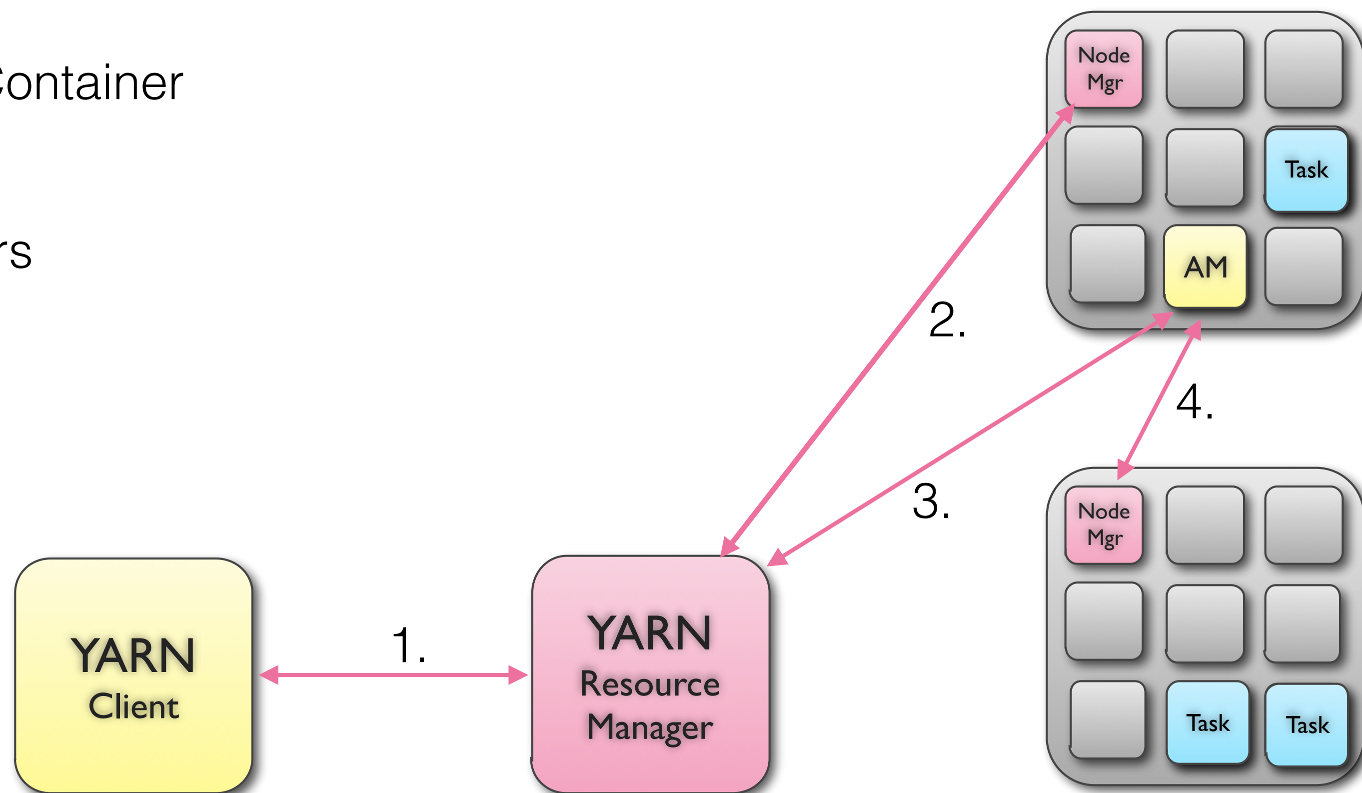


A Multi-Purpose Cluster

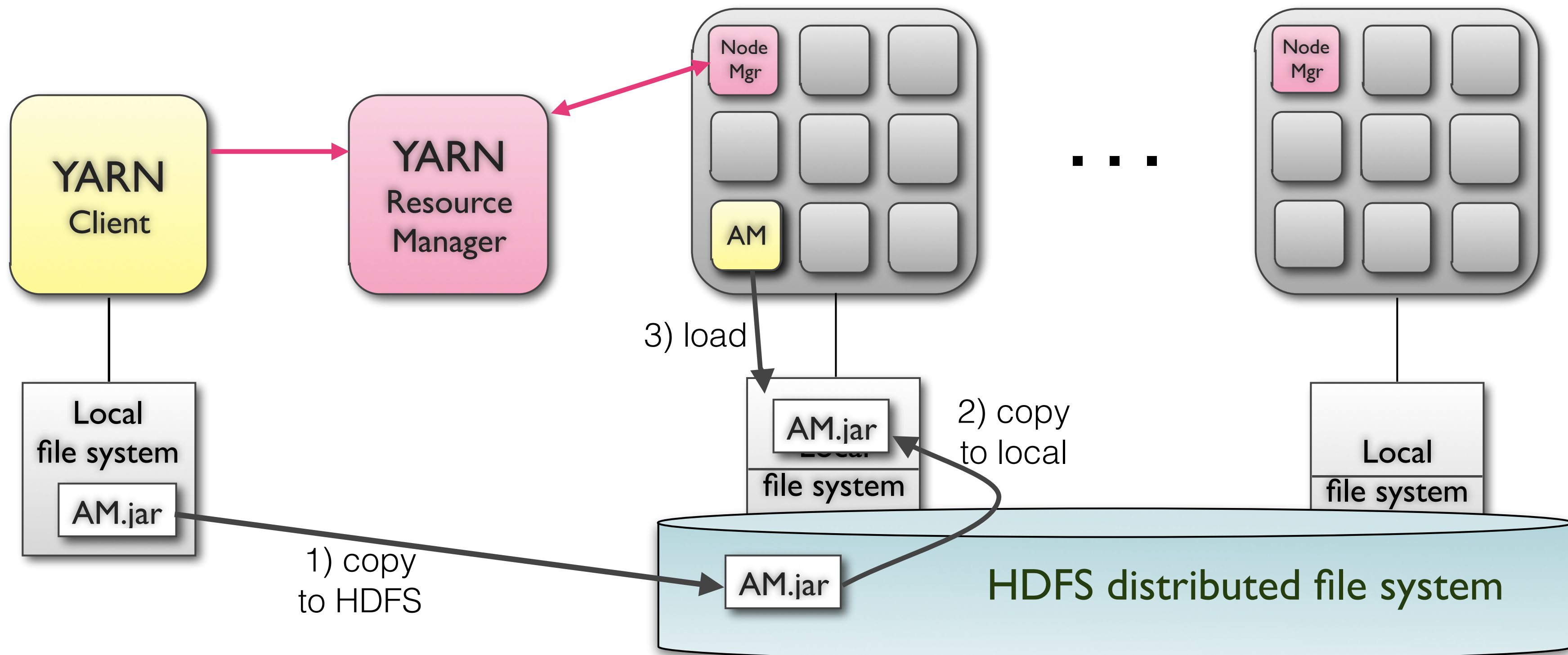


YARN - How it works

1. Submit App Master
2. Start App Master in a Container
3. Request Containers
4. Start Tasks in Containers



Starting the App Master



The YARN Client

1. Connect to the Resource Manager.
2. Request a new application ID.
3. Create a submission context and a container launch context.
4. Define the local resources for the AM.
5. Define the environment for the AM.
6. Define the command to run for the AM.
7. Define the resource limits for the AM.
8. Submit the request to start the app master.

Writing the YARN Client

1. Connect to the Resource Manager:

```
YarnConfiguration yarnConf = new YarnConfiguration(conf);
InetSocketAddress rmAddress =
    NetUtils.createSocketAddr(yarnConf.get(
        YarnConfiguration.RM_ADDRESS,
        YarnConfiguration.DEFAULT_RM_ADDRESS));
LOG.info("Connecting to ResourceManager at " + rmAddress);
configuration rmServerConf = new Configuration(conf);
rmServerConf.setClass(
    YarnConfiguration.YARN_SECURITY_INFO,
    ClientRMSecurityInfo.class, SecurityInfo.class);
ClientRMProtocol resourceManager = ((ClientRMProtocol) rpc.getProxy(
    ClientRMProtocol.class, rmAddress, appsManagerServerConf));
```

Writing the YARN Client

2) Request an application ID:

```
GetNewApplicationRequest request =
    Records.newRecord(GetNewApplicationRequest.class);
GetNewApplicationResponse response =
    resourceManager.getNewApplication(request);
LOG.info("Got new ApplicationId=" + response.getApplicationId());
```

3) Create a submission context and a launch context

```
ApplicationSubmissionContext appContext =
    Records.newRecord(ApplicationSubmissionContext.class);
appContext.setApplicationId(appId);
appContext.setApplicationName(appName);

ContainerLaunchContext amContainer =
    Records.newRecord(ContainerLaunchContext.class);
```

Writing the YARN Client

4. Define the local resources:

```
Map<String, LocalResource> localResources = Maps.newHashMap();
// assume the AM jar is here:
Path jarPath; // <- known path to jar file

// Create a resource with location, time stamp and file length
LocalResource amJarRsrc = Records.newRecord(LocalResource.class);
amJarRsrc.setType(LocalResourceType.FILE);
amJarRsrc.setResource(ConverterUtils.getYarnUrlFromPath(jarPath));
FileStatus jarStatus = fs.getFileStatus(jarPath);
amJarRsrc.setTimestamp(jarStatus.getModificationTime());
amJarRsrc.setSize(jarStatus.getLen());
localResources.put("AppMaster.jar", amJarRsrc);

amContainer.setLocalResources(localResources);
```

Writing the YARN Client

5. Define the environment:

```
// Set up the environment needed for the launch context
Map<String, String> env = new HashMap<String, String>();

// Setup the classpath needed.
// Assuming our classes are available as local resources in the
// working directory, we need to append "." to the path.
String classPathEnv = "$CLASSPATH:./*:";
env.put("CLASSPATH", classPathEnv);

// setup more environment
env.put(...);

amContainer.setEnvironment(env);
```

Writing the YARN Client

6. Define the command to run for the AM:

```
// Construct the command to be executed on the launched container
String command =
    "${JAVA_HOME}" + /bin/java" +
    " MyAppMaster" +
    " arg1 arg2 arg3" +
    " 1>" + ApplicationConstants.LOG_DIR_EXPANSION + "/stdout" +
    " 2>" + ApplicationConstants.LOG_DIR_EXPANSION + "/stderr";

List<String> commands = new ArrayList<String>();
commands.add(command);

// Set the commands into the container spec
amContainer.setCommands(commands);
```

Writing the YARN Client

7. Define the resource limits for the AM:

```
// Define the resource requirements for the container.
// For now, YARN only supports memory constraints.
// If the process takes more memory, it is killed by the framework.
Resource capability = Records.newRecord(Resource.class);
capability.setMemory(amMemory);
amContainer.setResource(capability);

// Set the container launch content into the submission context
appContext.setAMContainerSpec(amContainer);
```

Writing the YARN Client

8. Submit the request to start the app master:

```
// Create the request to send to the Resource Manager
SubmitApplicationRequest appRequest =
    Records.newRecord(SubmitApplicationRequest.class);
appRequest.setApplicationSubmissionContext(appContext);

// Submit the application to the ApplicationsManager
resourceManager.submitApplication(appRequest);
```

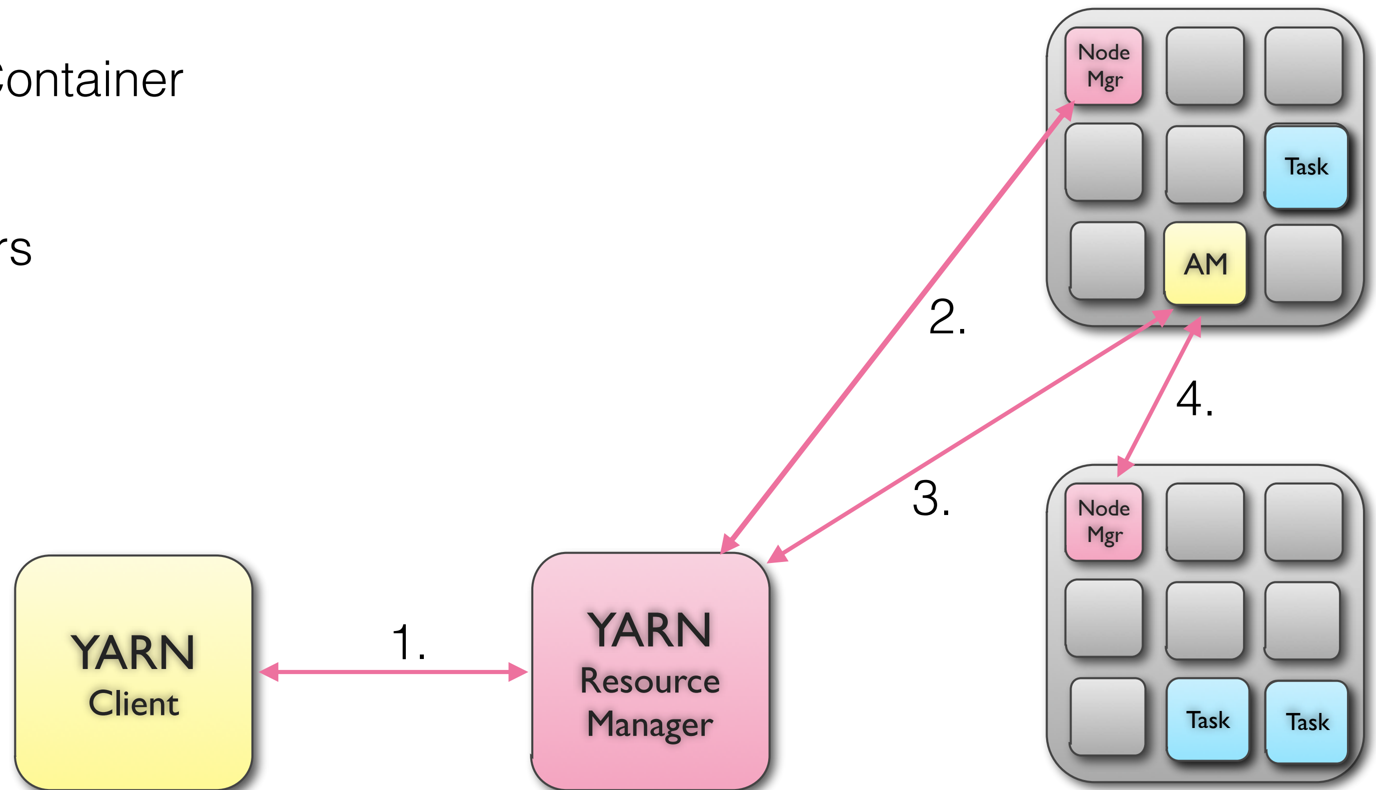
YARN - How it works

1. Submit App Master

2. Start App Master in a Container

3. Request Containers

4. Start Tasks in Containers



YARN is complex

- Three different protocols to learn
  - Client -> RM, AM -> RM, AM -> NM
- Asynchronous protocols
- Full Power at the expense of simplicity
- Duplication of code
  - App masters
  - YARN clients

Can it be easier?

YARN App vs Multi-threaded App

YARN Application	Multi-threaded Java Application
YARN Client	`java` command to launch application with options and arguments
Application Master	`main()` method preparing threads
Task in Container	`Runnable` implementation, each runs in its own Thread

Apache Twill

- Adds simplicity to the power of YARN
  - Java thread-like programming model
- Incubated at Apache Software Foundation (Nov 2013)
  - Current release: 0.2.0-incubating

Hello World

```
public class HelloWorld {
  static Logger LOG = LoggerFactory.getLogger(HelloWorld.class);

  static class HelloWorldRunnable extends AbstractTwillRunnable {
    @Override
    public void run() {
      LOG.info("Hello World");
    }
  }

  public static void main(String[] args) throws Exception {
    YarnConfiguration conf = new YarnConfiguration();
    TwillRunnerService runner = new YarnTwillRunnerService(conf, "localhost:2181");
    runner.startAndWait();

    TwillController controller = runner.prepare(new HelloWorldRunnable())
                                       .start();
    Services.getCompletionFuture(controller).get();
  }
```

Hello World

```
public class HelloWorld {
  static Logger LOG = LoggerFactory.getLogger(HelloWorld.class);

  static class HelloWorldRunnable extends AbstractTwillRunnable {
    @Override
    public void run() {
      LOG.info("Hello World");
    }
  }

  public static void main(String[] args) throws Exception {
    YarnConfiguration conf = new YarnConfiguration();
    TwillRunnerService runner = new YarnTwillRunnerService(conf, "localhost:2181");
    runner.startAndWait();

    TwillController controller = runner.prepare(new HelloWorldRunnable())
                                       .start();
    Services.getCompletionFuture(controller).get();
  }
```

Hello World

```
public class HelloWorld {
  static Logger LOG = LoggerFactory.getLogger(HelloWorld.class);

  static class HelloWorldRunnable extends AbstractTwillRunnable {
    @Override
    public void run() {
      LOG.info("Hello World");
    }
  }

  public static void main(String[] args) throws Exception {
    YarnConfiguration conf = new YarnConfiguration();
    TwillRunnerService runner = new YarnTwillRunnerService(conf, "localhost:2181");
    runner.startAndWait();

    TwillController controller = runner.prepare(new HelloWorldRunnable())
                                       .start();
    Services.getCompletionFuture(controller).get();
  }
```

Hello World

```
public class HelloWorld {
  static Logger LOG = LoggerFactory.getLogger(HelloWorld.class);

  static class HelloWorldRunnable extends AbstractTwillRunnable {
    @Override
    public void run() {
      LOG.info("Hello World");
    }
  }

  public static void main(String[] args) throws Exception {
    YarnConfiguration conf = new YarnConfiguration();
    TwillRunnerService runner = new YarnTwillRunnerService(conf, "localhost:2181");
    runner.startAndWait();

    TwillController controller = runner.prepare(new HelloWorldRunnable())
                                       .start();
    Services.getCompletionFuture(controller).get();
  }
```

Twill is easy.

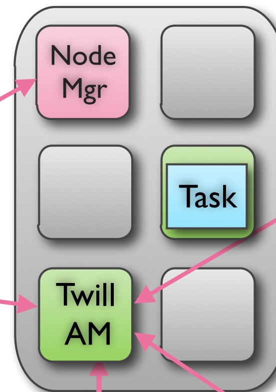
Architecture

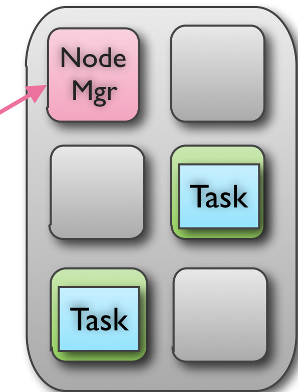
This is the only programming interface you need

1.

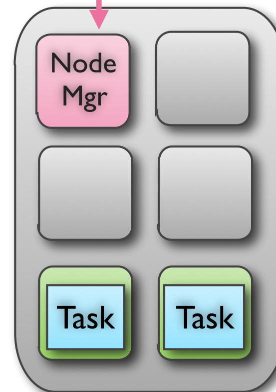
2.

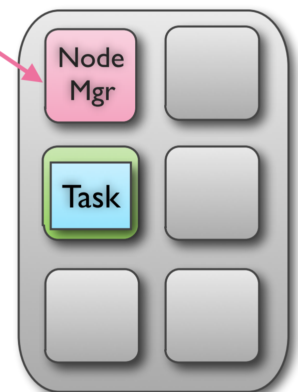
3.





4.





Twill Application

What if my app needs more than one type of task?

Twill Application

What if my app needs more than one type of task?

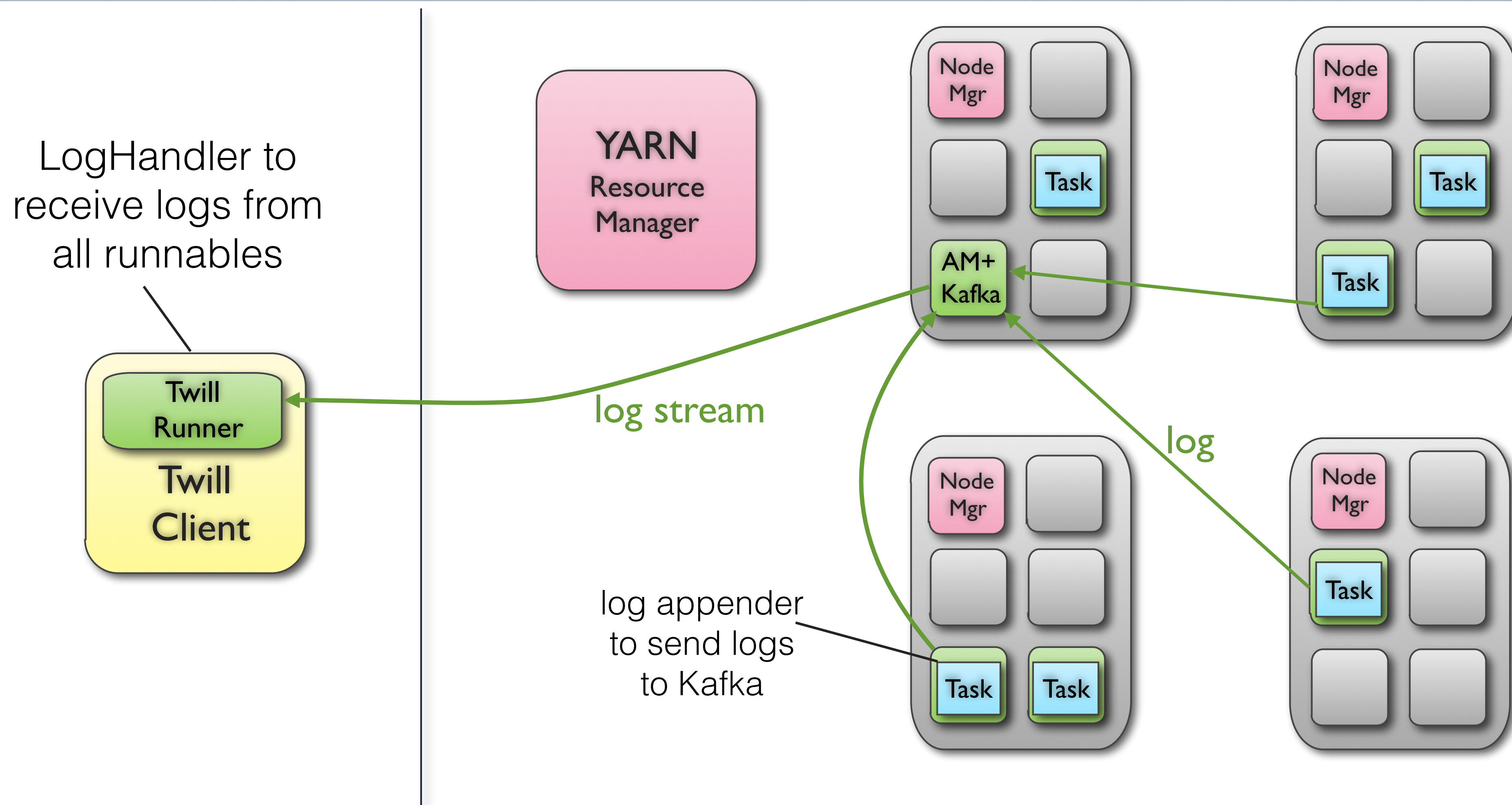
- Define a TwillApplication with multiple TwillRunnables inside:

```
public class MyTwillApplication implements TwillApplication {

  @Override
  public TwillSpecification configure() {
    return TwillSpecification.Builder.with()
      .setName("Search")
      .withRunnable()
        .add("crawler", new CrawlerTwillRunnable()).noLocalFiles()
        .add("indexer", new IndexerTwillRunnable()).noLocalFiles()
      .anyOrder()
      .build();
  }
}
```

Features

- Real-time logging
- Resource report
- State recovery
- Elastic scaling
- Command messages
- Service discovery
- Bundle jar execution

Real-Time Logging



Real-time Logging

```
TwillController controller =
  runner.prepare(new HelloWorldRunnable())
        .addLogHandler(new PrinterLogHandler(
            new PrintWriter(System.out, true)))
        .start();
```

OR

```
controller.addLogHandler(new PrinterLogHandler(
              new PrintWriter(System.out, true)));
```

Resource Report

- Twill Application Master exposes HTTP endpoint
  - Resource information for each container (AM and Runnables)
    - Memory and virtual core
    - Number of live instances for each Runnable
    - Hostname of the container is running on
    - Container ID
- Registered as AM tracking URL

Resource Report

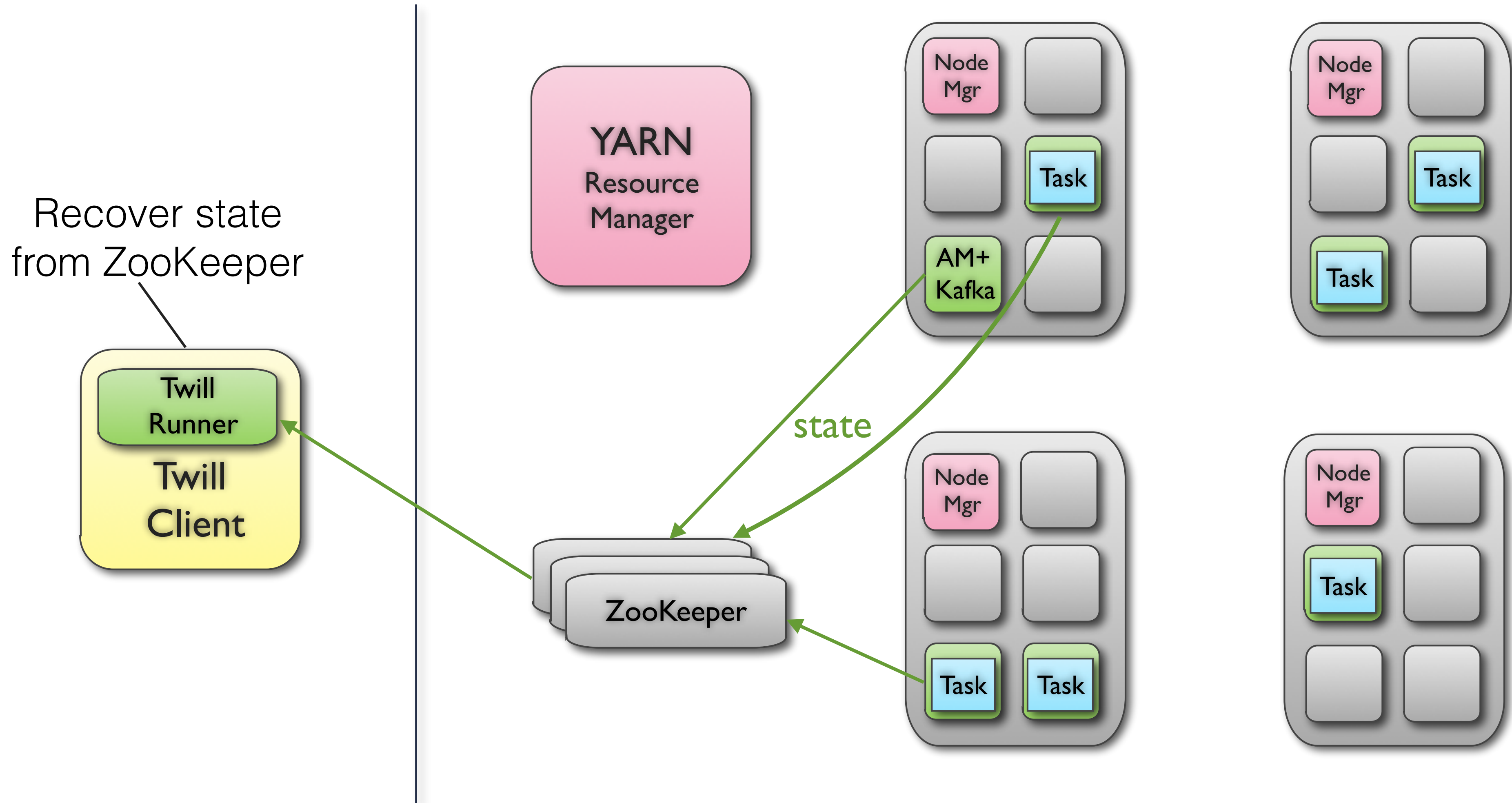
- Programmatic access to resource report.

```
ResourceReport report = controller.getResourceReport();
Collection<TwillRunResources> resources =
            report.getRunnableResource("MyRunnable");
```

State Recovery

- What happens to the Twill app if the client terminates?
  - It keeps running
  - Can a new client take over control?

State Recovery



State Recovery

- All live instances of an application

```
Iterable<TwillController> controllers =
    runner.lookup("HelloWorld");
```

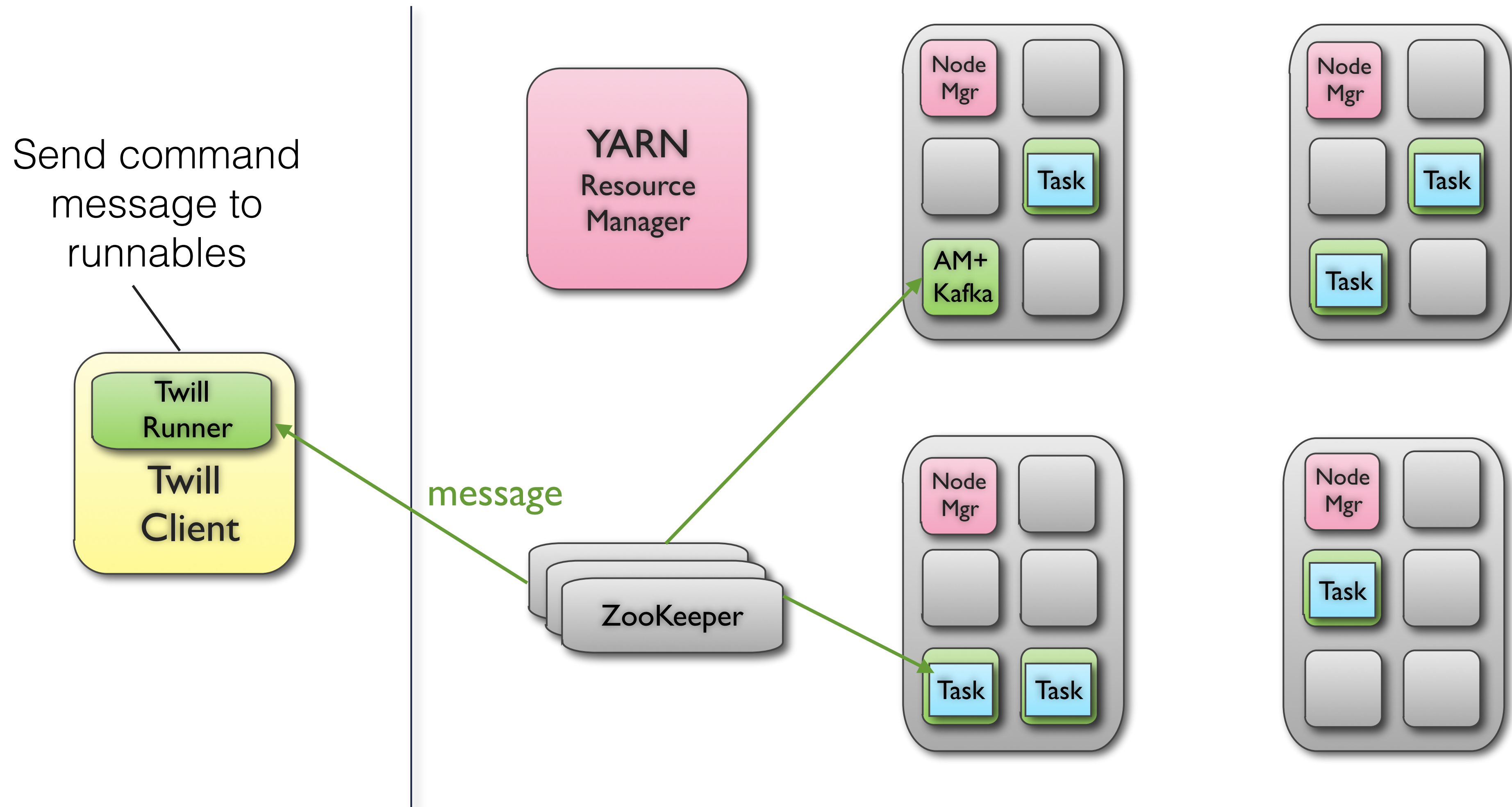
- A particular live instance

```
TwillController controller =
    runner.lookup("HelloWorld",
                  RunIds.fromString("lastRunId"));
```

- All live instances of all applications

```
Iterable<LiveInfo> liveInfos = runner.lookupLive();
```

Command Messages



Command Messages

- Send to all Runnables:

```
ListenableFuture<Command> completion =
  controller.sendCommand(
    Command.Builder.of("gc").build());
```

- Send to the “indexer” Runnable:

```
ListenableFuture<Command> completion =
  controller.sendCommand("indexer",
    Command.Builder.of("flush").build());
```

- The Runnable implementation defines how to handle it:

```
void handleCommand(Command command) throws Exception;
```

Elastic Scaling

- Change the instance count of a live runnable

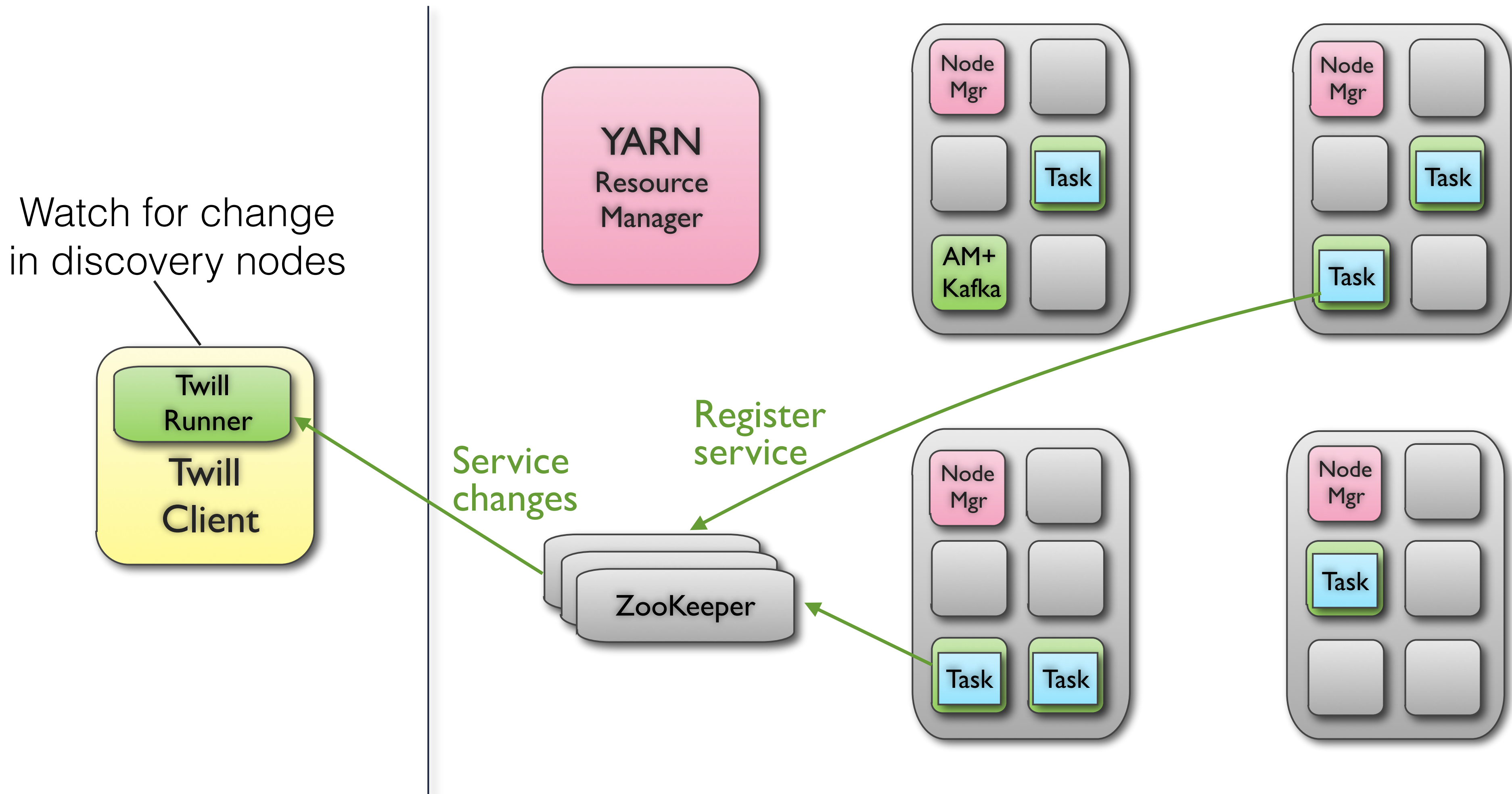
```
ListenableFuture<Integer> completion =
    controller.changeInstances("crawler", 10);

// Wait for change complete
completion.get();
```

- Implemented by a command message to the AM.

Service Discovery

- Running a service in Twill
  - What host/port should a client connect to?

Service Discovery



Service Discovery

- In TwillRunnable, register as a named service:

```
@Override
public void initialize(TwillContext context) {
  // Starts server on random port
  int port = startServer();
  context.announce("service", port);
}
```

- Discover by service name on client side:

```
ServiceDiscovered serviceDiscovered =
          controller.discoverService("service");
```

Bundle Jar Execution

- Different library dependencies than Twill
- Run existing application in YARN

Bundle Jar Execution

- Bundle Jar contains classes and all libraries depended on inside a jar.

```
MyMain.class
MyRecord.class
lib/guava-16.0.1.jar
lib/netty-all-4.0.17.jar
```

- Easy to create with build tools
    - Maven - maven-bundle-plugin
    - Gradle - apply plugin: 'osgi'

Bundle Jar Execution

- Execute with BundledJarRunnable
  - See BundledJarExample in the source tree.

```
java org.apache.twill.example.yarn.BundledJarExample
     <zkConnectStr> <bundleJarPath> <mainClass> <args…>
```

- Successfully run Presto on YARN
  - Non-intrusive, no code modification for Presto
  - Simple maven project to use Presto in embedded way and create Bundle Jar

The Road Ahead

- Scripts for easy life cycle management and scaling
- Distributed coordination within application
- Remote debugging
- Non-Java application
- Suspend and resume application
- Metrics
- Local runner service
- …

Summary

- YARN is powerful
  - Allows applications other than M/R in Hadoop cluster
- YARN is complex
  - Complex protocols, boilerplate code
- Twill makes YARN easy
  - Java Thread-like runnables
  - Add-on features required by many distributed applications
- Productivity Boost
  - Developers can focus on application logic

Thank You

- Twill is Open Source and needs your contributions
  - twill.incubator.apache.org
  - dev@twill.incubator.apache.org

- Continuuity is hiring
  - continuuity.com/careers