Finagle your application to prevent outages using Twitter's Finagle

Presenters: Brian Ko and Mike Pallas, Behr Paint Company

Behr Paint

RPC History

- Corba
- EJB
- Web Services (eg. SOAP)
- JMS
- REST
- ESB
- RPC Servers (eg. Finagle, Akka, etc.)

Finagle

Provides Service instances via **clients**.
Exposes Service instances via **servers**.
Adds behavior, is largely configurable
Retrying, connection pooling, load balancing, rate limiting, monitoring, stats keeping, …
Codecs implement wire protocols.
Manages resources for you.

Finagle

- Most of Finagle is protocol agnostic.
- Codecs for thrift, http, memcache, kestrel, redis, streaming HTTP, generic multiplexer.
- Supports many RPC styles: request/response, streaming, multiplexing.
- Writing new codecs is easy.
- Uses Netty for the event loop.
- Scala and Java parity.

Finagle provides a robust implementation of:

- connection pools, with throttling to avoid TCP connection churn;
- failure detectors, to identify slow or crashed hosts; failover strategies, to direct traffic away from unhealthy hosts;
- load-balancers, including “least-connections” and other strategies;
- and back-pressure techniques, to defend servers against abusive clients and dogpiling.

Additionally, Finagle makes it easier to build and deploy a service that

- publishes standard statistics, logs, and exception reports;
- supports distributed tracing (a la Dapper) across protocols;
- optionally uses ZooKeeper for cluster management; and
- supports common sharding strategies.

Finagle is Protocol Agnostic

- Http
- Streaming Http (Comet)
- Thrift
- Memcached/Kestrel
- MySQL
- More to come

Builders

ClientBuilder produces a Service Instance

```
val client = ClientBuilder()
.name(“httploadtest”)
.codec(Http)
.hosts("host1:80,host2:80,...")
.hostConnectionLimit(10)
.build()
```

ServerBuilder consumes a Service instance

```
val service: Service[HttpReq, HttpResp]
val server = ServerBuilder()
.name(“httpd”)
.codec(Http)
.bindTo(new InetSocketAddress(8080))
.build(service)


server.close() // when done
```

Simple Http Server

Scala

```
val service: Service[HttpRequest, HttpResponse] = new Service[HttpRequest, HttpResponse] {
    def apply(request: HttpRequest) = Future(new DefaultHttpResponse(HTTP_1_1, OK))
}

val address: SocketAddress = new InetSocketAddress(10000)

val server: Server = ServerBuilder()
 .codec(Http())
.bindTo(address)
.name("HttpServer")
.build(service)
```

Java

```
Service<HttpRequest, HttpResponse> service = new Service<HttpRequest, HttpResponse>() {
   public Future<HttpResponse> apply(HttpRequest request) {
    return Future.value(
        new DefaultHttpResponse(HttpVersion.HTTP_1_1, HttpResponseStatus.OK));
    }
};
ServerBuilder.safeBuild(service, ServerBuilder.get()
.codec(Http())
.name("HttpServer")
.bindTo(new InetSocketAddress("localhost", 10000)));
```

Simple Http Client

Scala

```
val client: Service[HttpRequest, HttpResponse] = ClientBuilder()
.codec(Http())
.hosts(address)
.hostConnectionLimit(5)
.build()

val request: HttpRequest = new DefaultHttpRequest(HTTP_1_1, GET, "/")
val responseFuture: Future[HttpResponse] = client(request)
onSuccess {
    response => println("Received response: " + response)
}
```

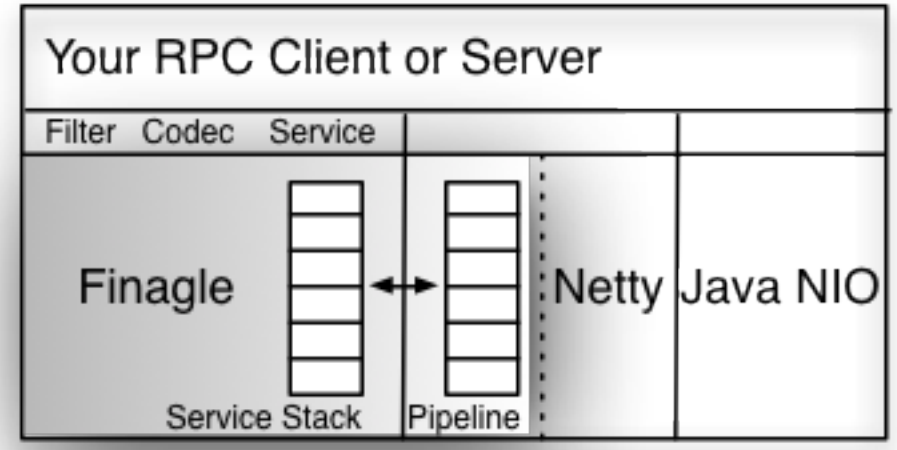
Java

```
Service<HttpRequest, HttpResponse> client = ClientBuilder.safeBuild(ClientBuilder.get()
 .codec(Http())
.hosts("localhost:10000")
.hostConnectionLimit(5));

HttpRequest request = new DefaultHttpRequest(HttpVersion.HTTP_1_1, HttpMethod.GET, "/");
client.apply(request).addEventListener(new FutureEventListener<HttpResponse>() {
    public void onSuccess(HttpResponse response) {
        System.out.println("received response: " + response);
    }
    public void onFailure(Throwable cause) {
        System.out.println("failed with cause: " + cause);
    }
});
```

Finagle extends the stream-oriented Netty model to provide asynchronous requests and responses for remote procedure calls (RPC). Internally, Finagle manages a service stack to track outstanding requests, responses, and the events related to them. Finagle uses a Netty pipeline to manage connections between the streams underlying request and response messages. The following diagram shows the relationship between your RPC client or server, Finagle, Netty, and Java libraries:

Relationship between Finagle, Netty, and NIO:



Finagle objects are the building blocks of RPC clients and servers:

- Future objects enable asynchronous operations required by a service
- Service objects perform the work associated with a remote procedure call
- Filter objects enable you to transform data or act on messages before or after the data or messages are processed by a service
- Codec objects decode messages in a specific protocol before they are handled by a service and encode messages before they are transported to a client or server.

Future Objects

In Finagle, Future objects are the unifying abstraction for all asynchronous computation. A Future represents a computation that has not yet completed, which can either succeed or fail. The two most basic ways to use a Future are to:

- block and wait for the computation to return
- register a callback to be invoked when the computation eventually succeeds or fails

Rather than passing a callback to a dispatch routine, a dispatch routine returns a Future which is a promise for the result in the future.

Simple Futures

```
val f: Future[String]

//wait indefinitely
val result = f.get()

//wait 1 second
val result = f.get(1.second)

// In this simple example, we create a block for a request to complete.
// Since there is not timeout specified, it will wait forever.
val request: HttpRequest = new DefaultHttpRequest(HTTP_1_1, GET, "/")
val responseFuture: Future[HttpResponse] = client(request)

 // In this example, the value of responseFuture will not be available
// until the scheduled job assigns it a value.
val responseFuture: Future[String] = executor.schedule(job)

// A promise is a writeable Future
val p = new Promise[Int]
```

Futures with timeouts

In this example, it waits 1 second for the response, but you don’t know if it was successful or timed out.

```
val request: HttpRequest = new DefaultHttpRequest(HTTP_1_1, GET, "/")
val responseFuture: Future[HttpResponse] = client(request)
println(responseFuture(1.second))
```

A timeout Filter

```
class TimeoutFilter[Req, Res](
      timeout: Duration, util.Timer)
    extends Filter[Req, Res, Req, Res]
{
    def apply
      request: Req, service: Service(Req, Res)
    ): Future[Res] = {
       service(request).timeout(timer, timeout) {
         Throw(new TimedoutRequestException)
       }
    }
 }
```

A timeout Filter can be implemented on the Client side or on the Server side and is completely independent of protocol.

Promise with Error Handling

```
def make() = {
...
val promise = new Promise[Service[Req, Rep]]
...
{ case Ok(myObject) =>
...
promise() = myConfiguredObject
   case Error(cause) =>
      promise() = Throw(new ... Exception(cause))
    case Cancelled =>
      promise() = Throw(new WriteException(new ...Exception))
    }
    promise
}
```

Codec

A Codec object encodes and decodes wire protocols, such as HTTP. You can use Finagle-provided Codec objects for encoding and decoding the Thrift, HTTP, memcache, Kestrel, HTTP chunked streaming (ala Twitter Streaming) protocols. You can also extend the CodecFactory class to implement encoding and decoding of other protocols.

Create you own Codec

```
class StringCodec extends CodecFactory[String, String] {
    def server = Function.const {
        new Codec[String, String] {
            def pipelineFactory = new ChannelPipelineFactory {
              def getPipeline = {
                  val pipeline = Channels.pipeline()
                  pipeline.addLast("line",
                      new DelimiterBasedFrameDecoder(
                          100, Delimiters.lineDelimiter: _*))
                  pipeline.addLast("stringDecoder",
                          new StringDecoder(CharsetUtil.UTF_8))
                  pipeline.addLast("stringEncoder",
                      new StringEncoder(CharsetUtil.UTF_8))
                  pipeline
            }
          }
        }
    }
…
```

cont.

```
    def client = Function.const {
    new Codec[String, String] {
      def pipelineFactory = new ChannelPipelineFactory {
        def getPipeline = {
          val pipeline = Channels.pipeline()
          pipeline.addLast("stringEncode",
              new StringEncoder(CharsetUtil.UTF_8))
          pipeline.addLast("stringDecode",
             new StringDecoder(CharsetUtil.UTF_8))
          pipeline
          }
      }
    }
}
}
```

Callbacks

```
val f: Future[String]
f onSuccess { s =>
      println(“got string “+s)
} onFailure { exc =>
     exc.printStackTrace()
}
```

Filter Objects

It is useful to isolate distinct phases of your application into a pipeline. For example, you may need to handle exceptions, authorization, and other phases before your service responds to a request. A Filter provides an easy way to decouple the protocol handling code from the implementation of the business rules. A Filter wraps a Service and, potentially, converts the input and output types of the service to other types.

A SimpleFilter is a kind of Filter that does not convert the request and response types.

Simple Filter Example

```
class Authorize extends SimpleFilter[HttpRequest, HttpResponse] {
    def apply(request: HttpRequest, continue: Service[HttpRequest, HttpResponse]) = {
        if ("shared secret" == request.getHeader("Authorization")) {
            continue(request)
        } else {
            Future.exception(new IllegalArgumentException("You don't know the secret"))
        }
    }
}
```

Filters to transform requests and responses

```
class RequireAuthentication(val p: ...)
    extends Filter[Request, HttpResponse, AuthenticatedRequest, HttpResponse]
    {
    def apply(request: Request, service: Service[AuthenticatedRequest, HttpResponse]) = {
        p.authenticate(request) flatMap {
          case AuthResult(AuthResultCode.OK, Some(passport: OAuthPassport), _, _) =>
            service(AuthenticatedRequest(request, passport))
         case AuthResult(AuthResultCode.OK, Some(passport: SessionPassport), _, _) =>
            service(AuthenticatedRequest(request, passport))
         case ar: AuthResult =>
           Trace.record("Authentication failed with " + ar)
           Future.exception(new RequestUnauthenticated(ar.resultCode))
         }
     }
}
```

Sequential Composition

A flatMap is the most important Combinator. The method signature tells the story: given the successful value of the future f must provide the next Future. The result of this operation is another Future that is complete only when both of these futures have completed. If either Future fails, the given Future will also fail. This implicit interleaving of errors allow us to handle errors only in those places where they are semantically significant. flatMap is the standard name for the combinator with these semantics.

```
def auth(token: String): Future[Long]
def getUser(id:Long)   : Future[User]
def getUser(token: String): Future[User]
```

```
def getUser(token: String): Future[User] =
   auth(token) flatMap { id =>
       getUser(id)
 }
```

Concurrent Composition

Concurrent combinators are available to convert a sequence of Futures into a Future of sequence. Three of the common ones are collect, join, and select.

collect

collect is the most straightforward one: given a set of Futures of the same type, we are given a Future of a sequence of values of that type. This future is complete when all of the underlying futures have completed, or when any of them have failed.

```
object Future {
  …
  def collect[A](fs: Seq[Future[A]]):
}
```

join

join takes a sequence of Futures whose types may be mixed, yielding a Future[Unit] that is completely when all of the underlying futures are (or fails if any of them do). This is useful for indicating the completion of a set of heterogeneous operations.

```
object Future {
    …
   def join(fs: Seq[Future[_]]): Future[Unit]
 }
```

select

select returns a Future that is complete when the first of the given Futures complete, together with the remaining uncompleted futures.

```
object Future {
   …
   def select(fs: Seq[Future[A]]) : Future[(Try[A], Seq[Future[A]])]
}
```

Combining Combinators

Combining combinators allows for powerful and concise expression of operations typical of network services. This hypothetical code performs rate limiting (in order to maintain a local rate limit cache) concurrently with dispatching a request on behalf of the user to the backend:

```
def serve(request: Request): Future[Response] = {
    val userLimit: Future[(User, Boolean)] =
        for {
           user <- auth(request)
           limited <- isLimit(user)
        } yield (user, limited)
        val done =
           dispatch(request) join userLimit
        done flatMap { case (rep, (usr, lim)) =>
        if (lim) {
            updateLocalRateLimitCache(usr)
            Future.exception(new Exception("rate limited"))
        } else {
            Future.value(rep)
        }
   }
}
```

Failover Detection and Clusters

An abstraction called a Cluster can be used to register Clients and Servers. There can be several implementations, the simplest is just a comma separated lists of hosts. Failure detection will mark hosts as dead.

A Directory is used to do dynamic registration on the server side and dynamic discovery on the client side.

Ajax Example

```
$.post(url,
     JSON.stringify(myformsData),
       function(msg); {
//add handle for ok response
           alert(msg);
       },
       "json");
```

Questions/Examples

http://github.com/twitter/util

http://github.com/twitter/finagle

http://twitter.github.com/finagle

Finaglers@googlegroups.com

Example of a 200 line search engine using Finagle:

http://twitter.github.com/scala_school/searchbird.html