YOUR SERVER AS A FUNCTION

Marius Eriksen, Twitter Inc.
@marius—PLOS’13—11/3/2013
AGENDA

Motivation

Futures

Services & Filters

Experiences
SERVER PROGRAMMING IN 2013

Networking everywhere: RPC, message passing, streaming, etc.

Concurrency is paramount.

Open source: failure-, protocol-, quality-, api- diversity.

Partial failures, latency variance (esp. tail).
Unified programming model; a small number of orthogonal & composable widgets.

Concurrency is first class.

Emphasize functional programming, value transformations. Maintain (some) separation of effects.
FUTURES

A Future is a placeholder for a result that may not yet exist. We say that

```scala
val f: Future[Int]
```

is an integer-valued future. (Futures are themselves first-class values.)

Futures occupy one of three states:

- **Empty**: no value is yet available
- **Succeeded**: a value is available
- **Failed**: producing a value failed; an exception is available
WHENCE FUTURES?

Any sort of **asynchronous operation**, for example:
- Pending RPC
- Disk I/O
- Pending timeout

Generally we are agnostic to the source of the operation: consumer and producer are **decoupled**.

A promise is a writable future; they are used rarely in an application.
FUTURES AS CONTAINERS

It is possible to view futures as an at-most one element container. As you’d expect, you can transform them.

`Future[T].map[U](f: T => U): Future[U]`

Converts a `Future[T]` to a `Future[U]` by applying `f` to the eventual value. (Exceptions are passed through.)

`Future[T].filter(p: T => Boolean): Future[T]`

Converts a successful `Future[T]` into a failed `Future[T]` unless the predicate `p` is met.
Futures may also be defined as a function of other futures. We call this dependent composition.

```scala
Future[T].flatMap[U](
  f: T => Future[U]): Future[U]
```


The returned `Future[U]` fails if the outer `Future[T]` fails.
def auth(id: Int, pw: String): Future[User]
def get(u: User): Future[UserData]

def getAndAuth(id: Int, pw: String): Future[UserData] = auth(id, pw) flatMap { u => get(u) }
Futures recover from errors by another form of dependent composition.

```scala
Future[T].rescue(
  PartialFunction[Throwable, Future[T]]
)
```

Like `flatMap`, but operates over exceptional futures.
val f = auth(id, pw) rescue {
  case Timeout => auth(id, pw)
}

(This amounts to a single retry.)
MULTIPLE DEPENDENT COMPOSITION

\[
\text{Future.collect}[T](fs: \text{Seq}[\text{Future}[T]]) \quad : \quad \text{Future}[\text{Seq}[T]]
\]

Waits for all futures to succeed, returning the sequence of returned values.

The returned future fails should any constituent future fail.
AN UNLIKELY, BUT NEVERTHELESS VALID* WEB CRAWLER

def fetch(url: String): Future[Buf]
def parse(buf: Buf): Seq[String]

def crawl(url: String): Future[Seq[Buf]] =
  fetch(url) flatMap { buf =>
    val ls =
      Future.collect(parse(buf) map crawl)
    ls map(_.flatten)
  }

* For DAGs.
FUNCTIONAL STYLE

We phrase our problems as a set of transformations of immutable values. Error handling, dependent composition, and the availability of recursion makes this very powerful.

We emphasize “what” over “how” — the meaning of the computation is specified independently of its execution details. Operational semantics of future operations are defined in a library.

We say that semantics are liberated from “mechanics”. This is good separation of concerns.
A service is a kind of asynchronous function.

trait Service[Req, Rep]
  extends (Req => Future[Rep])

val http:   Service[HttpReq, HttpRep]
val redis:  Service[RedisCmd, RedisRep]
val thrift: Service[TFrame, TFrame]
SERVICES ARE SYMMETRIC

// Client:
val http = Http.newService(..)

// Server:
Http.serve(..,
    new Service[HttpReq, HttpRep] {
        def apply(..) = ..
    }
)

// A proxy:
Http.serve(.., Http.newService(..))
Services represent logical endpoints; filters embody service agnostic behavior such as:

- Timeouts
- Retries
- Statistics
- Authentication
- Logging
trait Filter[ReqIn, ReqOut, RepIn, RepOut] extends ((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
class TimeoutFilter[Req, Rep](to: Duration) extends Filter[Req, Rep, Req, Rep] {
    def apply(req: Req, svc: Service[Req, Rep]) = svc(req).within(to)
}
class AuthFilter extends Filter[HttpReq, AuthHttpReq, HttpReq, HttpRep]
{
  def apply(
    req: HttpReq,
    svc: Service[AuthHttpReq, HttpRep]) =
  auth(req) match {
    case Ok(authreq) => svc(authreq)
    case Failed(exc) => Future.exception(exc)
  }
}
val timeout = new TimeoutFilter(1.second)
val auth = new AuthFilter
val authAndTimeout = auth andThen timeout
val service: Service[..] = ..
val authAndTimeoutService = authAndTimeout andThen service
REAL WORLD FILTERS

recordHandleTime andThen
traceRequest andThen
collectJvmStats andThen
parseRequest andThen
logRequest andThen
recordClientStats andThen
sanitize andThen
respondToHealthCheck andThen
applyTrafficControl andThen
virtualHostServer
A. Backup request filter

class BackupRequestFilter[Req, Rep] {
  quantile: Int,
  range: Duration,
  timer: Timer,
  statsReceiver: StatsReceiver,
  history: Duration,
  stopwatch: Stopwatch = Stopwatch
} extends SimpleFilter[Req, Rep] {
  require(quantile > 0 & quantile < 100)
  require(range < 1.hour)

  private[this] val histo = new LatencyHistogram(range, history)
  private[this] def cutoff() = histo.quantile(quantile)

  private[this] val timeouts = statsReceiver.counter("timeouts")
  private[this] val won = statsReceiver.counter("won")
  private[this] val lost = statsReceiver.counter("lost")
  private[this] val cutoffGauge =
    statsReceiver.addGauge("cutoff_ms") { cutoff().inMilliseconds.toFloat }

  def apply(req: Req, service: Service[Req, Rep]): Future[Rep] = {
    val elapsed = stopwatch.start()
    val howlong = cutoff()
    val backup = if (howlong == Duration.Zero) Future.never else {
      timer.doLater(howlong) {
        timeouts.incr()
        service(req)
      }
    }
    try {
      val orig = service(req)
      Future.select(Seq(orig, backup)) flatMap {
        case (Return(res), Seq(other)) =>
          if (other eq orig) lost.incr() else {
            won.incr()
            histo.add(elapsed())
          }
          other.raise(BackupRequestLost)
        case (Throw(_), Seq(other)) => other
      }
    } catch {
      case ThrownReason(e) =>
    }
  }
}
def apply(req: Req, service: Service[Req, Rep]): Future[Rep] = {
    val elapsed = stopwatch.start()
    val howlong = cutoff()
    val backup = if (howlong == Duration.Zero) Future.never else {
        timer.doLater(howlong) {
            timeouts.incr()
            service(req)
        } flatten
    }
    val orig = service(req)

    Future.select(Seq(orig, backup)) flatMap {
        case (Return(res), Seq(backup)) =>
            if (backup eq orig) lost.incr() else {
                won.incr()
                histo.add(elapsed())
            }

        other.raise(BackupRequestLost)
        Future.value(res)
        case (Throw(_, Seq(other))) => other
    }
}
FUTURES, SERVICES, & FILTERS

In combination, these form a sort of orthogonal basis on which we build our server software.

The style of programming encourages good modularity, separation of concerns.

Most of our systems are phrased as big future transformers.
HELLO, IT’S REALITY CALLING

As presented, futures are pure read-only constructs — they form a DAG where consumers are entirely separated from producers.

Inevitably, the real world is not so clean.

Example: a producer should know when a value is no longer needed to avoid needless work, queueing, and tying up resources.
INTERRUPTS

Idea: allow a consumer to **interrupt** a future. Producers optionally handle it. Preserves single-producer semantics.

```scala
val p = new Promise[T]
p setInterruptHandler {
  case exc =>
    if (p.updateIfEmpty(Throw(exc)))
      tearDown()
}

val f: Future[T]
f.raise(new TimedOutException)
```
INTERRUPTS

This breaks the clean model somewhat; compromises composability.

What happens when we interrupt a `Future.collect`? What if we cache futures?

In practice, we haven’t run into issues with this approach.

Probably due to the fact that interrupt actions usually result in some sort of failure, which anyway has to be handled.
CONCLUSION

Futures, services, & filters form an orthogonal basis for writing server software.

This strategy has paid off: picking a few composable abstractions has led to the emergence of small, reusable components: a “software tools” approach to modern server software.

Separating semantics from “mechanics” has been greatly beneficial.
Everything I talked about is open source.

http://github.com/twitter/util
http://github.com/twitter/finagle
THANK YOU. QUESTIONS?

marius@twitter.com
http://twitter.com/marius