



# Designing Actor Systems

Principles of Functional Programming

Roland Kuhn

## Starting Out with the Design

Imagine giving the task to a group of people, dividing it up.

Consider the group to be of very large size.

Start with how people with different tasks will talk with each other.

Consider these “people” to be easily replaceable<sup>1</sup>.

Draw a diagram with how the task will be split up, including communication lines.

---

<sup>1</sup>This is where our abstract people differ from real people.

## Example: the Link Checker

Write an actor system which given a URL will recursively download the content, extract links and follow them, bounded by a maximum depth; all links encountered shall be returned.

## Plan of Action

- ▶ Write web client which turns a URL into a HTTP body asynchronously.  
We will be using "com.ning" % "async-http-client" % "1.7.19"
- ▶ Write a Getter actor for processing the body.
- ▶ Write a Controller which spawns Getters for all links encountered.
- ▶ Write a Receptionist managing one Controller per request.

## The Web Client (1)

Let us start simple:

```
val client = new AsyncHttpClient
def get(url: String): String = {
  val response = client.prepareGet(url).execute().get
  if (response.getStatusCode < 400)
    response.getResponseBodyExcerpt(131072)
  else throw BadStatus(response.getStatusCode)
}
```

## The Web Client (1)

Let us start simple:

```
val client = new AsyncHttpClient
def get(url: String): String = {
  val response = client.prepareGet(url).execute().get
  if (response.getStatusCode < 400)
    response.getResponseBodyExcerpt(131072)
  else throw BadStatus(response.getStatusCode)
}
```

Blocks the calling actor until the web server has replied:

- ▶ actor is deaf to other requests, e.g. cancellation does not work
- ▶ wastes one thread—a finite resource

## The Web Client (2)

```
private val client = new AsyncHttpClient
def get(url: String)(implicit exec: Executor): Future[String] = {
  val f = client.prepareGet(url).execute();
  val p = Promise[String]()
  f.addListener(new Runnable {
    def run = {
      val response = f.get
      if (response.getStatusCode < 400)
        p.success(response.getResponseBodyExcerpt(131072))
      else p.failure(BadStatus(response.getStatusCode))
    }
  }, exec)
  p.future
}
```

## What we learned so far

- ▶ A reactive application is non-blocking & event-driven top to bottom.

## Finding Links

```
// using "org.jsoup" % "jsoup" % "1.8.1"
import org.jsoup.Jsoup
import scala.collection.JavaConverters._

def findLinks(body: String): Iterator[String] = {
  val document = Jsoup.parse(body, url)
  val links = document.select("a[href]")
  for {
    link <- links.iterator().asScala
  } yield link.absUrl("href")
}
```

## The Getter Actor (1)

```
class Getter(url: String, depth: Int) extends Actor {  
  
  implicit val exec = context.dispatcher  
  
  val future = WebClient.get(url)  
  future onComplete {  
    case Success(body) => self ! body  
    case Failure(err)  => self ! Status.Failure(err)  
  }  
  
  ...  
}
```

## The Getter Actor (2)

```
class Getter(url: String, depth: Int) extends Actor {  
  
    implicit val exec = context.dispatcher  
  
    val future = WebClient.get(url)  
    future.pipeTo(self)  
  
    ...  
}
```

## The Getter Actor (3)

```
class Getter(url: String, depth: Int) extends Actor {  
  
    implicit val exec = context.dispatcher  
  
    WebClient get url pipeTo self  
  
    ...  
}
```

## The Getter Actor (4)

```
class Getter(url: String, depth: Int) extends Actor {  
  ...  
  def receive = {  
    case body: String =>  
      for (link <- findLinks(body))  
        context.parent ! Controller.Check(link, depth)  
      stop()  
    case _: Status.Failure => stop()  
  }  
  def stop(): Unit = {  
    context.parent ! Done  
    context.stop(self)  
  }  
}
```

## What we learned so far

- ▶ A reactive application is non-blocking & event-driven top to bottom.
- ▶ Actors are run by a dispatcher—potentially shared—which can also run Futures.

## Actor-Based Logging

- ▶ Logging includes IO which can block indefinitely
- ▶ Akka's logging passes that task to dedicated actors
- ▶ supports ActorSystem-wide levels of debug, info, warning, error
- ▶ set level using setting akka.loglevel=DEBUG (for example)

```
class A extends Actor with ActorLogging {  
  def receive = {  
    case msg => log.debug("received message: {}", msg)  
  }  
}
```

## The Controller

```
class Controller extends Actor with ActorLogging {
  var cache = Set.empty[String]
  var children = Set.empty[ActorRef]
  def receive = {
    case Check(url, depth) =>
      log.debug("{} checking {}", depth, url)
      if (!cache(url) && depth > 0)
        children += context.actorOf(Props(new Getter(url, depth - 1)))
      cache += url
    case Getter.Done =>
      children -= sender()
      if (children.isEmpty) context.parent ! Result(cache)
  }
}
```

## What we learned so far

- ▶ A reactive application is non-blocking & event-driven top to bottom.
- ▶ Actors are run by a dispatcher—potentially shared—which can also run Futures.
- ▶ Prefer immutable data structures, since they can be shared.

## Handling Timeouts

```
import scala.concurrent.duration._

class Controller extends Actor with ActorLogging {
  context.setReceiveTimeout(10.seconds)
  ...
  def receive = {
    case Check(...)    => ...
    case Getter.Done    => ...
    case ReceiveTimeout => children foreach (_ ! Getter.Abort)
  }
}
```

The receive timeout is reset by every received message.

## Handling Abort in the Getter

```
class Getter(url: String, depth: Int) extends Actor {  
  ...  
  def receive = {  
    case body: String =>  
      for (link <- findLinks(body)) ...  
      stop()  
    case _: Status.Failure => stop()  
    case Abort              => stop()  
  }  
  def stop(): Unit = {  
    context.parent ! Done  
    context.stop(self)  
  }  
}
```

## The Scheduler

Akka includes a timer service optimized for high volume, short durations and frequent cancellation.

```
trait Scheduler {  
  def scheduleOnce(delay: FiniteDuration, target: ActorRef, msg: Any)  
    (implicit ec: ExecutionContext): Cancellable  
  
  def scheduleOnce(delay: FiniteDuration)(block: => Unit)  
    (implicit ec: ExecutionContext): Cancellable  
  
  def scheduleOnce(delay: FiniteDuration, run: Runnable)  
    (implicit ec: ExecutionContext): Cancellable  
  
  ... // the same for repeating timers  
}
```

## Adding an Overall Timeout (1)

```
class Controller extends Actor with ActorLogging {  
  import context.dispatcher  
  var children = Set.empty[ActorRef]  
  context.system.scheduler.scheduleOnce(10.seconds) {  
    children foreach (_ ! Getter.Abort)  
  }  
  ...  
}
```

## Adding an Overall Timeout (1)

```
class Controller extends Actor with ActorLogging {  
  import context.dispatcher  
  var children = Set.empty[ActorRef]  
  context.system.scheduler.scheduleOnce(10.seconds) {  
    children foreach (_ ! Getter.Abort)  
  } ... }
```

**Question:** What is the problem with this code?

- it does not compile
- it is not thread-safe
- the scheduled code will not run

## Adding an Overall Timeout (1)

```
class Controller extends Actor with ActorLogging {  
  import context.dispatcher  
  var children = Set.empty[ActorRef]  
  context.system.scheduler.scheduleOnce(10.seconds) {  
    children foreach (_ ! Getter.Abort)  
  }  
  ...  
}
```

Accessing an actor's state from outside its execution breaks encapsulation.

## Adding an Overall Timeout (2)

```
class Controller extends Actor with ActorLogging {
  import context.dispatcher
  var children = Set.empty[ActorRef]
  context.system.scheduler.scheduleOnce(10.seconds, self, Timeout)
  ...
  def receive = {
    ...
    case Timeout => children foreach (_ ! Getter.Abort)
  }
}
```

## How Actors and Futures Interact (1)

Future composition methods invite closing over the actor's state:

```
class Cache extends Actor {  
  var cache = Map.empty[String, String]  
  def receive = {  
    case Get(url) =>  
      if (cache contains url) sender() ! cache(url)  
      else  
        WebClient get url foreach { body =>  
          cache += url -> body  
          sender() ! body  
        }  
  }  
}
```

## How Actors and Futures Interact (2)

```
class Cache extends Actor {  
  var cache = Map.empty[String, String]  
  def receive = {  
    case Get(url) =>  
      if (cache contains url) sender() ! cache(url)  
      else  
        WebClient get url map (Result(sender(), url, _)) pipeTo self  
    case Result(client, url, body) =>  
      cache += url -> body  
      client ! body  
  }  
}
```

## How Actors and Futures Interact (3)

```
class Cache extends Actor {
  var cache = Map.empty[String, String]
  def receive = {
    case Get(url) =>
      if (cache contains url) sender() ! cache(url)
      else {
        val client = sender()
        WebClient get url map (Result(client, url, _)) pipeTo self
      }
    case Result(client, url, body) =>
      cache += url -> body
      client ! body
  }
}
```

## What we learned so far

- ▶ A reactive application is non-blocking & event-driven top to bottom.
- ▶ Actors are run by a dispatcher—potentially shared—which can also run Futures.
- ▶ Prefer immutable data structures, since they can be shared.
- ▶ Do not refer to actor state from code running asynchronously.

## The Receptionist (1)

```
class Receptionist extends Actor {  
  def receive = waiting  
  
  val waiting: Receive = {  
    // upon Get(url) start a traversal and become running  
  }  
  
  def running(queue: Vector[Job]): Receive = {  
    // upon Get(url) append that to queue and keep running  
    // upon Controller.Result(links) ship that to client  
    //   and run next job from queue (if any)  
  }  
}
```

## The Receptionist (2)

```
case class Job(client: ActorRef, url: String)
var reqNo = 0
def runNext(queue: Vector[Job]): Receive = {
  reqNo += 1
  if (queue.isEmpty) waiting
  else {
    val controller = context.actorOf(Props[Controller], s"c$reqNo")
    controller ! Controller.Check(queue.head.url, 2)
    running(queue)
  }
}
```

reqNo permeates all states but does not qualitatively change behavior: an example for when using var may benefit.

## The Receptionist (3)

```
def enqueueJob(queue: Vector[Job], job: Job): Receive = {  
  if (queue.size > 3) {  
    sender() ! Failed(job.url)  
    running(queue)  
  } else running(queue :+ job)  
}
```

## The Receptionist (4)

```
val waiting: Receive = {  
  case Get(url) => context.become(runNext(Vector(Job(sender(), url))))  
}
```

```
def running(queue: Vector[Job]): Receive = {  
  case Controller.Result(links) =>  
    val job = queue.head  
    job.client ! Result(job.url, links)  
    context.stop(sender())  
    context.become(runNext(queue.tail))  
  case Get(url) =>  
    context.become(enqueueJob(queue, Job(sender(), url)))  
}
```

## Summary

- ▶ A reactive application is non-blocking & event-driven top to bottom.
- ▶ Actors are run by a dispatcher—potentially shared—which can also run Futures.
- ▶ Prefer immutable data structures, since they can be shared.
- ▶ Prefer `context.become` for different states, with data local to the behavior.
- ▶ Do not refer to actor state from code running asynchronously.