CHAPTER 5

Actors in Scala

Author: This is the original version of the Actors chapter, written using Scala and Akka. After it was written, we took the decision to switch to Elixir in the book, but it seemed a shame to discard this version. Therefore, we've made it available as an online bonus chapter. Please be aware that it hasn't gone through the same degree of review and editing as the chapter in the book.

An actor is like a rental car—quick and easy to get hold of when you want one, and if it breaks down you don't bother trying to fix it, you just call the rental agency and another one is delivered to you.

The actor model is a general purpose concurrent programming model with particularly wide applicability. It can target both shared- and distributed-memory architectures, facilitates geographical distribution and provides especially strong support for fault-tolerance and resilience.

More Object-Oriented than Objects

Functional programming avoids the problems associated with shared mutable state by avoiding mutable state. Actor programming, by contrast, retains mutable state, but avoids sharing it.

An actor is like an object in an object-oriented program—it encapsulates state and communicates with other actors by exchanging messages. The difference is that actors run concurrently with each other and, unlike OO-style message passing (which is really just calling a method) actors *really* communicate by sending messages to each other.

Although actors are most commonly associated with Erlang, they can be used in just about any language. We're going to cover actors in Scala,[1] using the Akka concurrency library,[2] which is shipped with the standard Scala distri-

1. http://www.scala-lang.org
2. http://akka.io

bution. Akka supports many different concurrency models, but we will only look at its support for actors in this chapter.

Scala is a hybrid object/functional language that runs on the JVM. If you're familiar with either Java or Ruby, then you should find it easy enough to read. This isn't going to be a Scala tutorial (this is a book about concurrency, not programming languages), but I'll introduce the important Scala features we're using as we go along. There may be things you just have to take on trust if you're not already familiar with the language—I recommend *Programming in Scala* by Martin Odersky, Lex Spoon, and Bill Venners if you want to go deeper.

In day 1, we'll see the basics of the actor model—creating actors, and sending and receiving messages. In day 2 we'll see how failure detection, coupled with the "let it crash" philosophy, allows actor programs to be fault tolerant. Finally, in day 3 we'll see how actors' support for distributed programming allows us to both scale beyond a single machine, and recover from failure of one or more of those machines.

Day 1: Messages and Mailboxes

Today, we'll see how to create and stop actors, send and receive messages, and detect when an actor has terminated.

Our First Actor

Let's dive straight in with an example of creating a simple actor and sending it some messages. We're going to construct an actor called "talker." Here are the messages that it understands:

ActorsScala/HelloActors/src/main/scala/com/paulbutcher/HelloActors.scala

```
case class Greet(name: String)
case class Praise(name: String)
case class Celebrate(name: String, age: Int)
```

This defines three Scala *case classes* (we'll see why "case" classes shortly): Greet, Praise, and Celebrate, all of which have a name parameter of type String, to which Celebrate adds age of type Int.

Here's the code for our actor:

ActorsScala/HelloActors/src/main/scala/com/paulbutcher/HelloActors.scala

```
class Talker extends Actor {

  def receive = {
    case Greet(name) => println(s"Hello $name")
    case Praise(name) => println(s"$name, you're amazing")
```

```
    case Celebrate(name, age) => println(s"Here's to another $age years, $name")
  }
}
```

Now you can see why we used case classes to define our messages—case classes are classes that can be used within *case clauses*.

We'll pick through this code in more detail soon, but we're defining an actor that knows how to receive three different kinds of message, and prints an appropriate string when it receives each of them.

Finally, here's the main application:

ActorsScala/HelloActors/src/main/scala/com/paulbutcher/HelloActors.scala

```
object HelloActors extends App {

  val system = ActorSystem("HelloActors")

  val talker = system.actorOf(Props[Talker], "talker")

  talker ! Greet("Huey")
  talker ! Praise("Dewey")
  talker ! Celebrate("Louie", 16)

  Thread.sleep(1000)

  system.shutdown
}
```

First, we create an ActorSystem, and then use its actorOf() method to create an instance of Talker. Instances of actors are created by an *actor factory* created by Props—a polymorphic class that takes an actor type as an argument (Scala uses square brackets for type arguments where C++ or Java would use angle brackets). Because actorOf() takes a factory, it can create more than one instance of an actor if necessary—we'll see why this is important soon.

Next, we send three messages to our newly created actor with the ! (exclamation mark or tell) operator and sleep for a while to give it time to process those messages (using sleep() isn't the best approach—we'll see how to do this better soon). Finally, we shut down the actor system.

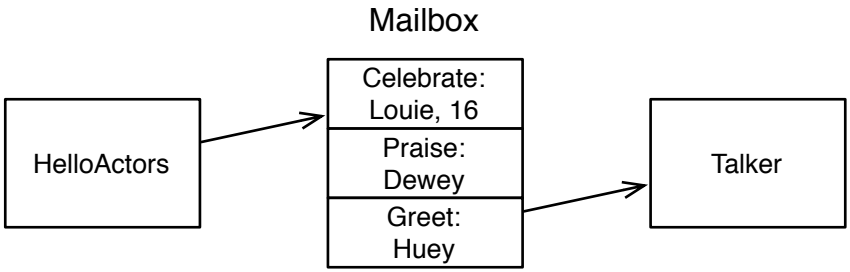
Here's what you should see when you run it:

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
```

Now that we've seen how to create an actor and send messages to it, let's see what's going on under the covers.

Mailboxes Are Queues

One of the most important features of actor programming is that messages are sent *asynchronously*. Instead of being sent directly to an actor, they are placed in a *mailbox*:



This means that actors are *decoupled*—actors run at their own speed and don't block when sending messages.

An actor runs concurrently with other actors, but handles messages sequentially, in the order they were added to the mailbox, moving on to next message only when it's finished processing the current message. We only have to worry about concurrency when sending messages.

Joe asks:

What About Other Kinds of Mailbox?

If you read the Akka documentation, you'll notice that an actor can be configured to use many different kinds of mailbox, including:

Priority mailboxes: allow high priority messages to be processed ahead of lower priority ones.

Durable mailboxes: store messages in a durable store (the file system, for example).

Bounded mailboxes: won't grow beyond a certain size.

Custom mailboxes: in case none of those provided as standard suffice.

Using one of these will affect the semantics of sending a message—sending a message to a bounded mailbox might block, for example.

These different types of mailbox are useful optimization tools, but they are not part of the core actor programming model and we will not cover them in this book.

The heart of an actor is its `receive()` method. Here's `Talker`'s again:

ActorsScala/HelloActors/src/main/scala/com/paulbutcher/HelloActors.scala
```
def receive = {
  case Greet(name) => println(s"Hello $name")
  case Praise(name) => println(s"$name, you're amazing")
  case Celebrate(name, age) => println(s"Here's to another $age years, $name")
}
```

Scala method definitions can look slightly odd if you're used to Java because much of the "noise" is missing. As receive() takes no arguments, Scala allows us to remove the parentheses following the method name, and there's no explicit return type because Scala's *type inference* has inferred it for us. Nor is there an explicit return statement—whatever is on the right hand side of the equals sign is returned from the method.

The return value is a *partial function* (a function that is defined for some values and not for others) that will be used to handle messages sent to the actor. In this case, a partial function that uses *pattern matching* to handle three different types of value: Greet, Praise and Celebrate.

Each case clause defines a *pattern*. Incoming messages are matched against each pattern in turn—if it matches, the variables in the pattern (name and age) are *bound* to the values in the message and the code to the right of the arrow (=>) executed. That code prints a message constructed using *string interpolation*—wherever a dollar sign appears in a string of the form s"...", the value to the right of the dollar sign is inserted into the string.

That's quite a bit of work for a humble 5-line method.

Joe asks:
Is an Actor a Thread?

As we saw in *Thread Creation Redux, on page 32*, creating too many threads can cause problems. Are we in danger of running into the same problems if we create too many actors?

The short answer is "no." Although it would be possible to create a naïve actor system where each actor ran on its own dedicated thread, that's not how Akka works. Instead, actors are scheduled to run on threads as needed (when they have messages to process). It's quite possible to have thousands of actors running concurrently without problems.

We won't cover them here, but dispatchers[a] allow you to tune exactly how actors are scheduled.

a. http://doc.akka.io/docs/akka/2.1.0/scala/dispatchers.html

The code on page 119 sleeps for a second to allow messages to be processed before shutting the actor system down. This is an unsatisfactory solution—happily, we can do better.

Poison Pills and Death Watch

We need two things to be able to shutdown cleanly. First, we need a way to tell talker to stop when it's finished processing all the messages in its queue. And second, we need some way to know when it has done so so we can then shut the system down.

We can achieve the first of these by sending talker a *poison pill*, and the second by establishing a *death watch* on it.

First, we have a little housekeeping to do—instead of creating an instance of Talker directly within our main thread, we're going to create a new "master" actor that will be notified when talker terminates:

ActorsScala/HelloActorsBetter/src/main/scala/com/paulbutcher/HelloActors.scala
```
object HelloActors extends App {

  val system = ActorSystem("HelloActors")

  system.actorOf(Props[Master], "master")
}
```

Here's the implementation of Master:

ActorsScala/HelloActorsBetter/src/main/scala/com/paulbutcher/HelloActors.scala
```
class Master extends Actor {

  val talker = context.actorOf(Props[Talker], "talker")

  override def preStart {
➤   context.watch(talker)

    talker ! Greet("Huey")
    talker ! Praise("Dewey")
    talker ! Celebrate("Louie", 16)
➤   talker ! PoisonPill
  }

  def receive = {
➤   case Terminated(`talker`) => context.system.shutdown
  }
}
```

The work of creating an instance of Talker and sending messages to it now takes place in Master's preStart() method, which as its name suggests is automatically called before the actor starts.

Akka provides a context member that an actor can use to gain access to the actor system it's running in, and various other features. Calling context.watch() establishes a death watch on talker, meaning that master will receive a Terminated message when talker terminates. The backticks (`) in the Terminated pattern mean that it matches the value of the existing talker variable, rather than binding to a new variable.

Finally, PoisonPill is a standard Akka message that all actors understand, which causes them to stop when they receive it. It's a normal message that gets added to the message queue just like any other, so we know that it will be handled after any previously sent messages.

Producer-Consumer with Actors

We've now got enough tools at our fingertips to create an actor-based version of our Wikipedia word-count program. As before, we'll split it into a producer that parses the XML into pages and a consumer that counts the words on each page. This time, however, both producer and consumer will be actors.

We can't simply create a producer that sends pages to the consumer as fast as it can parse them, however. Because we can parse pages much more quickly than we can count the words on them, we need to put some kind of *flow control* in place. If we didn't, the consumer's queue would grow until it exhausted memory.

With that in mind, here's the souce for Parser:

ActorsScala/WordCount/src/main/scala/com/paulbutcher/Parser.scala

```
Line 1  case object Processed
        -
        -  class Parser(counter: ActorRef) extends Actor {
        -
        5    val pages = Pages(100000, "enwiki.xml")
        -
        -    override def preStart {
        -      for (page <- pages.take(10))
        -        counter ! page
       10    }
        -
        -    def receive = {
        -      case Processed if pages.hasNext => counter ! pages.next
        -      case _ => context.stop(self)
       15    }
```

```
}
```

Parser's constructor takes a reference to the counter actor (in Scala, constructor arguments go after the class name) so it knows where to send pages. On line 8, we "prime the pump" by sending 10 messages, but we don't send any more until we receive a Processed message (line 13). Finally, when there are no more pages left, on line 14, we stop.

Here's the source for Counter:

ActorsScala/WordCount/src/main/scala/com/paulbutcher/Counter.scala

```
class Counter extends Actor {

  val counts = HashMap[String, Int]().withDefaultValue(0)

  def receive = {
    case Page(title, text) =>
      for (word <- Words(text))
        counts(word) += 1
      sender ! Processed
  }
}
```

Each time it receives a Page, it adds the words in that page to its counts map. It then makes use of the fact that each message has a sender associated with it to send a Processed message back to the parser.

Finally, here's Master, which creates instances of both Parser and Counter:

ActorsScala/WordCount/src/main/scala/com/paulbutcher/Master.scala

```
class Master extends Actor {

  val counter = context.actorOf(Props[Counter], "counter")
  val parser = context.actorOf(Props(new Parser(counter)), "parser")

  override def preStart {
    context.watch(counter)
    context.watch(parser)
  }

  def receive = {
    case Terminated(`parser`) => counter ! PoisonPill
    case Terminated(`counter`) => context.system.shutdown
  }
}
```

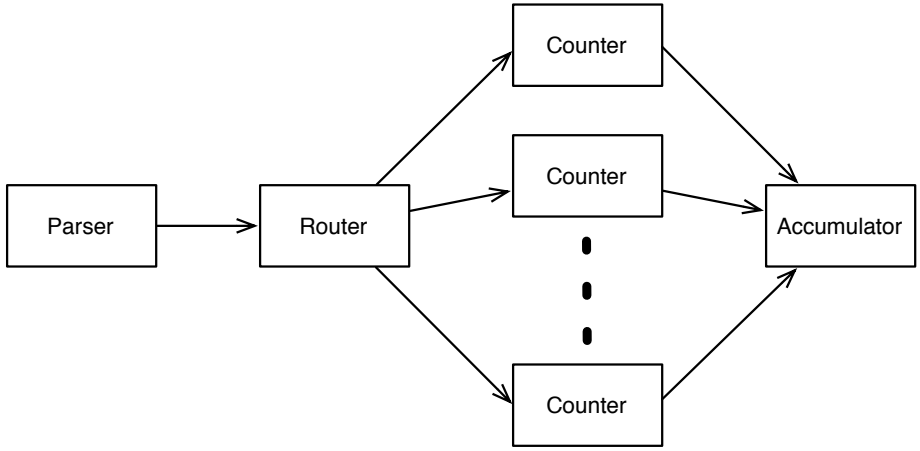
When Master notices that parser has stopped, it sends a PoisonPill to counter. And when counter stops, it shuts the entire system down.

As we did with our threads and locks version, now that we have a working producer-consumer implementation, we can speed it up by running multiple consumers.

Multiple Consumers

There are a couple of questions we need to answer before we can create a multiple consumer version of WordCount. Firstly, how will our producer know to which consumer it should send each message? And secondly, how are we going to accumulate the results from multiple consumers?

It turns out that Akka provides an out of the box answer to the first question in the form of *routers*. The second problem is solved by introducing another actor—an accumulator. Here's a diagram of where we're heading:



Let's start with the accumulator:

ActorsScala/WordCountMultipleCounters/src/main/scala/com/paulbutcher/Accumulator.scala

```
case class Counts(counts: Map[String, Int])

class Accumulator extends Actor {

  val counts = HashMap[String, Int]().withDefaultValue(0)

  def receive = {
    case Counts(partialCounts) =>
      for ((word, count) <- partialCounts)
        counts(word) += count
  }
}
```

When it receives a Counts message, it iterates through each (word, count) pair in the map of partial counts, adding them to its totals in counts.

The consumer is very similar to what we've already seen:

ActorsScala/WordCountMultipleCounters/src/main/scala/com/paulbutcher/Counter.scala

```
class Counter(accumulator: ActorRef) extends Actor {

  val counts = HashMap[String, Int]().withDefaultValue(0)

  def receive = {
    case Page(title, text) =>
      for (word <- Words(text))
        counts(word) += 1
      sender ! Processed
  }

  override def postStop() {
    accumulator ! Counts(counts)
  }
}
```

There are two changes. Firstly we're passing it a reference to the accumulator. Secondly, we override postStop() to send a Counts message to the accumulator. As its name suggests, postStop() is called after an actor has stopped.

Parser is unchanged from what we already have.

Finally, here's the new version of Master that wires everything together:

ActorsScala/WordCountMultipleCounters/src/main/scala/com/paulbutcher/Master.scala

```
class Master extends Actor {

  val accumulator = context.actorOf(Props[Accumulator], "accumulator")
  val counters = context.actorOf(
    Props(new Counter(accumulator)).withRouter(RoundRobinRouter(4)),
    "counter")
  val parser = context.actorOf(Props(new Parser(counters)), "parser")

  override def preStart {
    context.watch(accumulator)
    context.watch(counters)
    context.watch(parser)
  }

  def receive = {
    case Terminated(`parser`) => counters ! Broadcast(PoisonPill)
    case Terminated(`counters`) => accumulator ! PoisonPill
    case Terminated(`accumulator`) => context.system.shutdown
  }
}
```

Instead of creating a single Counter instance, we create an instance of RoundRobinRouter. This router then creates 4 Counter instances using the actor factory created by Props. Any messages sent to the router are routed to these counters, round robin fashion.

The reference returned by actorOf() is a reference to the router. The beauty is that we can pass this reference to our producer, and it's none the wiser—exactly the same implementation that worked with a single consumer works just fine with multiple consumers.

PoisonPill and Routers

It is possible to send a PoisonPill directly to a router, and doing so will appear to work—both the router and its routees will stop. Unfortunately, doing so introduces a subtle bug. When it receives a PoisonPill, a router shuts itself and its routees down immediately, whether or not those routees still have messages to process.

The solution is to broadcast the PoisonPill to the routees:

```
counters ! Broadcast(PoisonPill)
```

This time, the message is added to the tail of each routee's mailbox as normal, so they shut down only when they've processed the messages already in the mailbox. The router automatically shuts down when it notices that its routees have all shut down.

Day 1 Wrap-Up

This brings us to the end of day 1. In day 2, we'll see how the actor model helps with error handling and resilience.

What We Learned in Day 1

Actors run concurrently, do not share state, and communicate by asynchronously sending messages to mailboxes. We saw how to:

- Create actors.
- Send messages.
- Leverage Scala's pattern matching to match and process messages.
- Register for notification when an actor terminates (death watch).
- Send a poison pill to shut an actor down cleanly.
- Use a router to create and route messages to multiple actors.

Day 1 Self-Study

Find

- As well as the ! or tell operator, Akka also provides ? or ask. How does ask make use of futures? When might you choose to use it over tell?
- In addition to RoundRobinRouter, what other types of router are available? When might you use them?
- Scala/Akka programs run on top of the JVM and are therefore subject to the Java memory model. Why isn't memory visibility an issue for an actor program?

Do

- Use Akka's configuration mechanism to configure the number of counters created by WordCount in application.conf instead of in code.
- Research Akka's become/unbecome mechanism and implement the dining philosophers problem we saw in Chapter 2, Threads and Locks, on page 9 with it.
- Reimplement your dining philosophers solution using Akka's FSM mixin. What are the strengths and weaknesses of the two approaches?

Day 2: Error Handling and Resilience

As we saw in Concurrency Enables Resilience on page 6, one of the key benefits of concurrency is that it enables us to write fault tolerant code. Today we'll see the tools that actors provide to enable us to do so.

We'll start by looking at how one actor can find an instance of another by looking up its *path*.

Actor Paths

One of the great things about Scala is its *console* (sometimes called a "Read, Eval, Print Loop" or *REPL*). This allows you to type code and have it evaluated immediately without having to create source files and compile them, which can be amazingly helpful when experimenting with unfamiliar code. We'll use it now to experiment with actor paths.

The "Paths" project that accompanies this book defines TestActor as follows:

ActorsScala/Paths/src/main/scala/com/paulbutcher/TestActor.scala

```
case class CreateChild(name: String)
case object SayHello
```

```
case class SayHelloFrom(path: String)

class TestActor extends Actor {

  def receive = {
    case CreateChild(name) => context.actorOf(Props[TestActor], name)
    case SayHello => println(s"Hello from $self")
    case SayHelloFrom(path) => context.actorFor(path) ! SayHello
  }
}
```

You can start a Scala console by typing sbt console (sbt is Scala's standard build tool—the equivalent of rake for Ruby or ant or mvn for Java). You should see something like this:

```
Welcome to Scala version 2.10.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_09).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Any code you type after the scala> prompt will be evaluated immediately. Here's how you can create an ActorSystem and a couple of actors:

```
scala> val system = ActorSystem("Paths")
system: akka.actor.ActorSystem = akka://Paths

scala> val anActor = system.actorOf(Props[TestActor], "an-actor")
anActor: akka.actor.ActorRef = Actor[akka://Paths/user/an-actor]

scala> anActor ! SayHello

Hello from Actor[akka://Paths/user/an-actor]
```

Now you can see why we've been giving names to the actors we've been creating—they're used to create a *path* that can be used to refer to that actor.

Actor paths are URIs, and hierarchical. There are two ways to get hold of a reference to an actor—by holding on to the reference returned by actorOf(), or by looking up a path:

```
scala> anActor ! CreateChild("a-child")

scala> val aChild = system.actorFor("/user/an-actor/a-child")
aChild: akka.actor.ActorRef = Actor[akka://Paths/user/an-actor/a-child]

scala> aChild ! SayHello

Hello from Actor[akka://Paths/user/an-actor/a-child]
```

The CreateChild message we defined earlier instructs our actor to create a child actor with context.actorOf. Because it's a child, its path is a sub-path of its parent's. We then lookup a reference for that new actor with actorFor().

Paths can be relative, as well as absolute:

```
scala> anActor ! CreateChild("another-child")

scala> aChild ! SayHelloFrom("../another-child")

Hello from Actor[akka://Paths/user/an-actor/another-child]
```

The SayHelloFrom(path) message instructs our actor to send a SayHello message to the actor reference returned by context.actorFor(path).

Paths can even include wildcards, allowing us to reference more than one actor at a time with actorSelection():

```
scala> val children = system.actorSelection("/user/an-actor/*")
children: akka.actor.ActorSelection = akka.actor.ActorSelection$$anon$1@51a4bf7e

scala> children ! SayHello

Hello from Actor[akka://Paths/user/an-actor/a-child]
Hello from Actor[akka://Paths/user/an-actor/another-child]
```

Joe asks:

Why /user?

You'll have noticed that all our actor paths contain /user. Akka supports a number of *top level scopes* for actor paths, including:

/user the parent for user-created top level actors (those created by system.actorOf).

/system the parent for system-created actors.

/temp the parent for short-lived system-created actors.

Next, we'll look at exactly when actors are started, stopped and restarted.

The Actor Life-cycle

To understand how actors help with fault tolerance, we need to understand the life-cycle of an actor. The "Lifecycle" project that accompanies this book defines TestActor, which overrides the life-cycle hooks provided by Akka so we can see when they're called:

ActorsScala/Lifecycle/src/main/scala/com/paulbutcher/TestActor.scala

```
case class CreateChild(name: String)
```

```
case class Divide(x: Int, y: Int)

class TestActor extends Actor {

  def receive = {
    case CreateChild(name) => context.actorOf(Props[TestActor], name)
    case Divide(x, y) => log(s"$x / $y = ${x / y}")
  }

  override def preStart() { log(s"preStart") }

  override def preRestart(reason: Throwable, message: Option[Any]) {
    log(s"preRestart ($reason, $message)")
  }

  override def postRestart(reason: Throwable) { log(s"postRestart ($reason)") }

  override def postStop() { log(s"postStop") }

  def log(message: String) { println(s"${self.path.name}: $message") }
}
```

We've already seen preStart() and postStop()—the others are preRestart() and postRestart().

Let's see what happens if we create an actor with a child and then stop it:

```
scala> val anActor = system.actorOf(Props[TestActor], "an-actor")
anActor: akka.actor.ActorRef = Actor[akka://Paths/user/an-actor]

an-actor: preStart

scala> anActor ! CreateChild("a-child")

a-child: preStart

scala> anActor ! PoisonPill

a-child: postStop
an-actor: postStop
```

We can see that preStart() is called when an actor starts, and postStop() when it stops. And stopping a parent also stops its children.

So far, so unsurprising. Let's see what happens if we make the child perform a division by zero (and therefore throw an ArithmeticException):

```
scala> val child = system.actorFor("/user/an-actor/a-child")
child: akka.actor.ActorRef = Actor[akka://Paths/user/an-actor/a-child]

scala> child ! Divide(1, 0)
```

```
[ERROR] «...» [akka://Paths/user/an-actor/a-child] / by zero
java.lang.ArithmeticException: / by zero
  «stack trace»

a-child: preRestart (java.lang.ArithmeticException: / by zero, Some(Divide(1,0)))
a-child: postRestart (java.lang.ArithmeticException: / by zero)
```

So when the child experiences an error, it's automatically restarted. We'll see why next.

Supervision

An actor's parent is also its *supervisor*. Whenever an actor throws an exception, its supervisor is consulted to see what should happen. The supervisor chooses between the following options:

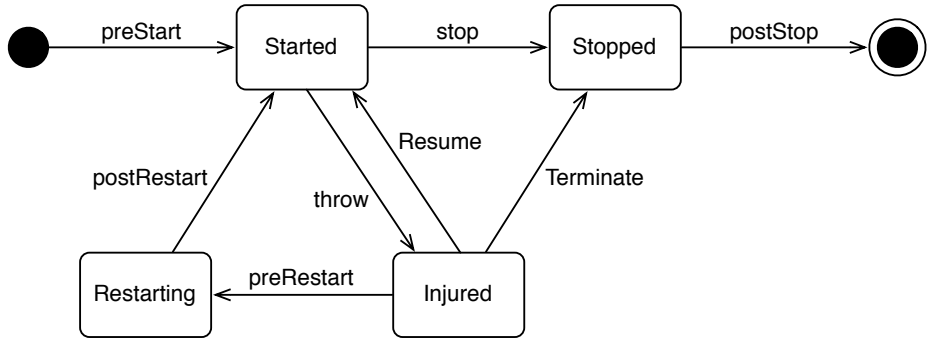
Resume: Discards the message that was being processed when the exception was thrown. Maintains the internal state of the actor.

Restart: Discards the message that was being processed when the exception was thrown. Destroys the original instance of the actor and creates a new one. This has the effect of resetting the internal state of the actor.

Terminate: Terminates the actor. Any further messages that were in the actor's mailbox will not be processed.

Escalate: Escalate the decision to the supervisor's supervisor. This might result in the supervisor itself being restarted or terminated.

The possible transitions are shown in the following diagram:



In addition, a supervisor chooses between a *one-for-one* or *all-for-one* strategy:

One-for-one: Only the child that experienced the error is restarted or terminated.

All-for-one: All of the supervisor's children are restarted or terminated when a single child experiences an error.

The default supervision strategy is one-for-one, and in most cases restarts the failing actor (see the Akka documentation for full details) but can be overridden.

Joe asks:
What's the Difference Between Supervision and Death Watch?

Supervision and death watch are related, but different. Death watch allows us to observe failure, supervision allows us to manage it.

Every actor has exactly one supervisor—it's parent, but can have zero or more death watches in place. A supervisor may establish a death watch on its children, but does not have to do so.

A Custom Supervisor Strategy

Let's experiment with creating our own customized supervisor strategy. Here's an actor that implements a simple cache (perhaps we want to cache web pages), but contains a few bugs:

ActorsScala/BuggyCache/src/main/scala/com/paulbutcher/BuggyCache.scala

```
case class Put(key: String, value: String)
case class Get(key: String)
case object ReportSize

class BuggyCache extends Actor {

  val cache = HashMap[String, String]()
  var size = 0

  def receive = {
    case Put(key, value) =>
      cache(key) = value
      size += value.length

    case Get(key) => sender ! Result(cache(key))

    case ReportSize => sender ! Result(size)
  }

  override def postRestart(reason: Throwable) {
    println("BuggyCache has restarted")
```

```
  }
}
```

It supports three messages: Put adds an entry to the cache, Get retrieves an entry, and ReportSize reports how much data the cache contains.

Here's an actor that we'll use as its supervisor:

ActorsScala/BuggyCache/src/main/scala/com/paulbutcher/Master.scala

```
case class Result(result: Any)

class Master extends Actor {

  val cache = context.actorOf(Props[BuggyCache], "cache")

  def receive = {
    case Put(key, value) => cache ! Put(key, value)
    case Get(key) => cache ! Get(key)
    case ReportSize => cache ! ReportSize
    case Result(result) => println(result)
  }

  override val supervisorStrategy = OneForOneStrategy() {
      case _: NoSuchElementException => Resume
      case _: NullPointerException => Restart
      case _ => Escalate
    }
}
```

This actor overrides supervisorStrategy with an instance of OneForOneStrategy. A child will be resumed if it throws NoSuchElementException, and restarted if it throws NullPointerException. If it throws anything else, the decision will be escalated.

Here's an example of things working:

```
scala> val master = system.actorOf(Props[Master], "master")
master: akka.actor.ActorRef = Actor[akka://BuggyCache/user/master]

scala> master ! Put("google.com", "Welcome to Google ...")

scala> master ! Get("google.com")

Welcome to Google ...

scala> master ! ReportSize

21
```

So far so good—we can put an entry into our cache, get it back again, and see how large the cache is.

What happens if we try to retrieve a non-existent entry?

```
scala> master ! Get("nowhere.com")

[ERROR] «...» [akka://BuggyCache/user/master/cache] key not found: nowhere.com
java.util.NoSuchElementException: key not found: nowhere.com
  «stack trace»
```

Damn—looks like our cache doesn't handle non-existent entries well, throwing an unhandled NoSuchElementException. Not good.

Nevertheless, because its supervisor's strategy says that this should result in a Resume, we can continue using the cache, and all its previous state still exists:

```
scala> master ! Get("google.com")

Welcome to Google ...

scala> master ! ReportSize

21
```

How about a different error? What happens if we try to insert bad data (a null value) into the cache?

```
scala> master ! Put("paulbutcher.com", null)

[ERROR] «...» [akka://BuggyCache/user/master/cache] null
java.lang.NullPointerException
  «stack trace»

BuggyCache has restarted

scala> master ! ReportSize

0
```

Looks like our cache doesn't handle that very well either. This time, the BuggyCache actor is restarted. Nevertheless, we can still send messages to the same reference after the restart, although the new instance has lost any state that was stored in the previous instance.

The Elements of Fault Tolerance

We've now seen all the building blocks that help us create fault tolerant code:

Actor Factories: Because actorOf() takes an actor factory, new instances of actors can be created when necessary, in particular when they need to be restarted.

Mailboxes: As-yet unhandled messages reside in a mailbox, meaning that they aren't lost, and can still be handled if an actor is restarted.

Actor References: Because actorOf() and actorFor() both return an actor reference, not the actor itself, we can continue to use the same reference even if the actor is restarted.

Supervision: Every actor has a supervisor, which decides what action should be taken if it experiences an error.

Death watch: Any actor can establish a death watch on any other, allowing it to know when the actor dies and take appropriate action.

These building blocks naturally lead to a hierarchical structure in which risky operations are pushed down the hierarchy, which we'll cover next.

The Error Kernel Pattern

Tony Hoare famously said:[3]

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

Actor programming naturally supports an approach to writing fault-tolerant code that leverages this observation—the *error kernel pattern*.

A software system's *error kernel* is the part that must be correct if the system is to function correctly. Well written programs make this error kernel as small and as simple as possible. So small and simple that there are obviously no deficiencies.

An actor program's error kernel is its top-level actors. These supervise their children, starting, stopping, resuming, and restarting them as necessary.

Each module of a program has its own error kernel in turn—the part of the module that must be correct for it to function correctly. Sub-modules also have error kernels, and so on.

This leads to a hierarchy of error kernels in which risky operations are pushed down towards the lower-level actors, as shown in the following figure:

3. http://awards.acm.org/images/awards/140/articles/4622167.pdf

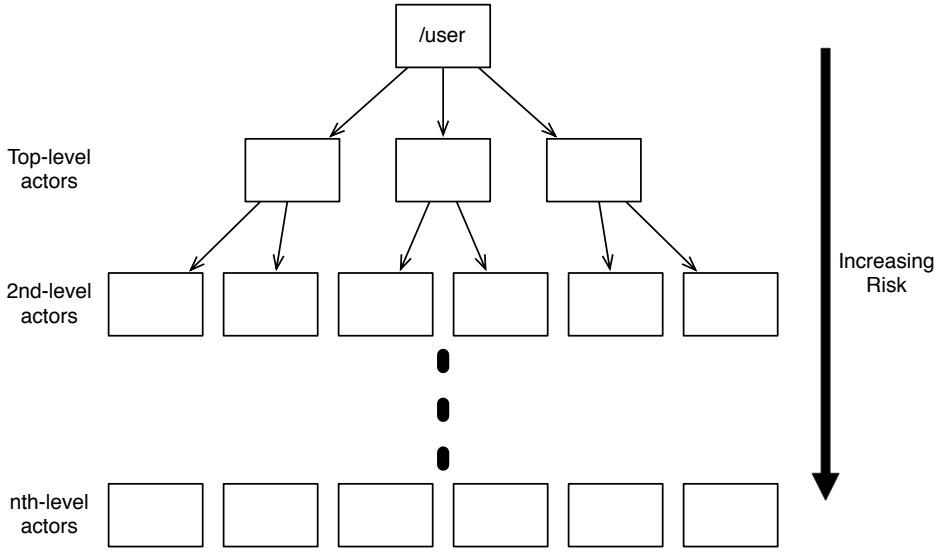


Figure 4—A Hierarchy of Error Kernels

Closely related to the error kernel pattern is the thorny subject of *defensive programming*.

Let It Crash!

Defensive programming is an approach to achieving fault-tolerance by trying to anticipate possible bugs. Imagine, for example, that we're writing a method that takes a string and returns `true` if it's all uppercase and `false` otherwise. Here's one possible implementation:

```
def allUpper(s: String) = s.forall(_.isUpper)
```

This is a perfectly reasonable method, but if for some reason we pass `null` to it, it will crash. With that in mind, some developers would add something along these lines to the beginning:

```
if (s == null) return false
```

So, now the code won't crash, but what does it *mean* to call this function with `null`? There's an excellent chance that any code that does so contains a bug—a bug that we've now masked, meaning that we're likely to remain unaware of it until it bites us at some time in the future.

Actor programs tend to avoid defensive programming and subscribe to the *let it crash* philosophy, allowing the fault tolerance mechanisms we've discussed to address the problem instead. This has multiple benefits, including:

- Our code is simpler and easier to understand, with a clear separation between "happy path" and fault-tolerance code.
- Actors are separate from each other and don't share state, so there's little danger that a failure in one actor will adversely affect another. In particular, a failed actor's supervisor cannot crash because the actor it's supervising crashes.
- Because the failure of an actor is logged, instead of sweeping problems under the carpet, we become aware of them and can take remedial action.

Although it can seem alien at first acquaintance, the let it crash philosophy has, together with the error kernel pattern, repeatedly been proven in production. Some systems have reported availability as high as 99.9999999% (that's nine nines—see *Programming Erlang: Software for a Concurrent World.* by Joe Armstrong).

Day 2 Wrap-Up

Day 1 introduced the basics of the actor model, and in day 2, we saw how the actor model facilitates fault-tolerance. In day 3, we'll see how the actor model helps with distributed programming.

What We Learned in Day 2

Actors form a hierarchy and can be referred to through paths. Actors facilitate the error kernel pattern and the let it crash philosophy. The building blocks of fault tolerance are:

- Actor factories, allowing new actor instances to be created during a restart.
- Mailboxes, ensuring that as-yet unhandled messages aren't lost.
- Actor references, which continue to be valid after a restart.
- Supervision, providing a structured way to decide how to react to errors.
- Death watch, allowing an actor that depends on another to know when it dies.

Day 2 Self-Study

Find

- What is Akka's default supervision strategy? How does it handle non-error shutdown?
- What is SupervisorStrategy.stoppingStrategy? When might you use it?
- What does the default implementation of preRestart() do?

Do

- Messages sent to actors that have terminated are sent to a virtual *dead letter* mailbox. Write code to intercept messages sent to the dead letter mailbox and display them.

Day 3: Distribution

Everything we've done so far has been on a single computer, but one of actors' primary benefits compared to other concurrency models is that they support distribution. All it takes is a little configuration to allow an actor on one machine to send messages to one running on another.

Clustering

A *cluster* is a set of machines that collaborate to solve a single problem. Cluster members can register to receive *member events* to be notified when new members join or existing members leave or fail.

The "HelloCluster" project that accompanies this book defines a simple actor system that we can use to experiment with clustering. It enables clustering through application.conf:

ActorsScala/HelloCluster/src/main/resources/application.conf

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"
  }

  extensions = ["akka.cluster.Cluster"]
}
```

Joe asks:
How Do I Manage My Cluster?

Akka provides a number of cluster management options, including a command-line interface and JMX. Two key areas to think about when designing a cluster are how new nodes join, and how to handle node failure. The "HelloCluster" example adds the following to the basic cluster configuration:

```
cluster {
  auto-join = off
  auto-down = on
}
```

For this example, I've chosen to have actor systems join the cluster by taking a hostname on the command line and then explicitly calling Cluster(system).join(address). An alternative would be to switch auto-join on and configure one or more *seed nodes*.

I've also chosen to switch auto-down on, which means that a cluster member that becomes unreachable is automatically marked as "down." This may not be the right choice in production, however.

Cluster design trade-offs are subtle, and beyond the scope of this book. Please make sure that you read the documentation about these questions before rolling out a production cluster.

Here's the implementation of TestActor:

ActorsScala/HelloCluster/src/main/scala/com/paulbutcher/TestActor.scala

```
case class HelloFrom(actor: ActorRef)

class TestActor extends Actor {

  def receive = {
    case MemberUp(member) =>
      println(s"Member is up: $member")
      val remotePath = RootActorPath(member.address) / "user" / "test-actor"
      val remote = context.actorFor(remotePath)
      remote ! HelloFrom(self)
      context.watch(remote)

    case HelloFrom(actor) => println(s"Hello from: $actor")
    case Terminated(actor) => println(s"Terminated: $actor")
    case event => println(s"Event: $event")
  }
}
```

It does little more than print out the messages that it receives. The exception is the MemberUp message that indicates that a new node has joined the cluster. When it receives this message, our actor looks up the test-actor instance on

the new cluster member, sends it a HelloFrom message and registers it for death watch.

Finally, here's an application that uses it:

ActorsScala/HelloCluster/src/main/scala/com/paulbutcher/HelloCluster.scala
```
object HelloCluster extends App {

  val opts = parseCommandline

  System.setProperty("akka.remote.netty.hostname", opts.localHost)
  System.setProperty("akka.remote.netty.port", opts.localPort)

  val system = ActorSystem("ClusterTest")

  val testActor = system.actorOf(Props[TestActor], "test-actor")
  Cluster(system).subscribe(testActor, classOf[MemberEvent])

  Cluster(system).join(
    Address("akka", "ClusterTest", opts.clusterHost, opts.clusterPort))
}
```

This parses the command line to find which hostname and port it should use, and the hostname and port of the cluster that it should join. After creating an instance of TestActor, it subscribes that actor to receive cluster member events.

Here's what happens if I create a single node cluster by having an actor system (running on a machine with IP address 172.16.129.1) join itself:

```
$ sbt "run --local-host 172.16.129.1 --cluster-host 172.16.129.1"
Event: CurrentClusterState(TreeSet(),Set(),Set(),None)
Event: MemberJoined(Member(
    address = akka://ClusterTest@172.16.129.1:2552, status = Joining))
Member is up: Member(address = akka://ClusterTest@172.16.129.1:2552, status = Up)
Hello from: Actor[akka://ClusterTest/user/test-actor]
```

We can see three events arriving at our test actor. First it receives CurrentClusterState, showing that the cluster is currently empty. Then it receives a MemberJoined event for the local actor system, followed by a MemberUp. The last thing we see is a HelloFrom message from the actor sent to itself.

Now let's see what happens if we fire up an actor system on another computer (172.16.129.137) and get it to join our cluster:

```
$ sbt "run --local-host 172.16.129.137 --cluster-host 172.16.129.1"
Event: CurrentClusterState(TreeSet(),Set(),Set(),None)
Member is up: Member(address = akka://ClusterTest@172.16.129.1:2552, status = Up)
Event: MemberJoined(Member(
    address = akka://ClusterTest@172.16.129.137:2552, status = Joining))
```

```
Member is up: Member(address = akka://ClusterTest@172.16.129.137:2552, status = Up)
Hello from: Actor[akka://ClusterTest/user/test-actor]
Hello from: Actor[akka://ClusterTest@172.16.129.1:2552/user/test-actor]
```

As before, it first receives CurrentClusterState followed by MemberUp for the first cluster node. Then there are MemberJoined and MemberUp events for the new node. Finally, we see two HelloFrom messages, one from the new instance of TestActor sent to itself, and one from the original actor.

On our first node, we see:

```
Event: MemberJoined(Member(
    address = akka://ClusterTest@172.16.129.137:2552, status = Joining))
Hello from: Actor[akka://ClusterTest@172.16.129.137:2552/user/test-actor]
Member is up: Member(address = akka://ClusterTest@172.16.129.137:2552, status = Up)
```

So we can see the new cluster member joining, and the HelloFrom message from the actor on that node.

Joe asks:
What If I Have Only One Computer?

If you have only one computer to hand and still want to experiment with clustering, you have a few options:

- Use virtual machines.
- Fire up Amazon EC2 or similar cloud instances.
- Run multiple actor systems locally. Note that if you choose this route, each actor system must run using a different port.

Finally, what if we kill the actor system on our new node? Initially, nothing happens, but after a few seconds (long enough for Akka's cluster system to notice that the system at the other end of the network connection has gone down) here's what we see on the original node:

```
Terminated: Actor[akka://ClusterTest@172.16.129.137:2552/user/test-actor]
Event: MemberDowned(Member(
    address = akka://ClusterTest@172.16.129.137:2552, status = Down))
```

So our death watch on the actor on the other node has triggered, and we've received a MemberDowned message.

It's worth taking a moment to think about what this simple example has demonstrated. We've shown that we can dynamically add nodes to a cluster, have actors in the cluster automatically discover actors running on other

nodes, and notice when a network connection or cluster member goes down. That's a pretty powerful toolbox for creating distributed fault tolerant systems.

Push or Pull?

We'll spend the rest of today developing a distributed, fault-tolerant version of our word count program on top of Akka's cluster support. Before we start writing code, we should think about the best way to distribute work between multiple actors when they're running on different machines.

We glossed over a potential problem with the version we built yesterday—what happens if some pages take much longer to process than others?

The round robin router we used will simply distribute work to the next actor in sequence, whether or not it's busy. If we're unlucky, and one actor happens to receive all the big work items, it could lag behind the others. Eventually this might lead to us having to wait for that actor to catch up while the others sit idle.

Although this *might* happen when our actors all run on a single machine, it's much more likely if they're running on different machines with different performance characteristics.

One potential solution would be to use an alternative router. `SmallestMailboxRouter`, for example, is designed to address exactly this kind of problem. We're going to look at a different solution, however—switching to a *pull model* in which consumers request work when they're idle. Not only does this naturally provide load balancing between consumers running at different speeds, but it also helps with fault tolerance.

A WordCount Cluster

Here's a diagram of the structure we're heading for:

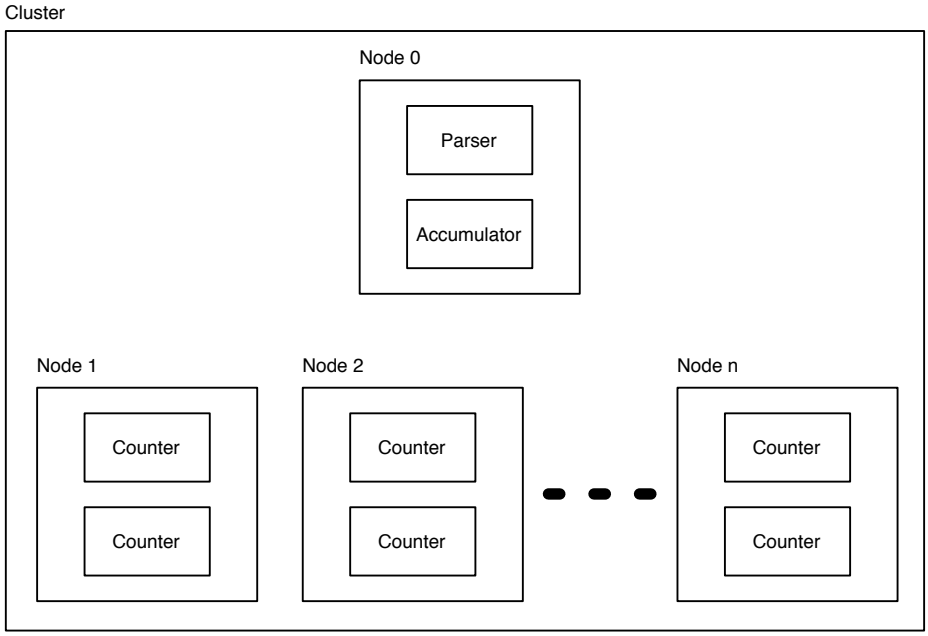
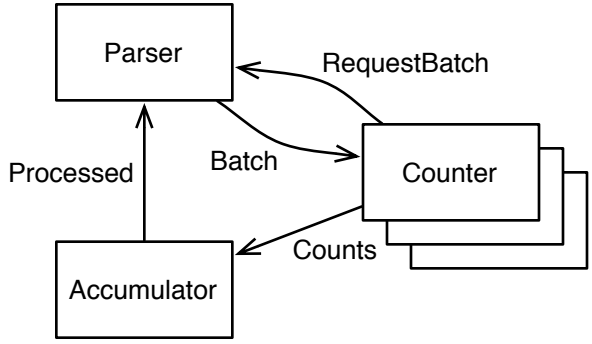


Figure 5—A WordCount Cluster

We're going to have a single node that hosts the parser and the accumulator, and then a number of nodes, each of which hosts a number of counters.

Here is the message flow between the three core types of actor:



Processing a page is kicked off by a counter sending a `RequestBatch` message to the parser. The parser responds with a `Batch` of work. When the counter completes the batch, it sends the `Counts` it collected to the accumulator.

Finally, the accumulator let's the parser know that the batch has been Processed.

Joe asks:
Why Batch Work?

The code we wrote yesterday sent pages one at a time. Why are we now batching multiple pages within a message?

The reason is efficiency. When a consumer wants work from a producer, it now needs to both send a message and receive a reply, potentially over a network connection. This introduces both overhead and latency. We can amortize this by batching multiple pages into a single message.

Let's start by looking at the implementation of Counter:

ActorsScala/WordCountFaultTolerant/src/main/scala/com/paulbutcher/Counter.scala

```
case class ParserAvailable(parser: ActorRef)
case class Batch(id: Int, pages: Seq[Page], accumulator: ActorRef)

class Counter extends Actor {

  def receive = {
    case ParserAvailable(parser) => parser ! RequestBatch

    case Batch(id, pages, accumulator) =>
      sender ! RequestBatch
      val counts = HashMap[String, Int]().withDefaultValue(0)
      for (page <- pages)
        for (word <- Words(page.text))
          counts(word) += 1
      accumulator ! Counts(id, counts)
  }
}
```

When it receives a ParserAvailable message, telling it that a new parser has become available (we'll see what sends this message soon) it sends a RequestBatch message to start work flowing.

Upon receiving a batch of work, it first requests more work. We do this first so the producer can prepare the next batch while this one is being worked on, minimizing the effect of network latency.

Next, it iterates over the pages contained within this batch, counting the words contained within. Finally, it sends those counts to the accumulator.

Note that each batch has an id associated with it. We'll see how this is used when we look at the implementation of the parser.

Next, let's see how instances of Counter are created:

ActorsScala/WordCountFaultTolerant/src/main/scala/com/paulbutcher/Counters.scala

```
class Counters(count: Int) extends Actor {

  val counters = context.actorOf(Props[Counter].
    withRouter(new BroadcastRouter(count)), "counter")

  override def preStart {
    Cluster(context.system).subscribe(self, classOf[MemberUp])
  }

  def receive = {
    case state: CurrentClusterState =>
      for (member <- state.members if (member.status == Up))
        counters ! ParserAvailable(findParser(member))

    case MemberUp(member) => counters ! ParserAvailable(findParser(member))
  }

  def findParser(member: Member) =
    context.actorFor(RootActorPath(member.address) / "user" / "parser")
}
```

Counters creates a set of Counter instances by using a BroadcastRouter. As its name suggests, this broadcasts any message it receives to its routees.

Counters registers to receive cluster events. When it receives either CurrentClusterState or MemberUp, it looks up the router instance on the nodes identified by those messages and notifies its routees with ParserAvailable.

Joe asks:
What About Cluster Members With No Parser?

Every time a member joins the cluster, we look up /user/parser on it and send RequestBatch messages to the resulting ActorRef. But just because a new member has joined the cluster doesn't mean that a Parser instance is running on it. So what happens to these messages?

Messages sent to non-existent actors (or actors that have terminated) go to a special virtual mailbox called *dead letters*, which by default discards them.

So if no parser is running on a new node, any message we send to it is discarded.

Next, here is Accumulator:

ActorsScala/WordCountFaultTolerant/src/main/scala/com/paulbutcher/Accumulator.scala

```
case class Counts(id: Int, counts: Map[String, Int])

class Accumulator(parser: ActorRef) extends Actor {

  val counts = HashMap[String, Int]().withDefaultValue(0)
  val processedIds = Set[Int]()

  def receive = {
    case Counts(id, partialCounts) =>
      if (!processedIds.contains(id)) {
        for ((word, count) <- partialCounts)
          counts(word) += count
        processedIds += id
        parser ! Processed(id)
      }
  }
}
```

When it receives the counts associated with a batch, it first checks to see if it's already received counts for that batch. If it has, it discards the new counts—this means that it won't double-count if a batch is processed more than once (we'll see how this might happen soon). If it hasn't already seen the id, it adds the counts to its totals and lets the parser know that the batch has been completely processed.

Finally, here's Parser:

ActorsScala/WordCountFaultTolerant/src/main/scala/com/paulbutcher/Parser.scala

```
Line 1 case object RequestBatch
- case class Processed(id: Int)
-
- class Parser(filename: String, batchSize: Int, limit: Int) extends Actor {
5
-   val pages = Pages(limit, filename)
-   var nextId = 1
-   val pending = LinkedHashMap[Int, Batch]()
-
10  val accumulator = context.actorOf(Props(new Accumulator(self)))
-
-   def receive = {
-     case RequestBatch =>
-       if (pages.hasNext) {
15       val batch = Batch(nextId, pages.take(batchSize).toVector, accumulator)
-        pending(nextId) = batch
-        sender ! batch
-        nextId += 1
-      } else {
20       val (id, batch) = pending.head  // The oldest pending item
-        pending -= id                   // Remove and re-add so it's now
```

```
-         pending(id) = batch             // the youngest
-         sender ! batch
-       }
25
-     case Processed(id) =>
-       pending.remove(id)
-       if (!pages.hasNext && pending.isEmpty)
-         context.system.shutdown
30   }
- }
```

This differs from the parser we created yesterday in a number of ways. Firstly, it maintains a record of pending work (line 8)—batches that have been sent to a consumer, but not yet fully processed. It also creates the Accumulator instance, passing it a reference to itself (line 10).

When it receives a RequestBatch message, there are two cases to consider:

- If there are still unparsed pages available (line 15) it builds a batch by parsing pages, records that batch as pending, and sends it to the consumer that made the request.
- If there are no unparsed pages available (line 20) it sends the oldest pending batch to the consumer that made the request.

Why this second case? Surely every pending batch will eventually be processed? What do we gain by sending it to another consumer?

What we gain is fault tolerance. If a consumer exits, or the network goes down, or the machine it's running on dies, we'll just end up sending the batch it was processing to another consumer. Because each batch has an id associated with it, we know which batches have been processed and won't double-count.

To convince yourself, try starting a cluster and verify that:

- You can add parsers and counters to a cluster in any order, and everything "just works."
- If you pull the network cable out the back of a machine running counters, or kill the process they're running in, the remaining counters continue to process pages, including those that were in progress on that machine.

This is a great example of the benefits of concurrent, distributed development. This program will hardly miss a beat when faced with a hardware failure that would kill a normal sequential or multi-threaded program.

Day 3 Wrap-Up

This brings us to the end of day 3, and our discusion of programming with actors.

What We Learned in Day 3

Because actors share no data and communicate through message passing, they naturally map on to a distributed architecture. We saw how to create a cluster in which actors can:

- Be notified of changes to the cluster, including when new members join.
- Look up actor instances running on other cluster members.
- Send messages to, and receive messages from, remote actors.
- Detect when remote actors fail.

Day 3 Self-Study

Find

- What guarantees does Akka make regarding message delivery? How do the guarantees for local messages differ from remote messages?
- How does Akka detect remote failures?

Do

- The fault tolerant word count program we developed can handle failure of a counter or the machine that it's running on, but not the parser or accumulator. Create a version that can handle failure of *any* actor or node.
- Research Akka's support for testing clusters and write tests for our word count program.

Wrap-Up

Alan Kay, the designer of Smalltalk and father of object-oriented programming, had this to say on the essence of object-orientation:[4]

> I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.
>
> The big idea is "messaging" ... The Japanese have a small word—ma—for "that which is in between"—perhaps the nearest English equivalent is "interstitial." The

4. http://c2.com/cgi/wiki?AlanKayOnMessaging

> key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

This captures the essence of actor programming very well—we can think of actors as the logical extension of object-oriented programming to the concurrent world. Indeed, you can think of actors as more object-oriented than objects, with stricter message passing and encapsulation.

Strengths

Actors have a number of features that make them ideal for solving a wide range of concurrent problems.

Messaging and Encapsulation

Actors do not share state and, although they run concurrently with each other, within a single actor everything is sequential. This means that we need only worry about concurrency when considering message flows between actors.

This is a huge boon to the developer. An actor can be tested in isolation and, as long as our tests accurately represent the types of, and order in which messages might be delivered, we can have high confidence that it behaves as it should. And if we do find ourselves faced with a concurrency-related bug, we know where to look—the message flows between actors.

Fault Tolerance

Fault tolerance is built into actor programs from the outset. This not only enables more resilient programs, but also (through the "let it crash" philosophy) simpler and clearer code.

Distributed Programming

Actors' support for both shared and distributed memory architectures brings a number of significant advantages:

Firstly, it allows an actor program to scale to solve problems of almost any size. We are not limited to problems that fit on a single system.

Secondly, it allows us to address problems where geographical distribution is an intrinsic consideration. Actors are an excellent choice for programs where different elements of the software need to reside in different geographical locations.

Finally, distribution is a key enabler for resilient and fault-tolerant systems.

Weaknesses

As with threads and locks, actors provide no direct support for parallelism. Parallel solutions need to be built from concurrent building blocks, raising the specter of non-determinism. And because actors do not share state, and can only communicate through message passing, they are not a suitable choice if you need fine-grained parallelism.

Other Languages

As with most good ideas, the actor model is not new—it was first described in the 1970s, most notably by Carl Hewitt. The language that has done most to popularize actor programming, however, is unquestionably Erlang.[5] In particular, Erlang's creator Joe Armstrong is the originator of the "let it crash" philosophy and its VM is specifically constructed to facilitate fault tolerance.

Most popular programming languages now have an actor library available. Indeed, as well as being used in Scala, the Akka toolkit can be used to add actor support to Java.

There are also languages that, although not strictly actor languages, make heavy use of message-passing to support concurrency and have much in common with actor programming. The most prominent such language is Go.[6]

Final Thoughts

The difficulties with multi-threaded programming arise from shared mutable state. Actor programs avoid those difficulties by avoiding shared mutable state altogether—actors share no data.

In the next chapter, we'll see an approach that goes even further, avoiding not only shared mutable state, but all mutable state.

5. http://www.erlang.org
6. http://golang.org