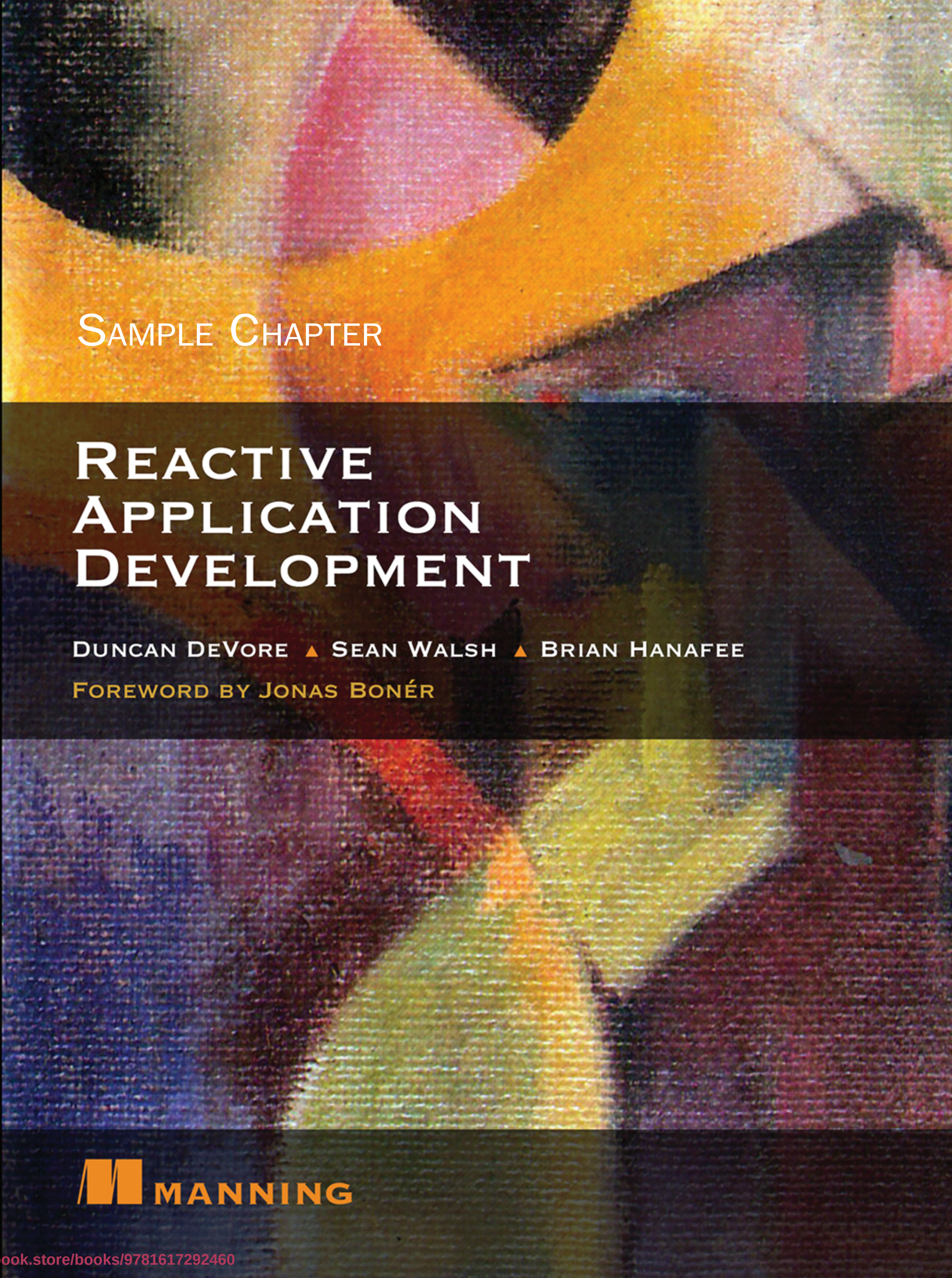

REACTIVE APPLICATION DEVELOPMENT

DUNCAN DEVORE ▲ SEAN WALSH ▲ BRIAN HANAFEE

FOREWORD BY JONAS BONÉR

MANNING

Reactive Application Development

by Duncan DeVore, Sean Walsh and Brian Hanafee

Sample Chapter 2

brief contents

2 Getting started with Akka

This chapter covers

- Building an actor system
- Distributing and scaling horizontally
- Applying reactive principles

You understand from chapter 1 the tenets of reactive design, but haven't yet seen them in practice. This chapter changes that situation. In this chapter, you build a simple reactive system by using the *actor* model that was introduced in chapter 1. The actor model is one of the most common reactive patterns. Actors can send and receive messages, make local decisions, create new actors, and do all that asynchronously and without locks. You build the example in this chapter with the Akka toolkit, which you also saw previously. Akka is a powerful system for creating and running actors. It's written in the Scala language, and the examples in this chapter are also written in Scala. Chapters 3 and 4 explain Akka in more depth.

The system you build consists of two actors passing messages to each other; you can use the same skills to create much larger applications. Next, you'll learn to scale the system horizontally by adding more copies of one of the actors. Finally, you'll see how this approach produces a system that's both message-driven and elastic—two of the four reactive properties from the Reactive Manifesto.

2.1 Understanding messages and actors

Reactive systems are message-driven, so it comes as no surprise that messages play a key role. Actors and messages are the building blocks of an actor system. An actor receives a message and does something in response to it. That something might include performing a computation, updating internal state, sending more messages, or perhaps initiating some I/O.

Much the same could be said of an ordinary function call. To understand what an actor is, it's useful first to consider some of the problems that can arise from an ordinary function call. A function accepts some input parameters, performs some processing, and returns a value. The processing may be quick or could take a long time. However long the processing takes, the caller is blocked while waiting for the return value.

2.1.1 Moving from functions to actors

If a function includes an I/O operation, control of the processor core most likely is handed off to another thread while the caller is waiting for a response. The caller won't be able to continue processing until the I/O operation is complete and the scheduler hands control back to the original processing thread, as shown in figure 2.1. The scheduling maintains the illusion for the caller that it made a simple synchronous call. What happened was that any number of other threads may have been running in the background, potentially even changing data structures referenced by the original input parameters.

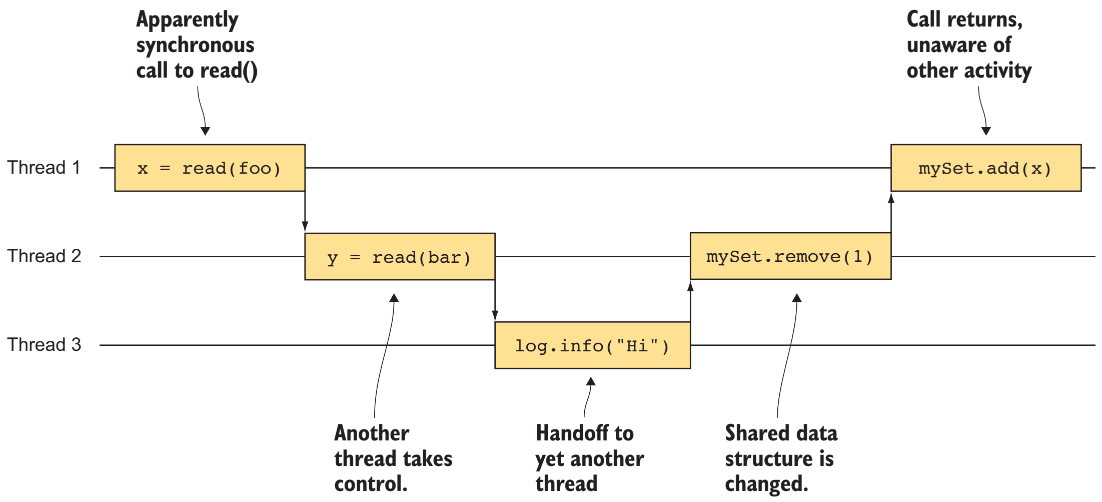
Apparently synchronous call to read()
Call returns, unaware of other activity
Thread 1
x = read(foo)
mySet.add(x)
Thread 2
y = read(bar)
mySet.remove(1)
Thread 3
log.info("Hi")
Another thread takes control.
Handoff to yet another thread
Shared data structure is changed.

Figure 2.1 The illusion of a synchronous call can be the source of unexpected behavior.

The developer may know that the function is liable to take a long time and may design the system to accommodate thread safety and timing. Sometimes, however, the developer can't predict the amount of time that the function requires. If the function has a cache of recently used data in memory but must go to a database if the data isn't in the cache, for example, the amount of time that the function requires may vary by many orders of magnitude from one call to the next. Ensuring that the caller and callee have the correct synchronization and thread safety without deadlocks can be extremely difficult. If the application programming interface (API) is properly encapsulated, the entire implementation may be replaced by one that has different characteristics. An excellent design around the original characteristics could become an inappropriate design with respect to the replacement.

The result is often complex code littered with exception handlers, callbacks, synchronized blocks, thread pools, timeouts, mysterious tuning parameters, and bugs that developers never seem to be able to replicate in the test environment. What all these things have in common is that they have nothing to do with the business domain. Rather, they're aspects of the computing domain imposing themselves on the application.

The actor model pushes these concerns out of the business domain and into the actor system.

Actors are asynchronous message handlers

The simplified view of an actor shown in figure 2.2 is a receive function that accepts a message and produces no return value; it processes each message as each message is received from the actor system. The actor system manages a mailbox of messages addressed to the actor, ensuring that the actor has to process only one message at a time. An important consequence of this design is that the sender never calls the actor directly. Instead, the sender addresses a message to the actor and hands it to the actor system for delivery.

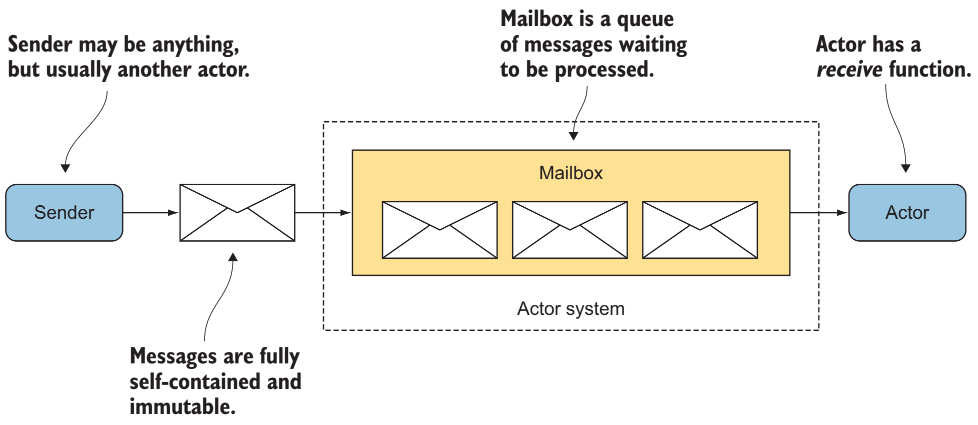
Sender may be anything, but usually another actor.
Mailbox is a queue of messages waiting to be processed.
Actor has a *receive* function.
Sender
Mailbox
Actor
Actor system
Messages are fully self-contained and immutable.

Figure 2.2 The sender obtains a *reference* to address messages to the actor through the actor system.

Actors remove many of the problems of function calls by abandoning the illusion that everything is synchronous. Instead, actors take the approach that everything is one-way and asynchronous. The underlying system takes responsibility for delivering the message to the receiving actor, immediately or some time later. An actor receives a new message only when it's ready to process that message. Until then, the actor system holds on to the message. The sender may proceed immediately to other tasks rather than wait for a response that may come some time later or perhaps not at all. If the receiving actor has a response for the sender, that response is handled with another asynchronous message.

TIP Senders never call actors directly. All interactions between senders and actors are mediated by the actor system.

MESSAGES ARE SELF-CONTAINED AND IMMUTABLE

As messages are passed among actors, they may move to an actor system on a different server. Messages must be designed so that they can be copied from one system to another, which means that all the information has to be contained within the message itself. Messages can't include references to data outside the message. Sometimes, a message doesn't make it to the destination and must be sent again. In chapter 4, you learn that the same message may be broadcast to more than one actor.

For this process to work, a message must be *immutable*. When it's sent, it's read-only and can't be allowed to change. If the message did change after being sent, there'd be no way to know whether the change happened before or after it was received, or perhaps while it was being processed by another actor. If the message happened to have been sent to an actor on another server, the change may have been made before or after it was transmitted, and there'd be no way to know. Worse, if the message had to be sent more than once, some copies of the message may include the change, and some may not. Immutable messages make all those worries go away.

2.1.2 Modeling the domain with actors and messages

Actors should correspond to real things in the domain model. The example in this chapter consists of a tourist who has an inquiry about a country and a guidebook that provides guidance to the tourist.

The example actor system is shown in figure 2.3. The system contains two actors: a *tourist* and a *guidebook*. The tourist sends an *inquiry* message to the guidebook, and the guidebook sends *guidance* messages back to the tourist. Messages are one-way affairs, so the inquiry and the guidance are defined as separate messages. As in real life, the tourist must be prepared to receive no guidance, a single guidance message, or even multiple guidance messages in response to a single inquiry. (The tourist in the example can receive multiple guidance messages, but deciding what to believe would be the subject of a different book.)

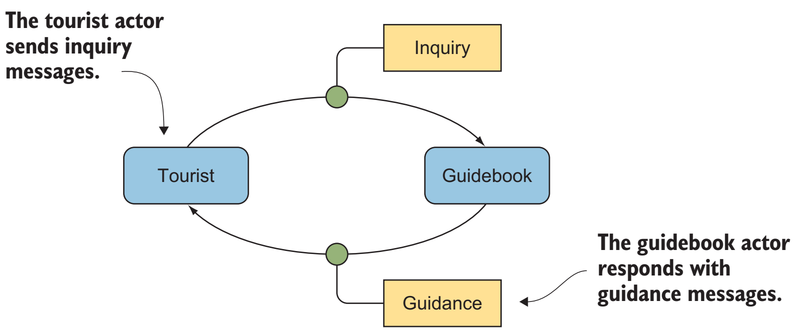
The tourist actor sends inquiry messages.
Inquiry
Tourist
Guidebook
Guidance
The guidebook actor responds with guidance messages.

Figure 2.3 The tourist actor sends inquiry messages to the guidebook. The guidebook actor responds with guidance messages returned to the tourist actor.

2.1.3 *Defining the messages*

You know already that messages are self-contained and immutable, and now you've identified some messages that are needed for the example. In Scala, case classes provide an easy way to implement messages. The example messages, shown in listing 2.1, define a case class for each message. The definitions follow the convention that messages are defined in the companion object to the actor that receives the message. The `Guidebook` actor receives `Inquiry` messages that include a `String` for the country code that's being inquired about. The `Tourist` actor receives `Guidance` messages, which contain the original country code and a description of the country. The original country code is included in the guidance so that the information in the message is fully self-contained. Otherwise, there'd be no way to correlate which guidance goes with which inquiry. Finally, the `Start` message is used later to tell the `Tourist` actor to start sending inquiries.

Scala case classes

For readers who aren't familiar with Scala, a case class is defined by a class name and some parameters. By default, instances of a case class are immutable. Each parameter corresponds to a read-only value that's passed to the constructor. The compiler takes care of generating the rest of the boilerplate for you. The concise Scala definition

```
case class Inquiry(code: String)
```

produces a class equivalent in Java:

```
public class Inquiry {
   private final String code;
```

(continued)

```
    public Inquiry(String code) {
        this.code = code;
    }

    public String getCode() {
        return this.code;
    }

    // …more methods are generated automatically
}
```

Case classes generate more than the getters. They automatically include correct equality, hash codes, a human-friendly `toString`, a `copy` method, a companion object, support for pattern matching, and additional methods that are useful for more advanced functional programming techniques.

Listing 2.1 Message definitions

```
object Guidebook {
  case class Inquiry(code: String)
}
object Tourist {
  case class Guidance(code: String, description: String)
  case class Start(codes: Seq[String])
}
```

The Inquiry and Guidance messages are simple case classes.

The Start message starts the tourist sending inquiries.

Now that the messages are defined, it's time to move on to the actors.

2.1.4 Defining the actors

The example requires a `Tourist` actor and a `Guidebook` actor. Most of the behavior of an actor is provided by extending the `akka.actor.Actor` trait. One thing that can't be built into the actor trait is what to do when a message is received, because that behavior is specific to the application. You provide that behavior by implementing the abstract `receive` method.

THE TOURIST ACTOR

As shown in listing 2.2, the `receive` method on the `Tourist` defines cases to handle the two types of message expected by the `Tourist`. In response to a `Start` message, it extracts codes and sends an `Inquiry` message to the `Guidebook` actor for each one it finds. It also receives `Guidance` messages, which it handles by printing the code and description to the console.

The `Tourist` needs to address messages to the `Guidebook`, but actors never keep direct references to other actors. Notice that the `Guidebook` is passed to the constructor as an `ActorRef`, not an `Actor`. An `ActorRef` is a *reference* to an actor. Because an actor may be on a different server, having a direct reference isn't always possible. In addition, actor instances may come and go over the lifetime of the actor system. The reference provides a level of isolation that allows the actor system to manage those events and prevents actors from directly changing the state of other actors. All communication between actors must occur through messages.

Listing 2.2 `Tourist` actor

```
import akka.actor.{Actor, ActorRef}

import Guidebook.Inquiry
import Tourist.{Guidance, Start}

class Tourist(guidebook: ActorRef) extends Actor {

  override def receive = {
    case Start(codes) =>                          // Extracts the codes from the message
      codes.foreach(guidebook ! Inquiry(_))       // For each code, send an inquiry message to the guidebook by using the ! operator.
    case Guidance(code, description) =>
      println(s"$code: $description")             // Prints the guidance to the console
  }
}
```

The `!` operator

The use of the `!` operator to send messages from one actor to another may be confusing the first few times you encounter it. The method is defined by the `ActorRef` trait. Writing

```
ref ! Message(x)
```

is equivalent to writing

```
ref.!(Message(x))(self)
```

Both methods use the `self` value, which is an `ActorRef` provided by the `Actor` trait as a reference to itself. The `!` operator takes advantage of Scala infix notation and the fact that `self` is declared as an implicit value.

The Guidebook actor

The `Guidebook` in listing 2.3 is similar to the `Tourist` from listing 2.2. It processes one message: an `Inquiry`. When it receives an inquiry, the `Guidebook` uses a few classes built into the `java.util` package to generate a rudimentary description suitable for the example. Then it produces a `Guidance` message to send back to the tourist.

The `Guidebook` needs to address messages back to the tourist that sent the inquiry. An important difference between the `Guidebook` and the `Tourist` is how each actor acquires a reference to the other. In the `Tourist`, a fixed reference to the `Guidebook` is provided as a parameter to the constructor. Because many `Tourists` could be consulting the same `Guidebook`, that approach doesn't work here. It wouldn't make sense to tell a `Guidebook` in advance about every `Tourist` who might use it. Instead, the `Guidebook` sends the guidance message back to the same actor that sent the inquiry. The `sender` inherited from the `Actor` trait provides a reference back to the actor that sent the message. This reference can be used for simple request-reply messaging.

> **NOTE** Knowing that Akka is used for concurrent applications, you might expect the `sender` reference to be synchronized to prevent the receive processing for one message from inadvertently responding to the sender of another message. As you'll learn in chapters 3 and 4, the Akka design prevents this situation from happening. For now, rest assured that you don't need to worry about it.

Listing 2.3 `Guidebook` actor

```
import akka.actor.Actor

import Guidebook.Inquiry
import Tourist.Guidance

import java.util.{Currency, Locale}

class Guidebook extends Actor {
  def describe(locale: Locale) =
    s"""In ${locale.getDisplayCountry},
  ➥ ${locale.getDisplayLanguage} is spoken and the currency
  ➥ is the ${Currency.getInstance(locale).getDisplayName}"""

  override def receive = {
    case Inquiry(code) =>
println(s"Actor ${self.path.name}
➥ responding to inquiry about $code")
    Locale.getAvailableLocales.
    filter(_.getCountry == code).
    foreach { locale =>
      sender ! Guidance(code, describe(locale))
    }
  }
}
```

Uses Java built-in packages to produce a rather basic description

Prints a log message to the console

Finds every locale with a matching country code. This implementation is rather inefficient.

Sends the guidance back to the sender

Now that you have two complete actors and some messages to pass between them, you'll want to try it yourself. First set up your development environment to build and run an actor system.

2.2 Setting up the example project

The examples in this book are built with *sbt*, which is a build tool commonly used for Scala projects. The home page for the tool is www.scala-sbt.org; there, you can find instructions to install the tool for your operating system. The example code is available online. You can retrieve a copy of the complete example by using the command

```
git clone https://github.com/ironfish/reactive-application-development-scala
```

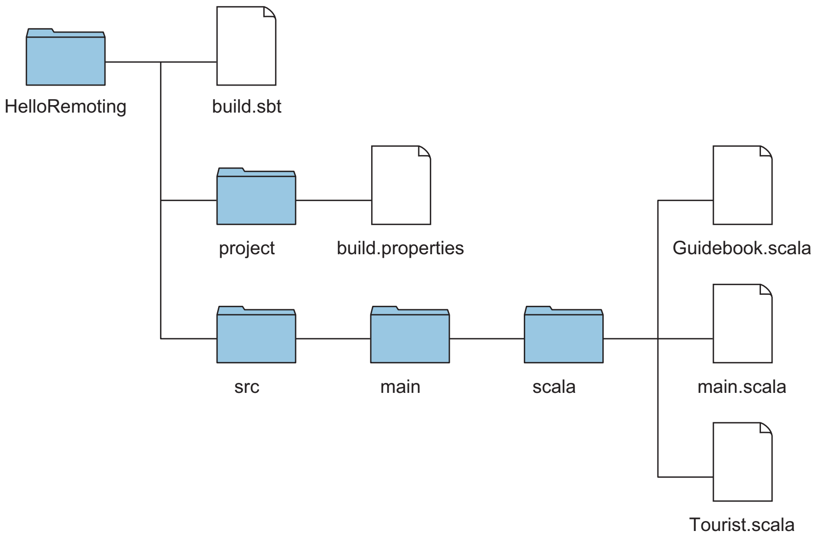

Figure 2.4 The layout of the sbt project follows the pattern used by other build systems, such as Maven and Gradle.

The source code is in the chapter2_001_guidebook_demo directory. The layout of the project is shown in figure 2.4 and is similar to that used by other build tools, such as Maven and Gradle. The project includes source code and the following files:

- *build.sbt*—Contains the build instructions
- *build.properties*—Tells sbt which version of sbt to use

As with other modern tools, sbt prefers convention to configuration. The build.sbt file shown in the following listing contains a project name and version, Scala version, repository URL, and a dependency on `akka-actor`.

Listing 2.4 build.sbt

```
name := "Guidebook"

version := "1.0"

scalaVersion := "2.12.3"

val akkaVersion = "2.5.4"

resolvers += "Lightbend Repository" at
➥ http://repo.typesafe.com/typesafe/releases/

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % akkaVersion
)
```

The example was tested with Scala 2.12.3.

Typesafe is now known as Lightbend, but the repository at typesafe.com is still supported.

The example was tested with Akka 2.4.8. The %% tells sbt to use a version of the library that was compiled for the version of Scala defined above.

The build.properties file allows sbt to use a different version of itself for each project. The default version is available by typing

```
sbt about
```

at the console. The complete one-line file is shown in the following listing.

Listing 2.5 build.properties

```
sbt.version=0.13.12
```

The example was tested with sbt 0.13.12.

You've already seen the source code for the messages and the two actors, which remain the same throughout this chapter. Whether the example actors are in one actor system or spread across multiple actor systems across many servers is determined entirely by configuration and the `Main` programs that drive the system. In the next section, you run both actors in one actor system. Then you learn to scale the system by using multiple actor systems.

2.3 Starting the actor system

Akka doesn't require much to get started. You create the actor system and add some actors; the Akka library does the rest. Sometimes, as shown in figure 2.5, it's useful to

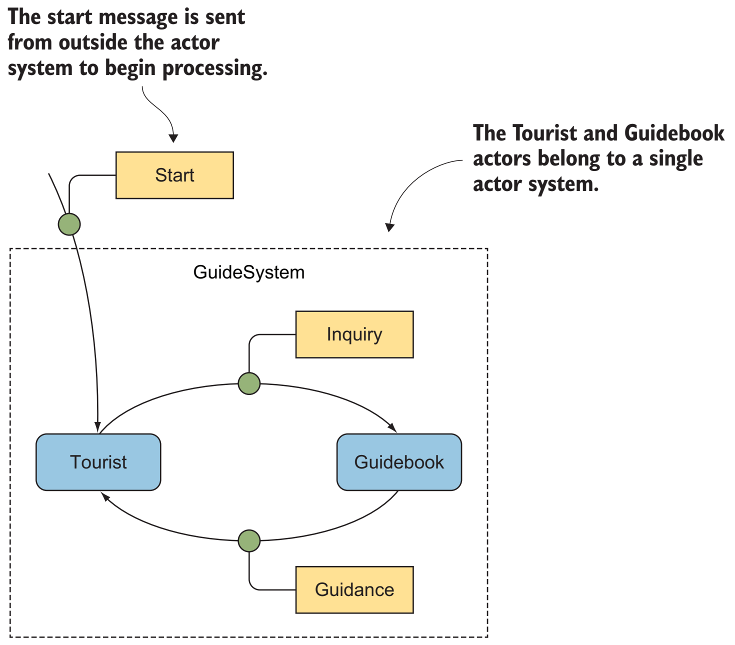
The start message is sent from outside the actor system to begin processing.
The Tourist and Guidebook actors belong to a single actor system.
Start
GuideSystem
Inquiry
Tourist
Guidebook
Guidance

Figure 2.5 Both the `Tourist` and `Guidebook` actors are deployed into the same actor system. The start message is sent from outside the actor system.

get things moving by sending a first message to the system. In chapters 3 and 4, you learn a bit more about what Akka is doing behind the scenes. To learn even more about the internals, see *Akka in Action*, by Raymond Roestenburg, Rob Bakker, and Rob Williams (Manning, 2016).

2.3.1 Creating the driver

The driver program shown in listing 2.6 does as expected: creates an actor system, defines the actors, and sends the `Start` message. The definition of the actors is interesting. Actor instances may come and go over the life of the actor system. The actor system takes responsibility for creating new instances, so it needs enough information to construct a new instance. That information is passed via `Props`. The steps are

1. Create a `Props` object that contains the class of the actor and the constructor parameters, if any.
2. Pass the `Props` to the `actorOf` method to create a new actor and assign it a name. This method is defined by the `ActorRefFactory` trait. That trait is extended by several classes, including `ActorSystem`, which is used in the example.
3. Record the `ActorRef` returned by `actorOf`. Callers don't receive a direct reference to the new actor.

Listing 2.6 The `Main` driver application

```
import java.util.Locale

import akka.actor.{ActorRef, ActorSystem, Props}        ◁ Akka library

import Tourist.Start                                     ◁ Start message shown previously

object Main extends App {
  val system: ActorSystem = ActorSystem("GuideSystem")  ◁ Creates the actor system that will contain the actors

  val guideProps: Props = Props[Guidebook]              ◁ Props define a recipe for creating instances of the Guidebook actor.

  val guidebook: ActorRef =
➥ system.actorOf(guideProps, "guidebook")              ◁ Creates an actor based on the Props, returning a reference to the actor

  val tourProps: Props =
➥ Props(classOf[TouristActor], guidebook)              ◁ The Props for a Tourist include a reference to the Guidebook actor.

  val tourist: ActorRef = system.actorOf(tourProps)     ◁ Creates an actor based on the Props, returning a reference to the actor

  tourist ! messages.Start(Locale.getISOCountries)      ◁ Sends an initial message to the tourist actor
}
```

There's nothing special about the driver. Because it extends the `App` trait, it automatically has a `Main` function, like any other Scala or Java application.

2.3.2 Running the driver

The output of the build is a Java Archive (JAR) file. You could use sbt to generate the build and then use the `java` command to launch it, but during development, it's easier to let sbt take care of that job too. Use

```
sbt run
```

to build and launch the application. As with nearly any framework, building the application the first time may take a while, because the dependencies need to be downloaded. The sbt tool uses Apache Ivy (http://ant.apache.org/ivy) for dependency management, and Ivy caches the dependencies locally.

Here's the part you've been waiting for: if you built everything successfully, you should see the `Guidebook` printing a message for every inquiry it receives, and the `Tourist` printing concise travel guidance for every country. Congratulations! You've started your first actor system. A more sophisticated application would send another message to tell the actors to shut themselves down gracefully. For now, press Ctrl-C to stop the actor system.

2.4 Distributing the actors over multiple systems

Actors are lightweight objects. At about 300 bytes, the memory overhead required per actor is a small fraction of the stack space consumed by a single thread. It's possible to hold a lot of actors in a single Java virtual machine (JVM). At some point, a single JVM still isn't enough. The actors have to scale across multiple machines.

You've already put into practice the most important concepts that make distributed actors possible. Actors refer to one another only via actor references. The `Tourist` actor refers to the `Guidebook` by using an `ActorRef` supplied to the constructor, and the `Guidebook` actor refers to the `Tourist` only through the sender `ActorRef`. An `ActorRef` may refer to a local actor or to a remote actor, so both actors are already capable of working with distributed actor systems. Whether the references are to local or remote actors makes no difference to the code.

The first step you took toward making the messages work across multiple machines was making them immutable. It wouldn't be possible to change the content of a message after it's been sent from one machine to another. The remaining step toward making the messages fully self-contained is making them serializable, so that they can be transmitted and reconstructed by the actor system that receives the message. Once again, Scala case classes come to the rescue. As long as the properties within the case class can be serialized, the whole class can be serialized, too.

Finally, the system needs some way to resolve references to actors in remote actor systems. You learn how in the following sections.

2.4.1 Distributing to two JVMs

When the example moves from one to two JVMs, the actors and messages remain the same. What changes? The new version is shown in figure 2.6. The primary difference

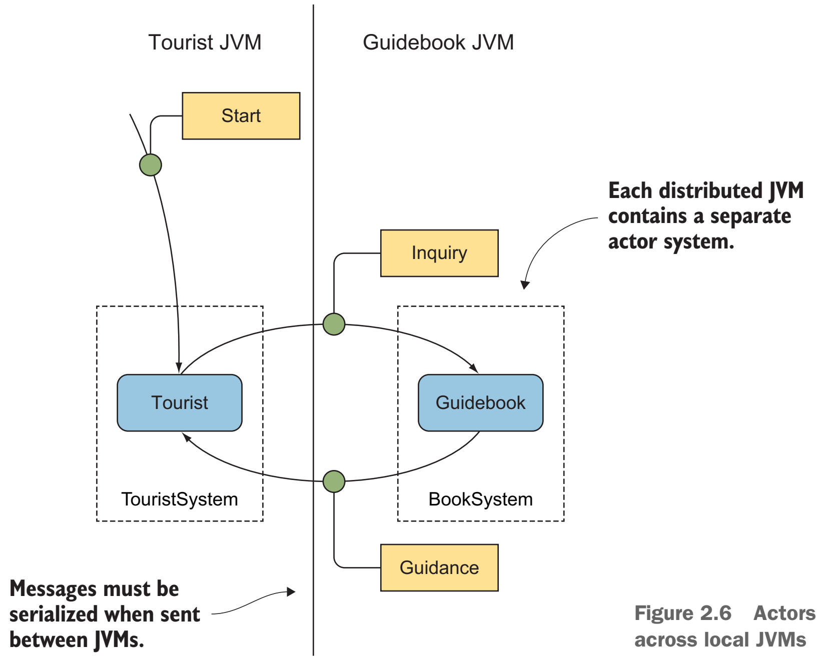
Tourist JVM
Guidebook JVM
Start
Each distributed JVM contains a separate actor system.
Inquiry
Tourist
Guidebook
TouristSystem
BookSystem
Guidance
Messages must be serialized when sent between JVMs.

Figure 2.6 Actors communicating across local JVMs

between figure 2.6 and the example shown in figure 2.5 is that figure 2.6 shows two actor systems. Each JVM needs its own actor system to manage its own actors.

If you cloned the original example from the Git repository, you can use the source in the chapter2_002_two_jvm directory.

2.4.2 Configuring for remote actors

As you might expect, distributing actors requires a little more setup than when everything is in one JVM. The process requires configuring an additional Akka library called `akka-remote`. The affected files are

- *build.sbt*—Adds the dependency on `akka-remote`
- *application.conf*—Provides some configuration information for remote actors

The change from the previous build.sbt example is nothing more than the inclusion of the additional library, as shown in the following listing.

Listing 2.7 build.sbt for remote actors

```
name := "Guidebook"

version := "1.0"

scalaVersion := "2.12.3"

val akkaVersion = “2.5.4”
```

```
resolvers += "Lightbend Repository" at "http://repo.typesafe.com/typesafe/
     releases/"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % akkaVersion,
  "com.typesafe.akka" %% "akka-remote" % akkaVersion
)
```

Adds dependency on remote actors. The Akka version numbers should match.

The configuration file shown in listing 2.8 is read automatically by Akka during startup. The tourist and guidebook JVMs in this example can use the same configuration file. More complex applications would require separate configuration files for each JVM, but the example is simple enough that one can be shared. The syntax is Human-Optimized Config Object Notation (HOCON), which is a JavaScript Object Notation (JSON) superset designed to be more convenient for humans to edit.

Listing 2.8 application.conf for remote actors

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = ${?PORT}
    }
  }
}
```

Replaces the default LocalActorRefProvider with the RemoteActorRefProvider

Enables remote communication by using the Transmission Control Protocol (TCP). Check the Akka documentation for other choices, such as Secure Sockets Layer (SSL) encryption.

The remote actors in the example will run on your local machine.

Obtains a port number from the PORT environment variable. If none is specified, the number defaults to 0, and Akka chooses a port automatically.

2.4.3 Setting up the drivers

Now that the configuration steps are complete, the next step is adding a program to act as a driver for the `Guidebook` actor system.

THE GUIDEBOOK DRIVER

The driver for the `Guidebook` actor system is a reduced version of the original driver for the entire system. Other than removing the `Tourist` actor, the only change is to provide unique names for the actor system and for the `Guidebook` actor. The names make it easier for the `Tourist` actor to obtain an `ActorRef` to the `Guidebook`. The complete code is shown in the following listing.

Listing 2.9 Driver for the `Guidebook` JVM

```
import akka.actor.{ActorRef, ActorSystem, Props}

object GuidebookMain extends App {
  val system: ActorSystem = ActorSystem("BookSystem")
```

Names the actor *system* uniquely

```
  val guideProps: Props =Props[Guidebook]                  Produces an ActorRef the same
  val guidebook: ActorRef =                                as in the single JVM example
➥ system.actorOf(guideProps, "guidebook")                 Names the actor uniquely
}
```

Now that the `Guidebook` driver is complete, you can move on to the `Tourist` driver.

The Tourist driver

The constructor for the `Tourist` actor requires a reference to the `Guidebook` actor. In the original example, this task was easy because the reference was returned when the `Guidebook` actor was defined. Now that the `Guidebook` actor is in a remote JVM, this technique won't work. To obtain a reference to the remote `Guidebook` actor, the driver

1. Obtains a URL-like path to the remote actor
2. Creates an `ActorSelection` from the path
3. Resolves the selection into an `ActorRef`

Resolving the selection causes the local actor system to attempt to talk to the remote actor and verify its existence. Because this process takes time, resolving the actor selection into a reference requires a timeout value and returns a `Future[ActorRef]`. You don't need to worry about the details of how a future works. For now, it's sufficient to understand that if the path resolves successfully, the resulting `ActorRef` is used as it was in the single JVM example. The complete driver is shown in listing 2.10.

NOTE The `scala.concurrent.Future[T]` used here isn't the same as a `java.util.concurrent.Future<T>`. It's closer to—though not the same as—the `java.util.concurrent.CompletableFuture<T>` in Java 8.

Listing 2.10 Driver for the `Tourist` JVM

```
import java.util.Locale


import akka.actor.{ActorRef, ActorSystem, Props}
import akka.util.Timeout
import tourist.TouristActor


import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.SECONDS
import scala.util.{Failure, Success}


object TouristMain extends App {                            Names the actor
  val system: ActorSystem = ActorSystem("TouristSystem")    system uniquely


                                                            Specifes the remote URL path
  val path =                                                for the Guidebook actor
    "akka.tcp://BookSystem@127.0.0.1:2553/user/guidebook"
    implicit val timeout: Timeout = Timeout(5, SECONDS)

    system.actorSelection(path).resolveOne().onComplete {
      case Success(guidebook) =>

        val tourProps: Props =
        ➥ Props(classOf[TouristActor], guidebook)
        val tourist: ActorRef = system.actorOf(tourProps)

        tourist ! messages.Start(Locale.getISOCountries)


      case Failure(e) => println(e)
    }
  }
```

Waits up to 5 seconds for the Guidebook to respond

Converts the path to an ActorSelection and resolves it

If the Guidebook is resolved successfully, continue as in the single JVM example.

If the Guidebook fails to resolve, fail with an error.

At this point, you have configuration and drivers to run the original `Tourist` and `Guidebook` actors in separate actor systems on separate JVMs. Notice that the messages and actors are unchanged from the original example, which isn't uncommon. Actors are designed to be distributable by default.

Now it's time to try the distributed actors.

2.4.4 Running the distributed actors

To run two JVMs, you need two command prompts. Start by opening a terminal session as you did for the single-actor system in section 2.3. This time, the `sbt` command line has to specify which `Main` class to use, because you have two. Recall that the application.conf file in listing 2.8 specifies that the listener port should be read from the `PORT` environment variable, so you have to specify the port as well.

Because the `Guidebook` waits forever for actors to contact it, but the `Tourist` waits for only a few seconds to find a `Guidebook`, the `Guidebook` is started first. The command line

The double quotes around the –D parameter are necessary in Windows but optional on other platforms.

```
sbt "-Dakka.remote.netty.tcp.port=2553" "runMain GuidebookMain"
```

should result in several messages to the console, ending with a log entry that tells you that the book system is now listening on port 2553.

Next open a second terminal window. The command line to run the tourist is almost the same:

```
sbt "runMain TouristMain"
```

The differences are the port number and the choice of `Main` class to run.

If everything has gone as expected, the `Tourist` should print the same `Guidebook` information as in the original example. Congratulations! You've created a distributed actor system.

As an exercise, try opening a third terminal and running another `Tourist` on a different port number. The code works because the `Guidebook` always responds to the *sender* of a message; it doesn't care whether one `Tourist` is sending a message or a thousand `Tourists` are sending messages. If you have thousands of `Tourists`, however, you may want to have more than one `Guidebook` actor too. In the next section, you learn how.

2.5 Scaling with multiple actors

Shortly after Akka 2.0 was released in 2012, a benchmark (http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single) demonstrated sending 50 million messages per second on a single machine—far more than a single `Guidebook` actor would be able to handle. Recall that the actor system guarantees that no more than one thread has access to the actor at a time. Eventually, there'd be too many incoming messages for a single actor, and it would be necessary to have multiple `Guidebook` actors to service all the requests.

Actor systems make adding multiple actors easy. An actor-based system handles scaling to multiple actors uniformly, whether the actors are local or remote. The additional `Guidebook` actors may run in the same JVM or in separate JVMs. In the rest of this chapter, you learn to put additional instances of the same actor in the same JVM; then you learn to scale horizontally to another JVM, which is a taste of things to come. Chapter 4 revisits these concepts in greater depth.

Before extending the actor system, take a look at how traditional systems that don't use actors approach the same problem.

2.5.1 Traditional alternatives

In a traditional system, scaling is handled quite differently, depending on the decision to put additional instances in the same JVM or to deploy them remotely. If the instances are in the same JVM, a system that doesn't use actors might instead use an explicit thread pool to balance requests, as shown in figure 2.7.

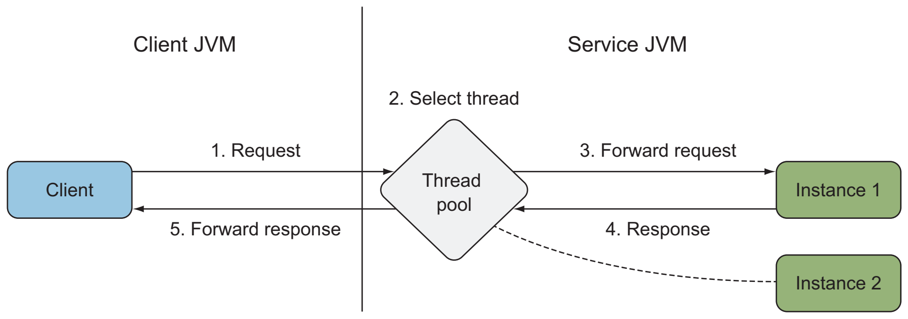

Figure 2.7 A thread pool can be used to manage access to multiple instances of a service.

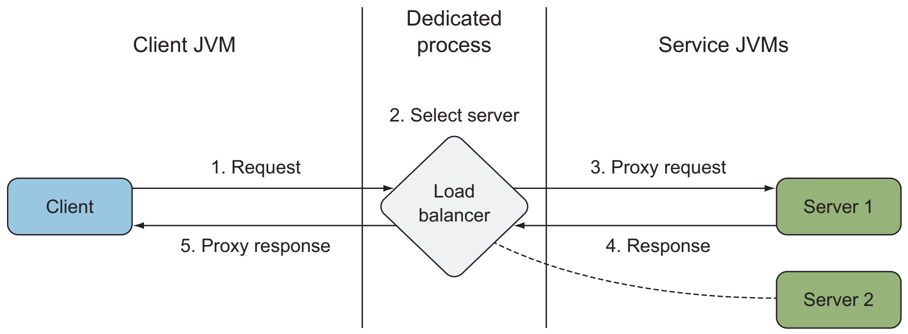

Figure 2.8 A traditional load balancer introduces a separate process between the client and server.

If the instances are in separate JVMs, the system could use a dedicated load balancer that sits between the client and service, as shown in figure 2.8. In most cases, the communication through the load balancer uses HTTP as the protocol.

Many excellent load-balancer implementations are available. HAProxy (www.haproxy.org) is a dedicated software solution, and NGINX (www.nginx.com) may be configured as a reverse proxy. Some companies even produce hardware solutions, such as the BIG-IP Local Traffic Manager from F5 Networks, Inc. Those solutions are outside the scope of this book, however, because they're not necessary. Instead, load balancing is handled by the actor system.

2.5.2 Routing as an actor function

In an actor-based system, the load balancer can be treated as an actor specialized for routing messages. The client treats the `ActorRef` to the router no differently than it treats a reference to the service itself. You've already seen that local and remote actors are treated uniformly, which continues to hold true here. The client doesn't need to concern itself with whether the router is local or remote. That decision can be made as part of system configuration, independent of how the client or service is coded.

Returning to the guidebook example, the `Tourist` actor sends an inquiry message to the router, the router selects a `Guidebook` actor, and the `Guidebook` sends the guidance directly back to the `Tourist`, all as shown in figure 2.9.

Recall from section 2.1 that the `Guidebook` actor sends its response back to the sender of a message. You may wonder how that process works when the message comes from the router rather than the original client. The answer is that the router doesn't pass a reference to itself as the sender. It forwards the *original* sender, so the routed message appears to have come directly from the client.

2.6 Creating a pool of actors

A single actor, such as the guidebook example, handles only one request at a time, which greatly simplifies coding, because the actor doesn't have to worry about synchronization. It also means that the `Guidebook` can become a bottleneck, because there's

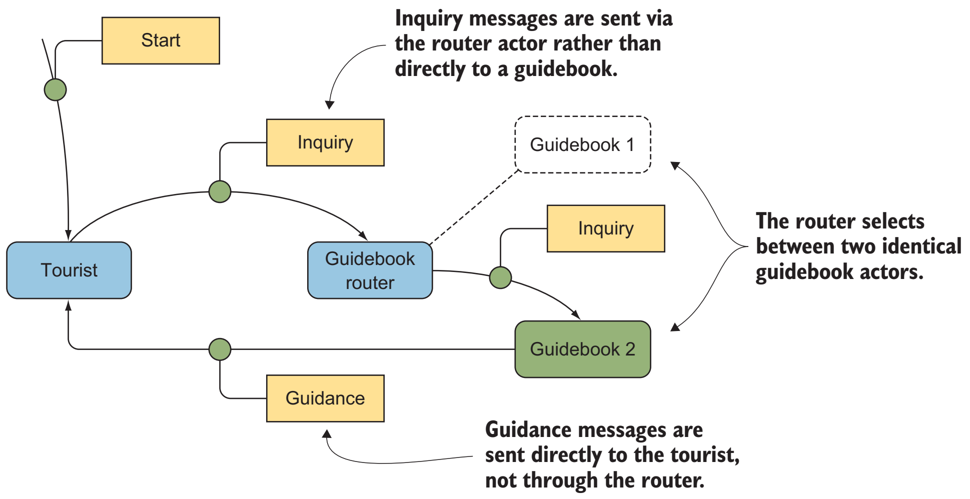
Start
Inquiry messages are sent via the router actor rather than directly to a guidebook.
Inquiry
Guidebook 1
Inquiry
The router selects between two identical guidebook actors.
Tourist
Guidebook router
Guidebook 2
Guidance
Guidance messages are sent directly to the tourist, not through the router.

Figure 2.9 Sending messages from the `Tourist` to the `Guidebook` through a router that performs load balancing

only one `Guidebook` and every request has to wait for it to become available. The simplest way to scale is to add a pool of `Guidebook` actors within the single-actor system and create a router to balance the inquiries. Figure 2.10 shows this approach.

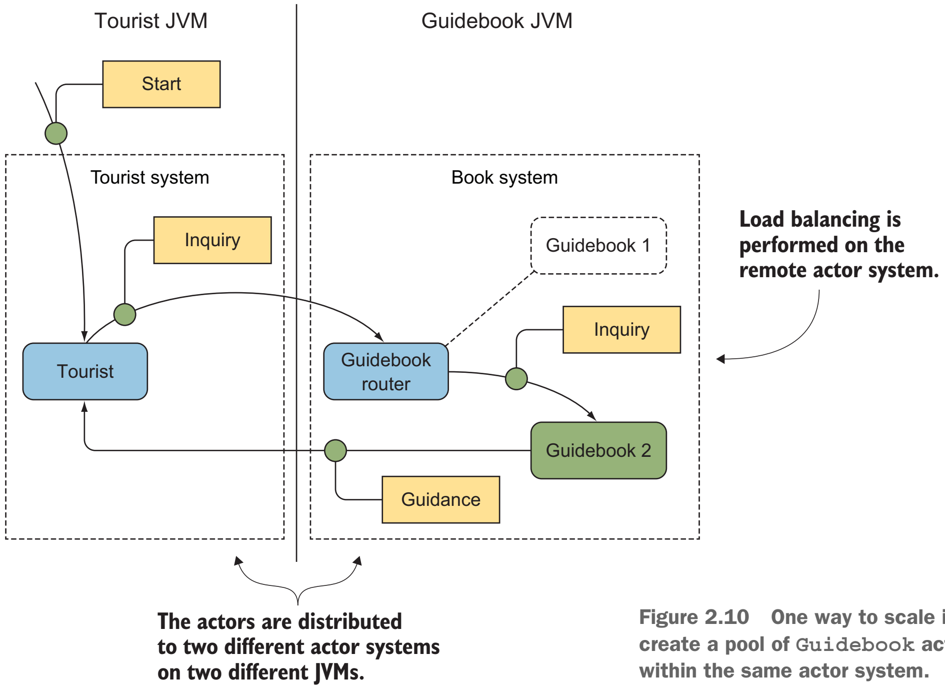
Tourist JVM
Guidebook JVM
Start
Tourist system
Book system
Inquiry
Guidebook 1
Load balancing is performed on the remote actor system.
Inquiry
Tourist
Guidebook router
Guidebook 2
Guidance
The actors are distributed to two different actor systems on two different JVMs.

Figure 2.10 One way to scale is to create a pool of `Guidebook` actors within the same actor system.

The Tourist and Guidebook actors remain unchanged from the previous examples. In fact, the entire tourist system remains the same. As you see in the next section, only the guidebook system needs to be changed to incorporate the change.

If you cloned the original example from the Git repository, you can use the source in the chapter2_003_pool directory, which uses two JVMs and an actor pool.

2.6.1 Adding the pool router

The *pool router* is an actor that takes the place of the original Guidebook actor. As with any other actor, you need a Props for the router actor. It's possible to configure an actor pool entirely within code, but it's preferable to use a configuration file. Akka routing includes a convenient FromConfig utility that tells Akka to configure a pool. The driver in the following listing passes the original guidebook Props to FromConfig so that Akka knows how to create new pool members, and everything else comes from the configuration file.

Listing 2.11 Driver for the guidebook JVM with a pool of guidebooks

```
import akka.actor.{ActorRef, ActorSystem, Props}
import akka.routing.FromConfig

object GuidebookMain extends App {
  val system: ActorSystem = ActorSystem("BookSystem")

  val guideProps: Props = Props[Guidebook]

  val routerProps: Props =
  ➥ FromConfig.props(guideProps)

  val guidebook: ActorRef =
  ➥ system.actorOf(routerProps, "guidebook")
}
```

Imports library to read the pool configuration from application.conf

Props for the Guidebook actor are unchanged.

Wraps the pool configuration around the original Props for the Guidebook actor

The name of the actor must match the name in the configuration file.

Akka includes several built-in pool routers. One of the most commonly used is the round-robin pool. This implementation creates a set number of instances of the actor and forwards requests to each actor in turn. Chapter 4 describes some of the other pool implementations.

The following listing shows how to configure a round-robin pool containing five instances of the Guidebook actor. These instances are called *routees*.

Listing 2.12 application.conf with a pool of Guidebook actors

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    deployment {
      /guidebook {
```

Configures a pool for the Guidebook actor

```
        router = round-robin-pool
        nr-of-instances = 5
      }
    }
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = ${?PORT}
    }
  }
}
```

Uses the built-in round-robin pool implementation with five instances in the pool

As you can see, Akka makes it easy to create many actors in a pool. You create a pool actor, give it the Props needed to create new pool entries, and configure the pool as needed through the configuration file.

2.6.2 Running the pooled actor system

Running the pool of actors is easy, too. The process is the same as running the two-actor system example shown previously in this section. As before, the command line

```
sbt "-Dakka.remote.netty.tcp.port=2553" "runMain GuidebookMain"
```

starts the guidebook system, and the command line

```
sbt "runMain TouristMain"
```

starts the tourist system, which is unchanged. The output on the tourist console should be the same as before. The difference is on the guidebook console. Before the pool was added, each inquiry resulted in the Guidebook actor's printing a line such as

```
Actor guidebook responding to inquiry about AD
```

Now that the actor named Guidebook is a router actor, each instance of the Guidebook actor in the pool is assigned a different, random name. The inquiries now result in each Guidebook actor's printing a line such as

```
Actor $a responding to inquiry about AD
```

Because five actors are configured, the console output should show five different names for the actor, such as $a, $b, $c, $d, and $e.

> **NOTE** The round-robin pool is one of several pool implementations included with Akka. You can try some of the other types by changing the configuration file. Other choices to try include random-pool, balancing-pool, smallest-mailbox-pool, scatter-gather-pool, and tail-chopping-pool.

In this section, you scaled an actor system by replacing a single actor with a router with a pool of identical actors. In the next section, you apply the same concepts to distribute messages across multiple-actor systems on multiple JVMs.

2.7 Scaling with multiple-actor systems

The router actor is responsible for keeping track of the routees that handle the messages that it receives. The pool routers in the preceding section handled this task by creating and managing the routees themselves. An alternative approach is to provide a group of actors to the router but manage them separately. This approach is similar to how a traditional load balancer works. The difference is that a traditional load balancer uses a dedicated process to manage the group membership and perform the routing, whereas in the actor-based system, the routing may be performed by a router actor within the client, as shown in figure 2.11.

The two approaches aren't mutually exclusive. It's reasonable to have a router on the client select a remote actor system to service the request and then have another tier of routing in the service actor system select a specific actor instance from a pool. The response message still flows from the service actor directly to the original client actor.

If you cloned the original example from the Git repository, you can use the source in the chapter2_004_group directory, which keeps the pool router on the guidebook systems and adds a group router to the tourist system.

2.7.1 Adding the group router

The driver for the `Tourist` actor system is simpler with a group than in the initial system with a single remote actor. In the original example (listing 2.10), the driver

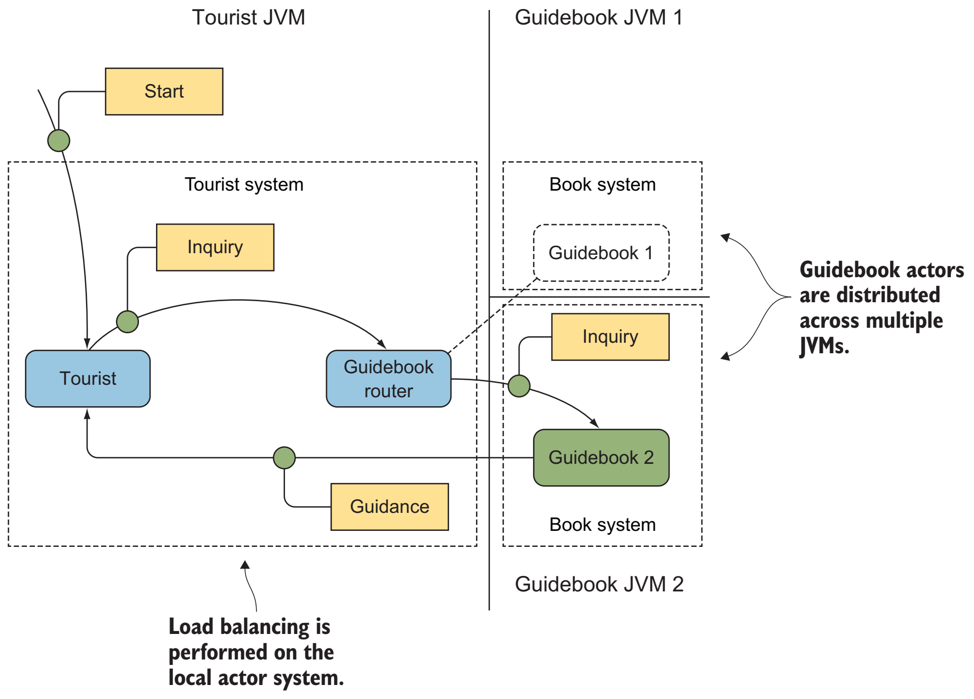

Figure 2.11 A local router can balance requests among actors on remote systems.

resolved the remote actor path by using a `Future[ActorRef]`, and the system waited for confirmation that the remote actor system had been verified before creating the `Tourist` actor. With a group router, all that work is handled by the group router, as shown in the following listing.

Listing 2.13 Driver for the `Tourist` JVM with a group of `Guidebook` systems

```
import java.util.Locale

import akka.actor.{ActorRef, ActorSystem, Props}
import akka.routing.FromConfig

import Tourist.Start

object TouristMain extends App {
  val system: ActorSystem = ActorSystem("TouristSystem")

  val guidebook: ActorRef =
    system.actorOf(FromConfig.props(), "balancer")

  val tourProps: Props =
    Props(classOf[Tourist], guidebook)

  val tourist: ActorRef = system.actorOf(tourProps)

  tourist ! Start(Locale.getISOCountries)
}
```

Imports library to read the pool configuration from application.conf

Uses a different name to distinguish this router from the router pool used by the Guidebook driver

The remaining steps are the same as the single JVM driver shown in listing 2.6.

Configuration of the router group is handled by the configuration file.

The group configuration for the balancer uses the round-robin-*group* rather than the round-robin-*pool*, as shown in the following listing. Group routers expect the routees to be provided, and they're provided via `routees.paths`. Some of the other group implementations are described in Chapter 4.

Listing 2.14 application.conf with a group of guidebook systems

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    deployment {
      /guidebook {
        router = round-robin-pool
        nr-of-instances = 5
      }
      /balancer {
        router = round-robin-group
        routees.paths = [
          "akka.tcp://BookSystem@127.0.0.1:2553/user/guidebook",
          "akka.tcp://BookSystem@127.0.0.1:2554/user/guidebook",
          "akka.tcp://BookSystem@127.0.0.1:2555/user/guidebook"]
      }
    }
```

Leave alone the pool for the guidebook actor. It will continue to be used by the guidebook.

Creates a round-robin group router named balancer with three group members

```
    }
    remote {
      enabled-transports = ["akka.remote.netty.tcp"]
      netty.tcp {
        hostname = "127.0.0.1"
        port = ${?PORT}
      }
    }
  }
```

The rest of the configuration file remains the same.

2.7.2 Running the multiple actor systems

The configuration in the preceding section instructed the balancer to look for three guidebooks by contacting the `BookSystem` actor systems listening on ports 2553, 2554, and 2555. Open three terminal windows, and start those three systems by using the following commands:

```
sbt "-Dakka.remote.netty.tcp.port=2553" "runMain GuidebookMain"
sbt "-Dakka.remote.netty.tcp.port=2554" "runMain GuidebookMain"
sbt "-Dakka.remote.netty.tcp.port=2555" "runMain GuidebookMain"
```

Run each of these commands in a separate terminal.

Next run the tourist system on a fourth terminal:

```
sbt "runMain TouristMain"
```

You should see the usual guidance on the tourist console.

Now switch to each of the terminal windows running the `guidebook` instances. You should see that all three of them have responded to some messages, and you should be able to verify that each received requests for different countries.

The pools within each of these actor systems are still in place, too, so the distributed actor system you're running now includes 16 actors: 1 `Tourist` actor plus 3 `Guidebook` systems that each have a pool of 5 `Guidebook` actors. That's quite a bit for such a small amount of code.

2.8 Applying reactive principles

The same `Tourist` and `Guidebook` actors remained unchanged throughout this chapter. At this point, both of the driver programs take their configuration from application.conf rather than having anything hardcoded. The complete system requires surprisingly little code. The two actors and their drivers are less than 100 lines of Scala, yet this system exhibits reactive attributes.

The four attributes that lay the foundation for reactive applications were introduced through the Reactive Manifesto in chapter 1. Reactive applications are message-driven, elastic, resilient, and responsive.

The examples in this chapter are *message-driven*. Everything occurs in response to the same three immutable messages. All communication among actors is accomplished asynchronously, and there's never a direct function call from one actor to another. Only the actor system calls the actor to pass it a message. The message passing exhibits location transparency. The sender doesn't concern itself with whether the recipient of a message is local or remote.

Location transparency also allows routers to be injected into the message flow, which helps achieve another reactive attribute. The application is *elastic*, too, capable of applying more resources by expanding a pool of local actors and capable of expanding remotely by adding remote actors.

In other words, you now have an elastic, message-driven system that you can scale horizontally without writing any new code, which is quite an accomplishment. Feel free to experiment with the configuration.

If you experiment, you'll find that that the system has some attributes of resilience and responsiveness, but it could be better. Chapters 3 and 4 show you the additional pieces needed to create a fully reactive application. You'll learn more about the design of Akka and how the components work together to deliver the underpinnings of the system you created in this chapter.

Summary

- Actors have a *receive* function that accepts a message and doesn't have a return value.
- Actors are called by the actor *system*, not directly by other actors. The actor system guarantees that there's never more than one thread at a time calling an actor's `receive` function, which simplifies the `receive` function because it doesn't have to be thread-safe.
- Actors are distributable by default. The same actors and the same messages were used throughout this chapter. Only the drivers and configuration changed as the example evolved from a pair of actors exchanging messages within a single JVM all the way to 16 actors distributed across 3 JVMs.
- Immutable messages flow among actors. Immutable messages are thread-safe and safe to copy, which is necessary when a message is serialized and sent to another actor system. Scala case classes offer a safe, easy way to define immutable messages.
- Senders address messages by using an `ActorRef`, which is obtained from the actor system. An `ActorRef` may refer to a local or a remote actor.
- Actor systems can be scaled via a router to balance requests among multiple actors. A router may be configured as a pool that creates and manages the actor instances for you. A group router requires the actors to be created and managed separately.

SOFTWARE DEVELOPMENT/JAVA

Reactive Application Development

DeVore ▲ Walsh ▲ Hanafee

Free eBook
SEE FIRST PAGE

Mission-critical applications have to respond instantly to changes in load, recover gracefully from failure, and satisfy exacting requirements for performance, cost, and reliability. That's no small task! Reactive designs make it easier to meet these demands through modular, message-driven architecture, innovative tooling, and cloud-based infrastructure.

Reactive Application Development teaches you how to build reliable enterprise applications using reactive design patterns. This hands-on guide begins by exposing you to the reactive mental model, along with a survey of core technologies like the Akka actors framework. Then, you'll build a proof-of-concept system in Scala, and learn to use patterns like CQRS and Event Sourcing. You'll master the principles of reactive design as you implement elasticity and resilience, integrate with traditional architectures, and learn powerful testing techniques.

What's Inside

- Designing elastic domain models
- Building fault-tolerant systems
- Efficiently handling large data volumes
- Examples can be built in Scala or Java

Written for Java or Scala programmers familiar with distributed application designs.

Duncan DeVore, **Sean Walsh**, and **Brian Hanafee** are seasoned architects with experience building and deploying reactive systems in production.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/reactive-application-development

"Packed with hard-won wisdom and practical advice that will set you on the path toward effective reactive application development."
—From the Foreword by Jonas Bonér
Creator of Akka

"The ultimate reference on reactive application development, with Scala code using Akka as a bonus!"
—Jean-François Morin, Laval University

"Explains complex concurrency problems in simple words with lots of realistic examples."
—Shabeesh Balan
Prime Focus Technologies

"Very well written and reflects the authors' expertise ... teaches you how to build modern distributed applications using reactive design patterns."
—Subhasis Ghosh, Alliant Credit Union

ISBN-13: 978-1-61729-246-0
ISBN-10: 1-61729-246-X
54999
9 781617 292460

MANNING US $ 49.99 | Can $ 65.99