If you are using a Unix system, you can load the list of words, separated by a newline character, from the file in the `/usr/share/dict/words` location. Alternatively, download the source code for this book and find the `words.txt` file, or create a dummy file with several words, and save it to the `src/main/resources/org/learningconcurrency/` directory. You can then test the correctness of the `DictionaryActor` class using the following program:

```
val dict = ourSystem.actorOf(Props[DictionaryActor], "dictionary")

dict ! DictionaryActor.IsWord("program")
Thread.sleep(1000)

dict ! DictionaryActor.Init("/org/learningconcurrency/words.txt")
Thread.sleep(1000)
```

The first message sent to the actor results in an error message. We cannot send an `IsWord` message before initializing the actor. After sending the `Init` message, we can check if words are present in the dictionary. Finally, we send an `End` message and shut down the actor system, as shown in the following code snippet:

```
dict ! DictionaryActor.IsWord("program")
Thread.sleep(1000)

dict ! DictionaryActor.IsWord("balaban")
Thread.sleep(1000)

dict ! DictionaryActor.End
Thread.sleep(1000)

ourSystem.shutdown()
```

Having learned about actor behaviors, we will study how actors are organized into a hierarchy in the following section.

Akka actor hierarchy

In large organizations, people are assigned roles and responsibilities for different tasks in order to reach a specific goal. The CEO of the company chooses a specific goal, such as launching a software product. He then delegates parts of the work tasks to various teams within the company-the marketing team investigates potential customers for the new product, the design team develops the user interface of the product, and the software engineering team implements the logic of the software product. Each of these teams can be further decomposed into sub-teams with different roles and responsibilities, depending on the size of the company. For example, the software engineering team can be composed into two developer sub-teams, responsible for implementing the backend of the software product, such as the server-side code, and the frontend, such as the website or a desktop UI.

Similarly, sets of actors can form hierarchies in which actors that are closer to the root work on more general tasks and delegate work items to more specialized actors lower in the hierarchy. Organizing parts of the system into hierarchies is a natural and systematic way to decompose a complex program into its basic components. In the context of actors, a correctly chosen actor hierarchy can also guarantee better scalability of the application, depending on how the work is balanced between the actors. Importantly, a hierarchy between actors allows isolating and replacing parts of the system that fail more easily.

In Akka, actors implicitly form a hierarchy. Every actor can have some number of child actors, and it can create or stop child actors using the `context` object. To test this relationship, we will define two actor classes to represent the parent and child actors. We start by defining the `ChildActor` actor class, which reacts to the `sayhi` messages by printing the reference to its parent actor. The reference to the parent is obtained by calling the `parent` method on the `context` object. Additionally, we will override the `postStop` method of the `Actor` class, which is invoked after the actor stops. By doing this, we will be able to see precisely when a child actor is stopped. The `ChildActor` template is shown in the following code snippet:

```
class ChildActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "sayhi" =>
      val parent = context.parent
      log.info(s"my parent $parent made me say hi!")
  }
  override def postStop() {
    log.info("child stopped!")
  }
}
```

We now define an actor class called `ParentActor`, which can accept the messages `create`, `sayhi`, and `stop`. When `ParentActor` receives a `create` message, it creates a new child by calling `actorOf` on the `context` object. When the `ParentActor` class receives a `sayhi` message, it forwards the message to its children by traversing the `context.children` list, and resending the message to each child. Finally, when the `ParentActor` class receives a `stop` message, it stops itself:

```
class ParentActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "create" =>
      context.actorOf(Props[ChildActor])
      log.info(s"created a kid; children = ${context.children}")
    case "sayhi" =>
      log.info("Kids, say hi!")
      for (c <- context.children) c ! "sayhi"
    case "stop" =>
      log.info("parent stopping")
      context.stop(self)
  }
}
```

We test the actor classes `ParentActor` and `ChildActor` in the following program. We first create the `ParentActor` instance, `parent`, and then send two `create` messages to `parent`. The `parent` actor prints that it created a child actor twice. We then send a `sayhi` message to `parent`, and witness how the child actors output a message after the parent forwards the `sayhi` message to them. Finally, we send a `stop` message to stop the `parent` actor. This is shown in the following program:

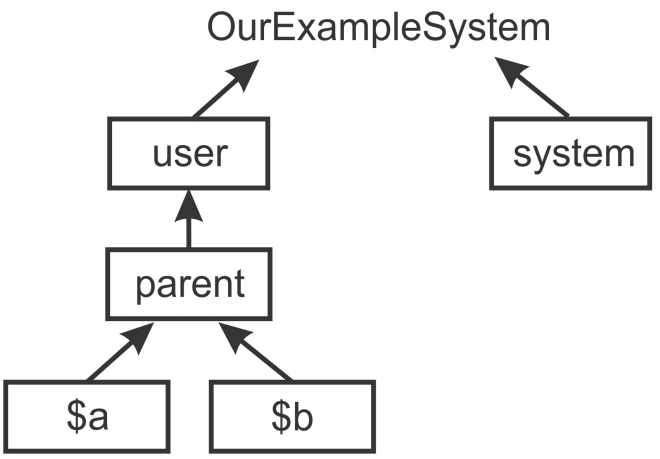
```
object ActorsHierarchy extends App {
  val parent = ourSystem.actorOf(Props[ParentActor], "parent")
  parent ! "create"
  parent ! "create"
  Thread.sleep(1000)
  parent ! "sayhi"
  Thread.sleep(1000)
  parent ! "stop"
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

By studying the standard output, we find that each of the two child actors output a `sayhi` message immediately after the `parent` actor prints that it is about to stop. This is the normal behavior of Akka actors-a child actor cannot exist without its parent. As soon as the parent actor stops, its child actors are stopped by the actor system as well.

When an actor is stopped, its child actors are also automatically stopped.

If you ran the preceding example program, you might have noticed that printing an actor reference reflects the actor's position in the actor hierarchy. For example, printing the child actor reference shows the `akka://OurExampleSystem/user/parent/$a` string. The first part of this string, `akka://`, denotes that this reference points to a local actor. The `OurExampleSystem` part is the name of the actor system that we are using in this example. The `parent/$a` part reflects the name of the parent actor and the automatically generated name `$a` of the child actor. Unexpectedly, the string representation of the actor reference also contains a reference to an intermediate actor, called `user`.

In Akka, an actor that resides at the top of the actor hierarchy is called the **guardian actor**, which exists to perform various internal tasks, such as logging and restarting user actors. Every top-level actor created in the application is placed under the `user` predefined guardian actor. There are other guardian actors. For example, actors internally used by the actor system are placed under the `system` guardian actor. The actor hierarchy is shown in the following figure, where the guardian actors `user` and `system` form two separate hierarchies in the actor system called `OurExampleSystem`:



In this section, we saw that Akka actors form a hierarchy, and learned about the relationships between actors in this hierarchy. Importantly, we learned how to refer to immediate neighbors of an actor using the `parent` and `children` methods of the `context` object. In the following section, we will see how to refer to an arbitrary actor within the same actor system.

Identifying actors

In the previous section, we learned that actors are organized in a hierarchical tree, in which every actor has a parent and some number of children. Thus, every actor lies on a unique path from the root of this hierarchy, and can be assigned a unique sequence of actor names on this path. The `parent` actor was directly beneath the `user` guardian actor, so its unique sequence of actor names is `/user/parent`. Similarly, the unique sequence of actor names for the `parent` actor's child actor is `$a` is `/user/parent/$a`. An **actor path** is a concatenation of the protocol, the actor system name, and the actor names on the path from the top guardian actor to a specific actor. The actor path of the `parent` actor from the previous example is `akka://OurExampleSystem/user/parent`.

Actor paths closely correspond to file paths in a filesystem. Every file path uniquely designates a file location, just as an actor path uniquely designates the location of the actor in the hierarchy. Just as a file path in a filesystem does not mean that a file exists, an actor path does not imply that there is an actor on that file path in the actor system. Instead, an actor path is an identifier used to obtain an actor reference if one exists. Also, parts of the names in the actor path can be replaced with wildcards and the `..` symbol, similar to how parts of filenames can be replaced in a shell. In this case, we obtain a **path selection**. For example, the path selection `..` references the parent of the current actor. The selection `../*` references the current actor and all its siblings.

Actor paths are different from actor references; we cannot send a message to an actor using its actor path. Instead, we must first use the actor path to identify an actor on that actor path. If we successfully find an actor reference behind an actor path, we can send messages to it.

To obtain an actor reference corresponding to an actor path, we call the `actorSelection` method on the context object of an actor. This method takes an actor path, or a path selection. Calling the `actorSelection` method might address zero actors if no actors correspond to the actor path. Similarly, it might address multiple actors if we use a path selection. Thus, instead of returning an `ActorRef` object, the `actorSelection` method returns an `ActorSelection` object, which might represent zero, one, or more actors. We can use the `ActorSelection` object to send messages to these actors.

Use the `actorSelection` method on the `context` object to communicate with arbitrary actors in the actor system.

If we compare the `ActorRef` object to a specific e-mail address, an `ActorSelection` object can be compared to a mailing list address. Sending an e-mail to a valid e-mail address ensures that the e-mail reaches a specific person. On the other hand, when we send an e-mail to a mailing list, the e-mail might reach zero, one, or more people, depending on the number of mailing list subscribers.

An `ActorSelection` object does not tell us anything about the concrete paths of the actors, in a similar way to how a mailing list does not tell us anything about its subscribers. For this purpose, Akka defines a special type of message called `Identify`. When an Akka actor receives an `Identify` message, it will automatically reply by sending back an `ActorIdentity` message with its `ActorRef` object. If there are no actors in the actor selection, the `ActorIdentity` message is sent back to the sender of `Identify` without an `ActorRef` object.

Send `Identify` messages to the `ActorSelection` objects to obtain actor references of arbitrary actors in the actor system.

In the following example, we define a `CheckActor` actor class, which describes actors that check and print actor references whenever they receive a message with an actor path. When the actor of type `CheckActor` receives a string with an actor path or a path selection, it obtains an `ActorSelection` object and sends it an `Identify` message. This message is forwarded to all actors in the selection, which then respond with an `ActorIdentity` message. The `Identify` message also takes a `messageId` argument. If an actor sends out multiple `Identify` messages, the `messageId` argument allows disambiguating between the different `ActorIdentity` responses. In our example, we use the `path` string as the `messageId` argument. When `CheckActor` receives an `ActorIdentity` message, it either prints the actor reference or reports that there is no actor on the specified path. The `CheckActor` class is shown in the following code snippet:

```
class CheckActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case path: String =>
      log.info(s"checking path $path")
      context.actorSelection(path) ! Identify(path)
    case ActorIdentity(path, Some(ref)) =>
      log.info(s"found actor $ref at $path")
    case ActorIdentity(path, None) =>
      log.info(s"could not find an actor at $path")
  }
}
```

Next, we instantiate a `checker` actor of the `CheckActor` class, and send it the path selection, `../*`. This references all the child actors of the `checker` parent-the `checker` actor itself and its siblings:

```
val checker = ourSystem.actorOf(Props[CheckActor], "checker")
checker ! "../*"
```

We did not instantiate any top-level actors besides the `checker` actor, so checker receives only a single `ActorIdentity` message and prints its own actor path. Next, we try to identify all the actors one level above the `checker` actor. Recall the earlier figure. Since `checker` is a top-level actor, this should identify the guardian actors in the actor system:

```
checker ! "../../*"
```

As expected, the `checker` actor prints the actor paths of the `user` and `system` guardian actors. We are curious to learn more about the system-internal actors from the `system` guardian actor. This time, we send an absolute path selection to `checker`:

```
checker ! "/system/*"
```

The `checker` actor prints the actor paths of the internal actors `log1-Logging` and `deadLetterListener`, which are used for logging and for processing unhandled messages, respectively. We next try identifying a non-existing actor:

```
checker ! "/user/checker2"
```

There are no actors named `checker2`, so `checker` receives an `ActorIdentity` message with the `ref` field set to `None` and prints that it cannot find an actor on that path.

Using the `actorSelection` method and the `Identify` message is the fundamental method for discovering unknown actors in the same actor system. Note that we will always obtain an actor reference, and never obtain a pointer to the actor object directly. To better understand the reasons for this, we will study the lifecycle of actors in the next section.

The actor lifecycle

Recall that the `ChildActor` class from the previous section overrode the `postStop` method to produce some logging output when the actor is stopped. In this section, we investigate when exactly the `postStop` method gets called, along with the other important events that comprise the lifecycle of the actor.

To understand why the actor lifecycle is important, we consider what happens if an actor throws an exception while processing an incoming message. In Akka, such an exception is considered abnormal behavior, so top-level user actors that throw an exception are by default restarted. Restarting creates a fresh actor object, and effectively means that the actor state is reinitialized. When an actor is restarted, its actor reference and actor path remain the same. Thus, the same `ActorRef` object might refer to many different physical actor objects during the logical existence of the same actor. This is one of the reasons why an actor must never allow its `this` reference to leak. Doing so allows other parts of the program to refer to an old actor object, consequently invalidating the transparency of the actor reference. Additionally, revealing the `this` reference of the actor can reveal the internals of the actor implementation, or even cause data corruption.

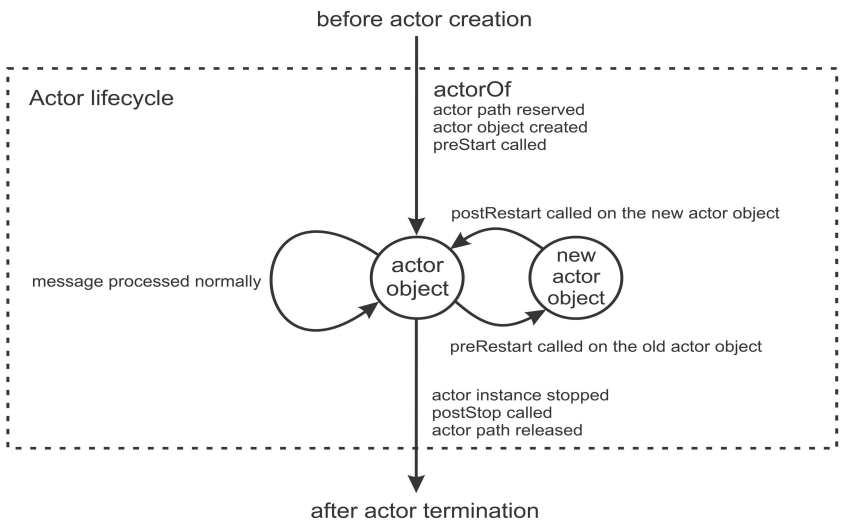
Never pass an actor's `this` reference to other actors, as it breaks actor encapsulation.

Let's examine the complete actor lifecycle. As we have learned, a logical actor instance is created when we call the `actorOf` method. The `Props` object is used to instantiate a physical actor object. This object is assigned a mailbox, and can start receiving input messages. The `actorOf` method returns an actor reference to the caller, and the actors can execute concurrently. Before the actor starts processing messages, its `preStart` method is called. The `preStart` method is used to initialize the logical actor instance.

After creation, the actor starts processing messages. At some point, an actor might need to be restarted due to an exception. When this happens, the `preRestart` method is first called. All the child actors are then stopped. Then, the `Props` object, previously used in order to create the actor with the `actorOf` method, is reused to create a new actor object. The `postRestart` method is called on the newly created actor object. After `postRestart` returns, the new actor object is assigned the same mailbox as the old actor object, and it continues to process messages that were in the mailbox before the restart.

By default, the `postRestart` method calls the `prestart` method. In some cases, we want to override this behavior. For example, a database connection might need to be opened only once during `preStart`, and closed when the logical actor instance is terminated.

Once the logical actor instance needs to stop, the `postStop` method gets called. The actor path associated with the actor is released, and returned to the actor system. By default, the `preRestart` method calls the `postStop` method. The complete actor lifecycle is illustrated in the following figure:



Note that, during the actor lifecycle, the rest of the actor system observes the same actor reference, regardless of how many times the actor restarts. Actor failures and restarts occur transparently for the rest of the system.

To experiment with the lifecycle of an actor, we declare two actor classes, `StringPrinter` and `LifecycleActor`. The `StringPrinter` actor prints a logging statement for each message that it receives. We override its `preStart` and `postStop` methods to precisely track when the actor has started and stopped, as shown in the following snippet:

```
class StringPrinter extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case msg => log.info(s"printer got message '$msg'")
  }
  override def preStart(): Unit = log.info(s"printer preStart.")
  override def postStop(): Unit = log.info(s"printer postStop.")
}
```

The `LifecycleActor` class maintains a `child` actor reference to a `StringPrinter` actor. The `LifecycleActor` class reacts to the `Double` and `Int` messages by printing them, and to the `List` messages by printing the first element of the list. When it receives a `String` message, the `LifecycleActor` instance forwards it to the `child` actor:

```
class LifecycleActor extends Actor {
  val log = Logging(context.system, this)
  var child: ActorRef = _
  def receive = {
    case num: Double  => log.info(s"got a double - $num")
    case num: Int     => log.info(s"got an integer - $num")
    case lst: List[_] => log.info(s"list - ${lst.head}, ...")
    case txt: String  => child ! txt
  }
}
```

We now override different lifecycle hooks. We start with the `preStart` method to output a logging statement and instantiate the `child` actor. This ensures that the `child` reference is initialized before the actor starts processing any messages:

```
override def preStart(): Unit = {
  log.info("about to start")
  child = context.actorOf(Props[StringPrinter], "kiddo")
}
```

Next, we override the `preRestart` and `postRestart` methods. In the `preRestart` and `postRestart` methods, we log the exception that caused the failure. The `postRestart` method calls the `preStart` method by default, so the new actor object gets initialized with a new `child` actor after a restart:

```
override def preRestart(t: Throwable, msg: Option[Any]): Unit = {
  log.info(s"about to restart because of $t, during message $msg")
  super.preRestart(t, msg)
}
override def postRestart(t: Throwable): Unit = {
  log.info(s"just restarted due to $t")
  super.postRestart(t)
}
```

Finally, we override the `postStop` method to track when the actor is stopped:

```
override def postStop() = log.info("just stopped")
```

We now create an instance of the `LifecycleActor` class called `testy`, and send a `math.Pi` message to it. The actor prints that it is about to start in its `preStart` method, and creates a new `child` actor. It then prints that it received the value `math.Pi`. Importantly, the `child about to start` logging statement is printed after the `math.Pi` message is received. This shows that actor creation is an asynchronous operation-when we call `actorOf`, creating the actor is delegated to the actor system, and the program immediately proceeds:

```
val testy = ourSystem.actorOf(Props[LifecycleActor], "testy")
testy ! math.Pi
```

We then send a string message to `testy`. The message is forwarded to the `child` actor, which prints a logging statement, indicating that it received the message:

```
testy ! "hi there!"
```

Finally, we send a `Nil` message to `testy`. The `Nil` object represents an empty list, so `testy` throws an exception when attempting to fetch the `head` element. It reports that it needs to restart. After that, we witness that the `child` actor prints the message that it needs to stop; recall that the child actors are stopped when an actor is restarted. Finally, `testy` prints that it is about to restart, and the new `child` actor is instantiated. These events are caused by the following statement:

```
testy ! Nil
```

Testing the actor lifecycle revealed an important property of the `actorOf` method. When we call the `actorOf` method, the execution proceeds without waiting for the actor to fully initialize itself. Similarly, sending a message does not block execution until the message is received or processed by another actor; we say that message sends are asynchronous. In the following section, we will examine various communication patterns that address this asynchronous behavior.

Communication between actors

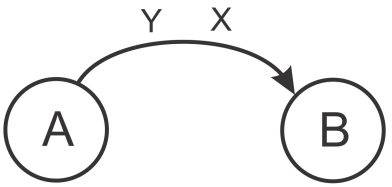
We have learned that actors communicate by sending messages. While actors running on the same machine can access shared parts of memory in the presence of proper synchronization, sending messages allows isolating the actor from the rest of the system and ensures location transparency. The fundamental operation that allows you to send a message to an actor is the `!` operator.

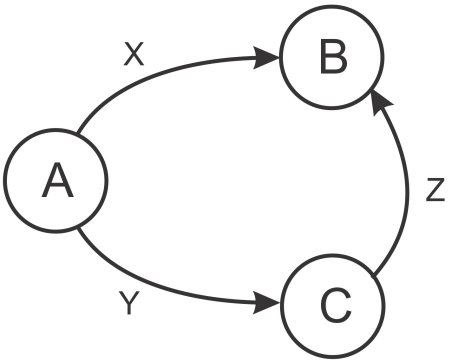
We have learned that the `!` operator is a non-blocking operation-sending a message does not block the execution of the sender until the message is delivered. This way of sending messages is sometimes called the **fire-and-forget** pattern, because it does not wait for a reply from the message receiver, nor does it ensure that the message is delivered.

Sending messages in this way improves the throughput of programs built using actors, but can be limiting in some situations. For example, we might want to send a message and wait for the response from the target. In this section, we learn about patterns used in actor communication that go beyond fire-and-forget.

While the fire-and-forget pattern does not guarantee that the message is delivered, it guarantees that the message is delivered **at most once**. The target actor never receives duplicate messages. Furthermore, the messages are guaranteed to be ordered for a given pair of sender and receiver actors. If an actor **A** sends messages **X** and **Y** in that order, the actor **B** will receive no duplicate messages, only the message **X**, only the message **Y**, or the message **X**, followed by the message **Y**.

This is shown on the left in the following figure:





$X_{send} < Y_{send} \rightarrow X_{receive} < Y_{receive}$

$X_{send} < Y_{send}$ and $Y_{receive} < Z_{send}$ does not imply $X_{receive} < Z_{receive}$

However, the delivery order is not ensured for a group of three or more actors. For example, as shown on the right in the preceding figure, actor **A** performs the following actions:

- Sends a message **X** to the actor **B**
- Sends a message **Y** to another actor, **C**
- Actor **C** sends a message **Z** to the actor **B** after having received **Y**

In this situation, the delivery order between messages **X** and **Z** is not guaranteed. The actor **B** might receive the messages **X** and **Z** in any order. This property reflects the characteristics of most computer networks, and is adopted to allow actors to run transparently on network nodes that may be remote.

The order in which an actor **B** receives messages from an actor **A** is the same as the order in which these messages are sent from the actor **A**.

Before we study various patterns of actor communication, note that the `!` operator was not the only non-blocking operation. The methods `actorOf` and `actorSelection` are also non-blocking. These methods are often called while an actor is processing a message. Blocking the actor while the message is processed prevents the actor from processing subsequent messages in the mailbox and severely compromises the throughput of the system. For these reasons, most of the actor API is non-blocking. Additionally, we must never start blocking the operations from third-party libraries from within an actor.

Messages must be handled without blocking indefinitely. Never start an infinite loop and avoid long-running computations in the `receive` block, the `unhandled` method, and within actor lifecycle hooks.

The ask pattern

Not being able to block from within an actor prevents the request-respond communication pattern. In this pattern, an actor interested in certain information sends a request message to another actor. It then needs to wait for a response message from the other actor. In Akka, this communication pattern is also known as the *ask pattern*.

The `akka.pattern` package defines the use of convenience methods in actor communication. Importing its contents allows us to call the `?` operator (pronounced ask) on actor references. This operator sends a message to the target actor, such as the tell operator. Additionally, the ask operator returns a future object with the response from the target actor.

To illustrate the usage of the ask pattern, we will define two actors that play ping pong with each other. A `Pingy` actor will send a `ping` request message to another actor, of type `Pongy`. When the `Pongy` actor receives the `ping` message, it sends a `pong` response message to the sender. We start by importing the `akka.pattern` package:

```
import akka.pattern._
```

We first define the `Pongy` actor class. To respond to the `ping` incoming message, the `Pongy` actor needs an actor reference of the sender. While processing a message, every actor can call the `sender` method of the `Actor` class to obtain the actor reference of the sender of the current message. The `Pongy` actor uses the `sender` method to send `ping` back to the `Pingy` actor. The `Pongy` implementation is shown in the following code snippet:

```
class Pongy extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "ping" =>
      log.info("Got a ping -- ponging back!")
      sender ! "pong"
      context.stop(self)
  }
  override def postStop() = log.info("pongy going down")
}
```

Next, we define the `Pingy` actor class, which uses the ask operator to send a request to the `Pongy` actor. When the `Pingy` class receives a `pongyRef` actor reference of `Pongy`, it creates an implicit `Timeout` object set to two seconds. Using the ask operator requires an implicit `Timeout` object in scope; the future is failed with an `AskTimeoutException` exception if the response message does not arrive within the given timeframe. Once `Pingy` class sends the `ping` message, it is left with an `f` future object. The `Pingy` actor uses the special `pipeTo` combinator that sends the value in the future to the sender of the `pongyRef` actor reference, as shown in the following code:

```
import akka.util.Timeout
import scala.concurrent.duration._
class Pingy extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case pongyRef: ActorRef =>
      implicit val timeout = Timeout(2 seconds)
      val f = pongyRef ? "ping"
      f pipeTo sender
  }
}
```

The message in the future object can be manipulated using the standard future combinators seen in `Chapter 4`, *Asynchronous Programming with Futures and Promises*. However, the following definition of the `Pingy` actor would not be correct:

```
class Pingy extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case pongyRef: ActorRef =>
      implicit val timeout = Timeout(2 seconds)
      val f = pongyRef ? "ping"
      f onComplete { case v => log.info(s"Response: $v") } // bad!
  }
}
```

Although it is perfectly legal to call the `onComplete` on the `f` future, the subsequent asynchronous computation should not access any mutable actor state. Recall that the actor state should be visible only to the actor, so concurrently accessing it opens the possibility of data races and race conditions. The `log` object should only be accessed by the actor that owns it. Similarly, we should not call the `sender` method from within the `onComplete` handler. By the time the future is completed with the response message, the actor might be processing a different message with a different sender, so the `sender` method can return arbitrary values.

When starting an asynchronous computation from within the `receive` block, the `unhandled` method, or a lifecycle hook, never let the closure capture any mutable actor state.

To test `Pingy` and `Pongy` in action, we define the `Master` actor class that instantiates them. Upon receiving the `start` message, the `Master` actor passes the `pongy` reference to the `pingy` reference. Once the `pingy` actor returns a `pong` message from `pongy`, the `Master` actor stops. This is shown in the following `Master` actor template:

```
class Master extends Actor {
  val pingy = ourSystem.actorOf(Props[Pingy], "pingy")
  val pongy = ourSystem.actorOf(Props[Pongy], "pongy")
  def receive = {
    case "start" =>
      pingy ! pongy
    case "pong" =>
      context.stop(self)
  }
  override def postStop() = log.info("master going down")
}
val masta = ourSystem.actorOf(Props[Master], "masta")
masta ! "start"
```

The ask pattern is useful because it allows you to send requests to multiple actors and obtain futures with their responses. Values from multiple futures can be combined within `for` comprehensions to compute a value from several responses. Using the fire-and-forget pattern when communicating with multiple actors requires changing the actor behavior, and is a lot more cumbersome than the ask pattern.

The forward pattern

Some actors exist solely to forward messages to other actors. For example, an actor might be responsible for load-balancing request messages between several worker actors, or it might forward the message to its mirror actor to ensure better availability. In such cases, it is useful to forward the message without changing the `sender` field of the message. The `forward` method on actor references serves this purpose.

In the following code, we use the `StringPrinter` actor from the previous section to define a `Router` actor class. A `Router` actor instantiates four child `StringPrinter` actors and maintains an `i` field with the index of the list child it forwarded the message to. Whenever it receives a message, it forwards the message to a different `StringPrinter` child before incrementing the `i` field:

```
class Router extends Actor {
  var i = 0
  val children = for (_ <- 0 until 4) yield
    context.actorOf(Props[StringPrinter])
  def receive = {
    case msg =>
      children(i) forward msg
      i = (i + 1) % 4
  }
}
```

In the following code, we create a `Router` actor and test it by sending it two messages. We can observe that the messages are printed to the standard output by two different `StringPrinter` actors, denoted with actors on the actor paths `/user/router/$b` and `/user/router/$a`:

```
val router = ourSystem.actorOf(Props[Router], "router")
router ! "Hola"
router ! "Hey!"
```

The forward pattern is typically used in router actors, which use specific knowledge to decide about the destination of the message; replicator actors, which send the message to multiple destinations; or load balancers, which ensure that the workload is spread evenly between a set of worker actors.

Stopping actors

So far, we have stopped different actors by making them call `context.stop`. Calling the `stop` method on the `context` object terminates the actor immediately after the current message is processed. In some cases, we want to have more control over how an actor gets terminated. For example, we might want to allow the actor to process its remaining messages or wait for the termination of some other actors. In Akka, there are several special message types that assist us in doing so, and we study them in this section.

In many cases, we do not want to terminate an actor instance, but simply restart it. We have previously learned that an actor is automatically restarted when it throws an exception. An actor is also restarted when it receives the `Kill` message-when we send a `Kill` message to an actor, the actor automatically throws an `ActorKilledException` and fails.

Use the `Kill` message to restart the target actor without losing the messages in the mailbox.

Unlike the `stop` method, the `Kill` message does not terminate the actor, but only restarts it. In some cases, we want to terminate the actor instance, but allow it to process the messages from its mailbox. Sending a `PoisonPill` message to an actor has the same effect as calling `stop`, but allows the actor to process the messages that were in the mailbox before the `PoisonPill` message arrives.

Use the `PoisonPill` message to stop the actor, but allow it to process the messages received before the `PoisonPill` message.

In some cases, allowing the actor to process its message using `PoisonPill` is not enough. An actor might have to wait for other actors to terminate before terminating itself. An orderly shutdown is important in some cases, as actors might be involved in sensitive operations, such as writing to a file on the disk. We do not want to forcefully stop them when we end the application. A facility that allows an actor to track the termination of other actors is called **DeathWatch** in Akka.

Recall the earlier example with the `Pingy` and `Pongy` actors. Let's say that we want to terminate the `Pingy` actor, but only after the `Pongy` actor has already been terminated. We define a new `GracefulPingy` actor class for this purpose. The `GracefulPingy` actor class calls the `watch` method on the `context` object when it gets created. This ensures that, after `Pongy` actor terminates and its `postStop` method completes, `GracefulPingy` actor receives a `Terminated` message with the actor reference to `Pongy` actor.

Upon receiving the `Terminated` message, `GracefulPingy` stops itself, as shown in the following `GracefulPingy` implementation:

```
class GracefulPingy extends Actor {
  val pongy = context.actorOf(Props[Pongy], "pongy")
  context.watch(pongy)
  def receive = {
    case "Die, Pingy!" =>
      context.stop(pongy)
    case Terminated(`pongy`) =>
      context.stop(self)
  }
}
```

Whenever we want to track the termination of an actor from inside an actor, we use DeathWatch, as in the previous example. When we need to wait for the termination of an actor from outside an actor, we use the *graceful stop pattern*. The `gracefulStop` method from the `akka.pattern` package takes an actor reference, a timeout, and a shutdown message. It returns a future and asynchronously sends the shutdown message to the actor. If the actor terminates within the allotted timeout, the future is successfully completed. Otherwise, the future fails.

In the following code, we create a `GracefulPingy` actor instance and call the `gracefulStop` method:

```
object CommunicatingGracefulStop extends App {
  val grace = ourSystem.actorOf(Props[GracefulPingy], "grace")
  val stopped =
    gracefulStop(grace, 3.seconds, "Die, Pingy!")
  stopped onComplete {
    case Success(x) =>
      log("graceful shutdown successful")
      ourSystem.shutdown()
    case Failure(t) =>
      log("grace not stopped!")
      ourSystem.shutdown()
  }
}
```

We typically use DeathWatch inside the actors, and the graceful stop pattern in the main application thread. The graceful stop pattern can be used within actors as well, as long as we are careful that the callbacks on the future returned by the `gracefulStop` method do not capture actor state. Together, DeathWatch and the graceful stop pattern allow safely shutting down actor-based programs.

Actor supervision

When studying the actor lifecycle, we said that top-level user actors are by default restarted when an exception occurs. We now take a closer inspection at how this works. In Akka, every actor acts as a supervisor for its children. When a child fails, it suspends the processing messages, and sends a message to its parent to decide what to do about the failure. The policy that decides what happens to the parent and the child after the child fails is called the **supervision strategy**. The parent might decide to do the following:

- Restart the actor, indicated with the `Restart` message
- Resume the actor without a restart, indicated with the `Resume` message
- Permanently stop the actor, indicated with the `Stop` message
- Fail itself with the same exception, indicated with the `Escalate` message

By default, the `user` guardian actor comes with a supervision strategy that restarts the failed children actors. User actors stop their children by default. Both supervision strategies can be overridden.

To override the default supervision strategy in user actors, we override the `supervisorStrategy` field of the `Actor` class. In the following code, we define a particularly troublesome actor class called `Naughty`. When the `Naughty` class receives a `String` type message, it prints a logging statement. For all other message types, it throws the `RuntimeException`, as shown in the following implementation:

```
class Naughty extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case s: String => log.info(s)
    case msg => throw new RuntimeException
  }
  override def postRestart(t: Throwable) =
    log.info("naughty restarted")
}
```

Next, we declare a `Supervisor` actor class, which creates a child actor of the `Naughty` type. The `Supervisor` actor does not handle any messages, but overrides the default supervision strategy. If a `Supervisor` actor's child actor fails because of throwing an `ActorKilledException`, it is restarted. However, if its child actor fails with any other exception type, the exception is escalated to the `Supervisor` actor. We override the `supervisorStrategy` field with the value `OneForOneStrategy`, a supervision strategy that applies fault handling specifically to the actor that failed:

```
class Supervisor extends Actor {
  val child = context.actorOf(Props[StringPrinter], "naughty")
  def receive = PartialFunction.empty
  override val supervisorStrategy =
    OneForOneStrategy() {
      case ake: ActorKilledException => Restart
      case _ => Escalate
    }
}
```

We test the new supervisor strategy by creating an actor instance, `super`, of the `Supervisor` actor class. We then create an actor selection for all the children of `super`, and send them a `Kill` message. This fails the `Naughty` actor, but `super` restarts it due to its supervision strategy. We then apologize to the `Naughty` actor by sending it a `String` message. Finally, we convert a `String` message to a list of characters, and send it to the `Naughty` actor, which then throws a `RuntimeException`. This exception is escalated by `super`, and both actors are terminated, as shown in the following code snippet:

```
ourSystem.actorOf(Props[Supervisor], "super")
ourSystem.actorSelection("/user/super/*") ! Kill
ourSystem.actorSelection("/user/super/*") ! "sorry about that"
ourSystem.actorSelection("/user/super/*") ! "kaboom".toList
```

In this example, we saw how the `OneForOneStrategy` works. When an actor fails, that specific actor is resumed, restarted, or stopped, depending on the exception that caused it to fail. The alternative `AllForOneStrategy` applies the fault-handling decision to all the children. When one of the child actors stops, all the other children are resumed, restarted, or stopped.

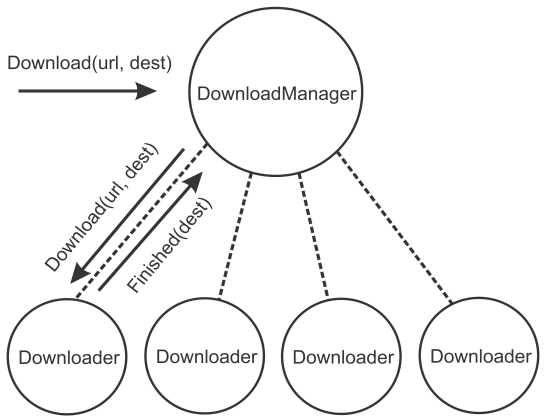
Recall our minimalistic web browser implementation from `Chapter 6`, *Concurrent Programming with Reactive Extensions*. A more advanced web browser requires a separate subsystem that handles concurrent file downloads. Usually, we refer to such a software component as a download manager. We now consider a larger example, in which we apply our knowledge of actors in order to implement the infrastructure for a simple download manager.

The download manager will be implemented as an actor, represented by the `DownloadManager` actor class. The two most important tasks of every download manager are to download the resources at the requested URL, and to track the downloads that are currently in progress. To be able to react to download requests and download completion events, we define the message types `Download` and `Finished` in the `DownloadManager` companion object. The `Download` message encapsulates the URL of the resource and the destination file for the resource, while the `Finished` message encodes the destination file where the resource is saved:

```
object DownloadManager {
  case class Download(url: String, dest: String)
  case class Finished(dest: String)
}
```

The `DownloadManager` actor will not execute the downloads itself. Doing so would prevent it from receiving any messages before the download completes. Furthermore, this will serialize different downloads and prevent them from executing concurrently. Thus, the `DownloadManager` actor must delegate the task of downloading the files to different actors. We represent these actors with the `Downloader` actor class. A `DownloadManager` actor maintains a set of `Downloader` children, and tracks which children are currently downloading a resource. When a `DownloadManager` actor receives a `Download` message, it picks one of the non-busy `Downloader` actors, and forwards the `Download` message to it.

Once the download is complete, the `Downloader` actor sends a `Finished` message to its parent. This is illustrated in the following figure:



We first show the implementation of the `Downloader` actor class. When a `Downloader` actor receives a `Download` message, it downloads the contents of the specified URL and writes them to a destination file. It then sends the `Finished` message back to the sender of the `Download` message, as shown in the following implementation:

```
class Downloader extends Actor {
  def receive = {
    case DownloadManager.Download(url, dest) =>
      val content = Source.fromURL(url)
      FileUtils.write(new java.io.File(dest), content.mkString)
      sender ! DownloadManager.Finished(dest)
  }
}
```

The `DownloadManager` actor class needs to maintain state to track which of its `Downloader` actors is currently downloading a resource. If there are more download requests than there are available `Downloader` instances, the `DownloadManager` actor needs to enqueue the download requests until a `Downloader` actor becomes available. The `DownloadManager` actor maintains a `downloaders` queue with actor references to non-busy `Downloader` actors. It maintains another queue, the `pendingWork` queue, with `Download` requests that cannot be assigned to any `Downloader` instances.

Finally, it maintains a map called `workItems` that associates actor references of the busy `Downloader` instances with their `Download` requests. This is shown in the following `DownloadManager` implementation:

```
class DownloadManager(val downloadSlots: Int) extends Actor {
  import DownloadManager._
  val log = Logging(context.system, this)
  val downloaders = mutable.Queue[ActorRef]()
  val pendingWork = mutable.Queue[Download]()
  val workItems = mutable.Map[ActorRef, Download]()
  private def checkDownloads(): Unit = {
    if (pendingWork.nonEmpty && downloaders.nonEmpty) {
      val dl = downloaders.dequeue()
      val item = pendingWork.dequeue()
      log.info(
        s"$item starts, ${downloaders.size} download slots left")
      dl ! item
      workItems(dl) = item
    }
  }
  def receive = {
    case msg @ DownloadManager.Download(url, dest) =>
      pendingWork.enqueue(msg)
      checkDownloads()
```

```
      case DownloadManager.Finished(dest) =>
        workItems.remove(sender)
        downloaders.enqueue(sender)
        log.info(
          s"'$dest' done, ${downloaders.size} download slots left")
        checkDownloads()
    }
  }
```

The `checkDownloads` private method maintains the `DownloadManager` actor's invariant- the `pendingWork` and the `downloaders` queue cannot be non-empty at the same time. As soon as both the queues become non-empty, a `Downloader` actor reference `dl` is dequeued from `downloaders` and a `Download` request item is dequeued from the `pendingWork` queue. The `item` value is then sent as a message to the `dl` actor, and the `workItems` map is updated.

Whenever the `DownloadManager` actor receives a `Download` message, it adds it to the `pendingWork` queue and calls the `checkDownloads` method. Similarly, when the `Finished` message arrives, the `Downloader` actor is removed from the `workItems` queue and enqueued on the `downloaders` list.

To ensure that the `DownloadManager` actor is created with the specified number of `Downloader` child actors, we override the `preStart` method to create the `Downloaders` list and add their actor references to the `downloaders` queue:

```
  override def preStart(): Unit = {
    for (i <- 0 until downloadSlots) {
      val dl = context.actorOf(Props[Downloader], s"dl$i")
      downloaders.enqueue()
    }
  }
```

Finally, we must override the `supervisorStrategy` field of the `DownloadManager` actor. We use the `OneForOneStrategy` field again, but specify that the actor can be restarted or resumed only up to 20 times within a two-second interval.

We expect that some URLs might be invalid, in which case the actor fails with a `FileNotFoundException`. We need to remove such an actor from the `workItems` collection and add it back to the `downloaders` queue. It does not make sense to restart the `Downloader` actors, because they do not contain any state. Instead of restarting, we simply resume a `Downloader` actor that cannot resolve a URL. If the `Downloader` instances fail due to any other messages, we escalate the exception and fail the `DownloadManager` actor, as shown in the following `supervisorStrategy` implementation:

```
override val supervisorStrategy =
  OneForOneStrategy(
    maxNrOfRetries = 20, withinTimeRange = 2 seconds
  ) {
    case fnf: java.io.FileNotFoundException =>
      log.info(s"Resource could not be found: $fnf")
      workItems.remove(sender)
      downloaders.enqueue(sender)
      Resume // ignores the exception and resumes the actor
    case _ =>
      Escalate
  }
```

To test the download manager, we create a `DownloadManager` actor with four download slots, and send it several `Download` messages:

```
val downloadManager =
  ourSystem.actorOf(Props(classOf[DownloadManager], 4), "man")
downloadManager ! Download(
  "http://www.w3.org/Addressing/URL/url-spec.txt",
  "url-spec.txt")
```

An extra copy of the URL specification cannot hurt, so we download it to our computer. The download manager logs that there are only three download slots left. Once the download completes, the download manager logs that there are four remaining download slots again. We then decide that we would like to contribute to the Scala programming language, so we download the `README` file from the official Scala repository. Unfortunately, we enter an invalid URL, and observe a warning from the download manager saying that the resource cannot be found:

```
downloadManager ! Download(
  "https://github.com/scala/scala/blob/master/README.md",
  "README.md")
```

The simple implementation of the basic actor-based download manager illustrates both how to achieve concurrency by delegating work to child actors, and how to treat failures in child actors. Delegating work is important, both for decomposing the program into smaller, isolated components, and to achieve better throughput and scalability. Actor supervision is the fundamental mechanism for handling failures in isolated components implemented in separate actors.

Remote actors

So far in this book, we have mostly concentrated on writing programs on a single computer. Concurrent programs are executed within a single process on one computer, and they communicate using shared memory. Seemingly, actors described in this chapter communicate by passing messages. However, the message passing used throughout this chapter is implemented by reading and writing to shared memory under the hood.

In this section, we study how the actor model ensures location transparency by taking existing actors and deploying them in a distributed program. We take two existing actor implementations, namely, `Pingy` and `Pongy`, and deploy them inside different processes. We will then instruct the `Pingy` actor to send a message to the `Pongy` actor, as before, and wait until the `Pingy` actor returns the `Pongy` actor's message. The message exchange will occur transparently, although the `Pingy` and `Pongy` actor's were previously implemented without knowing that they might exist inside separate processes, or even different computers.

The Akka actor framework is organized into several modules. To use the part of Akka that allows communicating with actors in remote actor systems, we need to add the following dependency to our build definition file:

```
libraryDependencies += "com.typesafe.akka" %% "akka-remote" % "2.3.2"
```

Before creating our ping-pong actors inside two different processes, we need to create an actor system that is capable of communicating with remote actors. To do this, we create a custom actor system configuration string. The actor system configuration string can be used to configure a range of different actor system properties; we are interested in using a custom `ActorRef` factory object called `RemoteActorRefProvider`. The `ActorRef` factory object allows the actor system to create actor references that can be used to communicate over the network. Furthermore, we configure the actor system to use the **Netty** networking library with the TCP network layer and the desired TCP port number. We declare the `remotingConfig` method for this task:

```
import com.typesafe.config._
def remotingConfig(port: Int) = ConfigFactory.parseString(s"""
akka {
  actor.provider = "akka.remote.RemoteActorRefProvider"
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = $port
    }
  }
}
""")
```

We then define a `remotingSystem` factory method that creates an actor system object using the given name and port. We use the `remotingConfig` method, defined earlier, to produce the configuration object for the specified network port:

```
def remotingSystem(name: String, port: Int): ActorSystem =
  ActorSystem(name, remotingConfig(port))
```

Now we are ready to create the `Pongy` actor system. We declare an application called `RemotingPongySystem`, which instantiates an actor system called `PongyDimension` using the network port `24321`. We arbitrarily picked a network port that was free on our machine. If the creation of the actor system fails because the port is not available, you can pick a different port in the range `1024` to `65535`. Make sure that you don't have a firewall running, as it can block the network traffic for arbitrary applications.

The `RemotingPongySystem` application is shown in the following example:

```
object RemotingPongySystem extends App {
  val system = remotingSystem("PongyDimension", 24321)
  val pongy = system.actorOf(Props[Pongy], "pongy")
  Thread.sleep(15000)
  system.shutdown()
}
```

The `RemotingPongySystem` application creates a `Pongy` actor and shuts down after 15 seconds. After we start it, we will only have a short time to start another application running the `Pingy` actor. We will call this second application `RemotingPingySystem`. Before we implement it, we create another actor called `Runner`, which will instantiate `Pingy`, obtain the `Pongy` actor's reference, and give it to the `Pingy` actor; recall that the ping-pong game from the earlier section starts when the `Pingy` actor obtains the `Pongy` actor's reference.

When the `Runner` actor receives a `start` message, it constructs the actor path for the `Pongy` actor. We use the `akka.tcp` protocol and the name of the remote actor system, along with its IP address and port number. The `Runner` actor sends an `Identify` message to the actor selection in order to obtain the actor reference to the remote `Pongy` instance. The complete `Runner` implementation is shown in the following code snippet:

```
class Runner extends Actor {
  val log = Logging(context.system, this)
  val pingy = context.actorOf(Props[Pingy], "pingy")
  def receive = {
    case "start" =>
      val pongySys = "akka.tcp://PongyDimension@127.0.0.1:24321"
      val pongyPath = "/user/pongy"
      val url = pongySys + pongyPath
      val selection = context.actorSelection(url)
      selection ! Identify(0)
    case ActorIdentity(0, Some(ref)) =>
      pingy ! ref
    case ActorIdentity(0, None) =>
      log.info("Something's wrong - ain't no pongy anywhere!")
      context.stop(self)
    case "pong" =>
      log.info("got a pong from another dimension.")
      context.stop(self)
  }
}
```

Once the `Runner` actor sends the `Pongy` actor reference to `Pingy`, the game of remote ping pong can begin. To test it, we declare the `RemotingPingySystem` application, which starts the `Runner` actor and sends it a `start` message:

```
object RemotingPingySystem extends App {
  val system = remotingSystem("PingyDimension", 24567)
  val runner = system.actorOf(Props[Runner], "runner")
  runner ! "start"
  Thread.sleep(5000)
  system.shutdown()
}
```

We now need to start the `RemotingPongySystem` application, and the `RemotingPingySystem` application after that; we only have 15 seconds until the `RemotingPongySystem` application shuts itself down. The easiest way to do this is to start two SBT instances in your project folder and run the two applications at the same time. After the `RemotingPingySystem` application starts, we soon observe a `pong` message from another dimension.

In the previous example, the actor system configuration and the `Runner` actor were responsible for setting up the network communication, and were not location-transparent. This is typically the case with distributed programs; a part of the program is responsible for initializing and discovering actors within remote actor systems, while the application-specific logic is confined within separate actors.

In larger actor programs, separate deployment logic from application logic.

To summarize, remote actor communication requires the following steps:

- Declaring an actor system with an appropriate remoting configuration
- Starting two actor systems in separate processes or on separate machines
- Using actor path selection to obtain actor references
- Using actor references to transparently send messages

While the first three steps are not location-transparent, the application logic is usually confined within the fourth step, as we saw in this section. This is important, as it allows separating the deployment logic from the application semantics and building distributed systems that can be deployed transparently to different network configurations.

Summary

In this chapter, we learned what actors are and how to use them to build concurrent programs. Using the Akka actor framework, we studied how to create actors, organize them into hierarchies, manage their lifecycle, and recover them from errors. We examined important patterns in actor communication and learned how to model actor behavior. Finally, we saw how the actor model can ensure location transparency, and serve as a powerful tool to seamlessly build distributed systems.

Still, there are many Akka features that we omitted in this chapter. Akka comes with detailed online documentation, which is one of the best sources of information on Akka. To obtain an in-depth understanding of distributed programming, we recommend the books *Distributed Algorithms*, *Nancy A. Lynch*, published by Elsevier and *Introduction to Reliable and Secure Distributed Programming*, *Christian Cachin*, *Rachid Guerraoui, Luis Rodrigues*, published by Springer.

In the following chapter, we will summarize the different concurrency libraries we learned about in this book, examine the typical use cases for each of them, and see how they work together in larger applications.

Exercises

The following exercises test your understanding of the actor programming model, and distributed programming in general. The first few exercises are straightforward, and deal with the basics of the actor API in Akka. Subsequent exercises are more involved, and go deeper into the territory of fault-tolerant distributed programming. Try to solve these exercises by first assuming that no machines fail, and then consider what happens if some of the machines fail during the execution of the program:

1. Implement the timer actor with the `TimerActor` class. After receiving a `Register` message containing the `t` timeout in milliseconds, the timer actor sends a `Timeout` message back after `t` milliseconds. The timer must accept multiple `Register` messages.
2. Recall the bank account example from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. Implement different bank accounts as separate actors, represented by the `AccountActor` class. When an `AccountActor` class receives a `Send` message, it must transfer the specified amount of money to the target actor. What will happen if either of the actors receives a `Kill` message at any point during the money transaction?

3. Implement the `SessionActor` class for actors that control access to other actors:

```
class SessionActor(password: String, r: ActorRef)
extends Actor {
  def receive = ???
}
```

   After the `SessionActor` instance receives the `StartSession` message with the correct password, it forwards all the messages to the actor reference `r`, until it receives the `EndSession` message. Use behaviors to model this actor.

4. Use actors to implement the `ExecutionContext` interface, described in `Chapter 3`, *Traditional Building Blocks of Concurrency*.
5. Implement the `FailureDetector` actor, which sends `Identify` messages to the specified actors every `interval` seconds. If an actor does not reply with any `ActorIdentity` messages within `threshold` seconds, the `FailureDetector` actor sends a `Failed` message to its parent actor, which contains the actor reference of the failed actor.
6. A distributed hash map is a collection distributed across multiple computers, each of which contains part of the data, called a **shard**. When there are `2^n` shards, the first `n` bits of the hash code of the key are used to decide which shard a key-value pair should go to. Implement the distributed hash map with the `DistributedMap` class:

```
class DistributedMap[K, V](shards: ActorRef*) {
  def update(key: K, value: V): Future[Unit] = ???
  def get(key: K): Future[Option[V]] = ???
}
```

   The `DistributedMap` class takes a list of actor references to the `ShardActor` instances, whose actor template you also need to implement. You might assume that the length of the `shards` list is a power of two. The `update` and `get` methods are asynchronous, and return the result in a future object.

7. Implement an abstract `BroadcastActor` class, which defines the `broadcast` method:

```
def broadcast(refs: ActorRef*)(msg: Any): Unit = ???
```

   The `broadcast` method sends the `msg` message to all the actors specified in the `refs` list. The actor invoking the `broadcast` method might, for reasons such as power loss, fail at any point during the execution of the `broadcast` method. Nevertheless, the `broadcast` method must have **reliable delivery**: if at least one actor from the `refs` list receives the `msg` message, then all the actors from the `refs` list must eventually receive `msg`.

8. Implement a `FlowRateActor` class for an actor that forwards incoming messages to a target actor. This actor must ensure that the number of messages forwarded per second does not exceed a rate specified in its constructor.
9. Implement a `Sequencer` actor, which forwards messages to the target actor. If the message is a two-element tuple where the first element is a `Long` value, then the `Long` value is interpreted as a sequence number. All such messages must be forwarded in the proper sequence number order, starting from number `0`.
10. Implement a `MasterWorker[T]` actor that, given a number of worker parameters, creates a set of worker actors and forwards task messages of type `() => T` to those workers. When the worker actors complete a task, they send the result back to the `MasterWorker` actor, which sends the reply back to the client actor that originally sent the task.

9
Concurrency in Practice

"The best theory is inspired by practice."
-Donald Knuth

We have studied a plethora of different concurrency facilities in this book. By now, you will have learned about dozens of different ways of starting concurrent computations and accessing shared data. Knowing how to use different styles of concurrency is useful, but it might not yet be obvious when to use which.

The goal of this final chapter is to introduce the big picture of concurrent programming. We will study the use cases for various concurrency abstractions, see how to debug concurrent programs, and how to integrate different concurrency libraries in larger applications. In this chapter, we will perform the following tasks:

- Summarize the characteristics and typical uses of different concurrency frameworks introduced in the earlier chapters
- Investigate how to deal with various kinds of bugs appearing in concurrent applications
- Learn how to identify and resolve performance bottlenecks
- Apply the previous knowledge about concurrency to implement a larger concurrent application, namely, a remote file browser

We start with an overview of the important concurrency frameworks we have learned about in this book, and a summary of when to use each of them.

Choosing the right tools for the job

In this section, we present an overview of the different concurrency libraries that we learned about. We take a step back and look at the differences between these libraries, and what they have in common. This summary will give us an insight into what different concurrency abstractions are useful for.

A concurrency framework usually needs to address several concerns:

- It must provide a way to declare data that is shared between concurrent executions
- It must provide constructs for reading and modifying program data
- It must be able to express conditional execution, triggered when a certain set of conditions are fulfilled
- It must define a way to start concurrent executions

Some of the frameworks from this book address all these concerns; others address only a subset, and transfer part of the responsibility to another framework.

Typically, in a concurrent programming model, we express concurrently shared data differently from data intended to be accessed only from a single thread. This allows the JVM runtime to optimize sequential parts of the program more effectively. So far, we've seen a lot of different ways to express concurrently shared data, ranging from the low-level facilities to advanced high-level abstractions. We summarize different data abstractions in the following table:

Data abstraction	**Datatype or annotation**	**Description**
Volatile variables (JDK)	**@volatile**	**Ensures visibility and the happens-before relationship on class fields and local variables that are captured in closures.**
Atomic variables (JDK)	AtomicReference[T] AtomicInteger AtomicLong	Provide basic composite atomic operations, such as **`compareAndSet`** and **`incrementAndGet`**.
Futures and promises (**`scala.concurrent`**)	Future[T] Promise[T]	Sometimes called single-assignment variables, these express values that might not be computed yet but will eventually become available.

Data abstraction	Datatype or annotation	Description
Observables and subjects (Rx)	Observable[T] Subject[T]	Also known as first-class event streams, these describe many different values that arrive one after another in time.
Transactional references (ScalaSTM)	Ref[T]	These describe memory locations that can only be accessed from within memory transactions. Their modifications only become visible after the transaction successfully commits.

The next important concern is providing access to shared data, which includes reading and modifying shared memory locations. Usually, a concurrent program uses special constructs to express such accesses. We summarize the different data access constructs in the following table:

Data abstraction	Data access constructs	Description
Arbitrary data (JDK)	synchronized	Uses intrinsic object locks to exclude access to arbitrary shared data.
Atomic variables and classes (JDK)	compareAndSet	Atomically exchanges the value of a single memory location. It allows implementing lock-free programs.
Futures and promises (`scala.concurrent`)	value tryComplete	Used to assign a value to a promise, or to check the value of the corresponding future. The `value` method is not a preferred way to interact with a future.
Transactional references and classes (Scala STM)	atomic orAtomic single	Atomically modifies the values of a set of memory locations. Reduces the risk of deadlocks, but disallows side effects inside the transactional block.

Concurrent data access is not the only concern of a concurrency framework. As we have learned in previous chapters, concurrent computations sometimes need to proceed only after a certain condition is met. In the following table, we summarize different constructs that enable this:

Concurrency framework	**Conditional execution constructs**	**Description**
JVM concurrency	wait notify notifyAll	Used to suspend the execution of a thread until some other thread notifies that the conditions are met.
Futures and promises	onComplete Await.ready	Conditionally schedules an asynchronous computation. The **`Await.ready`** method suspends the thread until the future completes.
Reactive extensions	subscribe	Asynchronously or synchronously executes a computation when an event arrives.
Software transactional memory	retry retryFor withRetryTimeout	Retries the current memory transaction when some of the relevant memory locations change.
Actors	receive	Executes the actor's **`receive`** block when a message arrives.

Finally, a concurrency model must define a way to start a concurrent execution. We summarize different concurrency constructs in the following table:

Concurrency framework	**Concurrency constructs**	**Description**
JVM concurrency	Thread.start	Starts a new thread of execution.
Execution contexts	execute	Schedules a block of code for execution on a thread pool.
Futures and promises	Future.apply	Schedules a block of code for execution, and returns the future value with the result of the execution.
Parallel collections	par	Allows invoking data-parallel versions of collection methods.

Reactive extensions	Observable.create observeOn	The **`create`** method defines an event source. The **`observeOn`** method schedules the handling of events on different threads.
Actors	actorOf	Schedules a new actor object for execution.

This breakdown shows us that different concurrency libraries focus on different tasks. For example, parallel collections do not have conditional waiting constructs, because a data-parallel operation proceeds on separate elements independently. Similarly, software transactional memory does not come with a construct to express concurrent computations, and focuses only on protecting access to shared data. Actors do not have special constructs for modeling shared data and protecting access to it, because data is encapsulated within separate actors and accessed serially only by the actor that owns it.

Having classified concurrency libraries according to how they model shared data and express concurrency, we present a summary of what different concurrency libraries are good for:

- The classical JVM concurrency model uses threads, the `synchronized` statement, volatile variables, and atomic primitives for low-level tasks. Uses include implementing a custom concurrency utility, a concurrent data structure, or a concurrency framework optimized for specific tasks.
- Futures and promises are best suited for referring to concurrent computations that produce a single result value. Futures model latency in the program, and allow composing values that become available later during the execution of the program. Uses include performing remote network requests and waiting for replies, referring to the result of an asynchronous long-running computation, or reacting to the completion of an I/O operation. Futures are usually the glue of a concurrent application, binding the different parts of a concurrent program together. We often use futures to convert single-event callback APIs into a standardized representation based on the `Future` type.
- Parallel collections are best suited for efficiently executing data-parallel operations on large datasets. Uses include file searching, text processing, linear algebra applications, numerical computations, and simulations. Long-running Scala collection operations are usually good candidates for parallelization.

- Reactive extensions are used to express asynchronous event-based programs. Unlike parallel collections, in reactive extensions, data elements are not available when the operation starts, but arrive while the application is running. Uses include converting callback-based APIs, modeling events in user interfaces, modeling events external to the application, manipulating program events with collection-style combinators, streaming data from input devices or remote locations, or incrementally propagating changes in the data model throughout the program.
- Use STM to protect program data from getting corrupted by concurrent accesses. An STM allows building complex data models and accessing them with the reduced risk of deadlocks and race conditions. A typical use is to protect concurrently accessible data while retaining good scalability between threads whose accesses to data do not overlap.
- Actors are suitable for encapsulating concurrently accessible data, and seamlessly building distributed systems. Actor frameworks provide a natural way to express concurrent tasks that communicate by explicitly sending messages. Uses include serializing concurrent access to data to prevent corruption, expressing stateful concurrency units in the system, and building distributed applications such as trading systems, P2P networks, communication hubs, or data-mining frameworks.

Advocates of specific programming languages, libraries, or frameworks might try to convince you that their technology is the best for any task and any situation, often with the intent of selling it. Richard Stallman once said that "computer science is the only industry more fashion-driven than women's fashion." As engineers, we need to know better than to succumb to programming fashion and marketing propaganda. Different frameworks are tailored towards specific use cases, and the correct way to choose a technology is to carefully weigh its advantages and disadvantages when applied to a specific situation.

There is no one-size-fits-all technology. Use your own best judgment when deciding which concurrency framework to use for a specific programming task.

Sometimes, choosing the best-suited concurrency utility is easier said than done. It takes a great deal of experience to choose the correct technology. In many cases, we do not even know enough about the requirements of the system to make an informed decision. Regardless, a good rule of thumb is to apply several concurrency frameworks to different parts of the same application, each best suited for a specific task. Often, the real power of different concurrency frameworks becomes apparent when they are used together. This is the topic we will cover in the following section.

Putting it all together – a remote file browser

In this section, we use our knowledge about different concurrency frameworks to build a remote file browser. This larger application example illustrates how different concurrency libraries work together, and how to apply them to different situations. We will name our remote file browser ScalaFTP.

The ScalaFTP browser is divided into two main components: the server and the client process. The server process will run on the machine whose filesystem we want to manipulate. The client will run on our own computer, and comprise of a graphical user interface used to navigate the remote filesystem. To keep things simple, the protocol that the client and the server will use to communicate will not really be FTP, but a custom communication protocol. By choosing the correct concurrency libraries to implement different parts of ScalaFTP, we will ensure that the complete ScalaFTP implementation fits inside just 500 lines of code.

Specifically, the ScalaFTP browser will implement the following features:

- Displaying the names of the files and the directories in a remote filesystem, and allowing navigation through the directory structure
- Copying files between directories in a remote filesystem
- Deleting files in a remote filesystem

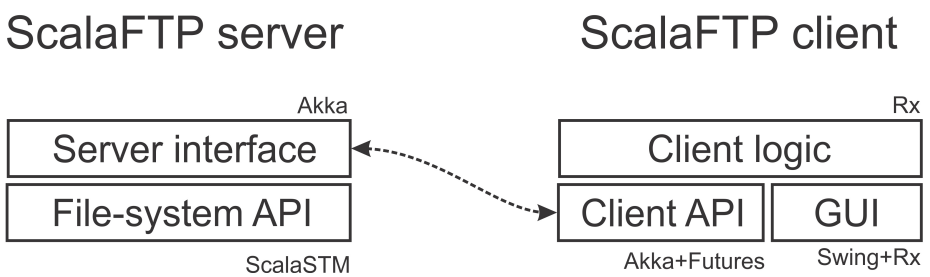
To implement separate pieces of this functionality, we will divide the ScalaFTP server and client programs into layers. The task of the server program is to answer to incoming copy and delete requests, and to answer queries about the contents of specific directories. To make sure that its view of the filesystem is consistent, the server will cache the directory structure of the filesystem. We divide the server program into two layers: the filesystem API and the server interface. The filesystem API will expose the data model of the server program, and define useful utility methods to manipulate the filesystem. The server interface will receive requests and send responses back to the client.

Since the server interface will require communicating with the remote client, we decide to use the Akka actor framework. Akka comes with remote communication facilities, as we learned in `Chapter 8`, *Actors*. The contents of the filesystem, that is, its state, will change over time. We are therefore interested in choosing proper constructs for data access.

In the filesystem API, we can use object monitors and locking to synchronize access to shared state, but we will avoid these due to the risk of deadlocks. We similarly avoid using atomic variables, because they are prone to race conditions. We could encapsulate the filesystem state within an actor, but note that this can lead to a scalability bottleneck: an actor would serialize all accesses to the filesystem state. Therefore, we decide to use the ScalaSTM framework to model the filesystem contents. An STM avoids the risk of deadlocks and race conditions, and ensures good horizontal scalability, as we learned in `Chapter 7`, *Software Transactional Memory*.

The task of the client program will be to graphically present the contents of the remote filesystem, and communicate with the server. We divide the client program into three layers of functionality. The GUI layer will render the contents of the remote filesystem and register user requests, such as button clicks. We will implement the GUI using the Swing and Rx frameworks, similarly to how we implemented the web browser in `Chapter 6`, *Concurrent Programming with Reactive Extensions*. The client API will replicate the server interface on the client side and communicate with the server. We will use Akka to communicate with the server, but expose the results of remote operations as futures. Finally, the client logic will be a gluing layer, which binds the GUI and the client API together.

The architecture of the ScalaFTP browser is illustrated in the following diagram, in which we indicate which concurrency libraries will be used by separate layers. The dashed line represents the communication path between the client and the server:



We now start by implementing the ScalaFTP server, relying on the bottom-up design approach. In the following section, we will describe the internals of the filesystem API.

Modeling the filesystem

In `Chapter 3`, *Traditional Building Blocks of Concurrency*, we used atomic variables and concurrent collections to implement a non-blocking, thread-safe filesystem API, which allowed copying files and retrieving snapshots of the filesystem. In this section, we repeat this task using STM. We will see that it is much intuitive and less error-prone to use STM.

We start by defining the different states that a file can be in. As in `Chapter 3`, *Traditional Building Blocks of Concurrency*, the file can be currently created, in the idle state, being copied, or being deleted. We model this with a sealed `State` trait, and its four cases:

```
sealed trait State
case object Created extends State
case object Idle extends State
case class Copying(n: Int) extends State
case object Deleted extends State
```

A file can only be deleted if it is in the idle state, and it can only be copied if it is in the idle state or in the copied state. Since a file can be copied to multiple destinations at a time, the `Copying` state encodes how many copies are currently under way. We add the methods `inc` and `dec` to the `State` trait, which return a new state with one more or one fewer copy, respectively. For example, the implementation of `inc` and `dec` for the `Copying` state is as follows:

```
def inc: State = Copying(n + 1)
def dec: State = if (n > 1) Copying(n - 1) else Idle
```

Similar to the `File` class in the `java.io` package, we represent both the files and directories with the same entity, and refer to them more generally as files. Each file is represented by the `FileInfo` class that encodes the path, its name, its parent directory and the date of the last modification to the file, a Boolean value denoting if the file is a directory, the size of the file, and its `State` object. The `FileInfo` class is immutable, and updating the state of the file will require creating a fresh `FileInfo` object:

```
case class FileInfo(path: String, name: String,
  parent: String, modified: String, isDir: Boolean,
  size: Long, state: State)
```

We separately define the factory methods `apply` and `creating` that take a `File` object and return a `FileInfo` object in the `Idle` or `Created` state, respectively.

Depending on where the server is started, the root of the ScalaFTP directory structure is a different subdirectory in the actual filesystem. A `FileSystem` object tracks the files in the given `rootpath` directory, using a transactional map called `files`:

```
class FileSystem(val rootpath: String) {
  val files = TMap[String, FileInfo]()
}
```

We introduce a separate `init` method to initialize the `FileSystem` object. The `init` method starts a transaction, clears the contents of the `files` map, and traverses the files and directories under `rootpath` using the Apache Commons IO library. For each file and directory, the `init` method creates a `FileInfo` object and adds it to the `files` map, using its path as the key:

```
def init() = atomic { implicit txn =>
  files.clear()
  val rootDir = new File(rootpath)
  val all = TrueFileFilter.INSTANCE
  val fileIterator =
    FileUtils.iterateFilesAndDirs(rootDir, all, all).asScala
  for (file <- fileIterator) {
    val info = FileInfo(file)
    files(info.path) = info
  }
}
```

Recall that the ScalaFTP browser must display the contents of the remote filesystem. To enable directory queries, we first add the `getFileList` method to the `FileSystem` class, which retrieves the files in the specified `dir` directory. The `getFileList` method starts a transaction and filters the files whose direct parent is equal to `dir`:

```
def getFileList(dir: String): Map[String, FileInfo] =
  atomic { implicit txn =>
    files.filter(_._2.parent == dir)
  }
```

We implement the copying logic in the filesystem API with the `copyFile` method. This method takes a path to the `src` source file and the `dest` destination file, and starts a transaction. After checking whether the `dest` destination file exists or not, the `copyFile` method inspects the state of the source file entry, and fails unless the state is `Idle` or `Copying`. It then calls `inc` to create a new state with the increased copy count, and updates the source file entry in the `files` map with the new state. Similarly, the `copyFile` method creates a new entry for the destination file in the `files` map. Finally, the `copyFile` method calls the `afterCommit` handler to physically copy the file to disk after the transaction completes. Recall that it is not legal to execute side-effecting operations from within the transaction body, so the private `copyOnDisk` method is called only after the transaction commits:

```
def copyFile(src: String, dest: String) = atomic { implicit txn =>
  val srcfile = new File(src)
  val destfile = new File(dest)
  val info = files(src)
  if (files.contains(dest)) sys.error(s"Destination exists.")
  info.state match {
    case Idle | Copying(_) =>
      files(src) = info.copy(state = info.state.inc)
      files(dest) = FileInfo.creating(destfile, info.size)
      Txn.afterCommit { _ => copyOnDisk(srcfile, destfile) }
      src
  }
}
```

The `copyOnDisk` method calls the `copyFile` method on the `FileUtils` class from the Apache Commons IO library. After the file transfer completes, the `copyOnDisk` method starts another transaction, in which it decreases the copy count of the source file and sets the state of the destination file to `Idle`:

```
private def copyOnDisk(srcfile: File, destfile: File) = {
  FileUtils.copyFile(srcfile, destfile)
  atomic { implicit txn =>
    val ninfo = files(srcfile.getPath)
    files(srcfile.getPath) = ninfo.copy(state = ninfo.state.dec)
    files(destfile.getPath) = FileInfo(destfile)
  }
}
```

The `deleteFile` method deletes a file in a similar way. It changes the file state to `Deleted`, deletes the file, and starts another transaction to remove the file entry:

```
def deleteFile(srcpath: String): String = atomic { implicit txn =>
  val info = files(srcpath)
  info.state match {
    case Idle =>
      files(srcpath) = info.copy(state = Deleted)
      Txn.afterCommit { _ =>
        FileUtils.forceDelete(info.toFile)
        files.single.remove(srcpath)
      }
      srcpath
  }
}
```

Modeling the server data model with the STM allows for the seamless addition of different concurrent computations to the server program. In the following section, we will implement a server actor that uses the server API to execute filesystem operations.

Use STM to model concurrently accessible data, as an STM works transparently with most concurrency frameworks.

Having completed the filesystem API, we now proceed to the server interface layer of the ScalaFTP browser.

The server interface

The server interface comprises of a single actor called `FTPServerActor`. This actor will receive client requests and respond to them serially. If it turns out that the server actor is the sequential bottleneck of the system, we can simply add additional server interface actors to improve horizontal scalability.

We start by defining the different types of messages that the server actor can receive. We follow the convention of defining them inside the companion object of the `FTPServerActor` class:

```
object FTPServerActor {
  sealed trait Command
  case class GetFileList(dir: String) extends Command
  case class CopyFile(src: String, dest: String) extends Command
  case class DeleteFile(path: String) extends Command
  def apply(fs: FileSystem) = Props(classOf[FTPServerActor], fs)
}
```

The actor template of the server actor takes a `FileSystem` object as a parameter. It reacts to the `GetFileList`, `CopyFile`, and `DeleteFile` messages by calling the appropriate methods from the filesystem API:

```
class FTPServerActor(fileSystem: FileSystem) extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case GetFileList(dir) =>
      val filesMap = fileSystem.getFileList(dir)
      val files = filesMap.map(_._2).to[Seq]
      sender ! files
    case CopyFile(srcpath, destpath) =>
      Future {
        Try(fileSystem.copyFile(srcpath, destpath))
      } pipeTo sender
    case DeleteFile(path) =>
      Future {
        Try(fileSystem.deleteFile(path))
      } pipeTo sender
  }
}
```

When the server receives a `GetFileList` message, it calls the `getFileList` method with the specified `dir` directory, and sends a sequence collection with the `FileInfo` objects back to the client. Since `FileInfo` is a case class, it extends the `Serializable` interface, and its instances can be sent over the network.

When the server receives a `CopyFile` or `DeleteFile` message, it calls the appropriate filesystem method asynchronously. The methods in the filesystem API throw exceptions when something goes wrong, so we need to wrap calls to them in `Try` objects. After the asynchronous file operations complete, the resulting `Try` objects are piped back as messages to the sender actor, using the Akka `pipeTo` method.

To start the ScalaFTP server, we need to instantiate and initialize a `FileSystem` object and start the server actor. We parse the network port command-line argument, and use it to create an actor system that is capable of remote communication. For this, we use the `remotingSystem` factory method that we introduced in `Chapter 8`, *Actors*. The remoting actor system then creates an instance of the `FTPServerActor`. This is shown in the following program:

```
object FTPServer extends App {
  val fileSystem = new FileSystem(".")
  fileSystem.init()
  val port = args(0).toInt
  val actorSystem = ch8.remotingSystem("FTPServerSystem", port)
  actorSystem.actorOf(FTPServerActor(fileSystem), "server")
}
```

The ScalaFTP server actor can run inside the same process as the client application, in another process in the same machine, or on a different machine connected with a network. The advantage of the actor model is that we usually need not worry about where the actor runs until we integrate it into the entire application.

When you need to implement a distributed application that runs on different machines, use an actor framework.

Our server program is now complete, and we can run it with the `run` command from SBT. We set the actor system to use the port `12345`:

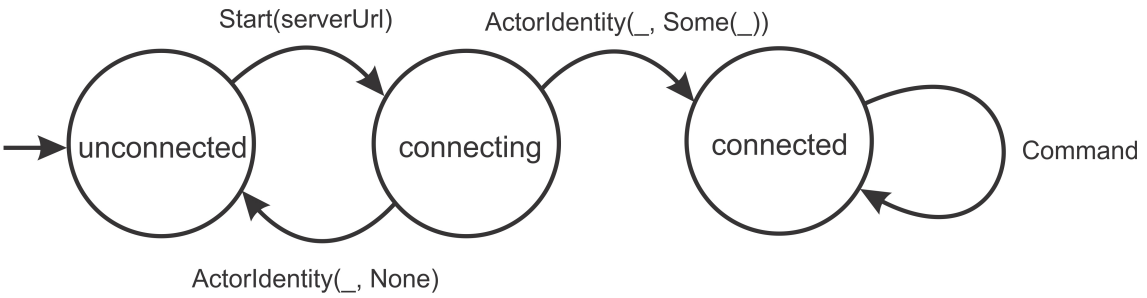
```
run 12345
```

In the following section, we will implement the file navigation API for the ScalaFTP client, which will communicate with the server interface over the network.

Client navigation API

The client API exposes the server interfaces to the client program through asynchronous methods that return future objects. Unlike the server's filesystem API, which runs locally, the client API methods execute remote network requests. Futures are a natural way to model latency in the client API methods, and to avoid blocking during network requests.

Internally, the client API maintains an actor instance that communicates with the server actor. The client actor does not know the actor reference of the server actor when it is created. For this reason, the client actor starts in an **unconnected** state. When it receives the `Start` message with the URL of the server actor system, the client constructs an actor path to the server actor, sends out an `Identify` message, and switches to the **connecting** state. If the actor system is able to find the server actor, the client actor eventually receives the `ActorIdentity` message with the server actor reference. In this case, the client actor switches to the **connected** state, and is able to forward commands to the server. Otherwise, the connection fails and the client actor reverts to the unconnected state. The state diagram of the client actor is shown in the following figure:



We define the `Start` message in the client actor's companion object:

```
object FTPClientActor {
  case class Start(host: String)
}
```

We then define the `FTPClientActor` class and give it an implicit `Timeout` parameter. The `Timeout` parameter will be used later in the Akka ask pattern, when forwarding client requests to the server actor. The stub of the `FTPClientActor` class is as follows:

```
class FTPClientActor(implicit val timeout: Timeout)
extends Actor
```

Before defining the `receive` method, we define behaviors corresponding to different actor states. Once the client actor in the unconnected state receives the `Start` message with the host string, it constructs an actor path to the server and creates an actor selection object. The client actor then sends the `Identify` message to the actor selection, and switches its behavior to `connecting`. This is shown in the following behavior method, named `unconnected`:

```
def unconnected: Actor.Receive = {
  case Start(host) =>
    val serverActorPath =
      s"akka.tcp://FTPServerSystem@$host/user/server"
    val serverActorSel = context.actorSelection(serverActorPath)
    serverActorSel ! Identify(())
    context.become(connecting(sender))
}
```

The `connecting` method creates a behavior given an actor reference to the sender of the `Start` message. We call this actor reference `clientApp`, because the ScalaFTP client application will send the `Start` message to the client actor. Once the client actor receives an `ActorIdentity` message with the `ref` reference to the server actor, it can send `true` back to the `clientApp` reference, indicating that the connection was successful. In this case, the client actor switches to the `connected` behavior. Otherwise, if the client actor receives an `ActorIdentity` message without the server reference, the client actor sends `false` back to the application and reverts to the `unconnected` state:

```
def connecting(clientApp: ActorRef): Actor.Receive = {
  case ActorIdentity(_, Some(ref)) =>
    clientApp ! true
    context.become(connected(ref))
  case ActorIdentity(_, None) =>
    clientApp ! false
    context.become(unconnected)
}
```

The `connected` state uses the `serverActor` server actor reference to forward the `Command` messages. To do so, the client actor uses the Akka ask pattern, which returns a future object with the server's response. The contents of the future are piped back to the original sender of the `Command` message. In this way, the client actor serves as an intermediary between the application, which is the sender, and the server actor. The `connected` method is shown in the following code snippet:

```
def connected(serverActor: ActorRef): Actor.Receive = {
  case command: Command =>
    (serverActor ? command).pipeTo(sender)
}
```

Finally, the `receive` method returns the `unconnected` behavior, in which the client actor is created:

```
def receive = unconnected
```

Having implemented the client actor, we can proceed to the client API layer. We model it as a trait with a `connected` value, the concrete methods `getFileList`, `copyFile`, and `deleteFile`, and an abstract `host` method. The client API creates a private remoting actor system and a client actor. It then instantiates the `connected` future that computes the connection status by sending a `Start` message to the client actor. The methods `getFileList`, `copyFile`, and `deleteFile` are similar. They use the ask pattern on the client actor to obtain a future with the response.

Recall that the actor messages are not typed, and the ask pattern returns a `Future[Any]` object. For this reason, each method in the client API uses the `mapTo` future combinator to restore the type of the message:

```
trait FTPClientApi {
  implicit val timeout: Timeout = Timeout(4 seconds)
  private val props = Props(classOf[FTPClientActor], timeout)
  private val system = ch8.remotingSystem("FTPClientSystem", 0)
  private val clientActor = system.actorOf(props)
  def host: String
  val connected: Future[Boolean] = {
    val f = clientActor ? FTPClientActor.Start
    f.mapTo[Boolean]
  }
  def getFileList(d: String): Future[(String, Seq[FileInfo])] = {
    val f = clientActor ? FTPServerActor.GetFileList(d)
    f.mapTo[Seq[FileInfo]].map(fs => (d, fs))
  }
  def copyFile(src: String, dest: String): Future[String] = {
    val f = clientActor ? FTPServerActor.CopyFile(src, dest)
    f.mapTo[Try[String]].map(_.get)
  }
  def deleteFile(srcpath: String): Future[String] = {
    val f = clientActor ? FTPServerActor.DeleteFile(srcpath)
    f.mapTo[Try[String]].map(_.get)
  }
}
```

Note that the client API does not expose the fact that it uses actors for remote communication. Moreover, the client API is similar to the server API, but the return types of the methods are futures instead of normal values. Futures encode the latency of a method without exposing the cause for the latency, so we often find them at the boundaries between different APIs. We can internally replace the actor communication between the client and the server with the remote `Observable` objects, but that would not change the client API.

In a concurrent application, use futures at the boundaries of the layers to express latency.

Now that we can programmatically communicate with the remote ScalaFTP server, we turn our attention to the user interface of the client program.

The client user interface

In this section, we create the static user interface for the ScalaFTP client program. This graphical frontend will make our ScalaFTP application easy and intuitive to use. We will rely on the Scala Swing library to implement the UI.

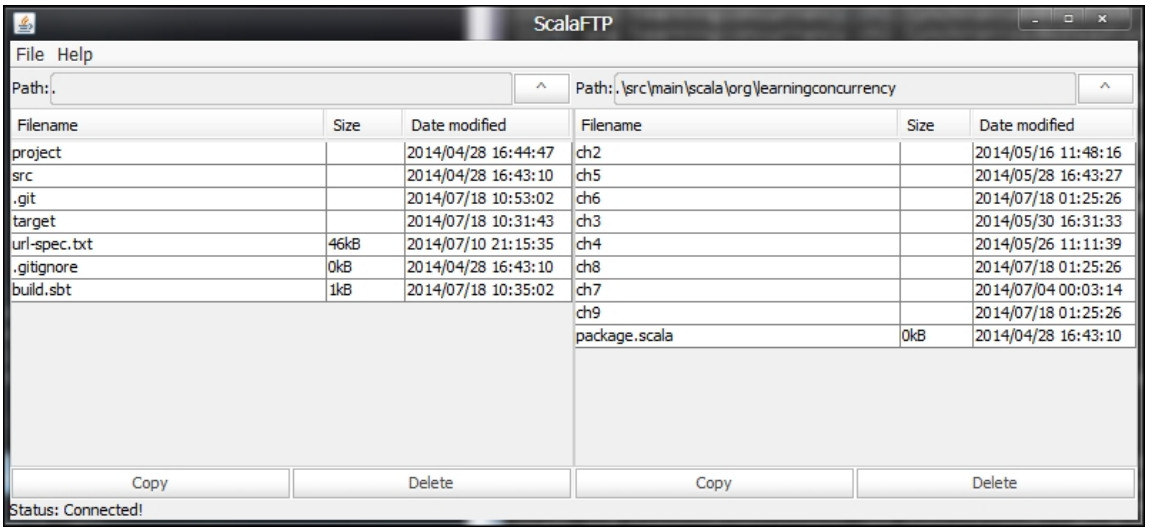
We will implement the client interface in an abstract `FTPClientFrame` class:

```
abstract class FTPClientFrame extends MainFrame {
  title = "ScalaFTP"
}
```

In the remainder of this section, we augment the `FTPClientFrame` class with different UI components. These UI components will enable the end user to interact with the client application, and ultimately with the remote server. Therefore, we will implement the following:

- A menu bar with common application options
- A status bar that displays various user notifications, such as the connection state, status of the last requested operation, and various error messages
- A pair of file panes that display the path to a specific directory in the filesystem, along with its contents, and buttons that start a copy or delete operation

After we are done, the ScalaFTP client program will look like the following screenshot:



We start by implementing the menu bar. When creating Swing components in our UI, we can instantiate an anonymous class that extends a `Menu` or `MenuBar` class, and assign it to a local variable. However, using an anonymous class does not allow access to its custom members. If the anonymous UI component class contains nested components, we are not able to refer to them. Therefore, we will use nested singleton objects to instantiate UI components, as doing this allows us to refer to the object's nested components.

In the following code snippet, we create the `menu` singleton object that extends the `MenuBar` class. We create the `file` and the `help` menu, with the `exit` and `about` menu items, respectively, and take care to add each `Menu` component to the `contents` collection of the enclosing component:

```
object menu extends MenuBar {
  object file extends Menu("File") {
    val exit = new MenuItem("Exit ScalaFTP")
    contents += exit
  }
  object help extends Menu("Help") {
    val about = new MenuItem("About...")
    contents += about
  }
  contents += file += help
}
```

Similarly, we implement the `status` object by extending the `BorderPanel` class. The `BorderPanel` components are used to hold other nested components: in our case, two nested `Label` objects. The anonymous `Label` object always contains the static `Status:` text, while the named `Label` object contains arbitrary status messages. We place the anonymous `Label` object to the left, and the `Label` object with the status messages in the center. This is shown in the following code snippet:

```
object status extends BorderPanel {
  val label = new Label("connecting...", null, Alignment.Left)
  layout(new Label("Status: ")) = West
  layout(label) = Center
}
```

Finally, we implement a custom `FilePane` component that displays the contents of a directory in the remote filesystem. We will have two `FilePane` instances in the client program, so we declare a custom `FilePane` class, which itself extends the `BorderPanel` component type:

```
class FilePane extends BorderPanel
```

We hierarchically decompose the `FilePane` class into three parts: the `pathBar` component that displays the path to the current directory, the `scrollPane` component that allows scrolling through the contents of the current directory, and the `buttons` component that contains the copy and delete buttons. In the following code snippet, we add a non-editable text field with the current path, and an `upButton` component that is used to navigate up the file hierarchy:

```
object pathBar extends BorderPanel {
  val label = new Label("Path:")
  val filePath = new TextField(".") { editable = false }
  val upButton = new Button("^")
  layout(label) = West
  layout(filePath) = Center
  layout(upButton) = East
}
```

The `scrollPane` component contains a `Table` object named `fileTable`. The `fileTable` object will contain the columns named `Filename`, `Size`, and `Date modified`, and each table row will contain a file or a subdirectory within the current working directory. To prevent the user from modifying filenames, sizes, or modification dates, we install a custom `TableModel` object that disallows editing in every row and column. The complete implementation of the `scrollPane` component is as follows:

```
object scrollPane extends ScrollPane {
  val columnNames =
    Array[AnyRef]("Filename", "Size", "Date modified")
  val fileTable = new Table {
    showGrid = true
    model = new DefaultTableModel(columnNames, 0) {
      override def isCellEditable(r: Int, c: Int) = false
    }
    selection.intervalMode = Table.IntervalMode.Single
  }
  contents = fileTable
}
```

The `buttons` singleton object is a `GridPanel` component with one row and two columns. Each column contains a single button, as shown in the following code snippet:

```
object buttons extends GridPanel(1, 2) {
  val copyButton = new Button("Copy")
  val deleteButton = new Button("Delete")
  contents += copyButton += deleteButton
}
```

We then place these custom components inside the `FilePane` component:

```
layout(pathBar) = North
layout(scrollPane) = Center
layout(buttons) = South
```

Finally, we add the `parent` directory field and the list of the files in the current directory, named `dirFiles`, into the `FilePane` class, as well as a few convenience methods to more easily access deeply nested UI components:

```
var parent: String = "."
var dirFiles: Seq[FileInfo] = Nil
def table = scrollPane.fileTable
def currentPath = pathBar.filePath.text
```

Recall that we need one `FilePane` instance on the left-hand side of the client program, and another one on the right. We declare the `files` singleton object inside the `FTPClientFrame` class to hold the two `FilePane` instances, as follows:

```
object files extends GridPanel(1, 2) {
  val leftPane = new FilePane
  val rightPane = new FilePane
  contents += leftPane += rightPane
  def opposite(pane: FilePane) =
    if (pane eq leftPane) rightPane else leftPane
}
```

Finally, we need to place the `menu`, `files`, and `status` components at the top, center, and bottom of the client program:

```
contents = new BorderPanel {
  layout(menu) = North
  layout(files) = Center
  layout(status) = South
}
```

We can already run the client program at this point, and try to interact with it. Unfortunately, the client program does not do anything yet. Clicking on the `FilePane` component, the buttons, or the menu items currently does not have any effect, as we have not yet defined callbacks for various UI actions. In the following section, we will use Rx to complete the functionality of the client application.

Implementing the client logic

We are now ready to add some life to the ScalaFTP client program. We will define the logic layer in the `FTPClientLogic` trait. We only want to allow mixing in the `FTPClientLogic` trait with classes that extend both the `FTPClientFrame` class and the `FTPClientApi` trait, as this allows the logic layer to refer to both UI components and use the client API. Therefore, we give this trait the self-type `FTPClientFrame` class with `FTPClientApi`:

```
trait FTPClientLogic {
  self: FTPClientFrame with FTPClientApi =>
}
```

Before we begin, recall that the Swing components can only be modified from the event-dispatching thread. Similar to how we ensured this using the `swingScheduler` object in `Chapter 6`, *Concurrent Programming with Reactive Extensions*, we now introduce the `swing` method, which takes a block of code and schedules it for execution on the Swing library's event-dispatching thread:

```
def swing(body: =>Unit) = {
  val r = new Runnable { def run() = body }
  javax.swing.SwingUtilities.invokeLater(r)
}
```

Throughout this section, we will rely on the `swing` method in order to ensure that the effect of asynchronous computations occur only on the Swing event-dispatching thread.

The Swing toolkit permits modifying UI components only from the event-dispatching thread, but does not ensure this restriction at compile time, and can unexpectedly fail during runtime.

We begin by relating the connection status to the user interface. Recall that we introduced the `connected` future as part of the client API. Depending on the result of the `connected` future, we either modify the `text` value of the status label to display an error message, or report that the client program has successfully connected to the server. In the latter case, we call the `refreshPane` method to update the contents of the `FilePane` components that we will look at shortly. The following code snippet shows the `onComplete` callback:

```
connected.onComplete {
  case Failure(t) =>
    swing { status.label.text = s"Could not connect: $t" }
  case Success(false) =>
    swing { status.label.text = "Could not find server." }
  case Success(true) =>
    swing {
      status.label.text = "Connected!"
      refreshPane(files.leftPane)
      refreshPane(files.rightPane)
    }
}
```

There are two steps involved in updating the `FilePane` component. First, we need to get the contents of the remote directory from the server. Second, once these contents arrive, we need to refresh the `Table` object in the `FilePane` component. In the following code, we call the `getFileList` method from the client API, and refresh the `Table` object with the `updatePane` method:

```
def refreshPane(pane: FilePane): Unit = {
  val dir = pane.pathBar.filePath.text
  getFileList(dir) onComplete {
    case Success((dir, files)) =>
      swing { updatePane(pane, dir, files) }
    case Failure(t) =>
      swing { status.label.text = s"Could not update pane: $t" }
  }
}
```

The `updatePane` method takes the `dir` directory name and the `files` list, and uses them to update the `FilePane` component `p`. It extracts the `DefaultTableModel` object, and clears its previous contents by setting the row count to `0`. It then updates the `parent` field in the `FilePane` object to the parent of the `dir` directory.

Finally, it stores the `files` list into the `dirFiles` field, and adds a row for each entry:

```
def updatePane(p: FilePane, dir: String, files: Seq[FileInfo]) = {
  val table = p.scrollPane.fileTable
  table.model match {
    case d: DefaultTableModel =>
      d.setRowCount(0)
      p.parent =
        if (dir == ".") "."
        else dir.take(dir.lastIndexOf(File.separator))
      p.dirFiles = files.sortBy(!_.isDir)
      for (f <- p.dirFiles) d.addRow(f.toRow)
  }
}
```

In the preceding method, we relied on the `toRow` method to convert the `FileInfo` object into an array of `String` objects, which the `Table` component works with:

```
def toRow = Array[AnyRef](
  name, if (isDir) "" else size / 1000 + "kB", modified)
```

So far, so good! Our client program is able to connect to the server and show the contents of the root directory. Next, we need to implement the UI logic that allows navigating through the remote filesystem.

When dealing with UI events in `Chapter 6`, *Concurrent Programming with Reactive Extensions*, we augmented our UI components with `Observable` objects. Recall that we added the `clicks` and `texts` methods in order to process events from the `Button` and `TextField` components. In the following code, we augment the `Table` component with the `rowDoubleClicks` method, which returns an `Observable` object with the indices of the rows that have been double-clicked on:

```
implicit class TableOps(val self: Table) {
  def rowDoubleClicks = Observable[Int] { sub =>
    self.peer.addMouseListener(new MouseAdapter {
      override def mouseClicked(e: java.awt.event.MouseEvent) {
        if (e.getClickCount == 2) {
          val row = self.peer.getSelectedRow
          sub.onNext(row)
        }
      }
    })
  }
}
```

To navigate through the remote filesystem, users need to click on the `FilePane` and `upButton` objects. We need to set up this functionality once for each pane, so we define the `setupPane` method for this purpose:

```
def setupPane(pane: FilePane): Unit
```

The first step when reacting to the clicks on the `FilePane` component is mapping each user double-click to the name of the file or directory that has been clicked on. Then, if the double-clicked file is a directory, we update the current `filePath` method, and call the `refreshPane` method:

```
val fileClicks =
  pane.table.rowDoubleClicks.map(row => pane.dirFiles(row))
fileClicks.filter(_.isDir).subscribe { fileInfo =>
  pane.pathBar.filePath.text =
    pane.pathBar.filePath.text + File.separator + fileInfo.name
  refreshPane(pane)
}
```

Similarly, when the user clicks on the `upButton` component, we call the `refreshPane` method to navigate to the parent directory:

```
pane.pathBar.upButton.clicks.subscribe { _ =>
  pane.pathBar.filePath.text = pane.parent
  refreshPane(pane)
}
```

Navigating through the remote filesystem is informative, but we also want to be able to copy and delete the remote files. This requires reacting to UI button clicks, each of which needs to be mapped to the correct, currently selected file. The `rowActions` method produces an event stream with the files that were selected at the time, at the point when a button was clicked:

```
def rowActions(button: Button): Observable[FileInfo] =
  button.clicks
    .map(_ => pane.table.peer.getSelectedRow)
    .filter(_ != -1)
    .map(row => pane.dirFiles(row))
```

Clicking on the copy button will copy the selected file to the directory selected in the opposite pane. We use the `rowActions` method to map the directory on the opposite pane, and call the `copyFile` method from the client API. Recall that the `copyFile` method returns a future, so we need to call the `onComplete` method to process its result asynchronously:

```
rowActions(pane.buttons.copyButton)
  .map(info => (info, files.opposite(pane).currentPath))
  .subscribe { t =>
    val (info, destDir) = t
    val dest = destDir + File.separator + info.name
    copyFile(info.path, dest) onComplete {
      case Success(s) =>
        swing {
          status.label.text = s"File copied: $s"
          refreshPane(pane)
        }
    }
  }
```

We use the `rowActions` method in a similar way, in order to react to clicks on the delete button. Finally, we call the `setupPane` method once for each pane:

```
setupPane(files.leftPane)
setupPane(files.rightPane)
```

Our remote file browser is now fully functional. To test it, we open two separate instances of the terminal, and run SBT in our project directory from both the terminals. We first run the server program:

```
> set fork := true
> run 12345
```

By making sure that the server is running on port `12345`, we can run the client from the second terminal as follows:

```
> set fork := true
> run 127.0.0.1:12345
```

Now, try copying some of our project files between different directories. If you've also implemented the delete functionality, make sure that you back up the project files before deleting anything, just in case. It's not always a good idea to test experimental file-handling utilities on our source code.

Improving the remote file browser

If you successfully ran both the ScalaFTP server, client programs, and copied files around, you might have noticed that, if you delete a file on the disk from an external application, such as your source-code editor, the changes will not be reflected in the ScalaFTP server program. The reason for this is that the server actor does not monitor the filesystem for changes, and the server filesystem layer is not updated when we delete the file.

To account for filesystem changes external to the ScalaFTP server program, we need to monitor the filesystem for changes. This sounds like an ideal case for event streams. Recall that we already did this in `Chapter 6`, *Concurrent Programming with Reactive Extensions*, when we defined the `modified` method to track file modifications. This time, we define the `FileCreated`, `FileDeleted`, and `FileModified` types to denote three different kinds of filesystem events:

```
sealed trait FileEvent
case class FileCreated(path: String) extends FileEvent
case class FileDeleted(path: String) extends FileEvent
case class FileModified(path: String) extends FileEvent
```

By implementing the additional methods in the `FileAlterationListener` interface, we ensure that the resulting `Observable` object produces any one of the three event types. In the following code snippet, we show the relevant part of the `fileSystemEvents` method that produces an `Observable[FileEvent]` object with the filesystem events:

```
override def onFileCreate(file: File) =
  obs.onNext(FileCreated(file.getPath))
override def onFileChange(file: File) =
  obs.onNext(FileModified(file.getPath))
override def onFileDelete(file: File) =
  obs.onNext(FileDeleted(file.getPath))
```

Now that we have an event stream of file events, we can easily modify the filesystem model. We subscribe to the file event stream, and start single-operation transactions to update the `fileSystem` transactional map:

```
fileSystemEvents(".").subscribe { e => e match {
    case FileCreated(path) =>
      fileSystem.files.single(path) = FileInfo(new File(path))
    case FileDeleted(path) =>
      fileSystem.files.single.remove(path)
    case FileModified(path) =>
      fileSystem.files.single(path) = FileInfo(new File(path))
  }
}
```

Now, you can run the server and the client again, and experiment with either deleting or copying files in your editor after the server has started. You will notice that the filesystem changes are detected on the server, and eventually shown when the client is refreshed.

Note that this example was chosen to illustrate how all the different concurrency libraries described in this book work together. However, there is no need to use all of these concurrency libraries in every program. In many situations, we only need a few different concurrency abstractions. Depending on your programming task, you should decide which ones are the best fit.

Never over-engineer your concurrent program. Only use those concurrency libraries that help you solve your specific programming task.

Having studied how to combine different concurrency libraries in a larger application, and having caught a glimpse of how to pick the correct concurrency library, we turn our attention to another aspect of dealing with concurrency, namely, debugging concurrent programs.

Debugging concurrent programs

Concurrent programming is much harder than sequential programming. There are multiple reasons for this. First, the details of the memory model are much more important in concurrent programming, resulting in increased programming complexity. Even on a platform with a well-defined memory model, such as the JVM, the programmer must take care to use proper memory access primitives in order to avoid data races. Then, it is harder to track the execution of a multithreaded program, simply because there are multiple executions proceeding simultaneously. Language debuggers are still focused on tracking the execution of a single thread at a time. Deadlocks and inherent nondeterminism are another source of bugs, neither of which is common in sequential programs. To make things worse, all these issues only have to do with ensuring the correctness of a concurrent program. Ensuring improved throughput and performance opens a separate set of problems, and is often harder than it sounds. Generally, a lot of effort is required to ensure that a concurrent program really runs faster, and performance debugging is an art of its own.

In this section, we survey some of the typical causes of errors in concurrent programs, and inspect different methods of dealing with them. We start with the simplest form of concurrency bugs, which are revealed by a lack of progress in the system.

Deadlocks and lack of progress

Despite the scariness typically associated with the term deadlock, when it comes to debugging concurrent programs, deadlocks are one of the more benevolent forms of concurrency bugs you will encounter. The reason for this is that deadlocks are easy to track down and analyze. In this section, we study how to identify and resolve a deadlock in a concurrent program.

Before we begin, we will make sure that SBT starts the example programs in a separate JVM process. To do this, we enter the following command into the SBT interactive shell:

```
> set fork := true
```

In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we discussed at length what deadlocks are and why they occur. Here, we recall the bank account example introduced in that chapter, which is a canonical example of a deadlock. The bank account example consisted of an `Account` class and the `send` method, which locks two `Account` objects, and transfers a certain amount of money between them:

```
class Account(var money: Int)

def send(a: Account, b: Account, n: Int) = a.synchronized {
  b.synchronized {
    a.money -= n
    b.money += n
  }
}
```

A deadlock nondeterministically occurs when we simultaneously make an attempt to transfer money from account `a` to account `b`, and vice versa, as shown in the following code snippet:

```
val a = new Account(1000)
val b = new Account(2000)
val t1 = ch2.thread { for (i <- 0 until 100) send(a, b, 1) }
val t2 = ch2.thread { for (i <- 0 until 100) send(b, a, 1) }
t1.join()
t2.join()
```

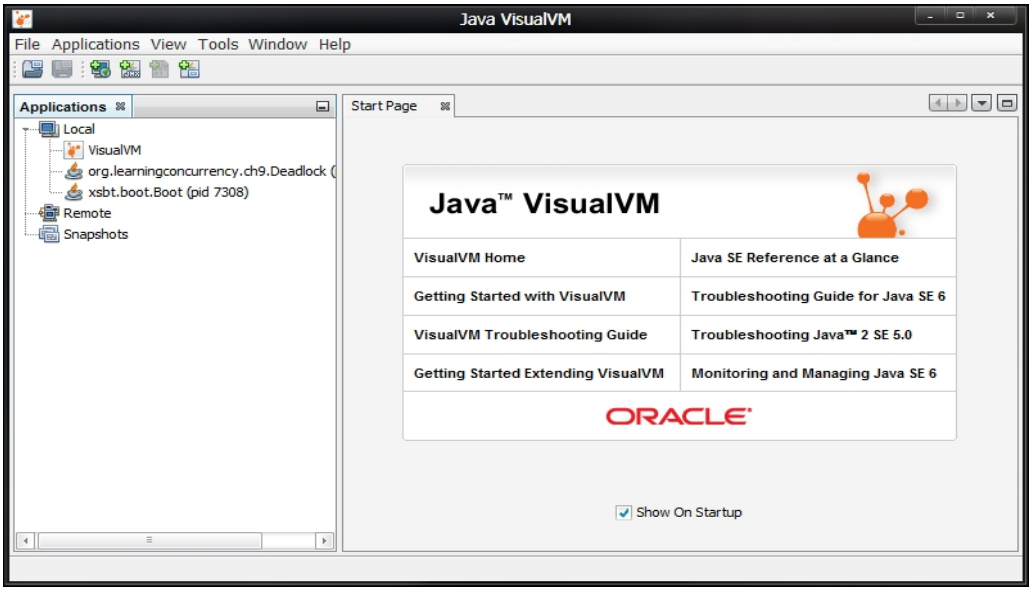
In the preceding snippet, we are using the `thread` method for the thread creation from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. This program never completes, as the `t1` and `t2` threads get suspended in the deadlock state. In a larger program, this effect manifests itself as a lack of response. When a concurrent program fails to produce a result or an end, this is a good indication that part of it is in the deadlock state.

Usually, the most difficult part in debugging a deadlock is localizing it. While this is easy to determine in our simple example, it is much harder in a larger application. However, a defining feature of a deadlock is the lack of any progress, and we can use this to our advantage to determine its cause; we simply need to find the threads that are in a blocked state, and determine their stack-traces.

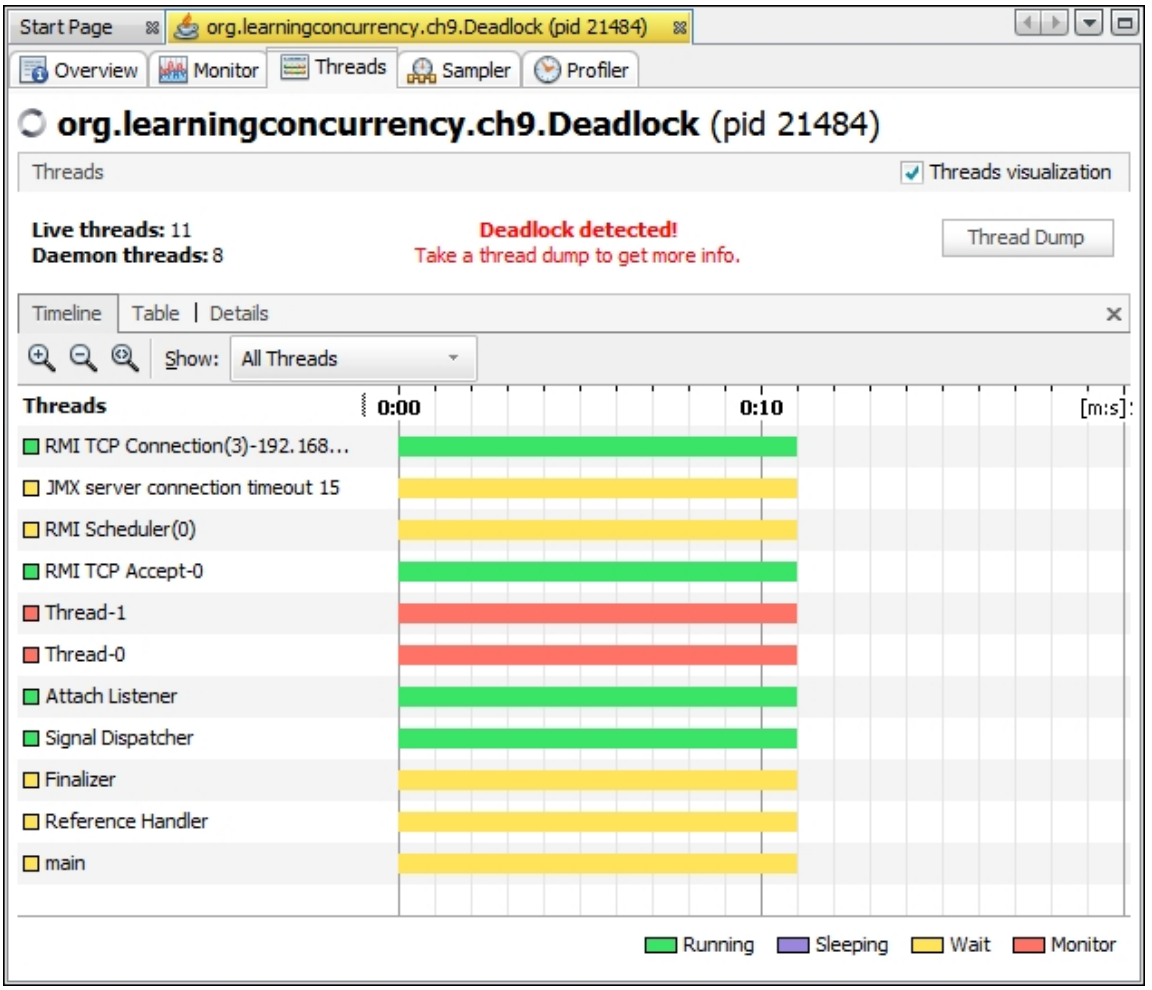
The Java VisualVM tool, which comes bundled with newer JDK distributions, is the simplest way to determine the state of the running Scala and Java applications. Without exiting our deadlocked program, we run the `jvisualvm` program from another terminal instance as follows:

```
$ jvisualvm
```

Once run, the Java VisualVM application shows all the active JVM processes on the current machine. In the following screenshot, the Java VisualVM application shows us the SBT process, our deadlock example program, and VisualVM itself, as the running instances:



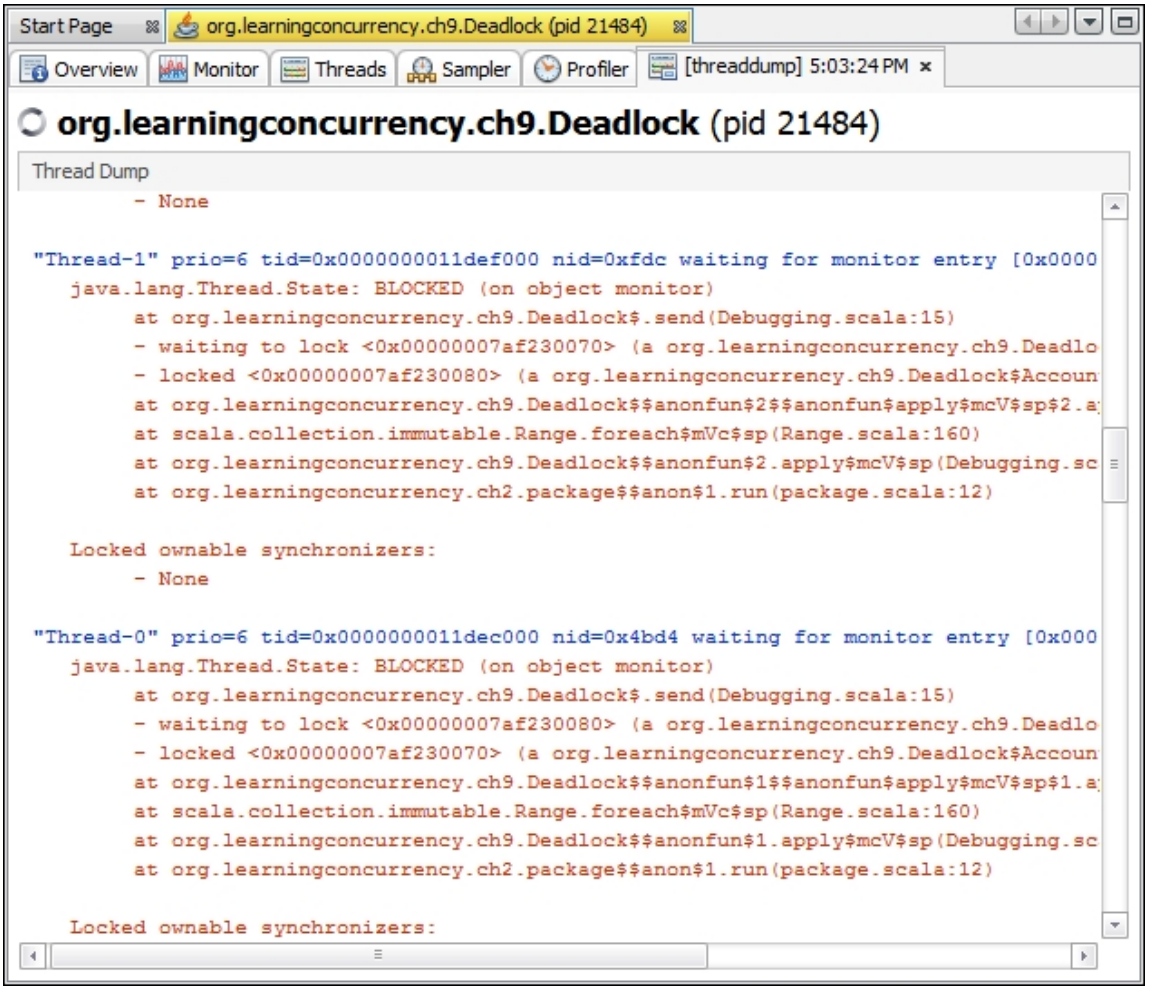
After clicking on the example process, we get the report shown in the following screenshot:



The preceding screenshot shows that there are multiple threads running inside the example process. Most of these threads are part of the virtual machine runtime, and not under the direct control of the programmer. Other threads, such as **main**, **Thread-0**, and **Thread-1** are created by our program.

To determine the cause of the deadlock, we need to inspect the threads in the `BLOCKED` state. By examining their stack-traces, we can determine the cycle that is causing the deadlock. In this case, Java VisualVM was smart enough to automatically determine the cause of the deadlock, and displays the deadlocked threads with the red bar.

After clicking on the **Thread Dump** button, Java VisualVM displays the stack traces of all the threads, as shown in the following screenshot:



The stack traces in the preceding screenshot tell us exactly where in the program the threads are blocked, and why. Both **Thread-0** and **Thread-1** threads are suspended in line 15 of the `Debugging.scala` file. Inspecting these lines of code in our editor reveals that both the threads are blocked on the nested `synchronized` statement. We now know that the cause of the deadlock is the inverted locking order in the `send` method.

We've already discussed how to deal with this type of a deadlock in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. Enforcing a locking order in the `send` method is a textbook example of dealing with deadlocks, and is easy to ensure by assigning unique identifiers to different locks.

In some cases, we are not able to enforce the locking order to avoid deadlocks. For example, in `Chapter 3`, *Traditional Building Blocks of Concurrency*, we learned that the lazy values initialization implicitly calls the `synchronized` statement without our control. There, we eluded deadlocks by avoiding the explicit `synchronized` statements on the object enclosing the lazy value. Another way of preventing deadlocks is to avoid blocking when a resource is not available. In `Chapter 3`, *Traditional Building Blocks of Concurrency*, we learned that custom locks can return an error value, letting the rest of the program decide how to proceed if a lock is not available.

Besides deadlocks, there are other kinds of concurrency bugs that are associated with a lack of progress. We've already seen examples of **starvation**, in which a concurrent computation is denied access to the required resources. In `Chapter 4`, *Asynchronous Programming with Futures and Promises*, we started many futures simultaneously, and suspended them by calling the `sleep` method. As a result, the thread-pool underlying the `ExecutionContext` object became exhausted, and no additional futures could execute until the `sleep` method returned.

In a **livelock**, different concurrent computations are not suspended, and constantly change their state, but are unable to make progress. A livelock is akin to the situation in which two people approach each other on the street, and constantly try to move to the opposite side in order to allow the other person to pass. As a result, neither person moves on, and they constantly move from one side to the other. What is common to these kinds of errors is that the system makes no or very little progress, making them easy to identify.

Looking for a deadlock is like hunting for a dead animal. Since it implies no progress, a deadlock is tracked down more easily than other kinds of concurrency bugs. In the following section, we will study a more malevolent class of concurrency errors that manifest themselves through incorrect program outputs.

Debugging incorrect program outputs

In this section, we study a broader range of concurrency bugs that manifest themselves as incorrect outputs of the program. Generally, these kinds of errors are harder to track, because their effects become apparent long after the actual error took place. A real-world example of such an error is a piece of broken glass lying on the road. You don't see the glass when you drive your car, and accidentally run over it. By the time your tire runs flat and you realize what happened, it is difficult to figure out where exactly along the road the glass was.

There are two main ways in which an error can appear. First, the concurrent program can consistently produce the same erroneous outputs. When this happens, we can consider ourselves lucky, as we are able to consistently reproduce the error to study it. Conversely, the incorrect output might appear only occasionally, in some executions of the program. This is a much less desired situation. A buggy concurrent program might exhibit incorrect behavior only occasionally due to its inherent nondeterminism. We will look at both deterministic and nondeterministic errors in the rest of the section.

The goal of this section will be to implement the `fold` method on futures. Given a sequence of future objects, a zero value, and the `folding` operator, the `fold` method will return a future object with the `folding` operator that is applied between all the values. We will require the `folding` operator to be commutative, associative, and without side effects. The `fold` method will closely correspond to the `foldLeft` method on collections. The signature of the `fold` method on futures will be as follows:

```
def fold[T](fs: Seq[Future[T]])(z: T)(op: (T, T) => T): Future[T]
```

One use case for the `fold` method is to compute the sum of the values in many different future objects, which cannot be done directly with the `foldLeft` method on collections. This is illustrated in the following code snippet:

```
val fs: Seq[Future[Int]] = for (i <- 0 until 5) yield Future { i }
val sum: Future[Int] = fold(fs)(0)(_ + _)
```

We will implement the `fold` method in two steps. First, we will accumulate the values from all the values in the `fs` sequence by applying the `op` operator on them. Accumulating the values will give us the accumulation value of the resulting future. Then, after all the futures complete, we will complete the resulting future with the accumulation value.

We start by implementing several basic concurrency abstractions that will help us implement the `fold` method. A **concurrent accumulator** is a concurrency facility that allows you to keep track of an accumulation of values. Here, the values can be integers, and the accumulation can be their sum. A concurrent accumulator comes with the `add` method that is used to add new values, and the `apply` method that is used to obtain the current state of the accumulation. We present the simplest possible lock-free implementation of a concurrent accumulator, which uses atomic variables from `Chapter 3`, *Traditional Building Blocks of Concurrency*. The `Accumulator` class takes the type `T` of the accumulation, a `z` initial value, and an `op` reduction operator, and is shown in the following code snippet:

```
class Accumulator[T](z: T)(op: (T, T) => T) {
  private val value = new AtomicReference(z)
  def apply(): T = value.get
  @tailrec final def add(v: T): Unit = {
    val ov = value.get
    val nv = op(ov, v)
    if (!value.compareAndSet(ov, nv)) add(v)
  }
}
```

The `Accumulator` implementation has a private atomic variable, named `value`, initialized with the `z` value, and is used to track the value of the accumulation. The `apply` method is easy to implement; we simply call the linearizable `get` method to obtain the current accumulation value. The `add` method must use the `compareAndSet` operation to atomically update the accumulation. Here, we read the `ov` current value of the atomic variable, call the `op` operator to compute the new `nv` accumulation value, and, finally, call the `compareAndSet` operation to replace the old `ov` accumulation value with the new `nv` value. If the `compareAndSet` operation returns `false`, then the accumulation was modified, as it was previously read, and the tail-recursive `add` operation must be retried. We studied this technique at length in `Chapter 3`, *Traditional Building Blocks of Concurrency*.

Note that, because of the retries, the `op` operator can be invoked multiple times with the same `v` argument. Therefore, our lock-free concurrent accumulator implementation only works correctly with a reduction operator that is free from side effects.

Next, we will need a facility that allows different futures to synchronize. A **countdown latch** is a synchronization primitive that performs a specific action once a specified number of threads agree that the action can be performed. Our `CountDownLatch` class takes the number of threads `n`, and an `action` block. The latch keeps an atomic integer variable, named `left`, with the current countdown value, and defines a `count` method, which decreases the value of the `left` atomic variable. After `n` calls of the `count` method, the `action` block is invoked once. This is shown in the following code snippet:

```
class CountDownLatch(n: Int)(action: =>Unit) {
  private val left = new AtomicInteger(n)
  def count() =
    if (left.decrementAndGet() <= 1) action
}
```

We now have all the prerequisites for implementing the `fold` method. This method needs to return a future object, so we start by instantiating a promise object. The promise will enable us to return the future object corresponding to the promise. We have seen this pattern many times in `Chapter 4`, *Asynchronous Programming with Futures and Promises*. Next, we need some way of combining the values from the different futures, so we instantiate an `Accumulator` object with the initial `z` value and the `op` reduction operator. We can complete the promise with the value of the accumulator only after all the futures complete, so we create a countdown latch with the countdown value set to the number of the futures. The action associated with the countdown latch completes the promise with the value of the accumulator, and we decide to use the `trySuccess` method for this purpose. Finally, we need to install callbacks on all the futures, which update the accumulator, and then call the `count` method on the latch. The complete implementation of the `fold` method is shown in the following code snippet:

```
def fold[T](fs: Seq[Future[T]])(z: T)(op: (T, T) => T) = {
  val p = Promise[T]()
  val accu = new Accumulator(z)(op)
  val latch = new CountDownLatch(fs.length)({
    p.trySuccess(accu())
  })
  for (f <- fs) f foreach { case v =>
    accu.add(v)
    latch.count()
  }
  p.future
}
```

If you paid close attention, you might have noticed that we deliberately introduced an error somewhere in the `fold` implementation. Don't worry if you did not notice this error yet, as we will now analyze how the error manifests itself, and how to identify it. To test the `fold` method, we run the following example program:

```
val fs = for (i <- 0 until 5) yield Future { i }
val folded = fold(fs)(0)(_ + _)
folded foreach { case v => log(s"folded: $v") }
```

On our machine, running this program prints the correct value `10`. We already feel confident that we implemented the program correctly, but we run the program again, just to be sure. This time, however, the program outputs the value `7`. It turns out that we have a bug in our implementation of the `fold` method. Even worse, the bug manifests itself nondeterministically!

In sequential programming, the normal response would be to use the debugger, and proceed stepwise through the program, until we reach the buggy behavior. In concurrent programming, this approach often does not help. By tracking the progress of one thread in the debugger, we are arbitrarily delaying it, and changing the execution schedule of the program. The bug appears nondeterministically, so it might not appear when we run the program in the debugger.

Instead of going forward through the program, to find the culprit, we work our way backwards through the code. The future is completed with the incorrect value, meaning that some thread must have inserted the incorrect value into the corresponding promise. We should insert a breakpoint at the promise completion point and observe what happens. To keep things simple, we avoid using the debugger, and insert a simple `println` statement to track the value with which the promise is completed:

```
val total = accu()
println(total)
p.trySuccess(total)
```

Running the program again gives the following output:

```
8
10
ForkJoinPool-1-worker-1: folded: 8
```

This reveals a surprising fact: the promise is, in fact, completed twice. The first time, some thread uses the value `8` of the accumulator, and the second time, another thread uses the value `10`. This also means that the `action` block of the countdown latch was called twice, so we need to find out why. We therefore modify the `count` method in order to track when the `action` block is called:

```
def count() = {
  val v = left.decrementAndGet()
  if (v <= 1) {
    println(v)
    action
  }
}
```

The program output now shows the following content:

```
1
0
ForkJoinPool-1-worker-15: folded: 7
```

It appears that the `action` block is called not only on the last decrement, but also on one before the last. This is because the `decrementAndGet` method first decrements the atomic integer, and then returns its value, rather than the other way around. The way to fix this is to either call the `getAndDecrement` method, or change the `if` statement. We reimplement the `count` method as follows:

```
def count() =
  if (left.decrementAndGet() == 0) action
```

Note that, if we had used the `success` method in place of `trySuccess`, we would have learned about the error much earlier. Let's change the implementation of the `action` block in the `fold` method to use the `success` method:

```
p.success(accu()))
```

Running the program with this change, and the previously incorrect `count` method, results in the following exception:

```
java.lang.IllegalStateException: Promise already completed.
```

This is much better. The output of the program is incorrect, but the exception consistently occurs each time that the program is run. Along with the cause of the error, we consistently get a full stack-trace to quickly determine where the error has occurred. We say that the error occurs deterministically.

Recall that, in `Chapter 4`, *Asynchronous Programming with Futures and Promises*, we used the `tryComplete` method to implement the `or` combinator on futures. This combinator was inherently nondeterministic, so we were forced to use the `tryComplete` method. However, there is no need to use any of the `tryXYZ` methods in the `fold` implementation, as the `fold` method should always return a future with the same result. Wherever possible, you should use the `complete`, `success`, and `failure` methods in place of the `tryComplete`, `trySuccess`, and `tryFailure` methods. More generally, always strive for deterministic semantics, unless the program itself is inherently nondeterministic.

Program defensively: check for consistency violations often, prefer determinism, and fail at an early stage. This simplifies the debugging process when program errors arise.

In the following section, we turn to a different correctness aspect in concurrent programs, namely, testing their performance.

Performance debugging

When it comes to performance debugging, the field is virtually endless. A separate book on the subject would barely scratch the surface. The goal of this section is to show you two basic examples that will teach you the basics of analyzing and resolving performance problems in concurrent Scala programs.

In recent years, processor clock rates have reached a limit, and processor vendors have struggled to improve single processor performance. As a consequence, multicore processors have overwhelmed the consumer market. Their main goal is to offer increased performance by increasing parallelism. Ultimately, the goal of concurrent and parallel computing is to increase the program performance.

There are two ways in which program performance can be improved. The first is through optimizing the program, so that its sequential instance runs as fast as possible. The second approach is to run parts of the program in parallel. In concurrent and parallel computing, both approaches are key to achieving optimal performance. It does not make sense to parallelize a program that is much slower than the optimal sequential program.

Thus, we will study both, how to optimize, and how to parallelize a concurrent program. We will start with a single-threaded version of the program that uses a concurrent accumulator, and ensure that it runs efficiently. Then, we will ensure that the program is also scalable, that is, adding additional processors makes it faster.

The first step in debugging the performance of a parallel program is to measure its running time. As stated in `Chapter 5`, *Data-Parallel Collections*, benchmarking the program performance is the only principled way of knowing how fast the program is and finding its bottlenecks. This task can be complicated on the JVM, due to effects such as garbage collection, JIT compilation, and adaptive optimizations.

Fortunately, the Scala ecosystem comes with a tool called ScalaMeter, which is designed to easily test the performance of both Scala and Java programs. The ScalaMeter tool can be used in two ways. First, ScalaMeter allows defining performance regression tests, which are essentially unit tests for performance. Second, ScalaMeter allows inline benchmarking that is used to benchmark parts of the running application. In this section, we will keep things simple, and only use ScalaMeter's inline benchmarking feature. We add the following line to our `build.sbt` file:

```
libraryDependencies +=
  "com.storm-enroute" %% "scalameter-core" % "0.6"
```

To use ScalaMeter inside our programs, we need to import the following package:

```
import org.scalameter._
```

This package gives us access to the `measure` statement that is used to measure various performance metrics. By default, this method measures the running time of a snippet of code. Let's use it to measure how long it takes to add one million integers to the `Accumulator` object defined in the preceding section:

```
val time = measure {
  val acc = new Accumulator(0)(_ + _)
  var i = 0
  val total = 1000000
  while (i < total) {
    acc.add(i)
    i += 1
  }
}
```

Printing the `time` value gives us the following output:

```
Running time: 34.60
```

From this, we might conclude that adding one million integers takes approximately 34 milliseconds. However, this conclusion is wrong. As discussed in `Chapter 5`, *Data-Parallel Collections*, after a JVM program is run, it goes through a warm-up phase. The program usually achieves the best possible performance only after the warm-up phase is completed. To measure the relevant running time more accurately, we need to first ensure that the JVM reached stable performance.

The good news is that ScalaMeter can do this automatically. In the following code, we configure the `measure` call to use the default warmer implementation, called `Warmer.Default`. We set several configuration parameters, such as the minimum number of warm-up runs, the maximum number of warm-up runs, and the number of benchmark runs that are used to compute the average running time. Finally, we set the `verbose` key to `true` in order to get more logging output about ScalaMeter's execution. This is shown in the following code snippet:

```
val accTime = config(
  Key.exec.minWarmupRuns -> 20,
  Key.exec.maxWarmupRuns -> 40,
  Key.exec.benchRuns -> 30,
  Key.verbose -> true
) withWarmer(new Warmer.Default) measure {
  val acc = new Accumulator(0L)(_ + _)
  var i = 0
  val total = 1000000
  while (i < total) {
    acc.add(i)
    i += 1
  }
}
println("Accumulator time: " + accTime)
```

When running this, make sure that there are no active applications running in the background on your computer. Running this snippet of code gives us the following output:

```
18. warmup run running time: 17.285859
GC detected.
19. warmup run running time: 21.460975
20. warmup run running time: 16.557505
21. warmup run running time: 17.712535
22. warmup run running time: 16.355897
Steady-state detected.
Accumulator time: 17.24
```

We can now see how the running time changes during the warm-up runs. Eventually, ScalaMeter detects a steady state and outputs the running time. We now have a value of `17.24` milliseconds, which is a good estimate.

A closer inspection of the ScalaMeter output reveals that, occasionally, a **Garbage Collection** (**GC**) cycle occurs. These GC cycles appear periodically during the execution of our code snippet, so we conclude that something in the `add` method allocates heap objects. However, the `add` implementation does not contain any `new` statements. The object allocation must be happening implicitly somehow.

Note that the `Accumulator` class is generic. It takes a `T` type parameter, which denotes the type of the accumulation. Scala allows using both the reference types, such as `String` or `Option`, and primitive types, such as `Int` or `Long`, as class-type parameters. Although this conveniently allows treating both the primitive and reference types in the same way, it has the unfortunate side effect that the primitive values passed to generic classes are converted into heap objects. This process is known as auto-boxing, and it hurts the performance in various ways. First, it is much slower than just passing a primitive value. Second, it causes GC cycles more frequently. Third, it affects cache-locality and might cause memory contention. In the case of the `Accumulator` class, each time we call the `add` method with a `Long` value, a `java.lang.Long` object is created on the heap.

In practice, boxing is sometimes problematic, and sometimes not. Generally, it should be avoided in high-performance code. In our case, we can avoid boxing by creating an accumulator specialized for the `Long` values. We show it in the following code snippet:

```
class LongAccumulator(z: Long)(op: (Long, Long) => Long) {
  private val value = new AtomicLong(z)
  @tailrec final def add(v: Long): Unit = {
    val ov = value.get
    val nv = op(ov, v)
    if (!value.compareAndSet(ov, nv)) add(v)
  }
  def apply() = value.get
}
```

Re-running the program reveals that the new accumulator is almost twice as fast:

```
Long accumulator time: 8.88
```

Boxing can slow down the program by a factor of anywhere between one and several dozen. This depends on the specific ratio of object allocations and other work, and it needs to be measured on a per-program basis.

An unfortunate side effect is that we can only use the new accumulator implementation for `Long` values. However, Scala allows us to retain the generic nature of the previous `Accumulator` implementation. The Scala specialization feature allows the annotation of class type parameters with the `@specialized` annotation, instructing the Scala compiler to automatically generate versions of the generic class for primitive types, such as `Long`, and avoid boxing. We will not dive any further into this topic, and instead let interested readers find out more on their own.

Now that we know how to identify performance issues and optimize sequential programs, we study how to improve the performance by increasing the parallelism level. Let's parallelize the previous program by adding one million integers from four separate threads. This is shown in the following code snippet:

```
val intAccTime4 = config(
  Key.exec.minWarmupRuns -> 20,
  Key.exec.maxWarmupRuns -> 40,
  Key.exec.benchRuns -> 30,
  Key.verbose -> true
) withWarmer(new Warmer.Default) measure {
  val acc = new LongAccumulator(0L)(_ + _)
  val total = 1000000
  val p = 4
  val threads = for (j <- 0 until p) yield ch2.thread {
    val start = j * total / p
    var i = start
    while (i < start + total / p) {
      acc.add(i)
      i += 1
    }
  }
  for (t <- threads) t.join()
}
println("4 threads integer accumulator time: " + intAccTime4)
```

In the preceding example, we distribute the work of adding 1 million integers across four different threads, so we expect the running time of the program to increase four times. Sadly, running the program reveals that our expectations were wrong:

```
4 threads integer accumulator time: 95.85
```

As pointed out in `Chapter 5`, *Data-Parallel Collections*, perpetually writing to the same memory location from multiple threads results in memory contention issues. In most computer architectures, cache-lines need to be exchanged between the processors writing to the same memory location, and this slows down the program. In our case, the contention point is the `AtomicLong` object in the `LongAccumulator` class. Simultaneously invoking the `compareAndSet` operation on the same memory location does not scale.

To address the issue of memory contention, we need to somehow disperse the writes throughout different cache-lines. Instead of adding the accumulated value to a single memory location, we will maintain many memory locations with partial accumulation values. When some processor calls the `add` method, it will pick one of these memory locations and update the partial accumulation. When a processor calls the `apply` method, it will scan all the partial accumulations and add them together. In this implementation, we trade the performance of the `apply` method for the improved scalability of the `add` method. This trade-off is acceptable in many cases, including our `fold` method, where we call the `add` method many times, but the `apply` method only once.

Furthermore, note that the new `apply` implementation is not linearizable, as explained in `Chapter 7`, *Software Transactional Memory*. If some processor calls the `apply` method when multiple processors are calling the `add` method, the resulting accumulation value can be slightly incorrect. However, if no other processor calls the `add` method when the `apply` method is called, the resulting accumulation value will be correct. We say that the new `apply` implementation is **quiescently consistent** with respect to the `add` method.

Note that this property is sufficient for ensuring the correctness of the preceding `fold` implementation, because the `fold` method only calls the `apply` method after all the `add` calls are completed.

We now show the implementation of the `ParLongAccumulator` class, which uses an `AtomicLongArray` object, named `values`, to keep the partial accumulation values. Atomic arrays are arrays on which we can call operations such as the `compareAndSet` method. Conceptually, an `AtomicLongArray` is equivalent to an array of `AtomicLong` objects, but is more memory-efficient.

The `ParLongAccumulator` class must choose a proper size for the `AtomicLongArray` object. Setting the size of the array to the number of processors will not make the memory contention problems go away. Recall from `Chapter 3`, *Traditional Building Blocks of Concurrency*, that a processor needs to own a cache-line in exclusive mode before writing to it. A cache-line size is typically 64 bytes. This means that on a 32-bit JVM, eight consecutive entries in an `AtomicLongArray` object fit inside a single cache-line. Even when different processors write to separate `AtomicLongArray` entries, memory contention occurs if these entries lie in the same cache-line. This effect is known as **false-sharing**. A necessary precondition in avoiding false-sharing is to make the array size at least eight times larger than the number of processors.

A `ParLongAccumulator` object is used by many different threads simultaneously. In most programs, there are many more threads than processors. To reduce false-sharing, as much as possible, we set the size of the `values` array to 128 times the number of processors:

```
import scala.util.hashing
class ParLongAccumulator(z: Long)(op: (Long, Long) => Long) {
  private val par = Runtime.getRuntime.availableProcessors * 128
  private val values = new AtomicLongArray(par)
  @tailrec final def add(v: Long): Unit = {
    val id = Thread.currentThread.getId.toInt
    val pos = math.abs(hashing.byteswap32(id)) % par
    val ov = values.get(pos)
    val nv = op(ov, v)
    if (!values.compareAndSet(pos, ov, nv)) add(v)
  }
  def apply(): Long = {
    var total = z
    for (i <- 0 until values.length)
      total = op(total, values.get(i))
    total
  }
}
```

The new `add` implementation is similar to the previous one. The main difference is that the new implementation needs to pick the `pos` memory location for the partial accumulation value. Different processors should pick different memory locations based on their index. Unfortunately, standard APIs on the JVM do not provide the index of the current processor. An adequate approximation is to compute the `pos` partial accumulation location from the current thread ID. We additionally use the `byteswap32` hashing function to effectively randomize the location in the array. This decreases the likelihood that two threads with adjacent IDs end up writing to adjacent entries in the array, and reduces the possibility of false-sharing.

Running the program demonstrates that we reached our goal, and improved the program performance by a factor of almost three:

```
Parallel integer accumulator time: 3.34
```

There are additional ways to improve our `ParLongAccumulator` class. One is to further reduce false sharing by choosing the entries in the `values` array more randomly. Another is to ensure that the `apply` method is not only quiescently consistent, but also linearizable. In the interest of keeping this section simple and clear, we do not dive further into these topics, but let interested readers explore them on their own.

In this and the preceding sections, we summarized the different styles of concurrency and studied the basics of dealing with concurrency bugs. This gave us a useful insight into the big picture, but the theory that we learned is only valuable if it can be applied in practice. We designed and implemented a remote file browser application, a practical example of a large concurrent application. This gave us insight into both the theoretical and practical side of concurrent programming.

Summary

Having seen the technical details of a variety of different concurrency libraries in the preceding chapters, we took a couple of steps back and presented a more cohesive view of Scala concurrency. After presenting a taxonomy of different styles of concurrency, we outlined the use cases for different concurrency frameworks. We then studied how to debug concurrent programs and analyze their performance. Finally, we combined the different concurrency frameworks together to implement a real-world distributed application: a remote file browser.

The best theory is inspired by practice, and the best practice is inspired by theory. This book has given you a fair amount of both. To deepen your understanding of concurrent computing, consider studying the references listed at the end of each chapter: you should already be able to grasp most of them. Importantly, to improve your practical concurrent programming skills, try to solve the exercises from this book. Finally, start building your own concurrent applications. By now, you must have understood both how high-level concurrency abstractions work and how to use them together, and are on the path to becoming a true concurrency expert.

Exercises

The following exercises will improve your skills in building practical concurrent applications. Some of them require extending the ScalaFTP program from this chapter, while others require implementing concurrent applications from scratch. Finally, several exercises are dedicated to testing the performance and scalability of concurrent programs:

1. Extend the ScalaFTP application to allow the addition of directories to the remote filesystem.
2. Extend the ScalaFTP application so that the changes in the server filesystem are automatically reflected in the client program.
3. Extend the ScalaFTP application so that it allows parallel regex searches over filenames in the remote filesystem.
4. Extend the ScalaFTP server so that it allows recursively copying directories.
5. Implement the download and upload functionality, and use `Observable` objects to display the file transfer progress in a Swing `ProgressBar` component.
6. Extend the ScalaFTP client implementation so that a `FilePane` can display either a remote or a local filesystem's contents.
7. Design and implement a distributed chat application.
8. Design and implement a Paint program with collaborative editing.
9. Compare the duration of creating and starting a new thread, and waiting for its termination, against the duration of starting a computation using `Future.apply` and waiting for the completion of the corresponding `Future` object.
10. A pool is one of the simplest collection abstractions, which allows the addition and extraction of elements. The `remove` operation returns any element that was previously added to the pool. A concurrent pool is represented by the `ConcurrentPool` class:

```
class ConcurrentPool[T] {
  def add(x: T): Unit = ???
  def remove(): T = ???
  def isEmpty(): Boolean = ???
}
```

    Implement the concurrent pool and make sure that its operations are linearizable. Measure and ensure high performance and scalability of your implementation.

11. Compare the performance and scalability of the Treiber stack from the exercise in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, against the transactional sorted list from `Chapter 7`, *Software Transactional Memory*. How are they compared to the concurrent pool from the previous exercise?
12. Implement the `getUniqueId` method from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. Measure and ensure high performance and scalability of your implementation.
13. Implement a lock-free concurrent linked list and a lock-based concurrent linked list that support linearizable prepend and append operations. Both implementations must be singly linked lists. Measure the performance of inserting many elements.
14. A barrier is a concurrent object that allows *N* threads to synchronize at some point in the program. A barrier exposes a single method, `await`, which effectively blocks the thread until all *N* threads call `await`. After all *N* threads call `await`, the `await` invocations of all the threads immediately return. Blocking can be done, for example, by busy-waiting. Use atomic integers to implement a barrier. Measure the performance of your implementation for one, two, four, and eight threads, and some large number of calls to `await`.

10
Reactors

"Simplicity is prerequisite for reliability."
-Edsger W. Dijkstra

Location-transparency, serializable event-handling, and non-blocking semantics of sends, make the actor model a powerful foundation for building distributed systems. However, the actor model has several important limitations, which only become apparent when building larger systems. First, actors cannot simultaneously contain multiple message entry points. All messages must arrive through the same `receive` block. Consequently, two different protocols cannot reuse the same message type, and must be aware of each other. The main example where we saw this was the `Identify` message, which required users to incorporate a unique token into the message. Second, actors cannot await specific combinations of messages. For example, it is cumbersome to simultaneously send a request message to two target actors, and proceed after both replies arrive. Third, the `receive` statement is not a first-class citizen. Event streams, which we saw in the Rx framework, are first-class citizens, and this improves program composition, modularity, and separation of concerns.

In this chapter, we study the reactor programming model for distributed computing, which retains the advantages of the actor model, but overcomes the above limitations. This framework allows creating complex concurrent and distributed applications more easily, by providing correct, robust, and composable abstractions for distributed programming. Similar to the actor model, the reactor model allows writing location-transparent programs. Clear separation between units of concurrency is achieved through special entities called reactors. This separation makes it easier to reason about concurrent programs, as was the case with actors. However, computations and message exchange patterns can be more easily subdivided into modular components in the reactor model. The improved composition at the core of the reactor model is the result of a careful integration of the traditional actor model and functional reactive programming concepts.

We use the Reactors framework throughout this chapter to learn about the reactor programming model. We will cover the following topics:

- Utilizing and composing event-streams to structure logic within a reactor
- Defining reactors and starting reactor instances
- Customizing reactor instances and using custom schedulers
- Using reactor system services to access non-standard events, and defining custom services
- The basics of protocol composition, along with several concrete protocol examples

We start by recounting what we learned about concurrent and distributed programming, and explaining why the reactor model is important.

The need for reactors

As you may have concluded by reading this book, writing concurrent and distributed programs is not easy. Ensuring program correctness, scalability, and fault-tolerance is harder than in a sequential program. Here, we recall some of the reasons for this:

- First of all, most concurrent and distributed computations are, by their nature, non-deterministic. This non-determinism is not a consequence of poor programming abstractions, but is inherent in systems that need to react to external events.
- Data races are a basic characteristic of most shared-memory multicore systems. Combined with inherent non-determinism, these lead to subtle bugs that are hard to detect or reproduce.
- When it comes to distributed computing, things get even more complicated. Random faults, network outages, or interruptions, present in distributed programming, compromise correctness and robustness of distributed systems.
- Furthermore, shared-memory programs do not work in distributed environments, and existing shared-memory programs are not easily ported to a distributed setup.

There is one more reason why concurrent and distributed programming is hard. When building large systems, we would like to compose simpler program components into larger entities. However, it is often hard to correctly compose concurrent and distributed programs. Correctness of specific components is no guarantee for global program correctness when those components are used together. Deadlocks inherent to locks are one such example, and potential race conditions in actors are another.

Frameworks that we have seen in this book strive to address the aforementioned problems in concurrent and distributed programming. Different concurrency models try to address these issues from different angles. The intent of the reactor model, described in this chapter, is to borrow some of the best characteristics of existing frameworks, such as location-transparency, serializability and data-race freedom, and especially address the issue of composability.

To achieve these goals, the reactor model employs several minimalist abstractions, which can compose into complex protocols, algorithms, and program components. In particular, the model is based on the following:

- Location-transparent **reactors**, lightweight entities that execute concurrently with each other, but are internally always single-threaded, and can be ported from a single machine to a distributed setting. Every reactor is created with one main event stream. A reactor is a generalization of an actor from the traditional actor model.
- Asynchronous first-class **event streams** that can be reasoned about in a declarative, functional manner, and are the basis for composing components. An event stream is the reading end of a channel. Only the reactor that owns the channel can read from the corresponding event stream. Event streams cannot be shared between different reactors. To borrow the analogy from the actor model, an event stream is a counterpart of the `receive` statement.
- **Channels** that can be shared between reactors, and are used to send events asynchronously. A channel is the writing end of the corresponding event stream, and any number of reactors can write to a channel. A channel is a close equivalent of the actor reference that we saw in the actor model.

These three unique abstractions are the core prerequisite for building powerful distributed computing abstractions. Most other utilities in the Reactors framework, which we study in this chapter, are built in terms of reactors, channels, and event streams.

Getting started with Reactors

This section contains instructions on how to get Reactors working in your project. The Reactors framework has multiple languages frontend, and works on multiple platforms. At the time of writing this book, Reactors can be used with Scala and Java as a JVM library, or alternatively on NodeJS or inside the browser if you are using the `Scala.js` frontend of Reactors.

If you are developing with SBT, the easiest way is to include Reactors into your project as a library dependency. To get started with `Reactors.IO`, you should grab the latest snapshot version distributed on **Maven**. If you are using SBT, add the following to your project definition:

```
resolvers ++= Seq(
  "Sonatype OSS Snapshots" at
"https://oss.sonatype.org/content/repositories/snapshots",
  "Sonatype OSS Releases" at
"https://oss.sonatype.org/content/repositories/releases"
)
libraryDependencies ++= Seq(
  "io.reactors" %% "reactors" % "0.8")
```

At the time of writing this, the latest version is `0.8` for Scala `2.11`. After a version of Reactors is released for Scala `2.12`, you might have to replace the `0.8` version in the preceding code.

The “Hello World” program

In this section, we go through a simple, working Hello World program. We will not go into too much, yet we will provide deeper information in the subsequent sections. For now, we will just define a reactor that waits for one incoming event, prints a message to the standard output once this event arrives, and then terminate.

We start by importing the contents of the `io.reactors` package:

```
import io.reactors._
```

This allows us to use the facilities provided by the Reactors framework. In the following snippet, we declare a simple reactor-based program:

```
object ReactorHelloWorld {
  def main(args: Array[String]): Unit = {
    val welcomeReactor = Reactor[String] { self =>
      self.main.events onEvent { name =>
        println(s"Welcome, $name!")
        self.main.seal()
      }
    }
    val system = ReactorSystem.default("test-system")
    val ch = system.spawn(welcomeReactor)
    ch ! "Alan"
  }
}
```

The program above declares an anonymous reactor called `welcomeReactor`, which waits for a name to arrive on its main event stream, prints that name, and then seals its main channel, therefore terminating itself. The main program then creates a new reactor system, uses the reactor template to start a new running instance of the previously defined `welcomeReactor`, and sends an event `"Alan"` to it.

By analyzing the previous program, we conclude the following:

- A reactor is defined using the `Reactor[T]` constructor, where `T` is the type of the events that can be sent to the reactor on its main channel.
- A reactor reacts to incoming events as specified in the callback function passed to the `onEvent` method. We can call `onEvent`, for example, on the main event stream of the reactor, which is obtained with the expression `main.events`.
- Calling `main.seal()` terminates the reactor.
- A reactor with a specific definition is started with the `spawn` method, which returns the reactor's main channel.
- Events are sent to the reactor by calling the `!` operator on one of its channels.

The subsequent sections will explain each of these features in greater depth.

Event streams

In this section, we study the basic data-type that drives most computations in the Reactors framework: an event stream. Event streams represent special program values that can occasionally produce events. Event streams are represented by the `Event[T]` type.

Semantically, an event stream is very similar to the `Observable` type, which we saw in `Chapter 6`, *Concurrent Programming with Reactive Extensions*. As we will see, the main difference between `Observable` and `Events` is that an `Observable` object can generally be used from different threads, and even emit events across different threads when the `observeOn` method is used. An `Events` object, by contrast, can only be used inside the reactor that owns that event stream.

Never share an event stream between two reactors. An event stream can only be used by the reactor that owns the corresponding channel.

In the following, we show an example event stream called `myEvents`, which produces events of type `String`:

```
val myEvents: Events[String] = createEventStreamOfStrings()
```

For now, we assume that the method `createEventStreamOfStrings` is already defined, and that it returns an event stream of type `Events[String]`.

To be useful, an event stream must allow the users to somehow manipulate the events it produces. For this purpose, every event stream has a method called `onEvent`, which takes a user callback function and invokes it every time an event arrives:

```
myEvents.onEvent(x => println(x))
```

The `onEvent` method is similar to what most callback-based frameworks expose: a way to provide an executable snippet of code that is invoked later, once an event becomes available. However, just like the `Observable` object in Reactive Extensions, the receiver of the `onEvent` method, that is, the event stream, is a first-class value. This subtle difference allows passing the event stream as an argument to other methods, and consequently allows writing more general abstractions. For example, we can implement a reusable `trace` method as follows:

```
def trace[T](events: Events[T]): Unit = {
  events.onEvent(println)
}
```

The `onEvent` method returns a special `Subscription` object. Events are propagated to the user-specified callback until the user decides to call the `unsubscribe` method of that `Subscription` object. These `Subscription` objects have similar semantics as those seen in the **Reactive Extensions** framework.

Before we continue, we note that event streams are entirely a single-threaded entity. The same event stream will never concurrently produce two events at the same time, so the `onEvent` method will never be invoked by two different threads at the same time on the same event stream. As we will see, this property simplifies the programming model and makes event-based programs easier to reason about.

To understand this better, let's study a concrete event stream called an emitter, represented by the `Events.Emitter[T]` type. In the following, we instantiate an emitter:

```
val emitter = new Events.Emitter[Int]
```

An emitter is simultaneously an event stream and an event source. We can imperatively tell the emitter to produce an event by calling its `react` method. When we do that, the emitter invokes the callbacks previously registered with the `onEvent` method.

```
var luckyNumber = 0
emitter.onEvent(luckyNumber = _)
emitter.react(7)
assert(luckyNumber == 7)
emitter.react(8)
assert(luckyNumber == 8)
```

By running the above snippet, we convince ourselves that the `react` call really forces the emitter to produce an event. Furthermore, the call `emitter.react(8)` will always execute after `emitter.react(7)`, and the callback will be first invoked with `7`, and then with `8`, but not concurrently. Event propagation will occur on the same thread on which `react` was called.

Lifecycle of an event stream

We now take a closer look at the events that an event stream can produce. An event stream of type `Events[T]` usually emits events of type `T`. However, type `T` is not the only type of events that an event stream can produce. Some event streams are finite. After they emit all their events, they emit a special event that denotes that there will be no further events. Sometimes, event streams run into exceptional situations, and emit exceptions instead of normal events.

The `onEvent` method that we saw earlier can only react to normal events. To listen to other event kinds, event streams have the more general `onReaction` method. The `onReaction` method takes an `Observer` object as an argument. An `Observer` object has three different methods used to react to different event types. In the following code snippet, we instantiate an emitter and listen to all its events:

```
var seen = List[Int]()
var errors = List[String]()
var done = 0
val e = new Events.Emitter[Int]
e.onReaction(new Observer[Int] {
  def react(x: Int, hint: Any) = seen ::= x
  def except(t: Throwable) = errors ::= t.getMessage
  def unreact() = done += 1
})
```

The type `Observer[T]` has three methods:

- The `react` method, which is invoked when a normal event gets emitted. The second, optional hint argument may contain an additional value, but is usually set to `null`.
- The `except` method, which is invoked when the event stream produces an exception. An event stream can produce multiple exceptions. An exception, however, does not terminate the stream, and many exceptions can be emitted by the same event stream. This is one big difference with respect to the type `Observable` from Reactive Extensions.
- The `unreact` method, which is invoked when the event stream stops producing events. After this method is invoked on the observer, no further events or exceptions will be produced by the event stream.

Let's assert that this contract is correct for `Events.Emitter`. We already learned that we can produce events with emitters by calling the `react` method. We can similarly call `except` to produce exceptions, or the `unreact` method to signal that there will be no more events. For example:

```
e.react(1)
e.react(2)
e.except(new Exception("^_^"))
e.react(3)
assert(seen == 3 :: 2 :: 1 :: Nil)
assert(errors == "^_^" :: Nil)
assert(done == 0)
e.unreact()
assert(done == 1)
e.react(4)
e.except(new Exception("o_O"))
assert(seen == 3 :: 2 :: 1 :: Nil)
assert(errors == "^_^" :: Nil)
assert(done == 1)
```

If you run the preceding code snippet, you will see that, after calling the `unreact` method, subsequent calls to the `react` or `except` methods have no effect, and the `unreact` call effectively terminates the emitter. Not all event streams are as imperative as emitters, however. Most other event streams are created by functionally composing different event streams.

Functional composition of event streams

Using event stream methods such as `onEvent` and `onReaction` can easily result in a callback hell: a program composed of a large number of unstructured `onXYZ` calls, which is hard to understand and maintain. Having first-class event streams is a step in the right direction, but it is not sufficient.

Event streams support functional composition, seen in the earlier chapters. This pattern allows declaratively forming complex values by composing simpler ones. Consider the following example, in which we compute the sum of squares of incoming events:

```
var squareSum = 0
val e = new Events.Emitter[Int]
e.onEvent(x => squareSum += x * x)
for (i <- 0 until 5) e react i
```

The example is fairly straightforward, but what if we want to make `squareSum` an event stream so that another part of the program can react to its changes? We would have to create another emitter and have our `onEvent` callback invoke the `react` method on that new emitter, passing it the value of `squareSum`. This could work, but it is not elegant, as shown in the following snippet:

```
val ne = new Events.Emitter[Int]
e onEvent { x =>
  squareSum += x * x
  ne.react(squareSum)
}
```

We now rewrite the previous snippet using event stream combinators. Concretely, we use the `map` and `scanPast` combinators. The `map` combinator transforms events in one event stream into events for a derived event stream. We use the `map` combinatory to produce a square of each integer event. The `scanPast` combinator combines the last and the current event to produce a new event for the derived event stream. We use `scanPast` to add the previous value of the sum to the current one. For example, if an input event stream produces numbers `0`, `1`, and `2`, the event stream produced by `scanPast(0)(_ + _)` would produce numbers `0`, `1`, and `3`.

Here is how we can rewrite the previous example:

```
val e = new Events.Emitter[Int]
val sum = e.map(x => x * x).scanPast(0)(_ + _)
for (i <- 0 until 5) e react i
```

The type `Events[T]` comes with a large number of predefined combinators. You can find other combinators in the online API documentation. A set of event streams composed using functional combinators forms a dataflow graph. Emitters are usually source nodes in this graph, event streams created by various combinators are inner nodes, and callback methods, such as `onEvent`, are sink nodes. Combinators such as `union` take several input event streams. Such event streams correspond to graph nodes with multiple input edges. Here is one example:

```
val numbers = new Events.Emitter[Int]
val even = numbers.filter(_ % 2 == 0)
val odd = numbers.filter(_ % 2 == 1)
val numbersAgain = even union odd
```

Dataflow graphs induced by event streams are similar in nature to dataflow graphs induced by Scala futures and `Observable` objects from Reactive Extensions, so we will not study them further in this chapter. The most important thing to remember about event streams in the reactor model is that they are single-threaded entities. As we will see in the next section, each event stream can only belong to a single reactor.

Reactors

As we learned previously, event streams always propagate events on a single thread. This is useful from the standpoint of program comprehension, but we still need a way to express concurrency in our programs. In this section, we will see how to achieve concurrency by using entities called reactors.

A reactor is the basic unit of concurrency. While actors receive messages, we will adopt the terminology in which reactors receive events, in order to disambiguate. However, while an actor a in particular state has only a single point where it can receive a message, namely, the `receive` statement, a reactor can receive an event from many different sources at any time. Despite this flexibility, one reactor will always process, at most, one event at any time. We say that events received by a reactor are **serialized**, similar to how messages received by an actor are serialized.

To be able to create new reactors, we need a `ReactorSystem` object, which tracks reactors in a single machine:

```
val system = new ReactorSystem("test-system")
```

Before we can start a reactor instance, we need to define its template. One way to do this is to call `Reactor.apply[T]` method, which returns a `Proto` object for the reactor. The `Proto` object is a reactor prototype, which can be used to start the reactor. The following reactor prints all the events it receives to the standard output:

```
val proto: Proto[Reactor[String]] = Reactor[String] { self =>
  self.main.events onEvent {
    x => println(x)
  }
}
```

Let's examine this code more closely. The `Reactor.apply` method is called with the type argument `String`. This means that the reactor encoded in the resulting `Proto` object by default receives events whose type is `String`. This is the first difference with respect to the standard actor model, in which actors can receive messages of any type. Events received by reactors are well typed.

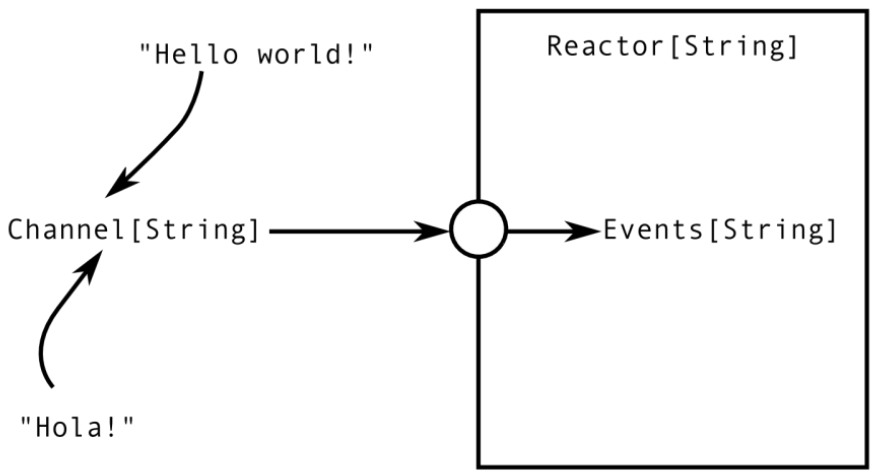
In the reactor model, every reactor can access a special event stream called `main.events`, which emits events that the reactor receives from other reactors. Since we are declaring an anonymous reactor with the `Reactor.apply` method, we need to add a prefix `self` to access members of the reactor. We previously learned that we can call `onEvent` to register callbacks to event streams, and we used it in this example to print the events using `println`.

After defining a reactor template, the next step is to spawn a new reactor. We do this by calling the `spawn` method on the reactor system:

```
val ch: Channel[String] = system.spawn(proto)
```

The `spawn` method takes a `Proto` object as a parameter. The `Proto` object can generally encode the reactor's constructor arguments, scheduler, name, and other options. In our example, we created a `Proto` object for an anonymous reactor with the `Reactor.apply` method, so we do not have access to any constructor arguments. We will later see alternative ways of declaring reactors and configuring prototypes.

The `spawn` method does two things. First, it registers and starts a new reactor instance. Second, it returns a `Channel` object, which is used to send events to the newly created reactor. We show the relationship between a reactor, its event stream, and the channel in the following figure:



The only way for the outside world to access the inside of a reactor is to send events to its channel. These events are eventually delivered to the corresponding event stream, which the reactor can listen to. The channel and event stream can only pass events whose type corresponds to the type of the reactor.

Let's send an event to our reactor. We do this by calling the bang operator `!` on the channel:

```
ch ! "Hola!"
```

Running the last statement should print the string `"Hola!"` to the standard output.

Defining and configuring reactors

In earlier sections, we saw how to define a reactor using the `Reactor.apply` method. In this section, we take a look at an alternative way of defining a reactor–by extending the `Reactor` base class. Recall that the `Reactor.apply` method defines an anonymous reactor template. Extending the `Reactor` class declares a named reactor template.

In the following, we declare the `HelloReactor` class, which must be top-level:

```
class HelloReactor extends Reactor[String] {
  main.events onEvent {
    x => println(x)
  }
}
```

To run this reactor, we first create a prototype to configure it. The method `Proto.apply` takes the type of the reactor and returns a prototype for that reactor type. We then call the `spawn` method with that `Proto` object to start the reactor:

```
val ch = system.spawn(Proto[HelloReactor])
ch ! "Howdee!"
```

We can also use the prototype to, for example, set the scheduler that the reactor instance should use. If we want the reactor instance to run on its own dedicated thread to give it more priority, we can do the following:

```
system.spawn(
  Proto[HelloReactor].withScheduler(JvmScheduler.Key.newThread))
```

Note that if you are running Reactors on `Scala.js`, you will need to use a `Scala.js` specific scheduler. The reason for this is because the JavaScript runtime, which `Scala.js` compiles to, is not multi-threaded. Asynchronous executions are placed on a single queue, and executed one after another. On `Scala.js`, you will need to use the `JsScheduler.Key.default` scheduler.

There are several other configuration options for `Proto` objects, and you can find out more about them in the online API documentation. We can summarize this section as follows. Starting a reactor is generally a three-step process:

1. A named reactor template is created by extending the `Reactor` class.
2. A reactor configuration object is created with the `Proto.apply` method.
3. A reactor instance is started with the `spawn` method of the reactor system.

For convenience, we can fuse the first two steps by using the `Reactor.apply` method, which creates an anonymous reactor template and directly returns a prototype object of type `Proto[I]`, for some reactor type `I`. Typically, this is what we do in the tests, or when trying things out in the Scala REPL.

Using channels

Now that we understand how to create and configure reactors in different ways, we can take a closer look at channels, which are the reactor's means of communicating with its environment. As noted before, every reactor is created with a default channel called `main`, which is often sufficient. But sometimes a reactor needs to be able to receive more than just one type of an event, and needs additional channels for this purpose.

Let's declare a reactor that stores key-value pairs. The reactor must react to requests for storing key-value pairs, and for retrieving a value under a specific key. Since the reactor's input channel will have to serve two purposes, we need the following data type:

```
trait Op[K, V]
case class Put[K, V](k: K, v: V) extends Op[K, V]
case class Get[K, V](k: K, ch: Channel[V]) extends Op[K, V]
```

The `Op` datatype has two type parameters called `K` and `V`, which denote the types of keys and values being stored. The `Put` case class is used to store a value into the reactor, so it contains the new key and value. The `Get` case class is used to retrieve the value that was previously stored with some key, so it encodes the key and the channel of type `V`. When the reactor receives the `Get` event, it must look up the value associated with the key, and send the value along the channel.

With the `Op[K, V]` data type, we can define `MapReactor`, shown in the following snippet:

```
class MapReactor[K, V] extends Reactor[Op[K, V]] {
  val map = mutable.Map[K, V]()
  main.events onEvent {
    case Put(k, v) => map(k) = v
    case Get(k, ch) => ch ! map(k)
  }
}
```

Let's start `MapReactor` and test it. We will use the `MapReactor` to store some DNS aliases. We will map each alias `String` key to a URL, where the URLs are represented with the `List[String]` type. We first initialize as follows:

```
val mapper = system.spawn(Proto[MapReactor[String, List[String]]])
```

We then send a couple of `Put` messages to store some alias values:

```
mapper ! Put("dns-main", "dns1" :: "lan" :: Nil)
mapper ! Put("dns-backup", "dns2" :: "com" :: Nil)
```

Next, we create a client reactor that we control by sending it `String` events. This means that the reactor's type will be `Reactor[String]`. However, the client reactor will also have to contact the `MapReactor` and ask it for one of the URLs. Since the `MapReactor` can only send it back `List[String]` events that do not correspond to the client's default channel type, the client's default channel is not be able to receive the reply. Therefore, the client will have to provide the `MapReactor` with a different channel. The following expression is used to create a new channel:

```
val c: Connector[EventType] = system.channels.open[EventType]
```

The expression `system.channels` returns a channel builder object, which provides methods such as `named` or `daemon`, used to customize the channel (see the online API docs for more details). In this example, we will create **daemon channel**, to indicate that the channel does not need to be closed (more on that a bit later). To create a new channel, we call the `open` method on the channel builder with the appropriate type parameter.

The resulting `Connector` object contains two members: the `channel` field, which is the newly created channel, and the `events` field, which is the event stream corresponding to that channel. The event stream propagates all events that were sent and delivered on the channel, and can only be used by the reactor that created it. The channel, on the other hand, can be shared with other reactors.

Use the `open` operation on the `system.channels` object to create new connectors. Each connector holds a pair of a channel and its event stream.

Let's define a client reactor that waits for a `"start"` message, and then checks a DNS entry. This reactor will use the `onMatch` handler instead of `onEvent`, to listen only to certain `String` events and ignore others:

```
val ch = system.spawn(Reactor[String] { self =>
  self.main.events onMatch {
    case "start" =>
      val reply = self.system.channels.daemon.open[List[String]]
      mapper ! Get("dns-main", reply.channel)
      reply.events onEvent { url =>
        println(url)
      }
    case "end" =>
      self.main.seal()
  }
})
```

In the preceding code snippet, when the reactor receives the `"start"` event from the main program, it opens a new `reply` channel that accepts `List[String]` events. It then sends a `Get` event to the `MapReactor` with the `"dns-main"` key and the `reply` channel. Finally, the reactor listens to events sent back along the `reply` channel, and prints the URL to the standard output. In the `"end"` case of the main pattern match, the reactor calls the `seal` method on the main channel to indicate that it will not receive any further events on that channel. Once all non-daemon channels become sealed, the reactor terminates.

A reactor terminates either when all its non-daemon channels are sealed, or when its constructor or some event handler throws an exception.

Let's start the client reactor and see what happens:

```
ch ! "start"
```

At this point, we should witness the URL on the standard output. Finally, we can send the `"end"` message to the client reactor to stop it.

```
ch ! "end"
```

In the next section, we will see how to customize reactors with custom scheduling policies.

Schedulers

Each reactor template can be used to start multiple reactor instances, and each reactor instance can be started with a different reactor scheduler. Different schedulers have different characteristics in terms of execution priority, frequency, latency, and throughput. In this section, we take a look at how to use a non-default scheduler, and how to define custom schedulers when necessary.

We start by defining a reactor that logs incoming events, reports every time it gets scheduled, and ends after being scheduled three times. We will use the `sysEvents` stream of the reactor, which will be explained in the next section. For now, all you need to know is that the system event stream produces events when the reactor gets some execution time (that is, gets scheduled), and pauses its execution (that is, gets pre-empted).

The `Logger` reactor is shown in the following snippet:

```
class Logger extends Reactor[String] {
  var count = 3
  sysEvents onMatch {
    case ReactorScheduled =>
      println("scheduled")
    case ReactorPreempted =>
      count -= 1
      if (count == 0) {
        main.seal()
        println("terminating")
      }
  }
  main.events.onEvent(println)
```

Before starting an instance of the `Logger` reactor, we need to create a reactor system, as we learned in the earlier sections:

```
val system = new ReactorSystem("test-system")
```

Every reactor system is bundled with a default scheduler and some additional predefined schedulers. When a reactor is started, it uses the default scheduler, unless specified otherwise. In the following, we override the default scheduler with the one using Scala's global execution context, that is, Scala's own default thread pool:

```
val proto = Proto[Logger].withScheduler(
  JvmScheduler.Key.globalExecutionContext)
val ch = system.spawn(proto)
```

Running the snippet above should start the `Logger` reactor and print the `"scheduled"` string once, because starting a reactor schedules it even before any events arrive. If we now send an event to the main channel, we will see the `"scheduled"` string printed again, followed by the event itself. We do this as follows:

```
ch ! "event 1"
```

Sending the event again decrements the reactor's counter. The main channel gets sealed, leaving the reactor in a state without non-daemon channels, and the reactor terminates:

```
ch ! "event 2"
```

Reactor systems also allow registering custom scheduler instances. In the following, we create and register a custom `Timer` scheduler, which schedules the `Logger` reactor for execution once every 1,000 milliseconds:

```
system.bundle.registerScheduler("customTimer",
  new JvmScheduler.Timer(1000))
val periodic = system.spawn(
  Proto[Logger].withScheduler("customTimer"))
```

By running the code above, we can see that the reactor gets scheduled even if no events were sent to it. The `Timer` scheduler ensures that the reactor gets scheduled exactly once every N seconds, and then processes some of its pending events.

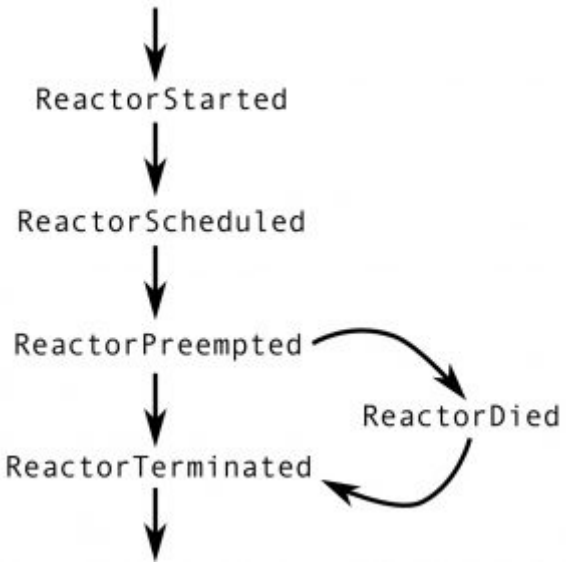
Reactor lifecycle

Every reactor goes through a certain set of stages during its lifetime, which are jointly called a **reactor lifecycle**. When the reactor enters a specific stage, it emits a lifecycle event. These lifecycle events are dispatched on a special daemon event stream called `sysEvents`. Every reactor is created with this special event stream.

The reactor lifecycle can be summarized as follows:

- After calling the `spawn` method, the reactor is scheduled for execution. Its constructor is started asynchronously, and immediately after that, a `ReactorStarted` event is dispatched.
- Then, whenever the reactor gets execution time, the `ReactorScheduled` event gets dispatched. After that, events get dispatched on normal event streams.
- When the scheduling system decides to pre-empt the reactor, the `ReactorPreempted` event is dispatched. This scheduling cycle can be repeated any number of times.
- Eventually, the reactor terminates, either by normal execution or exceptionally. If a user code exception terminates execution, a `ReactorDied` event is dispatched.
- In either normal or exceptional execution, a `ReactorTerminated` event gets emitted.

This reactor lifecycle is shown in the following diagram:



To test this, we define the following reactor:

```
class LifecycleReactor extends Reactor[String] {
  var first = true
  sysEvents onMatch {
    case ReactorStarted =>
      println("started")
    case ReactorScheduled =>
      println("scheduled")
    case ReactorPreempted =>
      println("preempted")
      if (first) first = false
      else throw new Exception
    case ReactorDied(_) =>
      println("died")
    case ReactorTerminated =>
      println("terminated")
  }
}
```

Upon creating the lifecycle reactor, the reactor gets the `ReactorStarted` event, and then the `ReactorStarted` and `ReactorScheduled` events. The reactor then gets suspended, and remains that way until the scheduler gives it more execution time.

```
val ch = system.spawn(Proto[LifecycleReactor])
```

The scheduler executes the reactor again when it detects that there are pending messages for that reactor. If we send an event to the reactor now, we will see the same cycle of `ReactorScheduled` and `ReactorPreempted` events from the standard output. However, the `ReactorPreempted` handler this time throws an exception. The exception gets caught, and a `ReactorDied` event is emitted, followed by the mandatory `ReactorTerminated` event.

```
ch ! "event"
```

At this point, the reactor is fully removed from the reactor system.