Concurrent programming for the multi-core era

Actors *in* Scala

artima

Philipp Haller
Frank Sommers

Actors in Scala

PrePrint™ Edition

Thank you for purchasing the PrePrint™ Edition of *Actors in Scala*.

A PrePrint™ is a work-in-progress, a book that has not yet been fully written, reviewed, edited, or formatted. We are publishing this book as a PrePrint™ for two main reasons. First, even though this book is not quite finished, the information contained in its pages can already provide value to many readers. Second, we hope to get reports of errata and suggestions for improvement from those readers while we still have time to incorporate them into the first printing.

As a PrePrint™ customer, you'll be able to download new PrePrint™ versions from Artima as the book evolves, as well as the final PDF of the book once finished. You'll have access to the book's content prior to its print publication, and can participate in its creation by submitting feedback. Please submit by clicking on the *Suggest* link at the bottom of each page.

Thanks for your participation. We hope you find the book useful and enjoyable.

Bill Venners
President, Artima, Inc.

Actors in Scala

PrePrint™ Edition

Philipp Haller, Frank Sommers

artima

ARTIMA PRESS

WALNUT CREEK, CALIFORNIA

Actors in Scala
PrePrint™ Edition, Version 2

Philipp Haller is a research assistant in the School of Computer and Communication Sciences at EPFL in Lausanne, Switzerland. Frank Sommers is president of Autospaces, Inc.

Artima Press is an imprint of Artima, Inc.
P.O. Box 305, Walnut Creek, California 94597

PrePrint™ Edition first published 2010
Build date of this impression March 3, 2011
Produced in the United States of America

to H.M. - P.H.

to Jungwon - F.S.

Overview

Contents

List of Figures

List of Listings

Actors in Scala

PrePrint™ Edition

Chapter 1

Concurrency Everywhere

The actor model of concurrency was born of a practical need: When Carl Hewitt and his team at MIT first described actors in the 1970s, computers were relatively slow.[1] While it was already possible to divide up work among several computers and to compute in parallel, Hewitt's team wanted a model that would not only simplify building such concurrent systems, but would also let them reason about concurrent programs in general. Such reasoning, it was believed, would allow developers to be more certain that their concurrent programs worked as intended.

Although actor-based concurrency has been an important concept ever since, it is only now gaining wide-spread acceptance. That is in part because until recently no widely used programming language offered first-class support for actors. An effective actors implementation places a great burden on a host language, and few mainstream languages were up to the task. Scala rises to that challenge, and offers a full-featured implementation of actor-based concurrency on the Java virtual machine (JVM).[2] Because Scala code seamlessly interoperates with code and libraries written in Java, or any other JVM language, Scala actors offer an exciting and practical way to build scalable and reliable concurrent programs.

Like many powerful concepts, the actor model can be understood and used on several levels. On one level, actor-based programming provides an easy way to exchange messages between independently running threads or

[1] Hewitt *et al.*, "A Universal Modular ACTOR Formalism for Artificial Intelligence" [Hew73]

[2] Haller and Odersky, "Scala Actors: Unifying thread-based and event-based programming" [Hal09]

processes. On another level, actors make concurrent programming generally simpler, because actors let developers focus on high-level concurrency abstractions and shield programmers from intricacies that can easily lead to errors. On an even broader level, actors are about building reliable programs in a world where concurrency is the norm, not the exception—a world that is fast approaching.

This book aims to explain actor-based programming with Scala on all those levels. Before diving into the details of Scala actors, it helps to take a step back and place actors in the context of other approaches to concurrent programming, some of which may already be familiar to you.

1.1 Scaling with concurrency

The mainstream computing architectures of the past decades focused on making the execution of a single thread of sequential instructions faster. That led to an application of Moore's Law to computing performance: Processor performance per unit cost has doubled roughly every eighteen months for the last twenty years, and developers counted on that trend to ensure that their increasingly complex programs performed well.[3]

Moore's Law has been remarkably accurate in predicting processor performance, and it is reasonable to expect processor computing capacity to double every one-and-a-half years for at least another decade. To make that increase practical, however, chip designers had to implement a major shift in their design focus in recent years. Instead of trying to improve the clock cycles dedicated to executing a single thread of instructions, new processor designs make it possible to execute many concurrent instruction threads on a single chip. While the clock speed of each computing core on a chip is expected to improve only marginally over the next few years, processors with dozens of cores are already showing up in commodity servers, and multicore chips are the norm even in inexpensive desktops and notebooks.

This shift in the design of high-volume, commodity processor architectures, such as the Intel x86, has at least two ramifications for developers. First, because individual core clock cycles will increase only modestly, we will need to pay renewed attention to the algorithmic efficiency of sequential code. Second, and more important in the context of actors, we will need to design programs such that they take maximum advantage of available pro-

[3] Sutter, "The free lunch is over: A fundamental turn toward concurrency" [Sut05]

cessor cores. In other words, we not only need to write programs that work correctly on concurrent hardware, but also design programs that opportunistically scale to all available processing units or cores.

1.2 Actors versus threads

In a concurrent program, many independently executing threads, or sequential processes, work together to fulfill an application's requirements. Investigation into concurrent programming has mostly focused on defining how concurrently executing sequential processes can communicate such that a larger process—for example, a program that executed those processes—can proceed predictably.

The two most common ways of communication among concurrent threads are synchronization on shared state, and message passing. Many familiar programming constructs, such as semaphores and monitors, are based on shared-state synchronization. Developers of concurrent programs are familiar with those structures. For example, Java programmers can find these structures in the `java.util.concurrent` library.[4] Among the biggest challenges for anyone using shared-state concurrency are avoiding concurrency hazards, such as data races and deadlocks, and scalability.[5]

Message passing is an alternative way of communication among cooperating threads. There are two important categories of systems based on message passing. In channel-based systems, messages are sent to *channels* (or *ports*) that processes can share. Several processes can then receive messages from the same shared channels. Examples of channel-based systems are MPI[6] and systems based on the CSP paradigm[7], such as the Go language. Systems based on actors (or agents, or Erlang-style processes[8]) are in the second category of message-passing concurrency. In these systems, mes-

[4] Goetz *et al.*, *Java Concurrency in Practice* [Goe06]

[5] One of the reasons why scalability is hard to achieve using locks (or Java-style synchronization) is the fact that coarse-grained locking increases the amount of code that is executed sequentially. Moreover, accessing a small number of locks (or, in the extreme case, a single global lock) from several threads may increase the cost of synchronization significantly.

[6] Gropp *et al.*, *Using MPI: Portable Parallel Programming with the Message–Passing Interface* [Gro99]

[7] Hoare, "Communicating sequential processes" [Hoa78]

[8] Armstrong *et al.*, *Concurrent Programming in Erlang* [Arm95]

sages are sent directly to actors; it is not necessary to create intermediary channels between processes.

An important advantage of message passing over shared-state concurrency is that it makes it easier to avoid data races. If processes communicate only by passing messages, and those messages are immutable, then race conditions are avoided by design. Moreover, anecdotal evidence suggests that this approach in practice also reduces the risk of deadlock. A potential disadvantage of message passing is that the communication overhead may be high. To communicate, processes have to create and send messages, and these messages are often buffered in queues before they can be received to support asynchronous communication. In contrast, shared-state concurrency enables direct access to shared memory, as long as it is properly synchronized. To reduce the communication overhead of message passing, large messages should not be transferred by copying the message state; instead, only a reference to the message should be sent. However, this reintroduces the risk for data races when several processes have access to the same mutable (message) data. It is an ongoing research effort to provide static checkers (for instance, the Scala compiler plug-in for uniqueness types[9] that can verify that programs passing mutable messages by reference do not contain data races.

Let's take a step back, and look at actor-based programming from a higher-level perspective. To appreciate the difference, and the relationship, between more traditional concurrency constructs and actors, it helps to pay a brief visit to the local railroad yard.

Imagine yourself standing on a bridge overlooking the multitude of individual tracks entering the rail yard. You can observe many seemingly independent activities taking place, such as trains arriving and leaving, cars being loaded and unloaded and so on.

Suppose, then, that your job was to design such a railroad yard. Thinking in terms of threads, locks, monitors, and so on, is similar to the problem of figuring out how to make sure that trains running on parallel tracks don't collide. It is a very important requirement; without that, the rail yard would be a dangerous place, indeed. To accomplish that task, you would employ specialized artifacts, such as semaphores, monitors, switches.

Actors illuminate the same rail yard from the higher perspective of ensuring that all the concurrent activities taking place at the rail yard progress

[9] Haller and Odersky, "Capabilities for Uniqueness and Borrowing" [Hal10]

smoothly: that all the delivery vehicles find their ways to train cars, all the trains can make their progress through the tracks, and all the activities are properly coordinated.

You will need both perspectives when designing a rail yard: Thinking from the relatively low-level perspective of individual tracks ensures that trains don't inadvertently cross paths; thinking from the perspective of the entire facility helps ensure that your design faciliates smooth overall operation, and that your rail yard can scale, if needed, to accommodate increased traffic. Simply adding new rail tracks only goes so far: you need some overall design principles to ensure that the whole rail yard can grow to handle increased traffic, and that greater traffic can scale to the full capacity of the tracks.

Working on the relatively low-level details of individual tracks (or problems associated with interleaving threads), on the one hand, and the higher-level perspective of the entire facility (actors) on the other, require somewhat different skills and experience. An actor-based system is often implemented in terms of threads, locks, monitors, and the like, but actors hide those low-level details, and allow you to think of concurrent programs from a higher vantage point.

1.3 The indeterministic soda machine

In addition to allowing you to focus on the scalability aspect of concurrent applications, actors' higher-level perspective on concurrency is helpful because it provides a more realistic abstraction for understanding how concurrent applications work. Specifically, concurrent programs exhibit two characteristics that, while also present in sequential applications, are especially pronounced when a program is designed from the ground up to take advantage of concurrency. To see what these are, we need only to stop by the office soda machine.[10]

A soda machine is convenient not only to quench our thirst on a hot summer day, but also because it's a good metaphor for a kind of program that moves from one well-defined state to another. To start out, a soda machine awaits input from the user, perhaps prompting the user to insert some coins. Inserting those coins causes the soda machine to enter a state where it can now ask the user to make a selection of the desired drink. As soon as the user

[10] Hoare, "Communicating sequential processes" [Hoa78]

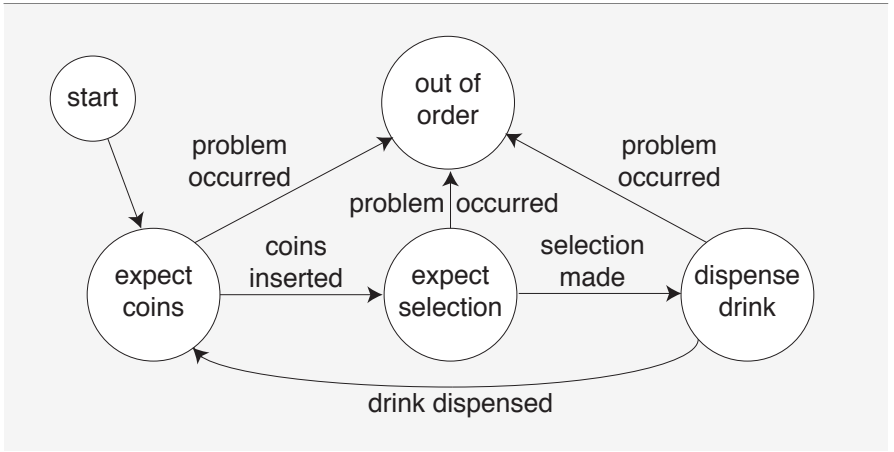

Figure 1.1 · State transitions in a soda machine.

makes that selection, the soda machine dispenses a can and moves back into its initial state. On occasion, it may also run out of soda cans—that would place it in an "out of service" state.

At any point in time, a soda machine is aware of only one state. That state is also global to the machine: Each component—the coin input device, the display unit, the selection entry keypad, the can dispenser, and so on—must consult that global state to determine what action to take next: For instance, if the machine is in the state where the user has already made his selection, the can dispenser component is allowed to release a soda can into the output tray.

In addition to always being in a well-defined state, our simple abstraction suggests two further characteristics of a soda machine: First, that the number of possible states the machine can enter is finite and, second, that given any one of those possible states, we can determine in advance what the next state will be. For instance, if you inserted a sufficient amount of coins, you would expect to be prompted for the choice of drink. And having made that choice, you expect the selected drink to be dispensed.

Of course, you've probably had occasions to experience soda machines that did not exactly behave in such a predictable, deterministic, way. You may have inserted plenty of coins, but instead of being prompted for your choice, you were presented with an unwelcoming "OUT OF ORDER" message. Or you may not have received any message at all—but also did not

receive your frosty refreshment, no matter how hard you pounded the machine. Real-world experience teaches us that soda machines, like most physical objects, are not entirely deterministic. Most of the time they move from one well-defined state to another in an expected, predetermined fashion; but on occasion they move from one state to another—to an error state, for instance—in a way that could not be predicted in advance.

A more realistic model of a soda machine, therefore, should include the property of some indeterminism: A model that readily admits a soda machine's ability to shift from one state to another in a way that could not be determined in advance with certainty.

Although we are generally adept at dealing with such indeterminism in physical objects—as well as when dealing with people—when we encounter such indeterminism in software, we tend to consider that behavior a bug. Examining such "bugs" may reveal that they crept into our code because some aspect of our program was not sufficiently specified.

Naturally, as developers we desire to create programs that are well-specified and, therefore, behave as expected—programs that act exactly in accord with detailed and exhaustive specifications. Indeed, one way to provide more or less exact specifications for code is by writing test cases for it.

Concurrent programs, however, are a bit more like soda machines than deterministic sequential code. That's because concurrent programs gain many of their benefits due to some aspects of a concurrent system intentionally being left unspecified.

The reason for that is easy to understand intuitively when considering a processor with four cores: Suppose that code running on the first core sends messages to code running on the three other cores, and then awaits replies back from each. Upon receiving a reply, the first core performs further processing on the response message.

In practice, the order in which cores 2, 3, and 4 send back their replies is determined by the order in which the three cores finish their computations. If that reply order is left unspecified, then core 1 can start processing a reply as soon as it receives one: it does not have to wait for the slowest core to finish its work.

In this example, leaving the reply order from cores 2, 3, and 4 unspecified helps to best utilize the available computing resources. At the same time, your program can no longer rely on any specific message ordering. Instead, your application must function deterministically even though its component

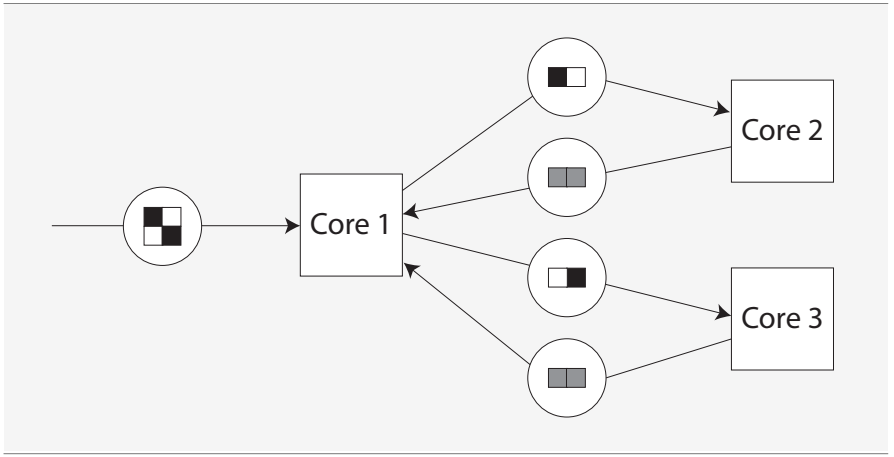

Figure 1.2 · Message passing between cores with indeterministic message ordering.

computations, or how those components interact, may not be fully specified.

One application of building deterministic systems out of indeterministic component computations are data centers constructed of commodity, off-the-shelf (COTS) components. Many well-known Web services companies have proven the economic advantages of using COTS hardware as basic building blocks for highly reliable data centers. Such an environment becomes practical when infrastructure software alleviates the need for developers to concern themselves with the intricacies of how such a data center partitions work between the various hardware components. Instead, application developers can focus on higher-level concerns, such as specifying the algorithms to use when servicing an incoming request.

A popular example of an infrastructure that makes programming on COTS clusters easier is MapReduce.[11] With MapReduce, a user provides some data, as well as some algorithms to operate on that data, and submits that as a request to the MapReduce infrastructure software. The MapReduce software, in turn, distributes the workload required to compute the specified request across available cluster nodes and returns a result to the user.

An important aspect of MapReduce is that, upon submitting a job, a user can reasonably expect some result back. For instance, should a node ex-

[11] Dean and Ghemawat, "MapReduce: Simplified data processing on large clusters" [Dea08]

ecuting parts of a MapReduce job fail to return results within a specified time period, the MapReduce software restarts that component job on another node. Because of its guarantee of returning a result, MapReduce not only allows an infrastructure to scale a compute-intensive job to a cluster of nodes, but more significantly, MapReduce lends reliability guarantees to the computation. It is that reliability aspect that makes MapReduce suitable for COTS-based compute clusters.

While a developer using MapReduce can expect to receive a result back, exactly when the result will arrive cannot be known prior to submitting the job: The user knows only that a result will be received, but he cannot, in advance, know when that will be. More generally, the system provides a guarantee that at some point a computation is brought to completion, but a developer using the system cannot in advance put a time bound on the length of time a computation would run.

Intuitively, it is easy to understand the reason for that: As the infrastructure software partitions the computation, it must communicate with other system components—it must send messages and await replies from individual cluster nodes, for instance. Such communication can incur various latencies, and those communication latencies impact the time it takes to return a result. You can't tell, in advance of submitting a job, how large those latencies will be.

Although some MapReduce implementations aim to ensure that a job returns some results—albeit perhaps incomplete results—in a specified amount of time, the actors model of concurrent computation is more general: It acknowledges that we may not know in advance just how long a concurrent computation would take. Put another way, you cannot place a time bound in advance on the length a concurrent computation would run. That's in contrast to traditional, sequential algorithms that model computations with well-defined execution times on a given input.

By acknowledging the property of unbounded computational times, actors aim to provide a more realistic model of concurrent computing. While varying communication latencies is easy to grasp in the case of distributed systems or clusters, it is also not possible in a four-core processor to tell in advance how long before cores 2, 3, and 4 will send their replies back to core 1. All we can say is that the replies will eventually arrive.

At the same time, unboundedness does not imply infinite times: While infinity is an intriguing concept, it lends but limited usefulness to realistically modeling computations. The actor model, indeed, requires that a concurrent

computation terminate in finite time, but it also acknowledges that it may not be possible to tell, in advance, just how long that time will be.

In the actor model, unboundedness and indeterminism—or, *unbounded indeterminism*—are key attributes of concurrent computing. While also present in primarily sequential systems, these are pervasive attributes of concurrent programs. Acknowledging these attributes of concurrency and providing a model that allows a developer to reason about a concurrent program in the face of those attributes are the prime goals of actors. The actor model accomplishes that by providing a surprisingly simple abstraction that can express program control structures you are familiar with from sequential programs—such as `if`, `while`, `for`, and so on—and make those control structures work predictably in a system that can opportunistically scale in a concurrent environment.

Chapter 2

Messages All the Way Up

Actor-based programming aims to model the complex world of pervasive concurrency with a handful of simple abstractions. Before diving into Scala's actors library, it is helpful to review briefly the most common Actor programming constructs. Scala's actors library implements many of these features. At the same time, like many Scala APIs, the actors API is constantly evolving, and future versions will likely provide even more capabilities. In the following birds-eye view of the actor programming model we refer to features that Scala actors already implement and, when relevant, point out differences between Scala actors and the more general model.

2.1 Control flow and data flow

The designers of the actor programming model started out by defining suitable abstractions for program control flow in concurrent systems. Informally, control flow in a program refers to the choice a program makes about what instructions to execute next. Branching, switch and case statements, as well as making decisions about what to return from a method invocation are all examples of control flow. All but the most trivial programs include some form of control flow.

Developers of sequential programs would not consider control flow a problematic task: After all, we routinely write `if`, `while`, and `for` expressions without thinking too much about the implications of those basic programming constructs. Concurrency, however, can make control flow more difficult to reason about. That's because control flow often depends on some logic, data, or state embedded in the program.

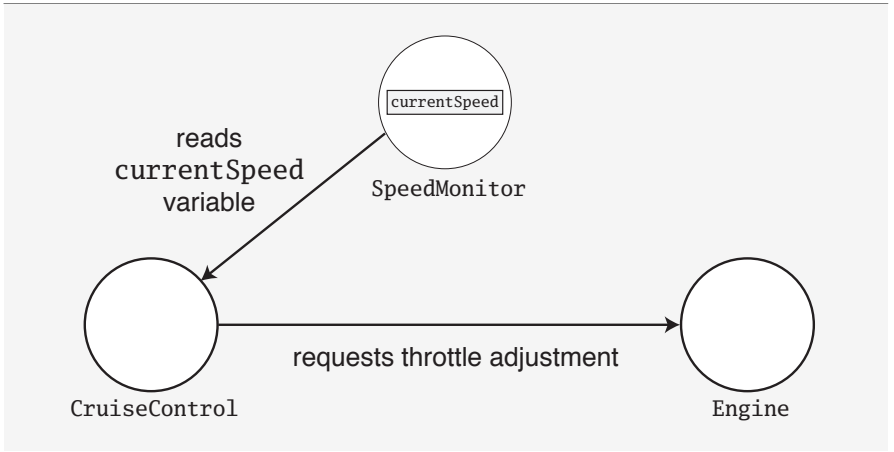

Figure 2.1 · Components holding shared state require synchronization.

In small programs, data and the control structures using that data may be defined close to each other, even in the same object. As a program grows in size, however, control flow decisions will need to consult bits of data—or program state—defined in other parts of the program. For example, the following expression requires access to the `currentSpeed` and `desiredSpeed` variables:

```
if (currentSpeed != dddesiredSpeed)
  channngeSpeed(dddesiSpeed - currentSpeed)
else
  mai tainSpee ()
```

The `currentSpeed` and `desiredSpeed` values are defined outside the `if-else` control structure, perhaps even outside the method or object containing the control flow expression. Similarly, the code implementing the `changeSpeed` and `maintainSpeed` methods may access, as well as alter, program state defined elsewhere in the program. Therefore, such methods require access to program state that other objects expose and share. In the above example, for instance, a `SpeedMonitor` object may have a public `currentSpeed` accessor method that any other object in the program can invoke. `currentSpeed` then becomes part of a globally visible program state.

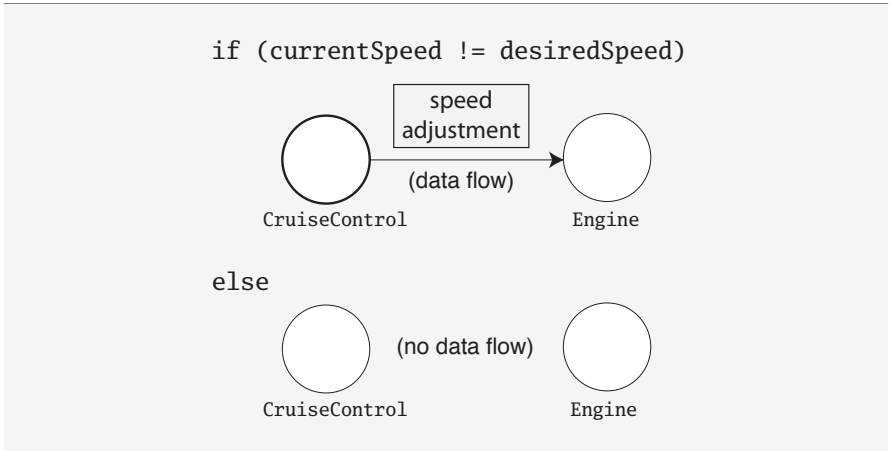

Figure 2.2 · Data flow and control flow interact in subtle ways: If speed adjustment is needed, data flows from one component to another. There is no data flow otherwise.

Globally visible program state defined across various objects is not a problem as long as only a single thread accesses that state. If many concurrently executing threads need to access globally visible state, however, a developer must carefully synchronize access to objects holding that state.

Synchronized access to shared data is not only difficult to get right, but can also reduce the amount of concurrency available in a system. To see why, consider that there are conceptually two different kinds of changes taking place as a concurrent program executes: First, various threads of execution, starting from the beginning of the program, wind their ways through possible paths based on program *control flow*. Those threads, in turn, can alter the values of variables holding the program's state. You can think of those state changes as defining the program's *data flow*. A developer must carefully identify every point in the program's data flow that can be altered by, and in turn affect, other execution threads, and guard against undesired side-effects.

A proven way to guard against unwanted conflicts between data flow and control flow is to serialize the program's data flow across concurrent threads of execution. Using special serialization constructs, such as locks, monitors, and semaphores, a developer specifies that threads must affect data flow in a strict order.

Defining such serialization in Java or Scala has become much easier with

the introduction of the `java.util.concurrent` package.

While serializing access to globally visible program state helps define correct program behavior, it may reduce some of the benefits of concurrent execution: As we mentioned in the previous chapter, the benefits of concurrency come about as a result of having few requirements about the order in which threads wind their way through a program and access program state. In effect, synchronization turns parts of a program into sequential code because only one thread at a time can access the global, or shared, state. Indeed, if control structures through a program rely on globally visible state, there is no way around serialized access to that state without risking incorrect behavior.

A key contribution of the actor model is to define control structures in a way that minimizes reliance on global program state. Instead, all the state—or knowledge—needed to make control flow decisions are co-located with the objects that make those decisions. Such objects, in turn, direct control flow only—or mostly—based on data locally visible to them. That *principle of locality* renders data flow and control flow in a program inseparable, reducing the requirement for synchronization. That, in turn, maximizes the potential for concurrency.

Although actor-based systems consider global state to be evil, in practice some control structures still need access to globally visible state. Recent additions to Scala's actors library make it easier to reason about such shared state in the context of actors, and we will highlight those features later in this book.

2.2 Actors and messages

The main mechanism for unifying control flow and data flow is a special abstraction, the actor, and the message-based communication that takes place between actors. An actor is any object with the capability to exchange messages with other actors. In the actor programming model, actors communicate solely by passing messages to each other.

In a pure actor system, every object is an actor. For instance, in Erlang, another language that defines an actor programming model, even atomic objects, such as `Int`s and `String`s, are actors. Scala's actors library, by contrast, allows you to easily turn any Scala object into an actor, but does not require that all objects be actors.

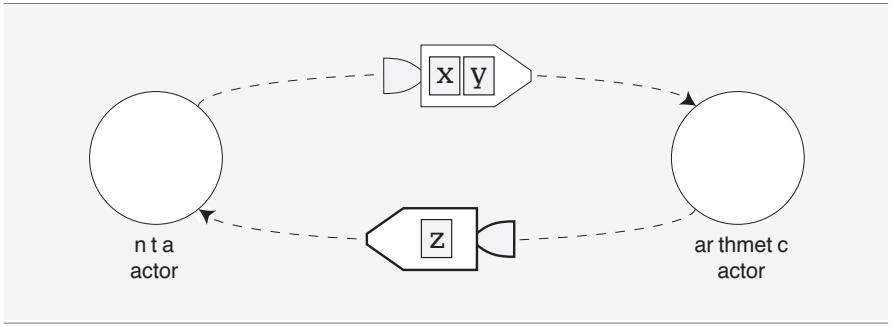

Figure 2.3 · The simplest actor computation: Adding `x` and `y` together.

Actors have a uniform public interface: An actor can, in general, accept any kind of message. When an actor receives a message from another actor, the receiving actor examines, or *evaluates*, the incoming message. Based on the contents and type of that message, the receiving actor may find the message interesting; otherwise, it simply discards the message. When an actor is interested in an incoming message, it may perform some action in response to that message. The action depends on the actor's internal script or program, as well as the actor's current state. The ability to perform actions in response to incoming messages is what makes an object an actor.

An actor's response to an incoming message can take different forms. The simplest response is to merely evaluate the message's content. Performing addition of integers `x` and `y` in an actor-based system, for instance, would consist of a message containing `x` and `y` sent to an actor capable of adding the integers together. In that case, the arithmetic actor would simply *evaluate* the sum of `x` and `y`.

Of course, merely adding two numbers together is of little use if the result is not visible outside the actor performing the evaluation. Thus, a more useful actor message would contain the address of another actor interested in receiving the result.

Reference to another actor in a message is the receiving actor's *continuation*: Upon evaluating a message according to an internal script, an actor can send the results of that evaluation to its continuation. Including references to a continuation in an actor's message means that the actor programming model implicitly supports continuation-passing style (CPS), but generalized

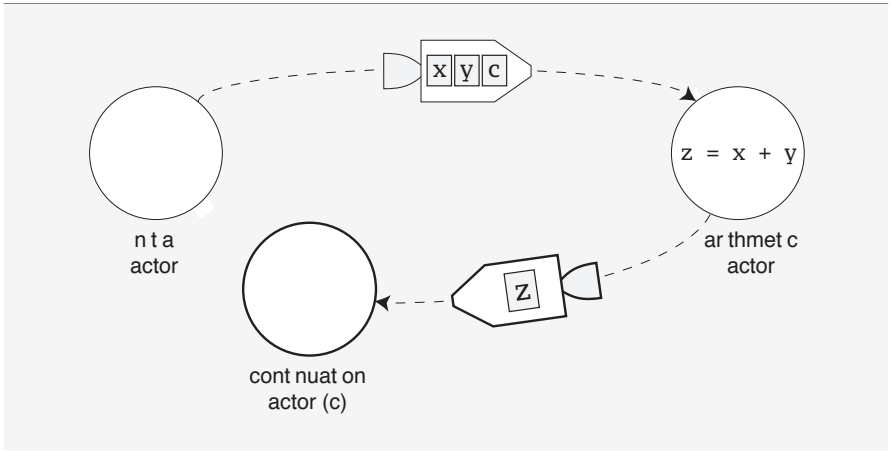

Figure 2.4 · Actor computation with continuation message passing.

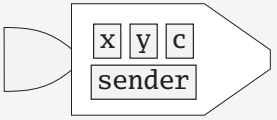

Figure 2.5 · Every message carries a sender reference.

to concurrent programming.[1]

The simplest kind of continuation is a reference to the sending actor. Having access to a message's sender is so convenient that the Scala actors library implicitly includes a reference to the sending actor in messages.

An actor's continuation is a key element of control flow in actor-based programming: Program control flows from one actor to another as continuations are passed between actors. At the same time, the actor message that sends possible continuations may also include the data required by the actor to determine control flow. The actor model unifies control flow and data flow in the sense that data as well as an actor's continuation can be passed inside the same message.

That unified view makes it easier to design actor-based programs. When

[1] Agha, "Concurrent Object-Oriented Programming" [Agh90]

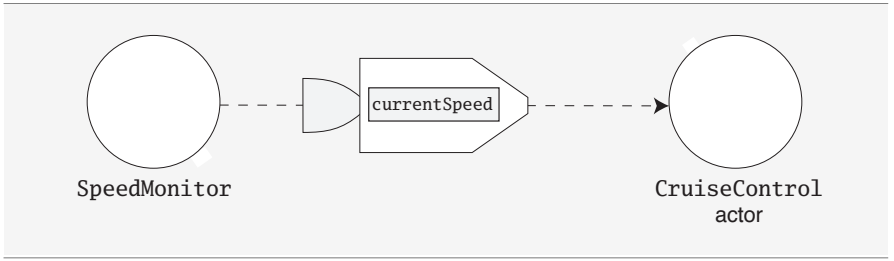

Figure 2.6 · `CruiseControl` actor receiving `currentSpeed` message.

designing a program with actors, it is helpful to first determine the kinds of control flow your code requires. Those control decisions would be made by actors. Thus, you would next define what data those control flow decisions require, and send that data to the appropriate actors inside messages.

The speed maintenance control structure in example above, for instance, requires a decision about whether to maintain or reduce the current speed. That decision needs just the current and desired speed values. The simplest implementation merely evaluates the values in the incoming message and takes appropriate action based on those values. Note that the message sender does not have to be an actor.

A more modular approach would define an actor responsible for deciding the required speed adjustment, and would then send the result to a continuation, as shown in Figure 2.7.

One advantage of the actor-based approach is that it allows the continuation of `CruiseControl`—`ThrottleControl`—to be defined after `CruiseControl` is already defined—and even after an instance of `CruiseControl` is already initialized and loaded into memory: `ThrottleControl` is simply an actor with the uniform actor interface to receive messages. Thus all `CruiseControl` needs is a reference to the continuation actor, such as that actor's address.

The ability to perform such extreme *late binding* of a continuation allows developers to incrementally add knowledge—such as control flow—to an actor-based system. Indeed, actors grew out of the desire to create large knowledge-based systems in an incremental fashion.

Late binding in actor control flow is also an important tool in lending robustness to an actor-based system. For instance, an actor may be redundantly defined, allowing a message sender to send replicated messages.

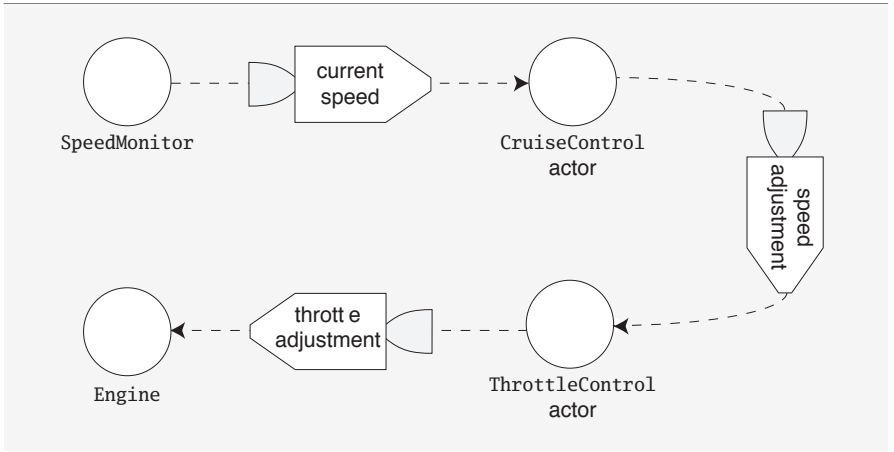

Figure 2.7 · A more modular approach to cruise control with further decomposition of responsibilities into actors.

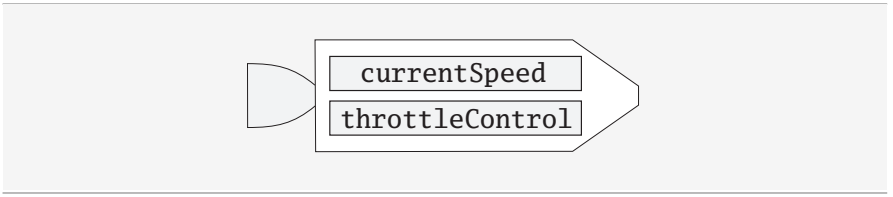

Figure 2.8 · A continuation actor included in a message affects control flow and accommodates late binding in an actor system.

If actors interacting via messages sounds similar to how objects communicate in an object-oriented system, that likeness is no mere coincidence. Indeed, the actor model was developed at the same time the first object-oriented languages were designed, and was, in turn, influenced by object-oriented concepts. Alan Kay, an inventor of object-oriented programming, noted that message passing between objects is more central to object-oriented programming than objects themselves are. In a note to a Smalltalk discussion group, Kay wrote:[2]

> The big idea is "messaging" – that is what the kernel of Smalltalk/Squeak is all about (and it's something that was never

[2] Kay, an email on messaging in Smalltalk/Squeak. [Kay98]

> quite completed in our Xerox PARC phase). The Japanese have a small word – ma – for "that which is in between" – perhaps the nearest English equivalent is "interstitial." The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be...

The actor model can be viewed as a special case of object-oriented programming where all communication between objects takes place via message passing, and when an object's internal state changes only in response to messages.

2.3 Actor creation

An actor can send a message only to its acquaintances—other actors whose addresses it knows. Continuation passing is one way in which an actor can learn the addresses of other actors. Another way is for an actor to create other actors as part of evaluating a message. Such newly created actors—child actors—can have an independent lifetime from that of the creating actor. Having created new actors, the creating actor can send messages to the new actors, possibly passing its own address as part of those messages.

An actor's ability to create other actors makes it easy to implement fork-join parallelism, for instance: Upon receiving a message, an actor may decide to divide up a potentially compute-intensive job and create child actors for the purpose of processing parts of that larger computation. A creator actor would divvy up work among its child actors, and wait for the children to complete their work and send their results back to the parent. Once all the results have been collected, the parent actor can summarize those results, possibly sending the results to yet another actor, or continuation. We will provide several examples of fork-join parallelism in later chapters.

2.4 Actor events

Although we have so far focused on an actor's ability to send messages to other actors, all the "action" in an actor takes place at the time a message is *received*. Receiving messages and creating other actors are two examples of events in an actor system.

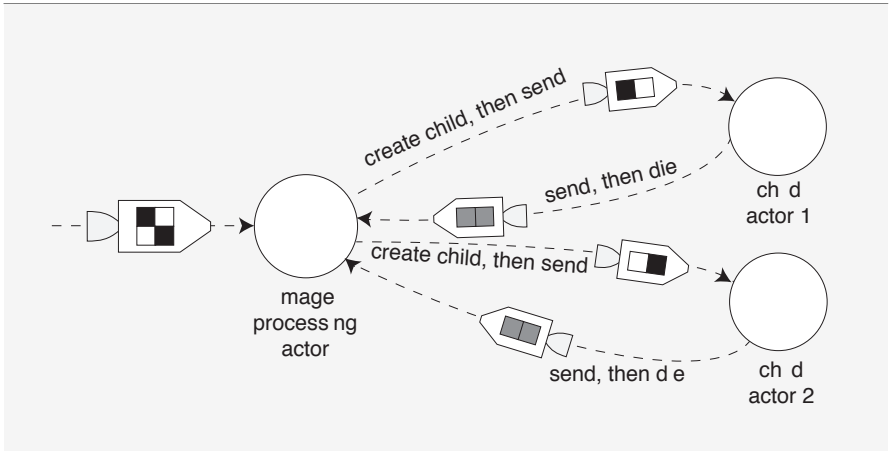

Figure 2.9 · An actor can create child actors as part of its message processing, and delegate work to those child actors.

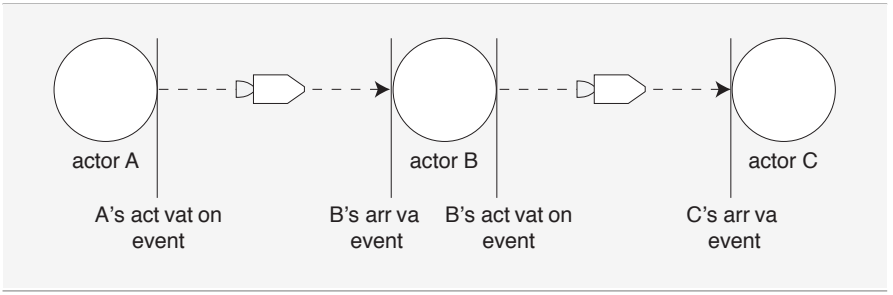

Figure 2.10 · C's arrival event is activated by B's arrival event.

Events and their relationships illuminate how physical phenomena inspired the actor programming model. For instance, when actor B receives a message from actor A, B can send a message to C as a result, defining an order of *arrival events*.

In this example, the message sent to B caused, or *activated*, event(c).

In their seminal paper on the "Laws for Communicating Parallel Processes," Carl Hewitt and Henry Baker noted that[3],

> Activation is the actor notion of causality… A crude analogy

[3] Hewitt and Baker, "Laws for Communicating Parallel Processes" [HB77]

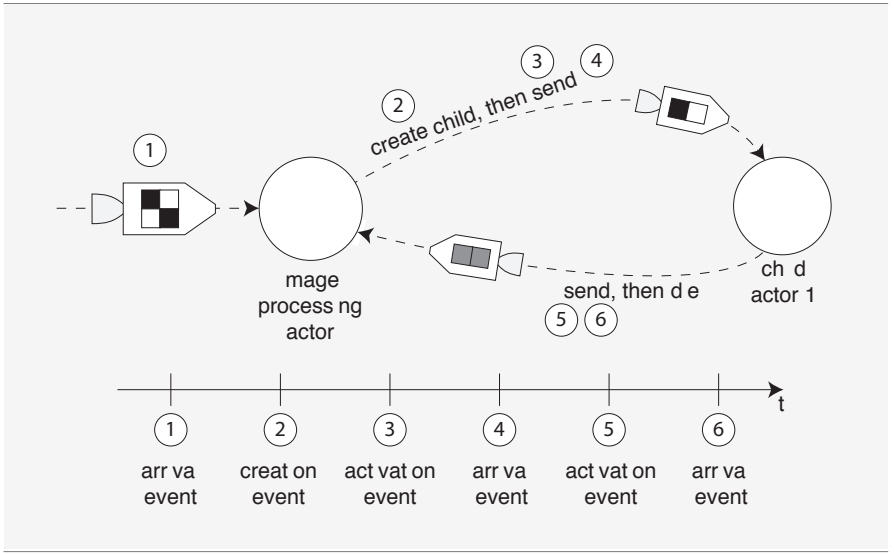

Figure 2.11 · Event causality in an actor system: an inital event is followed by arrival events, activations, and actor creation events.

> from physics may make activation more clear. A photon (message) is received by an atom (target) which puts it into an excited state. After a while, the atom gives off one or more photons and returns to its ground state. These emitted photons may be received by other atoms, and these secondary events are said to be activated by the first event.

In addition to arrival events and actor creation events, an actor-based system also includes some "initial event" that gets the ball rolling, so to speak. Causality extends to all three types of events: The initial event must precede all other events and may include a set of initial actors. Those actors can process activation events from each other in any order. Finally—and obviously—an actor's creation event must always precede activation events targeting that actor.

We can see that arrival and activation events nicely line up in a time-ordered sequence, one event always occurring before the other. Indeed, an actor-based computation can be described as a linear ordering of combined arrival and activation events: A computation starts with some event, which

is then followed by a finite number of other events, and is finally terminated by the computation's last event. The order of events is strict in the sense that an event can only be influenced by other events that preceded it.

When we say one event occurs before—or after—another event, we intuitively refer to some notion of time. In a sequential computation, when the entire program state is shared globally, the sequence of events that make up the computation refers to the global time: Time shared by all objects participating in the computation. An actor-based system, by contrast, splits the global program state into many local states held by each actor. Those actors interact merely by passing messages, and have no reference to a shared notion of time. Instead, arrival orderings of events in an actor-based system refer to the time *local* to an actor—there is no requirement to have a notion of global time.

Viewing computations as a partial order of actor events occurring in local time to an actor turns out to be a powerful abstraction. The designers of the actor programming model demonstrated that you can implement any control structure as a sequence of actor events. Because actor-based programming is designed with concurrency as its basic assumption, it is theoretically possible, therefore, to implement any sequential program in a concurrent manner with actor messaging.

2.5 Asynchronous communication

The reason actors ignore message sending as an event, and emphasize message arrival instead, is that message transmission between actors may incur some delay. For instance, actor A may send a message to B, and include C as a continuation. Although C's message is activated by A, there may be some delay between A sending the message and C receiving a message. The delay may be due to some processing time incurred at B as well as to communication latency. Considering message delay as an integral part of a computation is another way actor communication differs from simple object invocation in object-oriented programming.

Practical actor-based systems deal with message delay by offering the ability to asynchronously pass messages between actors: Once an actor dispatches a message to another actor, the sending actor can resume its work immediately. The sending actor does not need to wait for a reply. Indeed, some actor messages will never produce a reply. When replies are expected,

those will also be sent asynchronously. As Carl Hewitt noted, actors communicate via messages following the principle of "send it and forget it."

You might already be familiar with the concept of asynchronous message passing from modern Web programming models, such as AJAX. AJAX is based on asynchronous messages exchanged between a Web browser and a remote server. AJAX has proven practical in Web applications because an unknown amount of latency may be incurred both in the network communication as well as in the server processing an incoming message. A Web client can simply send a message to the server, register a listener for future replies from the server, and immediately return control to the user interface, keeping the application responsive to user interaction.

Similarly, asynchronous messages in actor communication means that the actor model works equally well across networks as it does in a single address space. Indeed, the Scala actors library defines both "local" as well as "remote" actors. Switching between local and remote actors is surprisingly simple because asynchronous messaging works well in either case. In addition to asynchronous messaging primitives, the Scala actors API provides for synchronous message sending as well.

The "send it and forget it" principle assumes that all messages sent are eventually received by the target actor. Although in many systems the notion of "lost" messages is real—for instance, the server hosting a target actor may crash resulting in the target never receiving the message—the actor model assumes that infrastructure components ensure reliable message transmission. In other words, the actor model assumes a finite—although initially unknown or *unbounded*—amount of time between message sending and message transmission.

How to achieve reliable message transmission is no more a part of the actor-based programming model than, say, the problem of high availability for database management systems is a part of relational algebra and SQL programming. The actor programming model nevertheless makes the implementation of highly reliable and available systems much easier: Reliability is often achieved through redundancy and replication, and actors' natural propensity to work well in distributed, concurrent systems serves those needs well. We will provide throughout this book examples and best practices for achieving reliable actor communication.

2.6 You've got mail: indeterminacy and the role of the arbiter

Although the actor model doesn't prescribe a mechanism for reliable message delivery, it acknowledges that many messages may be sent to a single actor in quick succession. Rapidly arriving messages could result in a sort of denial-of-service for the actor, rendering the actor incapable of processing the incoming message flow. To alleviate that problem, the actor model requires that a special object be provided by each actor for the purpose of receiving messages and holding those messages until the actor is able to process them. Such an arbiter is often called a mailbox, since it provides a function similar to, say, an email account: Messages can arrive in the mailbox at any time and will be held there until the recipient is ready to process them.

Email clients give you complete freedom in choosing the order in which you read new messages. In a similar way, an actor's mailbox may provide the actor with messages in any order. The only requirement is that an actor process one message at a time from its mailbox. Because the order in which messages are processed by an actor cannot be determined in advance—the order of message delivery is *indeterminate*—a developer must ensure that the correctness of an actor-based program does not depend on any specific message order.[4]

The actor model makes such programming practices easy, however, because any sort of data can be contained in an actor message, and also because an actor is able to maintain its own state. Consider, for instance, an actor that sums up two integers and sends the result to a third actor. In the simplest implementation, a single actor message would contain the two integers as well as the continuation:

That implementation assumes that both integers were known to the message sender. Another implementation may process integers from separate senders, expecting a message with a single integer, in addition to a name that uniquely identifies the calculatation to the calculation:

Since addition is commutative, the order of message transmission does not matter: The actor saves away the initial value received for a calcula-

[4] In Scala actors, the guarantees of message delivery are a bit stronger than the full indeterminacy of the pure actor programming model. If an actor sends several messages to the same receiver, those messages arrive in the receiving actor's mailbox in the order in which they have been sent.

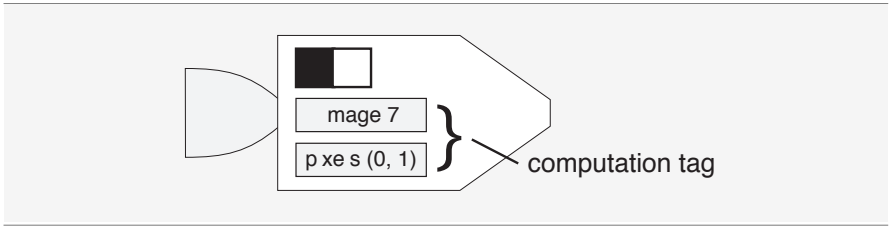

Figure 2.12 · Actor message with integer and reference to calculation.

tion and, upon receiving the second integer for the calculation, performs the arithmetic operation and sends the reply.

Consider, however, a version of the arithmetic actor designed to add a set of integers. One problematic approach would be to have each integer message include a `lastElement` flag indicating whether it is the terminal element of the series. As soon as the actor receives the last element in the series, it could send the result to the continuation. But since message delivery order is not guaranteed, the last element may be received in any order, resulting possibly in a premature sending of the result.

Reliance on message ordering can often be alleviated by refactoring the actor communication, *i.e.*, reworking the contents of the messages. For instance, the above message could include, instead of the `lastElement` flag, the number of elements in the series. Throughout this book, we will include tips and techniques to design actor communication that does not rely on message order.

Indeterminacy in the actor model results because an actor's mailbox, or arbiter, can receive and provide messages to the actor in any order. The order of message arrival can't be guaranteed—or even specified—due to the inevitable latencies in message transmission between actors: While a message is guaranteed to eventually arrive, the messages transmission time is unbounded.

As we mentioned in the previous chapter, a programming model based on unbounded indeterminism powerfully captures the nature of concurrent computation. In the actor model, concurrency is the norm, while sequential computation is a special case.

2.7 Actor lifecycle

Because of their readiness to process incoming messages, actors can be imagined as "live objects," or objects with a lifecycle. Unlike the lives of movie actors, the life of an actor object is rather boring: Once an actor is created, it typically starts processing incoming messages. Once an actor has exceeded its useful life, it can be stopped and destroyed, either of its own accord, or as a result of some "poison pill" message.

Creating and starting an actor are separate, although closely related, tasks. In Scala, actors are plain old Scala objects, and can, therefore, be created via their constructors. An actor starts processing incoming messages after it has been started, which is similar to starting a Java thread.

In practice, it is useful for actors to be able to monitor each others' lifecycle events. In the fork-join example, for instance, a child actor may decide to terminate upon sending its response to the parent, in order to free up memory. It could at that point send a message to its parent actor indicating its exit. In Scala Actors, lifecycle monitoring is supported through actor *links*. Actor linking is explained in detail in Section 6.2 in Chapter 6.

Chapter 3

Scala's Language Support for Actors

This chapter will explain how Scala's focus on scalable language features makes it possible to define the actor API as a domain-specific language embedded in the Scala language. This chapter will also cover the Scala features you need to understand in order to follow the examples later in the book.

Chapter 4

Actor Chat

The previous chapters illustrate the actor programming model's focus on message passing. Not surprisingly, much of Scala's actors library defines a rich set of programming constructs for sending and receiving messages. These constructs appear as an internal domain-specific language—DSL—to the developer. This chapter illustrates the key elements of Scala's actor DSL with a quintessential messaging application: a chat program.

A chat program allows users to communicate with each other by exchanging messages about various topics. Each topic is represented by a chat room. Users interested in following a discussion about a topic can subscribe to a chat room. Once subscribed, a user may send messages to the chat room and, in turn, receive messages from other chat room subscribers. Figure 4.1 provides an overview of the chat application developed in this chapter.

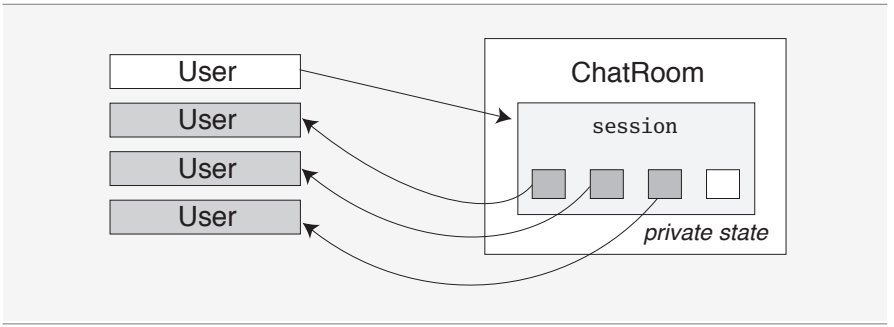

Figure 4.1 · Actor chat: Users subscribe to a chat room to received messages sent to that room. The chat room maintains a session of subscribers.

4.1 Defining message classes

The chat room application's communication centers around messages: a user sends a `Subscribe` message to indicate a desire to send and receive chat room messages, an `Unsubscribe` message to remove a user from the chat room's session, and a `UserPost` message to forward to the chat room's subscribers. When a chat room receives a user's `UserPost` message, it forwards the contents of that message to each of its subscribers inside a `Post` message. The chat room also makes sure not to send a message back to the user posting that message, lest an unfriendly "echo" effect appear.

A typical first step in developing an actor-based program is to define the message classes that represent the application's communication pattern. Scala's case classes come in handy for defining actor messages. As we'll shortly see, case classes are especially useful in the context of pattern matching, a key technique in actor message processing. Listing 4.1 shows how to define the message classes for our chat application.

```
cccaaasssdddUsssss(name: String)

            Subscribe(user: User)

cccaaasssdddUsssssscrib(user: User)

            Po t(msg: String)

case class UserPost(user: User, post: Post)
```

Listing 4.1 · Case classes for `Users` and messages

4.2 Processing messages

In addition to the messages, a key abstraction in the chat application is the `ChatRoom`. `ChatRoom`'s main responsibilities include keeping a session of actively logged-in users, receiving messages from users, and transmitting a user's message to other interested users, as shown in Figure 4.2.

The subscribers of a chat room are managed as private state of a `ChatRoom`. A `ChatRoom` modifies that state upon receiving a `Subscribe` or `Unsubscribe` message, illustrating an important concept of actor-based programming: some messages sent to an actor alter the actor's internal state

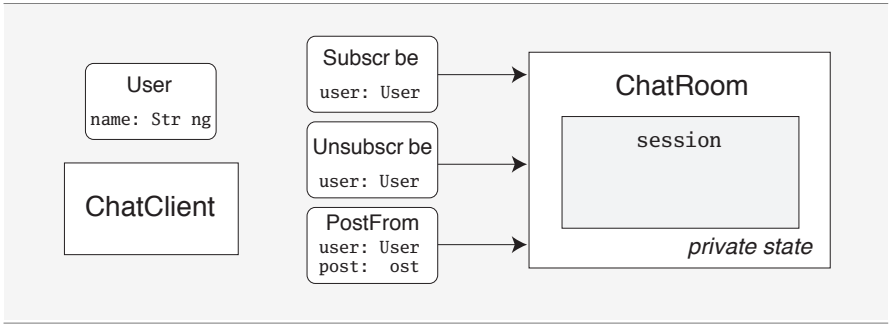

Figure 4.2 · All communication between the chatroom and users takes place via messages.

that, in turn, affects the actor's subsequent behavior. For instance, a new `Subscribe` message causes a `ChatRoom` to forward subsequent `Post` messages to the newly registered user, affecting the application's message flow.

`ChatRoom`'s message handling responsibilities are implemented by extending the `scala.actors.AAAcccttoorr` trait. Extending `Actor` means that a `ChatRoom` benefits from the trait's message handling infrastructure, such as the mailbox.

All message handling in an actor takes place inside the `act` method; listing 4.2 shows how to define it.

```
import scala.actors.Actor

class ChatRoom extends Actor {
  def act() {
    // the actor's behavior
  }
}
```

Listing 4.2 · Defining `act`

Message processing inside `act` starts when you invoke `start` on the actor:

```
val chatRoom = new ChatRoom
chatRoom.start()
```

A key task in actor message processing is to obtain the next available message from the actor's mailbox. `Actor`'s `receive` method accomplishes that by removing a message from the mailbox and making that message available to a series of pattern matching cases that you pass as a parameter to `receive`. The example in Listing 4.3 defines a pattern for each of the three types of messages a `ChatRoom` is expected to receive.

```
class ChatRoom extends Actor {
  def act() {
    while (true) {
      receive {
        cccaaaSSSuuubbbsssccrrriiibbbeee(user) =>        ///hhhaaannnddddlllesss
             UUUnsubscribe(user) =>                      uuunsubscriptions
              serPost(user, post) =>                      ser pos s
      }
    }
  }
}
```

Listing 4.3 · Incoming message patterns

Each invocation of `receive` obtains the next available message from the actor's mailbox, and passes that message to a list of pattern matching cases. Patterns are evaluated on a message starting from the first pattern and moving down in the list of patterns; if a pattern matches, the matching message is removed from the mailbox, and subsequent patterns are not evaluated on the message. If no match is found, the message is left in the mailbox.

In this example, `ChatRoom` expects either a `Subscribe`, an `Unsubscribe`, or a `UserPost` message. Upon receiving any such message, `ChatRoom` evalutes the expression on the right side of the pattern's `=>`.

The Scala actors library also provides a shorthand for defining and starting an actor in a single step without extending the `Actor` trait. Listing 4.4 shows how to write the above code using the shorthand notation.

Handling subscription messages

Upon receiving a `Subscribe` message, a `ChatRoom` must add the user to its subscribers session. At first, it may seem convenient to keep chat room sub-

```
val chatRoom =
  actor {
    while (true) {
      receive {
              Subscribe(user) =>
               nsubscribe(user) =>
        cccaaaIIIIsssseeePost(user, post) =>
      }
    }
  }
```

Listing 4.4 · Creating and starting an actor with `actor`

scribers in a list of `Users`. Note, however, that subscribers must be able to receive messages from the chat room. In our current design, when a `UserPost` arrives, `ChatRoom` iterates through its subscribers session, and sends the message's content to all subscribing users, except to the user sending the message.

So that a user can accept messages from `ChatRoom`, we can represent a user as an actor inside the subscriber session. When `ChatRoom` receives a `Subscribe` message, it creates a new actor representing the user, and associates the user with the newly created actor. That actor, in turn, will process `Post` messages sent to it from the chat room; this is shown in Listing 4.5.

4.3 Sending actor messages

At this point, `ChatRoom` is ready to process subscription messages. Scala's actors library supports both asynchronous and synchronous messages sending.

Asynchronous messages sending

You send a message *asynchronously* to an actor with the `!` method. In using `!` to denote message sending, Scala follows the tradition of Erlang actors:

```
val chatRoom = new ChatRoom
chatRoom ! Subscribe(User("Bob"))
```

```
var session = Map.empty[User, Actor]
while (true) {
  receive {
    case Subscribe(user) =>
      val sessionUser =
        actor {
          while (true) {
            self.receive {
              case Post(msg) => // Send message to sender
            }
          }
        }
      session = session + (user -> sessionUser)
      ///hhaaanndUsserlPosteeessage
                nsubscrib  message
  }
}
```

Listing 4.5 · Representing a user as an actor inside a session

! !sends a message to chatRoom and returns immediately: It does not wait for any confirmation or reply from the target actor. In addition to the message, also sends an implicit reference to the sender to the target actor. That reference is always available inside the target actor via the sender variable.

Listing 4.6 illustrates how the target actor uses the sender reference to process a Post message:

Note that there are *two* actors in the above example: ChatRoom and the actor representing the user inside the chat room, sessionUser. When the chat room actor receives a Subscribe message, it assigns the sender of that message to the subscriber variable. The closure passed to the actor method, in turn, captures that variable and makes it possible for subscriberActor to receive and process Post messages. Once sessionUser is initialized to the actor representing the user, it is saved away in the session map.

```
var session = Map.empty[User, Actor]

while (true) {
  receive {
    case Subscribe(user) =>
      val subscriber = sender
      val sessionUser =
        actor {
          while (true) {
            self.receive {
              case Post(msg) => subscriber ! Post(msg)
            }
          }
        }
      session = session + (user -> sessionUser)

    ///hhaannddllee  mmeessssaaggee
              nsubscrib  message
  }
}
```

Listing 4.6 · Using the `sender` reference

Synchronous messages

Scala also supports *synchronous* message sending via the `!?` operator:

```
val chatRoom = new ChatRoom
chatRoom !? Subscribe(User("Bob"))
```

Unlike with asynchronous message sending, `!?` blocks the calling thread until the message is sent and a reply received. Listing 4.7 shows how `ChatRoom` might return an acknowledgement when handling a subscription message with the `reply` method:

The client can capture and process the reply:

```
chatRoom !? Subscribe(User("Bob")) match {
  case response: String => println(response)
}
```

```
var session = Map.empty[User, Actor]

while (true) {
  receive {
    case Subscribe(user) =>
      val subscriber = sender
      val sessionUser =
      actor {
        while (true) {
          self.receive {
            case Post(msg) => subscriber ! Post(msg)
          }
        }
      }
      seeession= session + (user -> sessionUser)
      r ply("Subscribed " + user)
  }
}
```

Listing 4.7 · Using the `reply` method

Futures

In some cases, you want the calling thread to return immediately after sending a message, but you may also need access to the target actor's reply at a later time. For instance, you may want to quickly return from sending a subscription message, but also record the chatroom's acknowledgment message in the future.

Scala actors provide the concept of *futures* for such a scenario: Futures messages are sent with the `!!` method, which returns a `Future` without blocking the calling thread. A future's value may or may not be evaluated by the caller; if it is, and if the future value is not yet available, the calling thread blocks:

```
val future = chatRoom !! Subscribe(User("Bob"))

// Do useful work
println(future()) // Wait for the future
```

Message timeouts

In the examples so far, `receive` blocks the actor's main thread until a matching message is found. In some cases, you may want to wait for suitable messages only for a certain period of time. The `receiveWithin` method allows you to specify a message timeout, and to be notified if no message was received within that time.

You can use the `receiveWithin` method to automatically unsubscribe a user if the user hasn't received a post message within a specified amount of time. In the following example, `TIMEOUT` will match if no `Post` message is received within 3 minutes; the user is then unsubscribed from the chatroom:

```
val sseeessionUser actor {
  while (true) {
      lf.receiveWithin (1800 * 1000) {
      cccaaaPssseem(msg) => subscriber ! Post(msg)
           TIMEOUT =>
        room ! Unsubscribe(user)
        self.exit()
    }
  }
}
```

Listing 4.8 · Using message timeouts with `receiveWithin`

Processing user posts

All that remains from our chat room to be fully functional is to implement the processing of a user's post:

```
var session = Map.empty[User, Actor]

def act() {
  while (true) {
    receive {
      case PostFrom(user, msg) =>
        for (key <- session.keys; if key != user) {
          session(key) ! msg
        }
      /// Unsubscribe message
    }
  }
}
```

Listing 4.9 · Processing post messages

Chapter 5

Event-Based Programming

The constructs we introduced in Chapter 2 tie each actor to a JVM thread: each actor needs its own dedicated Java thread. The thread-per-actor approach works well if your program requires relatively few actors.

If you anticipate many actors, however, or if the number of actors in your program varies depending on input, defining one thread per actor incurs significant overhead: Not only does each JVM thread require memory for its execution stack–which is usually pre-allocated,– each JVM thread may be mapped to an underlying operating system process. Depending on platform, context-switching between those processes involves switching between kernel and user modes, an expensive operation.

To allow many actors in a JVM, you can make your actors *event-based*. Event-based actors are implemented as event handlers instead of as threads, and are therefore more light-weight than their thread-based cousins. Since event-based actors are not tied to Java threads, event-based actors can execute on a pool of a small number of worker threads. Typically, such a pool should contain as many worker threads as there are processor cores in the system. That maximizes parallelism while keeping the overhead of pool threads–memory consumption and context-switching–to a minimum.

5.1 Events vs. threads

Making an actor event-based is not entirely transparent to the programmer. That is because event-based programming follows a different paradigm from programming with threads. A typical actor spends some time waiting for incoming messages, and a key difference between event-based and thread-

based actors can be illustrated by an actor's waiting strategy.

A thread-based actor waits by invoking `wait` on an object for which its thread holds the associated lock. That thread resumes whenever another thread invokes `notify` (or `notifyAll`) on the same object.[1] An event-based actor, by contrast, registers an event-handler with the actor runtime. After that registration, the actor's computation usually finishes, and the thread initially running the computation is free to execute other tasks, or go to sleep if there is nothing else to do. Later, when an event of interest is fired–when a message of interest to the actor is received, for instance–the actor runtime schedules the actor's event-handler for execution on a thread pool, and the actor's computation resumes. In that manner, event-based actors are decoupled from underlying JVM threads.

5.2 Making actors event-based: `react`

Because event-based actors differ from thread-based actors in their waiting strategies, turning a thread-based actor into an event-based one is straightforward. The thread-based actors we have seen so far used `receive` to wait for a matching message to arrive in their mailbox. To make an actor event-based, replace all uses of `receive` with invocations of the `react` method. As with `receive`, `react` expects a block of message patterns that are associated with actions to process a matching message.

Although replacing `receive` with `react` is a simple code change, receive and react are used quite differently in programs. The following examples explore these differences.

Using `react` to wait for messages

Listing 5.1 shows the definition of a method that (recursively) builds a chain of actors and returns the first actor. Each actor in the chain uses `react` to wait for a `'Die` message. When it receives such a message, the actor checks to see if it is last in the chain (in this case, `next == null`). The last actor in the chain simply responds with `'Ack` to the sender of the `'Die` message and terminates.

[1] In practice, waiting is slightly more complicated, because threads may be interrupted during waiting.

```
def buildChain(size: Int, next: Actor): Actor = {
  val a = actor {
    react {
      case 'Die =>
        val from = sender
        if (next != null) {
          neeext! 'Die
          r act {
            case  Ack => from ! 'Ack
          }
        } else from ! 'Ack
    }
  }
  if (size > 0) buildChain(size - 1, a)
  else a
}
```

Listing 5.1 · Building a chain of event-based actors.

If the current actor is not the last in the chain, it sends a `'Die` message to the next actor, and waits for an `'Ack` message. When the `'Ack` arrives, it notifies the original sender of the `Die` and terminates. Note that we store the sender of the original `'Die` message in the local variable `from`, so that we can refer to this actor inside the nested `react`. Inside the nested `react`, `sender` refers to the next actor in the chain, whereas the current actor should send its `'Ack` to the previous actor in the chain, which is stored in `from`.

Let's use the `buildChain` method by putting it into an object with the `main` method shown in Listing 5.2. We store the first command-line argument in the `numActors` variable to control the size of the actor chain. Just for fun, we take the time to see how long it takes to build and terminate a single chain of size `numActors`. After building the chain using `buildChain`, we immediately send a `'Die` message to the first actor in the chain.

What happens is that each actor sends `'Die` to the next actor, waiting for an `'Ack` message. When the `'Ack` is received, the actor propagates it to the previous actor and terminates; the first actor is the last one to receive its `'AAAcc`. When the `receive` operation in the `main` method starts processing , all actors in the chain have terminated.

```
def main(args: Array[String]) {
  val numActors = args(0).toInt
  val start = System.currentTimeMillis
  buildChain(numActors, null) ! 'Die
  receive {
    case 'Ack  >
      val end ==System.currentTimeMillis
      println("Took "+(end-start)+" ms")
  }
}
```

Listing 5.2 · The `main` method.

How many actors are too many?

Actors that use `react` for receiving messages are very lightweight compared to normal JVM threads. Let's find out just how lightweight they actually are by creating chains of actors of ever increasing size until the JVM runs out of memory. Moreover, we can compare that chain with thread-based actors by replacing the two `reacts` with `receives`.

But first, how many event-based actors can we create? And, how much time does it take to create them? On a test system, a chain of one thousand actors is built and terminated in about 115 ms, while creating and destroying a chain of ten thousand actors takes about 540 ms. Building a chain with five hundred thousand actors takes 6,232 ms, but one with a million actors takes a little longer: about 26 seconds without increasing the default heap size of the JVM (Java HotSpot(TM) Server VM 1.6.0).

Let's try this now with thread-based actors. Since we are going to create lots of threads, we should configure the actor run-time to avoid unreasonable overheads.

Configuring the actor run-time's thread pool

Since we intend to use lots of threads with thread-bound actors, it is more efficient to create those threads in advance. Moreover, we can adjust the size of the actor runtime's internal thread pool to optimize actor execution. Scala's actor runtime allows its thread pool to resize itself according to the number of actors blocked in `receive`–each of those actors needs its own

thread–but that resizing may take a long time, since the thread pool is not optimized to handle massive resizing efficiently.

The internal thread pool is configured using two JVM properties, `actors.corePoolSize` and `actors.maxPoolSize`. The first property is used to set the number of pool threads that are started when the thread pool is initialized. The latter property specifies an upper bound on the total number of threads the thread pool will ever use (the default is 256).

To minimize the time it takes to resize the thread pool, we set both properties close to the actual number of threads that our application needs. For example, when running our chain example with one thousand thread-based actors, setting `actors.corePoolSize` to one thousand and `actors.maxPoolSize` to, say one thousand ten keeps the pool resizing overhead low.

With these settings in place, it takes about 12 seconds to create and destroy a chain of one thousand thread-based actors. A chain of two thousand threaded actors takes already over 97 seconds. With a chain of three thousand actors, the test JVM crashes with an `java.lang.OutOfMemoryError`.

As this simple test demonstrates, event-based actors are much more lightweight than thread-based actors. The following sections explore how to program with event-based actors effectively.

Using `react` effectively

As we mentioned above, with `react` an actor waits for a message in an event-based manner: Under the hood, instead of blocking the underlying worker thread, `react`'s block of pattern-action pairs is registered as an event-handler. That event handler is then invoked by the actor runtime when a matching message arrives.

The event-handler is all that is retained before the actor goes to sleep. In particular, the call stack, as it is maintained by the current thread, is discarded when the actor suspends. That allows the runtime system to release the underlying worker thread so that thread can be reused to execute other actors. By running a large number of event-based actors on a small number of threads, the context-switching overhead and the resource consumption of thread-bound actors is reduced dramatically.

That the current thread's call-stack is discarded when an event-based actor suspends bears an important consequence on the event-based actor programming model: a call to `react` never return normally. `react`, like any

```
def waitFor(n: Int): Unit = if (n > 0) {
  react {
    case 'Die =>
      val from = sender
      if (next != null) {
        n xt !  Die
        reeeact{
          case 'Ack => from ! 'Ack; waitFor(n - 1)
        }
      } else { from ! 'Ack; waitFor(n - 1) }
  }
}
```

Listing 5.3 · Sequencing `react` calls using a recursive method.

Scala or Java method, could return normally only if its full call-stack was available when it executed. But that isn't the case with event-based actors. In fact, a call to `react` does not return at all.

That `react` never returns means that no code can follow a `react` method invocation: Since `react` doesn't return, code following `react` would never execute. Thus, invoking `react` must always be the last thing an event-based actor does before it terminates.

Since an actor's main job is to handle interesting messages, and since `react` defines an event-based actor's message handling mechanism, you might think that `react` will always be the last, and even only, thing an actor needs to do. However, it is sometimes convenient to perform several `react` invocations in succession. In those situations, you could nest `react` invocations in sequence, as we saw in Listing 5.1.

Alternatively, you could define a recursive method that runs several `reacts` in sequence. For instance, we can extend our simple chain example so that an actor waits for a specified number of `'Die` messages before it terminates. We can do this by replacing the body of the chain actors with a call to the `waitFor` method as it is shown in Listing 5.3. `waitFor` method tests up-front whether the current actor should terminate (if `n == 0`) or continue waiting for messages. The protocol logic is the same as before. The only difference is that after each of the message sends to `from`, we added a recursive call to `waitFor`.

Recursive methods with `react`

Looking at the example in Listing 5.3, you might be concerned that calling a recursive method in this way could quickly lead to a stack overflow. The good news, however, is that `react` plays extremely well with recursive methods: Whenever an invocation of `react` resumes due to a matching message in the actor's mailbox, a task item is created and submitted to the actor runtime's internal thread pool for execution.

The thread that executes that task item doesn't have much else on its call stack, apart from the basic logic of being a pool worker thread. As a result, every invocation of `react` executes on a call stack that is as good as empty. The call stack of a recursive method like `waitFor` in Listing 5.3, therefore, doesn't grow at all thanks to the invocations of `react`.

Composing `react`-based code with combinators

Sometimes it is difficult or impossible to use recursive methods for sequencing multiple `react`s. That is the case when reusing classes and methods that make use of `react`: By their very nature, reusable components cannot be changed after they have been built. In particular, we cannot simply perform invasive changes such as when we added iteration through a recursive method in the example in Listing 5.3. This section illustrates several ways in which `react`-based code can be reused.

```
def sleep(delay: Long) {
  rreeegister(timedelay, self)
    act { case 'Awake => /* OK, continue */ }
}
```

Listing 5.4 · A `sleep` method that uses `react`.

For example, suppose our project contains the `sleep` method shown in Listing 5.4. It registers the current actor, `self`, with a timer service (not shown) to be woken up after the specified `delay` that is provided as a parameter. The `timer` notifies the registered actor using an `'Awake` message. For efficiency, `sleep` uses `react` to wait for the `'Awake` so that the sleeping actor does not require the resources of a JVM thread while it is sleeping.

Using the above `sleep` method invariably requires executing something after its `react` invocation. However, since we want to reuse the method as

```
actor {
  val period = 1000
  { // code before going to sleep
    sleep(period)
    andThen {
    // code after waking up
  }}}
}
```

Listing 5.5 · Using `andThen` to continue after `react`.

is, we cannot simply go ahead and insert something in the body of its `react`. Instead, we need a way to combine the `sleep` method with the code that should run after the `'Awake` message has been received *without changing the implementation of `sleep`*.

That is where the control-flow combinators of the `Actor` object come into play. These combinators allow expressing common communication patterns in a relatively simple and concise way. The most basic combinator is `andThen`. The `andThen` combinator combines two code blocks to run after each other even if the first one invokes `react`.

Listing 5.5 shows how you can use `andThen` to execute code that runs after invoking the `sleep` method. `andThen` is used as an operator that is written infix between two blocks of code. The first block of code invokes `sleep` as its last action, which, in turn, invokes `react`.

Note that the `period` parameter of `sleep` is declared outside the code block that `andThen` operates on. This is possible, because the two code blocks are actually *closures* that may capture variables in their environment. The second block of code is run when the `react` of the first code block–the one inside `sleep`–is finished. However, note that the second code block is really the last thing executed by the actor. The use of `andThen` does not change the fact that invocations of `react` do not return. `andThen` merely allows one to combine two pieces of code in sequence.

Another useful combinator is `loopWhile`. As its name suggests, it loops running a provided closure while some condition holds. Thanks to Scala's flexible syntax, `loopWhile` feels almost like a native language primitive. Listing 5.6 shows a variation of our actor chain example that uses `loopWhile` to wait for multiple `'Die` messages. Again, we make use of the fact that the

```
def buildChain(size: Int, next: Actor, waitNum: Int)
  : Actor = {
  val a = actor {
    var n = waitNum
    loopWhile (n > 0) {
      n -= 1
      react {
        case 'Die =>
          val from = sender
          if (next != null) {
            neeext! 'Die
            r act { case 'Ack => from ! 'Ack }
          } else from ! 'Ack
      }
    }
  }
  if (size > 0) buildChain(size - 1, a, waitNum)
  else a
}
```

Listing 5.6 · Using `loopWhile` for iterations with `react`.

two code block parameters of `loopWhile`: the condition `(n > 0)` and the body, are closures, since both code blocks access the local variable `n`. Note that the top-level `react` in the body of `loopWhile` is unchanged from the very first example that did not support iteration. The body might as well be extracted to a method—`loopWhile` works in either case.

5.3 Event-based futures

In Chapter 4, we illustrated how to use futures for result-bearing messages. Some of the methods to wait for the result of a future rely on the thread-based `receive` under the hood. While waiting for the result, those methods monopolize the underlying worker thread. It is also possible to wait for a future in an event-based way with `react`.

For example, suppose we would like to render a summary of all images linked from a web page at a given URL. It is possible to render each image

```
def renderImages(url: String) {
  val imageInfos = scanForImageInfo(url)
  val dataFutures = for (info <- imageInfos) yield {
    val loader = actor {
      react { case Download(info) =>
        reply(info.downloadImage())
      }
    }
    loader !! Download(info)
  }
  for (i <- 0 until imageInfos.size) {
    dataFutures(i)() match {
      case dddat@ ImageData(_) =>
        ren erImage(data)
    }
  }
  println("OK, all images rendered.")
}
```

Listing 5.7 · Image renderer using futures.

individually once the image finished downloading. To increase the throughput of the application, each image is downloaded by its own actor. Since each downloading actor performs a result-bearing task, it is convenient to use futures to keep track of the expected results. Listing 5.7 shows the code for rendering images in this way.

First, the URL provided as a parameter is scanned for image information. For each image, we start a new actor that downloads the image, and replies with image data. We obtain a future using the `!` message send variant. Once all the futures have been collected in `dataFutures`, the current actor waits for each of the futures in turn by invoking the future's `apply` method (with empty parameter list).

Example: image renderer with `react` and futures

The implementation that we just described blocks the underlying thread while waiting for a future. However, it is also possible to wait for a future

```
def renderImages(url: String) {
  val imageInfos = scanForImageInfo(url)
  val dataFutures = for (info <- imageInfos) yield {
    val loader = actor {
      react { case Download(info) =>
        reply(info.downloadImage())
      }
    }
    loader !! Download(info)
  }
  var i = 0
  loopWhile (i < imageInfos.sizeee){
    i += 1
    dataFutures(i-1).inputChann l.rrreeeact
      case data @ ImageData(_) =>   nderImage(data)
    }
  } andThen { println("OK, all images rendered.") }
}
```

Listing 5.8 · Using `react` to wait for futures.

in a non-blocking, event-based way using `react`. The key for this to work is the `InputChannel` associated with each `Future` instance. This channel is used to transmit the result of the future to the actor that created the future. Invoking a future's `apply` method waits to receive the result on that channel, using the thread-based `receive`. However, we can also wait for the results in an event-based way using `react` on the future's `InputChannel`.

Listing 5.8 shows an implementation that does just that. Since it is necessary to invoke `react` several times in sequence, you have to use one of the control-flow combinators of Section 5.2. In this example we use `loopWhile` to emulate the indexing scheme of the previous version in Listing 5.7. The main difference is that in this implementation the index variable `i` is declared and incremented explicitly, and that the generator in the for-comprehension has been replaced with a termination condition.

You can also build custom control-flow combinators that allow you to use `react` inside for-comprehensions. In the following section we explain how this can be done.

```
def renderImages(url: String) {
  val imageInfos = scanForImageInfo(url)
  val dataFutures = for (info <- imageInfos) yield {
    val loader = actr  {
      react { case Download(info) =>
        reply(info.dooownloadImage())
      }
    }
    loader !! Dooownload(info)
  }
  (for (ft <- F rEach(datttaFutures))
    ft.innnputChaaannel.reac
      case dat  @ ImageData(_) => renderImage(data)
    }) a dThen {
    println("OK, all images rendered.")
  }
}
```

Listing 5.9: A custom `ForEach` operator enables `react` in for-comprehensions.

```
caseeelllassForEach[T](iter: Iterable[T]) {
  df  foreach(fun: T => Unit): Unit = {
    va  it = iter...elemeeenttts
    loopWhiiile(itthasN x)  {
      fun( t.nex )
    }
  }
}
```

Listing 5.10 · Implementing the custom `ForEach` operator.

Building custom control-flow operators

Sometimes the existing control-flow combinators provided by the `Actor` object are not well-suited for the task at hand. In such cases building custom control-flow operators can help. In this section you will learn how the control-flow combinators provided by the `Actor` object can be used to build custom operators that enable the use of `react`–and methods using `react`–inside for-comprehensions.

Listing 5.9 shows the usage of a custom `ForEach` operator that allows iterating over a list, while invoking `react` for each element in the list. In this case, we want to iterate over the futures in `dataFutures`. We use `ForEach` to convert the plain `dataFutures` list into an object that acts as a generator in for-comprehensions. It generates the same values as the `dataFutures` list, namely all of the list's elements. However, it does so in a way that allows continuing the iteration even after an invocation of `react` in the body of the for-comprehension.

Listing 5.10 shows the implementation of `ForEach`. Making `ForEach` a case class allows omitting `new` when creating new instances. The constructor takes a parameter of type `Iterable[T]`—the collection that generates the elements for our iteration.

The `ForEach` class has a single method `foreach` that takes a parameter of function type `T => Unit`. Implementing the `foreach` method enables instances of the `ForEach` class to be used as generators in simple for-comprehensions like the one in Listing 5.9. The variable that is bound to the generated elements in the for-comprehension corresponds to the parameter of the function `fun`. The body of the for-comprehension corresponds to the body of `fun`.

Inside `foreach` we first obtain an iterator `it` from the `Iterable`. Then, we iterate over the collection using `it` and the `loopWhile` combinator of Section 5.2. In each iteration we apply the parameter function `fun` to the current element of the collection. Since we are using `loopWhile`, it is safe to invoke `react` inside `fun`.

Chapter 6

Exception Handling, Actor Termination and Shutdown

In this chapter we will take a look at how to handle errors in concurrent, actor-based programs. Actors provide several additional ways to handle exceptions compared to sequential Scala code. In particular, we will show how an actor can handle exceptions that are thrown but not handled by other actors. More generally, we will look at ways in which an actor can monitor other actors to detect whether they terminated normally or abnormally (for instance, through an unhandled exception). Finally, we introduce a number of concepts and techniques that can simplify termination management of actor-based programs.

6.1 Simple exception handling

An actor terminates automatically when an exception is thrown that is not handled inside the actor's body. One possible symptom of such a situation is that other actors wait indefinitely for messages from the dead actor. Since, by default, terminating actors do not generate any feedback, it can be quite time-consuming to find out what happened and why.

The simplest way of guarding against actors that silently terminate because of unhandled exceptions is to provide a global exception handler that is invoked whenever an exception propagates out of the actor's body. This is done by subclassing `Actor` (or `Reactor`) and overriding its `exceptionHandler` member. It is defined in `Reactor` as follows (omitting the modifiers):

```
object A extends Actor {
  def act() {
    react {
      case 'hello =>
        throw new Exception("Error!")
    }
  }
  override def exceptionHandler = {
    case e: Exception =>
      println(e.getMessage())
  }
}
```

Listing 6.1 · Defining an actor-global exception handler.

```
def exceptionHandler: PartialFunction[Exception, Unit]
```

As you can see, it is a parameterless method that returns a partial function that can be applied to instances of `java.lang.Exception`. Whenever an exception is thrown inside the body of an actor that would normally cause the actor to terminate, the run-time system checks whether the actor's `exceptionHandler` matches the given exception. If so, the `exceptionHandler` partial function is applied to the exception. After that the actor terminates normally.

Listing 6.1 shows how to override the `exceptionHandler` method so that it returns a custom partial function. Let's interact with the `A` actor using Scala's interpreter shell:

```
scala> A.start()
res0: scala.actors.Actor = A$@1ea414e

scala> A ! 'hello
Error!
```

As expected, `A`'s overridden `exceptionHandler` method runs, thereby printing the message string attached to the thrown exception, which is just `"Error!"`.

This form of exception handling using `exceptionHandler` works well together with control-flow combinators, such as `loop`. The combinators can

be used to resume the normal execution of an actor after handling an exception. For example, let's modify the `act` method of the `A` actor in Listing 6.1 as follows:

```
def act() {
  var lastMsg: Option[Symbol] = None
  loopWhile (lastMsg.isEmpty || lastMsg.get != 'stop) {
    react {
      cccaaasssee =>
        throw new Exception("Error!")))
           any: Symbol =>
        println("your message: "+any
        lastMsg = Some(any)
    }
  }
}
```

The invocation of `react` is now wrapped inside a `loopWhile` that tests whether the last received message is equal to `'stop`, in which case the actor terminates. Now, if the actor receives a `'hello` message, it throws the exception, which is handled as before. However, instead of terminating, the actor simply resumes its execution by continuing with the next loop iteration. This means that the actor is ready to receive more messages after the exception has been handled.

Let's try this out in the interpreter:

```
scala> A.start()
res0: scala.actors.Actor = A$@1cb048e

scala> A ! 'hello
Error!

scala> A.getState
res2: scala.actors.Actor.State.Value = Suspended

scala> A ! 'hi
your message: 'hi

scala> A ! 'stop
your message: 'stop

scala> A.getState
```

```
res5: scala.actors.Actor.State.Value = Terminated
```

Note that after sending `'hello` the actor eventually suspends waiting for the next message. The `getState` method can be used to query an actor's execution state. It returns values of the `Actor.State` enumeration, which is defined in the `Actor` object. The `Suspended` state indicates that the actor has invoked `react` and is now waiting for a matching message. Therefore, we can continue to interact with the actor by sending it a `'hi` message. After the actor receives a `'stop` message its `loopWhile` loop finishes and the actor terminates normally. The final state value is `Terminated`.

6.2 Monitoring actors

There are a number of scenarios where it is necessary to monitor the life cycle of (a group of) actors. In particular, error handling and fault tolerance in a concurrent system can be significantly simplified through monitoring. Here are some examples:

Scenario A. We want to be notified when an actor terminates normally or abnormally. For instance, we might want to replace an actor that terminated because of an unhandled exception. Or we might want to rethrow the exception in a different actor that can handle it.

Scenario B. We want to express that an actor *depends* on some other actor in the sense that the former cannot function without the latter. For instance, in a typical master-slave architecture the work done by a slave is useless if the master has crashed. In this case it would be desirable if all slaves would terminate automatically whenever the master crashes to avoid needless consumption of resources, such as memory.

- *Scenario A.* We want to be notified when an actor terminates normally or abnormally. For instance, we might want to replace an actor that terminated because of an unhandled exception. Or we might want to rethrow the exception in a different actor that can handle it.

- *Scenario B.* We want to express that an actor *depends* on some other actor in the sense that the former cannot function without the latter. For instance, in a typical master-slave architecture the work done by a

slave is useless if the master has crashed. In this case it would be desirable if all slaves would terminate automatically whenever the master crashes to avoid needless consumption of resources, such as memory.

Both of the above scenarios require monitoring the life cycle of an actor. In particular, they require us to be notified when an actor terminates, normally or abnormally. The actors package provides special support for managing such notifications. However, before diving into those monitoring constructs it is helpful to take a look at the ways in which actors can terminate.

Actor termination

There are three reasons why an actor terminates:

1. The actor finishes executing the body of its `act` method;
2. The actor invokes `exit`;
3. An exception propagates out of the actor's body.

The first reason is really a special case of the second one: after executing an actor's body, the run-time system invokes `exit` implicitly on the terminating actor. The `exit` method can be invoked with or without passing an argument. The `Actor` trait contains the following definitions (omitting the modifiers):

```
def exit(): Nothing
def exit(
        reason: AnyRef): Nothing
```

Both methods have result type `Nothing`, which means that invocations do not return normally, because an exception is thrown in all cases. In this case, the particular instance of `Throwable` should never be caught inside the actor, since it is used for internal life-cycle control. Invoking `exit` (with or without argument) terminates the execution of the current actor. The `reason` parameter is supposed to indicate the reason for terminating the actor. Invoking `exit` without an argument is equivalent to passing the `Symbol 'normal` to `exit`; it indicates that the actor terminated normally. Examples for arguments that indicate abnormal termination are:

- Exceptions that the actor cannot handle;

- Message objects that the actor cannot process;
- Invalid user input.

Exceptions that propagate out of an actor's body lead to abnormal termination of that actor. In the following section you will learn how actors can react to the termination of other actors. We will show the difference between normal and abnormal termination as seen from an outside actor. Importantly, we will see how to obtain the exit reason of another actor that terminated abnormally.

Linking actors

An actor that wants to receive notifications when another actor terminates must *link* itself to the other actor. Actors that are linked together implicitly monitor each other.
For example, Listing 6.2 shows a slave actor, which is supposed to do work on behalf of a master actor. The work done by the slave is useless without the master, since the master manages all results produced by the slave–the slave *depends* on its master. This means that whenever the master crashes its dependent slave should terminate, since otherwise it would only needlessly consume resources. This is where *links* come into play. Using the `link` method the slave actor links itself to the master actor to express the fact that it depends on it. As a result, the slave is notified whenever its master terminates.

By default, termination notifications are not delivered as messages to the mailbox of the notified actor. Instead, they have the following effect:

- If the exit reason of the terminating actor is `'normal`, no action is taken;
- If the exit reason of the terminating actor is different from `'normal`, the notified actor automatically terminates with the same exit reason.

In our master-slave example this means that the termination of the master actor caused by the unhandled exception results in the termination of the slave actor; the exit reason of the slave actor is the same as for the master actor, namely an instance of `UncaughtException`. The purpose of class `UncaughtException` is to provide information about the context in which

```
object Master extends Actor {
  def act() {
    Slave ! 'doWork
    react {
      case 'done =>
        throw new Exception("Master crashed")
    }
  }
}
objeeectSlave extends Actor {
  df  act() {
    llinkM(aster)
     oop {
      react {
        case 'doWork =>
          println("Done")
          reply('done)
      }
    }
  }
}
```

Listing 6.2 · Linking dependent actors.

the exception was thrown, such as the actor, the last message processed by that actor, and the sender of that message. The next section shows how to use that information effectively.

Let's use the interpreter shell to interact with the two actors.

```
scala> Slave.start()
res0: scala.actors.Actor = Slave$@190c99

scala> Slave.getState
res1: scala.actors.Actor.State.Value = Suspended

scala> Master.start()
Done
res2: scala.actors.Actor = Master$@395aaf
```

```
scala> Master.getState
res3: scala.actors.Actor.State.Value = Terminated

scala> Slave.getState
res4: scala.actors.Actor.State.Value = Terminated
```

Right after starting the `Slave` its state is `Suspended`. When the `Master` starts it sends a `'doWork` request to its `Slave`, which prints `Done` to the console and replies to the `Master` with `'done`. Once the `Master` receives `'done`, it throws an unhandled exception causing it to terminate abnormally. Because of the link between `Slave` and `Master`, this causes the `Slave` to terminate automatically. Therefore, at the end both actors are in state `Terminated`.

Trapping termination notifications. In some cases, it is useful to receive termination notifications as messages in the mailbox of a monitoring actor. For example, a monitoring actor may want to rethrow an exception that is not handled by some linked actor. Or, a monitoring actor may want to react to normal termination, which is not possible by default.

Actors can be configured to receive all termination notifications as normal messages in their mailbox using the Boolean `trapExit` flag. In the following example actor `b` links itself to actor `a`:

```
val a = actor {
    b
  self.trapExit = true
  link(a)
  ...
}
```

Note that before actor `b` invokes `link` it sets its `trapExit` member to `true`; this means that whenever a linked actor terminates (normally or abnormally) it receives a message of type `Exit` (see below). Therefore, actor `b` is going to be notified whenever actor `a` terminates (assuming that actor `a` did not terminate before `b`'s invocation of `link`).
Listing 6.3 makes this more concrete by having actor `a` throw an exception. The exception causes `a` to terminate, resulting in an `Exit` message to actor `b`. Running it produces the following output:

```
Actor 'a' terminated because of UncaughtException(...)
```

```
val a = actor {
  react {
    case 'start =>
      val somethingBadHappened = true
      if (somethingBadHappened)
        throw new Exception("Error!
      println("Nothing bad happened"))
  }
}
val b = actor {
  self.trapExit = true
  link(a)
  a ! 'start
  react {
    case Exit(((from,reason) if from == a =>
      println "Actor 'a' terminated because of " + reason)
  }
}
```

Listing 6.3 · Receiving a notification because of an unhandled exception.

`Exit` is a case class with the following parameters:

```
case class Exit(from: AbstractActor, reason: AnyRef)
```

The first parameter tells us which actor has terminated. In Listing 6.3 actor `b` uses a guard in the message pattern to only react to `Exit` messages indicating that actor `a` has terminated. The second parameter of the `Exit` case class indicates the reason why actor `from` has terminated.

The termination of a linked actor caused by some unhandled exception results in an `Exit` message where `reason` is equal to an instance of `UncaughtException`; it is a case class with the following fields:

`actor: Actor`: the actor that threw the uncaught exception;

`message: Option[Any]`: the (optional) message the actor was processing; `None` if the actor did not receive a message;

`sender: Option[OutputChannel[Any]]`: the (optional) sender of the most recently processed message;

`cause: Throwable`: the uncaught exception that caused the actor to terminate.

- `actor: Actor`: the actor that threw the uncaught exception;
- `message: Option[Any]`: the (optional) message the actor was processing; `None` if the actor did not receive a message;
- `sender: Option[OutputChannel[Any]]`: the (optional) sender of the most recently processed message;
- `cause: Throwable`: the uncaught exception that caused the actor to terminate.

Since `UncaughtException` is a case class, it can be matched against when receiving an `Exit` message. For instance, in Listing 6.3 we can extract the exception that caused actor `a` to terminate directly from the `Exit` message:

```
react {
  case Exit(((fromUncaughtException(_, _, _, _, cause))))
    if from == a =>
    println "Actor 'a' terminated because of " + cause
}
```

Running Listing 6.3 with the above change results in the following output:

```
Actor 'a' terminated because of java.lang.Exception: Error!
```

When the `trapExit` member of an actor is `true`, the actor is also notified when a linked actor terminates normally, for instance, when it finishes the execution of its body. In this case, the `Exit` message's `reason` field is the `Symbol 'normal`.[1] You can try this out yourself by changing the local variable `somethingBadHappened` to `false`. The output of running the code should then look like this:

```
Nottthingbad happeneeeddd
Ac or 'a' terminat   because of 'normal
```

[1] In Scala, `Symbol`s are similar to strings, except that they are always interned, which makes equality checks fast. Also, the syntax for creating `Symbol`s is slightly more lightweight compared to strings.

Restarting crashed actors

In some cases it is useful to restart an actor that has terminated because of an unhandled exception. By resetting (parts of) the state of a crashed actor, chances are that the actor can successfully process outstanding messages in its mailbox. Alternatively, upon restart the outstanding messages could be retrieved from the crashed actor's mailbox and forwarded to a healthy actor.

```
// assumes ‘self‘ linked to ‘patient‘ and ‘self.trapExit == true‘
def keepAlive(((patient:Actor): Nothing = {
  react {
    case Exit from, reason) if from == patient =>
      if (reason != 'normal) {
        link(patieeenttt)
        patient.r s art()
        keepAlive(patient)
      }
  }
}
```

Listing 6.4 · Monitoring and restarting an actor using `link` and `restart`.

Listing 6.4 shows how to create a keep-alive actor that monitors another actor, restarting it whenever it crashes. The idea is that the keep-alive actor first links itself to the monitored actor (the `patient`), and then invokes `keeeepAlive`. The `keepAlive` method works as follows. When receiving an `Exit` message indicating the abnormal termination of `patient` (in this case, `r ason != 'normal`), we re-link `self` to `patient` and restart it. Finally, `keepAlive` invokes itself recursively to continue monitoring the `patient`.

You may wonder why we link `self` to the `patient` actor before restarting it. After all, `keepAlive` assumes that this link already exists. The reason is that `self` automatically unlinks itself when receiving an `Exit` message from `patient`. This is done to avoid leaking memory through links that are never removed. Since in most cases terminated actors are not restarted, this behavior is a good default.

Listing 6.5 shows how to use our `keepAlive` method to automatically restart an actor whenever it crashes. Actor `crasher` is the actor that we want to monitor and restart. It maintains a counter such that whenever the counter is even, handling a `'request` message results in an exception being thrown.

```
val crasher = actor {
  println("I'm (re-)born")
  var cnt = 0
  loop {
    cnt += 1
    react {
      case 'request =>
        println("I try ttt servicccca request")
        if (cnnnt%t2 == 0) {
          pri  ln("some imes I  rash...")
          throw new Exception
        }
      case 'stop =>
        exit()
    }
  }
}

val client = actor {
  react {
    case 'start =>
      for (_ <- 1 to 6) { crasher ! 'request }
      crasher ! 'stop
  }
}

actor {
  self.tttrrraaapExtrue
  link(c  sherrr)
  clien  ! 'start
  keepAlive(c asher)
}
```

Listing 6.5 · Using `keepAlive` to automatically restart a crashed actor.

Since the exception is not handled, it causes the actor to crash. We can also tell the `crasher` to stop, thereby terminating it normally. The `client` actor waits for a `'start` message, and then sends a number of requests to `crasher`, some of which cause crashes.

The last actor, the keep-alive actor, links itself to the `crasher` with `t pExit` set to `true`. It is important that the keep-alive actor links itself to the `crasher` before the client starts. Otherwise, the client could cause the `crrraaasher` terminate without sending an `Exit` message to the keep-alive actor; since the `Exit` message would never be received, the `crasher` actor would not be restarted. Running the code in Listing 6.5 produces the following output:

```
I'm (re-)borrrn
  tttrrriiiccceeeeqqquuueeessstttt

sometimes I crash...
I'm (re-)borrrn
  tttrrriiiccceeeeqqquuueeessstttt

sometimes I crash...
I'm (re-)borrrn
  tttrrriiiccceeeeqqquuueeessstttt

sometimes I crash...
I'm (re-)born
```

As you can see, the `crasher` actor processes six `'request` messages. Every second message results in a crash, causing the keep-alive actor to restart it. Restarting the `crasher` re-runs its body, producing a rebirth message.

Exception handling using futures

One advantage of futures over simple asynchronous messages is that they make it easy to identify to which request they correspond. Basically, each future is a representation of the asynchronous request that created the future. We can leverage this property of futures for exception handling. Let's revisit the image downloader example of chapter 5. In the following we will show how you can extend Listing 5.8 to handle exceptions that may be thrown during the retrieval of the images (for instance, `IOExceptions`).

```
def renderImage (url: String) {
  vvvaaariiiaaabbbleeInfosss= scanForImageInfo(url)
  self.traaapExit= true
      dat Futures = for (info <- imageInfos) yield {
    val loader = link {
      react { case Download(((info)=>
        throw new Exception "no connection")
        reply(info.downloadImage())
      }: Unit
    }
    loader !! Download(info)
  }
  var i = 0
  loopWhile (i < imaaageInfosss.size(e)
    i += 1
    val Input = dat Futttture(i-1).inputChannel
    react {
      cccaaaIIIsssuuuee! (((dddaaaImageData(_)) =>
        renderImage   t )
           Exit(((fromUncaughtException(___,Some(Download(info)),
                                          _, cause)) =>
        println """Couldn't download image "+info+
                  because of "+cause)
    }
  }
}
```

Listing 6.6 · Reacting to `Exit` messages for exception handling.

Listing 6.6 shows the `renderImages` method extended with code to handle uncaught exceptions in the downloader actors. The idea is as follows. First, the actor that renders the images sets its `trapExit` member to `true`, which enables it to receive termination notifications from linked actors. Second, the renderer actor links itself to each downloader actor. For this, we use one of the `link` methods defined in the `Actor` object. The variant we use takes a code block (more precisely, a by-name parameter of type `=> Unit`) as an argument, creates a new actor to execute that block, and links the caller to the newly created actor. Importantly, linking and starting the new actor is done in a single, atomic operation to avoid a subtle race condition: between starting the new actor and linking to it, the newly created actor could already have died, which would result in a lost `Exit` message. This is the main reason why the `Actor` object provides a `link` method that takes a code block as an argument. Note that you have to add an explicit type annotation to the `react` expression. The reason is that the return type of `react` is `Nothing` which is compatible with both `link` methods, since `Nothing` is a subtype of every other type. By adding the `: Unit` *type ascription*, we force the compiler to select the `link` method that takes a code block.

After the renderer actor has sent out all download requests, it loops trying to receive `ImageData` objects from each future's input channel. To handle uncaught exceptions in the downloader actors, the renderer also reacts to `Exiiit`messages: whenever an `Exit` message that has an `UncaughtExcept on` as its reason is received, we extract the message that the terminated actor was processing using a nested pattern match. This enables us to easily get access to the corresponding `ImageInfo`, since it was passed as part of the `Download` message.

Chapter 7

Customizing Actor Execution

Actors are executed using a run-time system which is backed by an efficient task execution framework. You have seen that such a run-time system enables concurrent programs to scale to a large number of fairly lightweight actors. In this chapter you will learn how to customize the run-time system, to improve the integration with threads and thread-local data, to simplify testing, and more (Section 7.1).

In chapter 5 you have seen that event-based actors require only a small number of worker threads for their execution. However, when actors use blocking operations, the number of worker threads often must be increased to avoid locking up the thread pool. *Managed blocking* provides a way to automatically adjust the thread pool size depending on the blocking behavior of operations. In this chapter you will learn how to use managed blocking to enable a safe use of existing blocking concurrency classes (Section 7.2).

7.1 Pluggable schedulers

In some cases the way in which actors are executed must be customized. There are a number of examples where this is the case:

- Maintaining thread-bound properties such as `ThreadLocals`;
- Interfacing with existing event dispatch threads;
- Daemon-style actors;
- Deterministic execution of message sends/receives for reproducible testing;

- Fine-grained control over resources consumed by the underlying thread pool.

In all these cases it is necessary to customize the way in which actors are executed. The part of the runtime system that is responsible for executing an actor's behavior is called *scheduler*. Each actor is associated with a scheduler object that executes the actor's actions, that is, its body as well as its reactions to received messages. By default, a global scheduler executes all actors on a single thread pool. However, in principle each actor may be executed by its own scheduler.

To customize an actor's scheduler, it suffices to override the `scheduler` method inherited from the `Reactor` trait. The method returns an instance of `IScheduler`, which is used to execute the actor's actions. By returning a custom `IScheduler` instance the default execution mechanism can be overridden. In the following we are going to show how this is done in each of the above cases.

Maintaining thread-bound properties

When an application is run on the JVM, certain properties are maintained on a by-thread basis. Examples for such properties are the context class loader, the access control context, and programmer-defined `ThreadLocals`. In applications that use actors instead of threads, these properties are still useful or maybe even necessary to interoperate with JVM-based libraries and frameworks.

Using `ThreadLocals` or other thread-bound properties is unchanged compared to threads when using thread-based actors. For event-based actors, the situation is slightly more complicated, since the underlying thread that is executing a single event-based actor may change over time. Remember that each time an actor suspends in a `react`, the underlying thread is released. When this actor is resumed it may be executed by a different (pool) thread. Thus, without some additional logic, `ThreadLocals` could change unexpectedly during the execution of an event-based actor, which would be very confusing. In this section we show you how to correctly maintain thread-bound properties, such as `ThreadLocals`, over the lifetime of event-based actors.

Example: thread-local variables

Listing 7.1 shows an attempt to use a `ThreadLocal` to keep track of a name string that is associated with the current actor. The name is stored in a `ThreadLocal[String]` called `tname`. In Java, `ThreadLocals` are typically declared as static class members, since they hold data that is not specific to a class instance, but to an entire thread. In Scala, there are no static class members. Instead, data that is not specific to a class instance is held in singleton `objects`. Object members are translated to static class members in the JVM bytecode. Therefore, we declare the thread-local `tname` as a member of the application's object, as opposed to a member of a class or trait. We override the `initialValue` method to provide the initial value `"john"`. The `joeActor` responds to the first `'YourName` request by first setting its name to `"joe jr."`, and then sending it back in a reply to the `sender`. Upon the second `'YourName` request the thread-local name is sent back unchanged as a reply. The other actor simply sends two requests and prints their responses. We expect the program to produce the following output:

```
yyyoooommmaaarrrjjjeeejjjeeerrr...
```

However, surprisingly, some executions produce the following output:

```
yyyoooommmaaarrrjjjeeejre::
                hn
```

Apparently, in this execution the second `'YourName` request returns the initial value of the `ThreadLocal`, even though it has been set previously by the actor. As already mentioned, the underlying problem is that parts of an event-based actor are not always executed by the same underlying thread. After resuming the second `rrreeeaaacct`, it is possible that the actor is executed by a thread that is different from the thread that executed the reaction to the first request. In that case, the `Th      dLocal` has not been updated, yet, containing the initial value.

The above problem can be avoided as follows. First, we create a subclass of the `Actor` trait that stores a copy of the thread-local variable. The idea is to restore the actual `ThreadLocal` using this copy whenever the actor resumes. Conversely, we save the current value of the `ThreadLocal` to the actor's copy whenever the actor suspends. This way, we make sure that the

```
object ActorWithThreadLocalWrong extends Application {
  val tname = new ThreadLocal[String] {
    override protected def initialValue() = "john"
  }
  val joeActor = actor {
    react { case 'YourName =>
      tname set "joe jr."
      sender ! tname.get
      react { case 'YourName =>
        sender ! tname.get
      }
    }
  }
  actor {
    println("your name: " + (joeActor !? 'YourName))
    println("your name: " + (joeActor !? 'YourName))
  }
}
```

Listing 7.1 · Incorrect use of ThreadLocal.

```
abstract class ActorWithThreadLocal(private var name: String)
  extends Actor {
  override val scheduler = new SchedulerAdapter {
    def execute(block: => Unit): Unit =
      ActorWithThreadLocal.super.scheduler.execute {
        tname set name
        block
        name = tname.get
      }
  }
}
```

Listing 7.2 · Saving and restoring a ThreadLocal.

`ThreadLocal` holds the correct value while the actor executes, namely the value associated with the current actor.

The solution that we just outlined requires us to run custom code upon an actor's suspension and resumption. We can achieve this by overriding the scheduler that is used to execute the actor. However, we only want to override a specific method of the scheduler, namely the method that receives the code to be executed after an actor resumes and before it suspends. Since this is the most common use case when overriding an actor's scheduler, there is a helper trait `SchedulerAdapter`, which allows us to override only this required method.

This implementation is shown in Listing 7.2. The abstract `ActorWithThreadLocal` class overrides the `scheduler` member with a new instance of a `SchedulerAdapter` subclass. This subclass provides an implementation of the `execute` method that receives the code `block`, which is executed after `this` actor resumes and before it suspends. To insert the required code we invoke the `execute` method of the inherited `scheduler`, passing a closure that surrounds the evaluation of the by-name `block` argument with additional code. Before the actor resumes, we restore the thread-local `tname` with the value of the actor's copy in its private `name` member. Normally, after running the code `block`, the actor would suspend. In our extended closure we additionally save the current value of `tname` in the actor's `name` member before we suspend. By making `joeActor` in Listing 7.1 an instance of `ActorWithThreadLocal` its thread-local state is managed correctly.

```
abstract class SwingActor extends Actttor{
  override val scheduleeer = new SchedulerAdapter {
    def executtte(((block: Unit): Uni  =
      java.aw .EventQu ue.invokeLater(new Runnable() {
        def run)  = block
      })
  }
}
```

Listing 7.3 · Executing actors on the Swing event dispatch thread.

Interfacing with event dispatch threads

Some frameworks restrict certain actions to special threads that are managed by the framework. For example, the event queue of Java's Swing class library is managed by a special event dispatch thread. For thread safety, Swing UI components may only be accessed inside event handlers that are executed by this dispatch thread. Therefore, an actor that wants to interact with Swing components must run on the event dispatch thread. This is just one example where it is necessary to "bind" actors to specific threads that are provided by some framework. Another example is a library that interacts with native code through JNI where all accesses must be done from a single JVM thread.

By overriding an actor's scheduler we can ensure that its actions are executed on a specific thread, instead of an arbitrary worker thread of the actor runtime system. For this, we can again use the `SchedulerAdapter` trait, which we have already seen in the previous section. Listing 7.3 shows the implementation of a subclass of `Actor` that executes its instances only on the Swing event dispatch thread. For this, we override the `scheduler` member with a new instance of a `SchedulerAdapter` that executes the actor's actions by submitting `Runnables` to the Swing event dispatch thread. This is done using the `invokeLater` method of `java.awt.EventQueue`.

Daemon-style actors

In many cases, we don't have to care about the termination of an actor-based program: when all actors have finished their execution, the program terminates. However, when actors are long-running or react to messages inside an infinite loop, orderly termination of actors and the underlying thread pool can become challenging.

Some applications use actors that are always ready to accept requests for work to be processed in the background. To simplify termination in such cases, it can help to make those actors daemons: the existence of active daemon actors does not prevent the main program from terminating. This means that as soon as all non-daemon actors have terminated, the application terminates.

Listing 7.4 shows how to create actors with daemon-style semantics. It suffices to override the actor's `scheduler` method to return the `DaemonScheduler` object. `DaemonScheduler` uses the exact same configuration as the default `Scheduler` object, except that the actors that it manages

```
import scala.actors.Actor
import scala.actors.scheduler.DaemonScheduler

object DaemonActors {

  class MyDaemon extends Actor {
    override def scheduler = DaemonScheduler
    def act()
      loop {
        react {{case num: Int => reply(num + 1) }
      }
    }
  }

  def main(args: Arraaay[String]){
    val d = (new MyD emon).start()
    println(d !? 41)
  }
}
```

Listing 7.4 · Creating daemon-style actors.

do not prevent the application from terminating. In Listing 7.4 the `d` actor is again waiting for a message after the synchronous send has been served. However, since the main thread finishes, the `DaemonScheduler` also terminates, and with it the `d` actor.

Deterministic actor execution

By default, the execution of concurrent actors is not deterministic. This means that two actors that are ready to react to a received message may be executed in any order, or, depending on the number of available processor cores, in parallel. Since actors do not share state,[1] you do not have to worry about data races even if the actual execution order is not known in advance. In fact, for best performance and scalability we would like to have as many actors as possible executed in parallel!

[1] Currently this is a mere convention; however, efforts exist to have actor isolation checked using an annotation checker plug-in for the Scala compiler.

However, in some cases it can be very helpful to execute actors deterministically. The main reason is that a deterministic execution enables reproducing program executions that are not influenced by timing-dependent variations in thread scheduling, which make multi-threaded programs extremely hard to test.

```
class Gear(val id: Int, var speed: Int, val controller: Actor)
exteeendsActor {
  df  act() {{{
    loop {
      react
        case SyncGear(targetSpeed: Int) =>
          println("""[Gear"+id+
                  ]  synchronize from current speed "+speed+
                    to target speed "+targetSpeed)
          adjustSpeedTo(targetSpeed)
      }
    }
  }
  def adjustSpeedTo(targeeetSpeedInt) {
    if (targetSpeed > sp ed) {
      ssspeed= 1
       elf ! SyncGear(targetSpeeeed)
    } else if (targetSpeed < sp ed) {
      ssspeed= 1
       elf ! SyncGear(targetSpeed)
    } else if (targetSpeed == speed) {
      priiintln("[Gear "+id+"] has target speed")
      contttroller! SyncDone(this)
      ex  ()
    }
  }
}
```

Listing 7.5 · Synchronizing the speed of `Gear` actors.

For example, consider a concurrent application simulating mechanical gears and motors. Assume that the speed of each gear is adjusted using a

controller. To model the concurrency of the real-world, each gear as well as the controller is represented as an actor. Listing 7.5 shows the implementation of a gear as an actor. The `Gear` actor responds to `SyncGear` messages (not shown for brevity) that cause the gear to adjust its speed in a step-wise manner. For instance, a gear with current speed 7 units requires two steps to adjust its speed to 5 units. Each step is initiated by a `SyncGear` message. To process each message, the gear decrements its speed by a fixed amount and sends itself another `SyncGear` message if it has not reached its target speed. Otherwise, the gear reports to its controller that it has reached the target speed using a `SyncDone` message. Let's write a little driver to test the `Gear` actor:

```
object NonDeterministicGears {
  def main(args: Array[String]) {
    actor {
      val g1 = (new Gear(1, 7, self)).start()
      val g2 = (new Gear(2, 1, self)).start()
      g1 ! SyncGear(5)
      g2 ! SyncGear(5)
      react { case SyncDone(_) =>
        react { case SyncDone(_) => }
      }
    }
  }
}
```

The above driver creates two gears that, initially, are running at speeds 7 and 1, respectively. Afterwards the controller actor instructs the gears to adjust their speed to 5 by sending asynchronous `SyncGear` messages. Finally, it waits until both gears have synchronized their speeds. Running the driver produces output such as the following:

```
[Gear 2] synchronizing from speed 1 to 5
[Gear 1] synchronizing from speed 7 to 5
[Gear 2] synchronizing from speed 2 to 5
[Gear 1] synchronizing from speed 6 to 5
[Gear 2] synchronizing from speed 3 to 5
[Gear 1] has target speed 5
[Gear 1] has target speed
```

```
[Gear 2] synchronize from current speed 4 to target speed 5
[Gear 2] synchronize from current speed 5 to target speed 5
[Gear 2] has target speed
```

As you can see, the speed-adjusting steps of the different gears are interleaved, since the gear actors are running concurrently. A subsequent program run may produce an entirely different interleaving of the steps. However, this means that you could not use the above driver for a unit test that compares the actual output to some expected output.

Using the `SingleThreadedScheduler` (in package `scala.actors.scheduler`) it is possible to make the execution of concurrent actors deterministic. As its name suggests, this scheduler runs the behavior of all actors on a single thread. Since the execution of that thread is deterministic, the entire actor system executes deterministically. In particular, any side effects that actors might do as part of their reaction to messages, such as I/O, is done in the same order in all program runs.

This scheduler works by running the behavior of actors on a single thread. Usually, this means that the reaction of an actor receiving a message is immediately executed on the same thread that has been executing the sender of that message. Since it is a valid pattern to have an actor sending messages to itself in a loop, sometimes the scheduler must delay the processing of a message to avoid a stack overflow. For this, the scheduler maintains a queue of reactions that are executed when there is nothing else left to be done. However, it is possible that some reactions remain in the scheduler's queue just before the application should terminate. Therefore, to make sure that the scheduler processes all tasks, one has to invoke `shutdown` explicitly.

7.2 Managed blocking

The actor run-time system uses a thread pool, which is initialized to use a relatively small number of worker threads. By default, the number of workers used is twice the number of processor cores available to the JVM. In many cases this configuration allows executing actors with a maximum degree of parallelism while consuming only little system resources for the thread pool. In particular, actor programs that use only event-based operations such as `react` can always be executed using a fixed number of worker threads.

However, in some cases actors use a mix of event-based code and thread-based code. For instance, some methods like `receive` are imple-

mented using thread-blocking operations. Moreover, actor-based code may have to interoperate with code using the Java concurrency utilities (the `java.util.concurrent` package). In both cases, operations that may block the underlying thread have to be used with care, so as to avoid locking up the entire thread pool.

```
       scala.actors.Actor._
iiimmmpppjjaaavvvaaa.uuutttiiilll.concurrent.CountDownLatch

objeeecPoolLockup {

  df  main(args: Array[String]) {
    val numCores = Runtime.getRunnntime().availableProcessors()
    println("available cores: "+ umCores)

    val latch = new CountDownLatch(1)
    for (i <- 1 ttto(numCores * 2))actor {
      latch.awai ()
      println("actor "+i+" done"
    }

    actor { latch.countDown() }
  }

}
```

Listing 7.6 · Blocked actors may lock up the thread pool.

For example, Listing 7.6 shows what happens if too many actors are blocked simultaneously. To simplify the demonstration we use the `CountDownLatch` class in the `java.util.concurrent` package. Note that even though the actual code example may not be very useful in and of itself, there are probably places in your actor-based program where the Java concurrency utility classes come in handy. Therefore, the following discussion should be useful to anyone who wants to reuse blocking concurrency code in her actor code. Basically, we use a `CountDownLatch` to notify a bunch of actors once the "main actor" reaches a certain point. To do this, we initialize the latch to one, and tell our actors to wait until the latch becomes zero. Once the main actor sets the latch to zero, the other actors can continue and print a message before terminating. Now, the problem is that if too many actors wait for the latch to become zero, it is possible that all worker threads

in the underlying thread pool are blocked, so that there is no thread left to execute the main actor. As a result, the blocked actors wait indefinitely, the thread pool is locked up, and the program fails to terminate. Note that in the example we took care to start twice as many blocking actors as there are processor cores available to the VM. This corresponds exactly to the number of pool threads created by default. Therefore, starting fewer blocking actors will not cause problems, since there will be a pool thread left to execute the main actor, which releases the blocked actors.

There are several ways to prevent the thread pool from locking up our actor-based program:

- Configure the thread pool to create more worker threads on start up;
- Use *managed blocking* to dynamically resize the thread pool before invoking a blocking operation.

The first alternative can be implemented either (1) using the `actors.corePoolSize` and `actors.maxPoolSize` JVM properties (see Chapter 5), or (2) using a customized scheduler (see Section 7.1). However, pre-configuring the thread pool size can be fragile if the number of blocking actors is hard to predict. Moreover, overprovisioning of thread pool resources is likely to negatively impact the performance of your application.

The second alternative is a much more efficient way of dealing with blocking operations, since the thread pool grows only on demand, and (usually) only for a short period of time. It also avoids the problem of having to predict the maximum number of actors that may be blocked simultaneously. The basic idea of managed blocking is to invoke blocking operations indirectly through an interface that allows the thread pool to resize itself before blocking. Additionally, the interface allows the pool to query a wrapped blocking operation to check whether it no longer needs to block. This enables shrinking the pool back to the size it had before growing to accommodate the blocking operation.

Listing 7.7 shows how you can use the `ManagedBlocker` interface to avoid locking up the thread pool. Managed blocking requires the use of methods that are not accessible when defining actors inline using `actor {...}`. Therefore, you have to create your blocking actors by subclassing the `Actor` trait. Note that inside the body of `act` we replaced the invocation of `latch.await()` with a call to `managedBlock`, a method declared

```
                actors.Ac or
                actors.Ac or._
iiimmmppsssoccaatllaaa...concurrenttt.ManagedBlocker
        java.util.concurrent.CountDownLatch

object ManagedBlocccking{

  class BlockingAAttt(ir Int, latch: CountDownLatch)
        extends  c     {
    val blockeeer = new ManagedBlocker {
      ddeeblfffk() = { latch.aaawait();true }
          isR leasable = { l tch.getCount() == 0 }
    }
    def acttt(){
      scheduler.managedBlock(blocker)
      prin ln("actor "+i+" done")
    }
  }

  def main(args: Array[String]) {
    val numCores = Runtime.getRunnntime().availableProcessors()
    println("available cores: "+ umCores)

    val latch = new CountDownLatch(1)
    for (i <- 1 to (numCores * 2))))
      (new BlockingActor(i, latch ).start()

    actor { latch.countDown() }
  }

}
```

Listing 7.7 · Using managed blocking to prevent locking up the thread pool.

in the `IScheduler` trait. It is invoked on the `scheduler` instance that is used to execute the current actor (`this`). `managedBlock` takes an instance of `ManagedBlocker` as an argument. The `ManagedBlocker` trait is used to wrap blocking operations in a way that allows the underlying thread pool to choose when and how to invoke that operation. The trait contains the following two methods:

- `dddeeffff(): Boolean`
- `isReleasable: Boolean`

The two methods are supposed to be implemented in the following way. The `block` method invokes a method that possibly blocks the current thread. The underlying thread pool makes sure to invoke `block` only in a context where blocking is safe; for instance, if there are no idle worker threads left, it first creates an additional thread that can process submitted tasks in the case all other workers are blocked. The Boolean result indicates whether the current thread might still have to block even after the invocation of `block` has returned. In most cases it is sufficient to just return `true`, which indicates that no additional blocking is necessary. The `isReleasable` method, like `block`, indicates whether additional blocking is necessary. Unlike `block`, it should not invoke possibly blocking operations itself. Moreover, it can (and should) return `true` even if a previous invocation of `block` returned `false`, but blocking is no longer necessary.

The implementations of `block` and `isReleasable` in Listing 7.7 are straightforward. The `block` method simply invokes `latch.await` and returns `true` after that; clearly, once `await` has returned no additional blocking is necessary. In `isReleasable` we use the `getCount` method of `CountDownLatch` to determine whether the call to `await` has already unblocked the thread or not. Running the program extended with managed blocking in this way shows that the pool no longer locks up.

Managed blocking and `receive`

The `receive` method allows actors to receive messages in a thread-based way. This means that `receive` can be used just like any other possibly blocking operation. This is unlike the `react` method, which is more lightweight, but also more restricted. (Chapter 5 shows how to do event-based programming using `react`.) Since `receive` uses standard JVM monitors under the

hood, it has the same potential problems as any other blocking code when invoked from within actors. However, since all variants of `receive` are implemented in objects and types in the `scala.actors` package, it uses managed blocking internally to avoid thread pool lock-ups. Consequently, there is no need to wrap invocations of `receive` in `ManagedBlockers` in user code.

Chapter 8

Remote Actors

Scala actors can communicate with each other not only within the same Java Virtual Machine address space, but also across virtual machines, and even across network nodes. To explain the constructs involved in using remote actors, in this chapter we are revisiting the chat example application of Chapter 4. You are going to learn how to create remote actors, and how to address and communicate between remote actors.

The chat application of Chapter 4 creates an actor that is responsible for managing a chat room. Clients send various types of messages to the chat room actor, such as `Subscribe` and `Unsubscribe` messages. By making the chat room actor remotely accessible, the chat service can be used across a network.

8.1 Creating remote actors

Listing 8.1 shows how to turn the chat room actor into a remote actor. First, the actor runtime system needs to be informed that the actor wants to engage in remote communication with other actors. This is done by invoking the `alive` method of the `RemoteActor` object. It requires specifying a port number which is used to listen for incoming TCP connections. Actors running on different machines in the network use this port number to obtain a remote reference to the chat room actor.

The port number is not enough to uniquely identify the actor, though; several remote actors may be accessible via the same port. Therefore, remote actors must be registered under a name which is unique for a given port number. This is done using the `register` method of `RemoteActor`:

```
import scala.actors.Actor
import Actor._
import scala.actors.remote.RemoteActor.{alive, register}

class ChatRoom extends Actor {
  def act() {
    alive(9000)
    register('chatroom, self)
    while (true) {
      receive {
        cccaaSubseceibe(user) =>        ///hhhaaanmubdsllpetions
             UUUnsubscribe(user) =>               uuunsubscriptions
              serPost(user, post) =>               ser pos s
      }
    }
  }
}
```

Listing 8.1 · Making the chat room actor remotely accessible.

```
def register(name: Symbol, a: Actor): Unit
```

The method expects two arguments: the first argument is the name under which the actor should be registered. Note that names are Symbols, which are similar to strings that are always interned, but with a more lightweight syntax. The second argument is the actor that should be registered, in the example in Listing 8.1 simply self. Subsequently, it is possible to obtain a remote reference to the chat room actor using the port number, the IP address of the machine that the actor is running on, and the name under which it is registered on that machine.

Note that it is possible to change the name under which an actor is registered by repeatedly invoking register, passing different symbols. However, at any point in time an actor is registered under a single name only. The most recent invocation of register "wins."

Messages for remote communication

To communicate with the chat room, messages must be serialized and sent over the network. Therefore, it is necessary that the message classes are

serializable. Fortunately, the message classes defined in Listing 4.1 are case classes which are serializable by default.

8.2 Remote communication

The chat room actor can now receive messages from actors running on different nodes on the network. However, its clients first have to obtain a remote reference to it. This is done using the `select` method of the `RemoteActor` object:

```
def select(node: Node, sym: Symbol): AbstractActor
```

The first argument is a `Node` instance which specifies the IP address and port number of the target node. The second argument is the name of the target actor. Invoking `select` returns an object of type `AbstractActor`. You can think of `AbstractActor` as a trait that contains all the functionality of `Actor` except for methods that are not supported by remote actors, such as `start`, `restart`, and `getState`.

`Node` is a case class defined as follows:

```
case class Node(address: String, port: Int)
```

For example, let's select the chat room actor running on the local node on port 9000:

```
val chatRoom = select(Node("127.0.0.1", 9000), 'chatroom)
```

The `AbstractActor` reference returned by `select` can then be used to communicate with the chat room using the usual message send operations:

```
cchhhaaatttRRRoooomm ! Subscribe(User("Alice"))
          ?  Subscribe(User("Bob")
val future = chatRoom !! Subscribe(User("Charly"))
...
```

Just like with local actors, in all the above cases the remote actor implicitly receives a reference to the sending actor. As before, the remote actor can access the reference via the `sender` method of the `Actor` object. This means that the chat room actor's code to process incoming messages does not have to change.

Selecting an actor using a name that has no actor registered on the target node will *not* cause the `select` invocation to fail (by throwing an exception, say). Instead, the `AbstractActor` reference returned by `select` is *lazy* (or *delayed*) in the sense that no attempt to communicate with the remote node is made at this point. The `select` method merely creates a proxy object which forwards all messages it receives to the remote actor. Sending a message to a remote actor (or rather, its proxy) will result in a lost message if the symbol passed to `select` is not registered with an existing actor on the target node.

Linking to remote actors An actor can link itself to a remote actor just like a local actor. It does not matter whether the receiver of an invocation of `link` is local or (also) remote. Reacting to the termination of linked remote actors is unchanged compared to the non-remote case as discussed in Chapter 6.

8.3 A remote start service

In a distributed application it is often useful to have a parent actor manage several child actors that run on different nodes on the network. Moreover, the number of child actors usually is not fixed when the application (or the parent actor) starts; instead, the parent actor must be able to start child actors dynamically (depending on input data, the state of the application etc.). This design typically requires a service that allows actors to be started remotely. Using such a remote start service a parent actor can start new child actors on nodes different from its own node. In the following we are going to explain how to implement such a remote start service.

We can model our remote start service as a remote actor which responds to the following two types of messages:

```
case class Start(clazz: Class[_ <: Actor])
case object Stop
```

An instance of the `Start` case class should instruct the remote start service to create and start an actor of the type specified by the `clazz` argument. Instances of type `Class[_ <: Actor]` are run-time representations of classes that extend the `Actor` trait. In general, an object of type `Class[A]` can be used to create instances of class `A`.[1] This means that we can use the argument

[1] Java's reflection framework adds the constraint that the instantiated class type must define a no-argument constructor.

of a `Start` message to create `Actor` instances which can then be started on the remote node. The `Stop` case object is a message instructing the remote start service to terminate itself.

```
class Server extends Actor {
  var numStarted = 0
  def act()  {
    aliiive(19000)
    register('serveee, rthis)
    pr ntln("remot  start server running...")
    loop {
      react {
        case Start(clazz) =>
          val aaaActor = clazz.newInstance()
          a.st rt()
          numStarted += 1
          reply()
        case Stop =>
          priiintln("remote start server started " +
                  numStarted + " remote actors")
          ex t()
      }
    }
  }
}
```

Listing 8.2 · A server actor implementing a remote start service.

Listing 8.2 shows a `Server` actor which implements a basic remote start service. Inside the `act` method we use `alive` and `register` to make the actor remotely accessible on port 19000 under the name `'server`. After that the actor loops, reacting to the above `Start` and `Stop` messages. When the next message matches the `Staaart(clazz)` pattern, we create a new instance of the `Actor` subclass that `cl zz` represents by invoking `newInstance`. The object returned by `newInstance` has type `Actor` since the type parameter in the type of `clazz` is constrained to be a subtype of `Actor`. After starting the new actor, the remote start actor replies to the sender of the `Start` message. The reason is that `Start` messages are supposed to be sent synchronously;

this ensures that the new actor has been started when the synchronous message send completes.

```
class EchoActor extends Actor {
  def act() {
    alive(19000)
      gister('echo, this)
    rrreeeact
      case any => reply("echo: " + any)
    }
  }
}
```

Listing 8.3 · An echo actor that can be started remotely.

Listing 8.3 shows an `EchoActor` class that is suitable for remote starting. To be able to interact with new `EchoActor` instances remotely, we use the aaallland register methods like in previous examples. Note that calling twice on the same node with the same argument port number has no effect. This is important, since the `alive` invocation of a new `EchoActor` instance will be called while running on the same node as the `Server` actor. Let's use our remote start service to start a new `EchoActor`:

```
vvvaserver = seeelectNode("localhost", 19000), 'server)
serveeer!? Start(classOf[EchoActor])
     cho ==sel ct(Node("localhost", 1900 ), 'echo)
val resp   echo !? "hellllo"
println("remote start c ient received " + resp)
```

First, we obtain the `server` remote reference to the remote start service using `select`. To start a new instance of `EchoActor` on the node that runs the remote start service, we send a `Start(classOf[EchoActor])` message to the `server`. After that the remotely started actor can be referenced using `select` using the name that the `EchoActor` used to register itself on the remote node. The `echo` reference is a normal remote actor reference; as expected, running the above code results in the following message printed to the console:

```
remote start client received echo: hello
```

Chapter 9

Using Scala Actors with Java APIs

One of Scala's promises to developers is that Scala code can seamlessly invoke Java APIs. That allows you to combine the best of both worlds: make use of thousands of existing Java APIs, but also take advantage of Scala's advanced language features. This chapter will describe, and illustrate with examples, how to use Scala actors with existing Java APIs, including Java EE 6.

Chapter 10

Distributed and Parallel Computing

The Scala Actors API puts at your disposal a powerful, yet simple, parallel computing framework built on top of the JVM. This chapter illustrates how to accomplish some common parallel and distributed computing tasks with actors. In particular, we are focusing on two patterns that are useful in many applications: MAPREDUCE, and reliable broadcasting. MAPREDUCE is a paradigm for parallel and distributed programming which has been established as a de facto standard to accomplish a wide variety of tasks, such as (hypertext) document processing, machine learning, and data mining. Reliable broadcasting, on the other hand, is often necessary in distributed applications where machines in a cluster can fail due to hardware outages or communication delays.

10.1 MapReduce

MAPREDUCE is a parallel computing framework originally developed at Google to simplify programming large-scale distributed computations while providing fault tolerance and excellent scalability.[1] MAPREDUCE simplifies parallel programming, since the programmer does not have to manage parallelism explicitly. Instead, the MAPREDUCE framework is taking care of creating parallel tasks, synchronizing them, and distributing the work load. Moreover, a MAPREDUCE implementation typically also provides fault tolerance. This means that it is possible to successfully complete a MAPRE-

[1] Dean and Ghemawat, "MapReduce: simplified data processing on large clusters" [Dea08]

DUCE computation even if some of the machines in the cluster fail to compute or to communicate their results.

MapReduce history Why was MAPREDUCE invented at Google? Jeffrey Dean and Sanjay Ghemawat, the Google engineers that invented MAPREDUCE, recount that the abstraction emerged after they had written hundreds of special-purpose computations to process large amounts of raw data, such as crawled web pages, web server logs, etc. While the computations performed on the data were simple, the input data was so large that the computation had to be performed in parallel if it was to finish within a reasonable amount of time.

Google's data centers do not consist of large, expensive supercomputers. Instead, they are populated with large clusters of inexpensive commodity hardware, typically Linux desktop machines connected via an ethernet network. Computations thus needed to be parallelised for a distributed environment in which network bandwidth is scarce and machine failures are very common.

As a result, the simplicity of the computations was lost in the complexity of recurring issues such as how to distribute the data, how to parallelise the computation across machines, how to deal with load imbalance, machine failures and so on. Inspired by the *map* and *reduce* higher-order functions from functional programming, Dean and Ghemawat identified a way to separate out the computation-specific parts into higher-order functions. The programmer supplies just these functions, and the MAPREDUCE framework calls them on an appropriate machine, with part of the input data, hiding most of the complexity of the parallel and distributed computing environment. MAPREDUCE truly is a success story based on the principles of higher-order functional programming.

Let's take a look at a concrete example which is amenable to parallel processing. Consider the task of building an inverted index for a collection of text files. An inverted index can be used to quickly look up the (list of) files in which a given word occurs. To create such an index the mapping from files to their contents must be "inverted"–hence the name "inverted index." Our strategy for building it is as follows. For each file `f` we will create a list that contains pairs `(word, f)` where `word` is a word occurring in `f`. This means that each of these lists contains pairs that all have the same second component–the file in which the word occurs. In the next step, we will go

through all of the lists, and fill a `Map` that maps words to lists of files in which they occur; the file lists should not contain duplicates. The reason why we first create lists of word/file pairs (instead of directly building the `Map`) is that this allows us to parallelize the task: each file can be processed in parallel to create the intermediate lists.

```
def invertedIndex(input: List[(File, List[String])]) = {
  val master = self
    val worker = for ((file, words) <- input) yield
    actor {
      val wordsAndFiles = for (word <- words) yield (word, file)
      master ! Intermediate(wordsAndFiles)
    }

  var intermediates = List[(String, File)]()
  for (_ <- 1 to input.length)
    receive {
      case Intermediate(list)  => intermediates :::= list
    }

  var dict = Map[String, List[File]]() withDefault (k => List())
  for ((word, file) <- intermediates)
    dict += (word -> (file :: dict(word)))

  var result = Map[String, List[File]]()
  for ((word, files) <- dict) yield
    result += (word -> files.distinct)

  result
}
```

Listing 10.1 · A function for building an inverted index.

Listing 10.1 shows a parallel implementation using actors. For each file in the input list, we create a worker actor. A worker generates the list of word/file pairs, and sends them in an `Intermediate` message back to the `master` actor (`master` is the actor invoking the `invertedIndex` method.) The `Intermediate` class is defined as follows:

```
case class Intermediate(list: List[(String, File)])
```

The master actor concatenates all the intermediate results it receives which yields a list of type `List[(String, File)]`. The word/file pairs in this list are inserted into a `Map`, so that each word gets mapped to a list of files in which it occurs. Finally, we remove duplicates from these file lists, yielding the `result` map which represents our inverted index (the inferred result type of the `invertedIndex` method is `Map[String, List[File]]`.)

The parallel construction of our inverted index follows a certain pattern which–as it turns out–is useful for many applications. Let's walk through that pattern step-by-step. In the first step, a function, let's call it the `mapping` function, is applied to each pair of the input list in order to produce another list of pairs.

In the example, we generate for each input pair of type `(File, List[String])` a list of pairs of type `(String, File)`–each pair associates a word with the file in which it occurs. We can encapsulate just this computation in the following function:

```
def mapIndex(file: File, words: List[String]) =
  for (word <- words) yield (word, file)
```

(The inferred result type of `mapIndex` is `List[(String, File)]`.) The `mapping` function (`mapIndex` or some other function) is applied in parallel by creating a worker actor for each input pair:

```
val worker = for ((key, value) <- input) yield actor {
  master ! Intermediate(mapping(key, value))
}
```

Note that we changed the code to call the (generic) `mapping` function to produce the list of intermediate pairs. Moreover, we renamed the components of an input pair, replacing `(file, words)` by `(key, value)`. The reason is that besides factoring out the `mapping` function, this "mapping stage" can be generalized further to operate on arbitrary inputs of type `List[(K, V)]`. The type of the intermediate output pairs can be generic, too. Instead of always producing a `List[(String, File)]`, a `mapping` function may produce a `List[(K2, V2)]`. Taken together, the generic `mapping` function has type `(K, V) => List[(K2, V2)]`.

The next step is to collect the intermediate results sent to the `master` actor in `Intermediate` messages:

```
var intermediates = List[(K2, V2)]()
for (_ <- 1 to input.length)
  receive {
    case Intermediate(list) => intermediates :::= list
  }
```

After the `mapping` function has been applied to each input pair (a file paired with its contents), we group the intermediate output pairs by their first component, inserting them into a `Map`:

```
varrdict = Map[K2, List[V2]]() withDefault (k => List())
fo  ((key, value) <- intermedddiiiatttes)
  dict += (key -> (value ::   c (key)))
```

Subsequently, the intermediate results which have been grouped by their key are further processed using a function–let's call it the `reducing` function–to yield the final result.

In the example, the list of files corresponding to each word is reduced by removing duplicates. Again, it is easy to define a function which encapsulates just this reduction step:

```
def reeeduceIndex(keyString, files: List[File]) =
  fil s.distinct
```

The `reducing` function is applied to each entry in the `dict` map of intermediate results, yielding the `result` map:

```
varrresult = Map[K2, List[V2]]()
fo  ((key, value) <- dict)
  result += (key -> reducing(key, value))
```

According to its use in the above reduction step, the `reducing` function has type `(K2, List[V2]) => List[V2]`.

Listing 10.2 shows a generic function, called `mapreduce`, which implements the parallel programming pattern that we just described. As its name suggests, it is a basic (in-memory) MAPREDUCE implementation.

Note that we have moved the declaration of the `Intermediate` case class into the method body. The reason is that this allows us to recover more type information when pattern matching on `Intermediate` messages. Let's see what happens if `Intermediate` is defined outside the `mapreduceBasic` method, like this:

```
def mapreduceBasic[K, V, K2, V2](
  inpppuuuttt: List[(K, V)],
  ma ping: (K, V) => List[(K2, V2)],
  red cing: (K2, List[V2]) => List[V2]
): Map[K2, List[V2]] = {
  caaasse class Intermediate(list: List[(K2, V2)])

  vvvl masteeer = self
  a   wooorrrk = for ((key, value) <- input) yield
    act   {
      master ! Intermediate(mapping(key, value))
    }

  varr intermediates = List[(K2, V2)]()
  fo  (_ <- 1 to input.lengttth)
    receive {
      case Intermediate(lis)  => intermediates :::= list
    }

  varr dict = Map[K2, List[V2]]() withDefault (k => List())
  fo  ((key, value) <- intermedddiiiatttes)
    dict += (key -> (value ::   c (key)))

  varr result = Map[K2, List[V2]]()
  fo  ((key, value) <- dict)
    result += (key -> reducing(key, value))
  result
}
```

Listing 10.2 · A basic MAPREDUCE implementation.

```
case class Intermediate[K2, V2](list: List[(K2, V2)])
```

Now, when pattern matching against an instance of `Intermediate`, an instantiation of the type parameters has to be found. When matching against `Intermediate` messages received inside `mapreduceBasic`, `K2` and `V2` are both instantiated to `Any`. As a result, we get the following type error:

```
...: errrrortype mismatch;
 found   ::LLLiiissstttAny)](Any,
 requi ed        K2, V2)]
        case Intermediate(list) => intermediates :::= list

one error found
```

By moving the definition of the `Intermediate` class into the method body, the type of its `list` argument is `List[(K2, V2)]` where `K2` and `V2` are no longer generic, but fixed to the type arguments of the enclosing method. Then, in the pattern match, `list` has type `List[(K2, V2)]` which makes it compatible with the type of the `intermediates` list.

Parallel reductions The basic MAPREDUCE implementation shown above can be improved by parallelizing the reduction stage in addition to the mapping stage. Listing 10.3 shows how to modify our previous implementation to apply the `reducing` function in parallel. Similar to the mapping stage, we create an actor for each key in the `dict` map. The actor applies the `reducing` function to the key and the values with which the key is associated in `dict`. The result is sent to the `master` actor in a message of type `Reduced`. The `master` collects these messages like the `Intermediate` messages before, this time yielding the final `result` map.

Now that we have parallelized the reduction stage, let us summarize the basic execution model of MAPREDUCE:

- The MAPREDUCE computation is supervised by a single actor, called the *master*.
- Input data is represented as a list of "records", represented here as pairs of type `(K, V)`. The master partitions the input data across a set of "mapper" workers.

```
def mapreduce[K, V, K2, V2](
  input: List[(K, V)],
  mapping: (K, V) => List[(K2, V2)],
  reducing: (K2, List[V2]) => List[V2]
): Map[K2, List[V2]] = {
  case class Intermediate(list: List[(K2, V2)])
  case class Reduced(key: K2, values: List[V2])

  // ...

  val reducers = for ((key, values) <- dict) yield
    actor {
      master ! Reduced      red cing(key, values))
    }

  var result = Map[K2, List[V2]]()
  fo  (_ <- 1 to dict.size)
    rec ive {
      case Reduced(key, values) =>
        result +=       -> values)
    }

  result
}
```

Listing 10.3 · Applying the `reducing` function in parallel.

- Each mapper worker is a separate actor that applies the `mapping` function to a different part of the input data in parallel. For each input pair `(k, v)`, a mapper generates a new list of pairs, which may be of a different type `(K2, V2)`.

- The master collects the intermediate results of the mapping stage and then sorts this data according to the intermediate key type `K2`.

- For each such key, the master asks a "reducer" worker to reduce the list of values of type `V2`. Again, each reducer is a separate actor that can perform this step in parallel. Reducers may return a `List[V2]`, but it is not uncommon for this list to contain a single, reduced, value.

- The master collects the reduced `V2` values and simply combines them in a result `Map`.

Figure 10.1 illustrates the flow of data between master, mapper and reducer actors for the inverted index example.

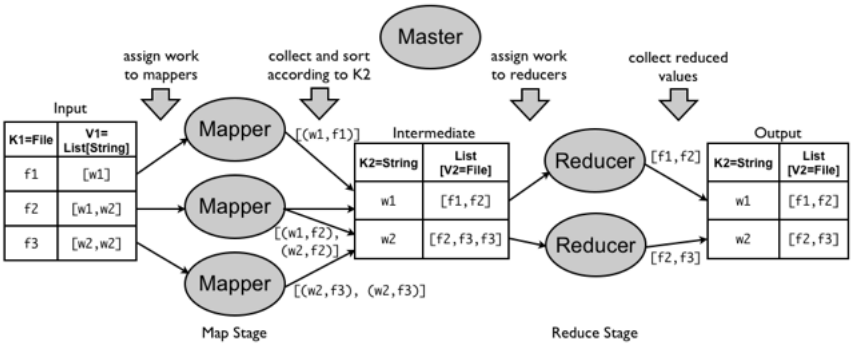

Figure 10.1 · Dataflow in a basic MapReduce implementation.

Fault-tolerance Our basic MapReduce implementation is not fault-tolerant, that is: it cannot tolerate the failures of either the master actor or any of the mapper or reducer worker actors. If a worker crashes, the master will block waiting for a reply indefinitely.

In our example implementation, since we are assuming a shared-memory environment, chances are that a system failure will bring all of the actors to a halt. However, in a distributed MapReduce implementation, the master and workers execute on different machines. In such an environment, partial failure, e.g. a single machine failure, can be common. Let us therefore extend our sample implementation to at least tolerate worker failures.

We will use the mechanisms described in section 6.2 to implement fault tolerance: the master actor will link itself to all of the workers it spawns, and is configured to trap exits (using `self.trapExit = true`). This causes the worker actors to send a special `Exit` message to the master when they terminate, allowing the master to identify crashed workers.

Listing 10.4 extends the basic MapReduce implementation of Listing 10.2 with this fault-tolerance mechanism. Notice that the mapper actors are spawned using `link` instead of `actor` such that they are automatically linked to the master actor.

```
def mapreduce[K, V, K2, V2](
  input: List[(K, V)],
  mapping: (K, V) => List[(K2, V2)],
  reducing: (K2, List[V2]) => List[V2]
): Map[K2, List[V2]] = {
  cccaaasssseeeIIInnnttteeerrrmmmeeedddiiiaaattteee(list: List[(K2, V2)]))
            Reduced(key: K2, values: List[V2

  vvvaaarrr mmmaaasssttteeerr= self
  self.trapExit = true
  r   as ignedMappers = Map[Actor, (K, V)]()

  def spawnMapper(key: K, value: V) = {
    val mapper = link {
      masteeer! Intermediate(mapping(key, value)))
    }
    assign dMappers += (mapper -> (key, value)
    mapper
  }

  for ((key, valueee)<- input)
    spawnMapper(k y, value)

  vvvaaarrr iiinnnttteeerrrmmmeeddiiiaaattteeesss= List[(K2, V2)]()
      nleft =  nput.length
  while (nleft > 0)
    receive {
      cccaaaIIInnnttteeerrrmmmeeedddiiiaaattteee(list))=>>>iiinnnttteeerrrmmmeeedddiiiaaattteeesss= list
          EEExxxiiitttrrreeeaaasssooonnn(((fffrrroooommm),,=>  l ft -= 1
                    reason
        ////rrreeetttrrriiieeevvveee assigned worrrk
        val (key, value) = assignedMappers(from)
          spawn nw  worker to  e-execute the work
        spawnMapper(key, value)
    }

  // ...
```

Listing 10.4 · A MAPREDUCE implementation that tolerates mapper faults.

For each spawned worker, the master also keeps track of what key-value pair it assigned to that worker. When a worker terminates with an abnormal reason, the master looks up what pair it assigned to the terminated worker and spawns a new worker to process the same input. When a worker terminates with a `'normal` reason, the master decrements a count identifying the number of outstanding jobs. When that number becomes zero, the master knows that all workers (either the original or restarted ones) have finished.

Our technique of simply re-executing a failed mapping is sound as long as the `mapping` function is really a function, i.e. if it has no side-effects. Otherwise, failures may affect the outcome of a MAPREDUCE computation. As Scala provides the right building blocks to program in a functional style, restricting oneself to a purely functional subset to implement the mapping and reducing functions is usually not a problem.

Another point worth mentioning is that this simple re-execution strategy does not cope with deterministic errors. Say, for example, that there is a bug in the `mapping` function that only manifests itself for particular values of type `K`. The bug may cause an exception, terminating the worker. In this case, simply starting a new worker to process the same input will cause the exception again, leading to endless re-execution, and no progress for the master. Actual MAPREDUCE implementations deal with such cases by skipping over such "bad" input data, giving up after a few retries. This makes sense, since MAPREDUCE is typically used for workloads where the loss of a little input data can still lead to a useful answer (e.g., indexing, search, data mining, ...).

Note that even in this extended implementation, the master is still a single point of failure: if it crashes, the entire MAPREDUCE computation is brought to a halt. If the chance of a master failure is small and the MAPREDUCE computation is not too large, it may not be worth dealing with this case. Otherwise, one has to either replicate the master, or to periodically checkpoint the state of the master to persistent storage, but these techniques are beyond what we can address in this chapter.

Coarse-grained worker tasks In our previous implementations of MAPREDUCE, a new mapper actor is spawned for each input key `K` and a new reducer actor is spawned for each intermediate key `K2`. When the input data is large (e.g., when indexing thousands of files each containing thousands of words), this simple strategy may introduce too much overhead. To

reduce this overhead, we can make tasks assigned to the mapper and reducer workers more "coarse-grained" by having each of them process multiple key-value pairs.

Listing 10.5 builds on the MAPREDUCE implementation from Listing 10.3 and adds support for coarse-grained tasks. The coarse-grained implementation takes two extra arguments: `numMappers`, the number of parallel mapper workers to spawn, and `numReducers`, the number of parallel reducer workers to spawn.

The magic of this implementation is hidden inside a useful function from the Scala `Seq` API: `grouped`. Calling `grouped(n)` on a sequence returns a new sequence that returns elements from the original sequence, grouped in groups of size `n`. To create more coarse-grained tasks, we split the input and the intermediate data into groups of an appropriate size, and then spawn a worker actor per group. Each such worker processes all key-value pairs in its assigned group.

To ensure that we spawn only `numMappers` mappers, it suffices to group the input data into groups of size `input.length / numMappers`. For example, if `numMappers` is 10 and we need to process 5000 files, then each mapper will need to process 500 pairs. Similarly, we partition the dictionary `dict` containing the sorted intermediate data into groups of size `dict.size / numReducers`. If the size of the data is not exactly divisible by the required number of workers, the last group returned by `grouped` will contain less elements, so the last worker will get assigned less work.

Our simple strategy of dividing work equally among the workers is fine as long as the amount of processing to be done is approximately the same for all keys. Actual MAPREDUCE implementations will often employ more elaborate data distribution techniques to balance the load between the workers at run time if the work is not evenly distributed.

10.2 Reliable broadcast

When building a distributed application it is often necessary that more than two actors operate in a coordinated manner. For example, you may want to ensure that all of your remote actors carry out some action, or none of them, while providing a way to handle the error case. For example, if a set of remote actors should save their internal state to a data base, then typically you want all of them to do it, or none, so that there is always a consistent

```
def coarseMapReduce[K, V, K2, V2](
  input: List[(K, V)],
  mapping: (K, V) => List[(K2, V2)],
  reducing: (K2, List[V2]) => List[V2],
  numMappers: Int, numReducers  Int)  Map[K,  List[V2]] = {
  cccaaassslddeaatssssesdiate(list::List[(K2, V222)])]
             Reduced(key: K2, values::List[V2

  val master = self
  for (grrroup<- input.grouped(input.length / numMappers))
    acto  {
      for ((key, value) <- group)
        master ! Intermediate(mapping(key, value))
    }

  vvvaaintermediates = ...

      dict = Map[K2, List[V2]]() withDefault (k => List())
  fffoooor((key,value) <- inteeermedddiiiatttes)
    dict += (key -> (valuuue :   c (key)))
      grrroup<- dict.grop  d(dict.size / numReducers))
    acto  {
      for ((key, values) <- group)
        master ! Reduced(key, reducing(key, values))
    }

  varrreeesult= Map[K2, List[V2]]()
  fo  (_ <- 1 to dict.size)
    rec ive {
      case Reduced(((kkkeeyvalues) =>
        result +=       -> values)
    }
  result
}
```

Listing 10.5: A MapReduce implementation with coarse-grained worker tasks.

view persisted to the data base.

```
abstract class BroadcastActor extends Actor {
  // can be set by external actor, therefore @volatile
  @volatile var isBroken = false
  private var canRun = true
    ivate var counter = 0L

  ppprrrotected def broadcasttt(mBSend) = if (!isBroken) {
    for (a <- m.recipien s) a ! BDeliver(m.data)
  } else if (canRun) {
    canRun = false // simulate it being broken
    for (a <- m.recipients.take(2)) a ! BDeliver(m.data)
    println("error at " + this)
  }

  // to be overridden in subtraits
  protected def reaction: PartialFunction[Any, Unit] = {
    case BCast(msg, receiverrrs)=>
      counter +==1
      broadcast(BSend(msg,  eceivers, counter))
    case 'stop  >
      exit()
  }

  def act = loopWhile (canRun) { react(reaction) }
}
```

Listing 10.6 · Best-effort broadcasting.

Basic broadcasting

Instructing a number of actors to carry out some action can be done by *broadcasting* a message to these actors. Listing 10.6 shows a simple broadcast implementation. First, we define a `BroadcastActor` which implements `actin` in a way that allows reacting to messages of type `BCast` and the special `'s op` message which causes the actor to terminate. The `BCast` case class is defined as follows:

```
case class BCast(data: Any, recipients: Set[Actor])
```

A `BCast` message tells the actor to send some data to a set of actors specified in the message. Note how the message handlers are defined using the `reaction` member which is a partial function that is passed to `react` inside the `act` method. This has the advantage that subclasses can override `reaction` to handle additional message patterns while inheriting some of the message handling logic from the supertrait. The `broadcast` method implements the actual message sending. `broadcast` is invoked passing a `BSend` message which contains the data, the set of recipients, and a time stamp (initially, we will not make use of the time stamp, though):

```
case class BSend(data: Any, recipients: Set[Actor], timestamp: Long)
```

To make things more interesting, we allow an actor to be "broken" which is expressed using the volatile `isBroken` field (the field is volatile to safely allow changing its value from a different actor.) A broken actor fails to send the message to all of the recipients. The actual data is wrapped in a message of type `BDeliver` which is defined as follows:

```
case class BDeliver(data: Any)
```

A `BDeliver` message indicates to the recipient that the data was delivered using a broadcast.

```
class MyActor extends BroadcastActor {
  override def reaction ==super.reaction orElse {
    case BDeliver(data)  >
      println("Received broadcast message: " +
              data + " at " + this)
  }
}
```

Listing 10.7 · Using the broadcast implementation in user code.

To use the broadcast implementation in actual user code, we extend the `BroadcastActor` and override its `reaction` member as shown in Listing 10.7. The message handling logic of `BroadcastActor` must be enabled alongside the new handler for `BDeliver` messages. To do this, we combine `super.reaction` with the new handler using `orElse`. As a result, `MyActor` instances respond to `Broadcast` and `'stop` messages as defined in `BroadcastActor` in addition to `BDeliver` messages.

Let's try out this basic broadcast implementation:

```
val a1 = new MyActor; a1.start()
val a2 = new MyActor; a2.start()
val a3 = new MyActor; a3.start()
val a4 = new MyActor; a4.start()
a1 ! Broadcast("Hello!", Set(a1, a2, a3, a4))
```

As expected, running the above code will produce output like the following:

```
RReeeccc                                                    0003c3c9217
                                                            15af33d666
                                                            54520eb
Received broadcast message: Hello! at MyActor@2c9b42e
```

However, let's try and set actor `a1`'s `isBroken` field to `true` and initiate another broadcast:

```
aaa111.isBroken = true
   ! Broadcast("Hello again!", Set(a1, a2, a3, a4))
```

Then, the output will look differently:

```
error at MyActor@15af33d6
Received broadcast message: Hello again! at MyActor@2c9b42e6
```

In the above run, only one other actor besides `a1` itself received the "Hello again!" message, because `a1` failed before sending out more messages. In an actual distributed application the reason could be a machine failure or a network link which is down. To guarantee that all recipients receive the broadcast message or none, it is necessary to implement a *reliable broadcast*.

Reliable broadcasting

To make the message broadcasting reliable we will extend our code to implement an algorithm known as "eager reliable broadcast." The idea of this algorithm is that every recipient of a broadcast message should forward that message to every other recipient. Since the forwarding should be done regardless of any failure, broadcasting is "eager."

We add this layer of reliability as follows. First, we extend the `BroadcastActor` as shown in Listing 10.8. Using the `delivered` set, an

```
class RbActor extends BroadcastActor {
  var delivered = Set[BSend]()
  override def reaction = super.reaction orElse {
    case m @ BSend(data, _, _) =>
      if (!delivered.contains(m)) {
        delivered += m
        broadcast(m)
        this ! BDeliver(data)
      }
  }
}
```

Listing 10.8 · A reliable broadcast actor.

`RbActor` keeps track of the `BSend` messages that it has received. Whenever it receives one, the actor checks whether it has already processed the message. This is the case if the condition `delivered.contains(m)` is true, since after processing a `BSend` message, it is added to the `deliverrrecet`. However, if the actor receives a fresh `BSend` message, it'll invoke `b oadcast` to send it to all recipient actors. Moreover, it'll send itself a `BDeliver` message, indicating that it received the `data` via a (reliable) broadcast.

It is crucial here that each `BSend` message contains a time stamp. The time stamp is set in the `BroadcastActor` when creating a new `BSend` message as a reaction to receiving a (broadcast-initiating) `BCast` message. The time stamp let's us identify to which broadcast a particular `BSend` message belongs. The messages forwarded by each `RbActor` do not change that time stamp. This way, each actor knows when it has already received a broadcast message, in which case it does not forward it further. However, for the `RbActor` to work properly we need to slightly change the implementation of the `BroadcastActor`. Basically, it is no longer sufficient to send a `BDeliver` message inside the `broadcast` method. The reason is that the `RbActor` needs to receive `BSend` messages, since only those contain time stamps. Therefore, we have to change the `broadcast` method accordingly; this is shown in Listing 10.9.

As you can see, the `BroadcastActor` now simply sends the `BSend` messages to the recipients (`a ! m`), instead of sending a `BDeliver` message which only contains the data (`a ! BDeliver(m.data)`).

```
protected def broadcast(m: BSend) = if (!isBroken) {
  for (a <- m.recipients) a ! m
} else if (canRun) {
  canRun = false // simulate it being broken
  for (a <- m.recipients.take(2)) a ! m
  println("error at " + this)
}
```

Listing 10.9: Sending messages with time stamps inside the `broadcast` method.

Having made these changes, let's re-run our client code. Before we can do that, however, we first have to change `MyActor` to extend `RbActor` instead of `BroadcastActor`:

```
class MyActor extends RbActor {
  override def reaction ==super.reaction orElse {
    case BDeliver(data)  >
      println("Received broadcast message: " +
              data + " at " + this)
  }
}
```

Running our short test code from above (setting `a1.isBroken` to `true`) should now produce output like the following:

```
error at MyActor@28e70e30
RRReeecccedobirrivvoeaddleecsasssHHeeeagggeggaaaaiMMyy!A!@005954864arr
                                                            3c3c9217
                                                            1ff82982
```

As you can see, even though actor `a1` failed after sending the broadcast message to itself and actor `a2`, actors `a3` and `a4` also received the message. The reason is that `a2` sent the `BSend` message it received from `a1` to all its recipients which includes `a3` and `a4`. In fact, it is possible to prove mathematically that the strategy of eager reliable broadcast will deliver a message to all recipients or none, provided the network communication between actors is based on a reliable transport protocol such as TCP which remote actors use by default.

Chapter 11

API Overview

This chapter provides a detailed overview of the API of the `scala.actors` package of Scala 2.8. The organization follows groups of types that logically belong together as well as the trait hierarchy. The focus is on the run-time behavior of the various methods that these traits define, thereby complementing the existing Scaladoc-based API documentation.

11.1 The actor traits `Reactor`, `ReplyReactor`, and `Actor`

Actors can be created based on several traits which form a simple hierarchy: `Actor <: ReplyReactor <: Reactor` (read "`<:`" as "extends"). There are two main reasons to prefer using a simpler trait (that is, a trait further to the right in the hierarchy) instead of a subtrait:

Types For example, the `Reactor` trait has a type parameter which restricts the type of messages that can be received by instances of the trait.

Scalability A `Reactor` (or `ReplyReactor`) maintains fewer instance variables than a `ReplyReactor` (or `Actor`, respectively.) This means that an application scales to a larger number of `Reactors` than `Actors`, say.

Efficiency The communication primitives provided by a trait are more efficient than those provided by its supertrait (if any). For example, message sends and `reacts` are faster between `Reactors` than `Actors`.

The `Reactor` trait

`Reactor` is the super trait of all actor traits. It has a type parameter `Msg` which indicates the type of messages that the actor can receive. Extending the `Reactor` trait allows defining actors with basic capabilities to send and receive messages.

The behavior of a `Reactor` is defined by implementing its `act` method. The `act` method is executed once the `Reactor` is started by invoking `start`, which also returns the `Reactor`. The `start` method is *idempotent* which means that invoking it on an actor that has already been started has no effect.

Invoking the `Reactor`'s `!` method sends a message to the receiver. Sending a message using `!` is asynchronous which means that the sending actor does not wait until the message is received; its execution continues immediately. For example, `a ! msg` sends `msg` to `a`. All actors have a *mailbox* which buffers incoming messages until they are processed.

The `Reactor` trait also defines a `forward` method. This method is inherited from `OutputChannel`. It has the same effect as the `!` method. Subtraits of `Reactor`, in particular the `ReplyReactor` trait, override this method to enable implicit reply destinations (see below).

A `Reactor` receives messages using the `react` method.[1] `react` expects an argument of type `PartialFunction[Msg, Unit]` which defines how messages of type `Msg` are handled once they arrive in the actor's mailbox. In the following example, the current actor waits to receive the string "Hello", and then prints a greeting:

```
react {
  case "Hello" => println("Hi there")
}
```

Invoking `react` never returns. Therefore, any code that should run after a message has been received must be contained inside the partial function that is passed to `react`. For example, two messages can be received in sequence by nesting two invocations of `react`:

```
react {
  case Get(from) =>
```

[1] By default, this method does not show up in the ScalaDoc API documentation, because its visibility is `protected[actors]` . This can be changed by selecting visibility "All" on the ScalaDoc page.

```
      react {
        case Put(x) => from ! x
      }
  }
```

The `Reactor` trait also provides control structures (see 11.2) which simplify programming with `react`.

Termination and execution states

The execution of a `Reactor` terminates when the body of its `act` method has run to completion. A `Reactor` can also terminate itself explicitly using the `exit` method. The return type of `exit` is `Nothing`, because `exit` always throws an exception. This exception is only used internally, and should never be caught.

A terminated `Reactor` can be restarted by invoking its `restart` method. Invoking `restart` on a `Reactor` that has not terminated, yet, throws an `IllegaaalStateException` Restarting a terminated actor causes its `act` method to be rerun.

`Re ctor` defines a method `getState` which returns the actor's current execution state as a member of the `Actor.Statte` enumeration. An actor that has not been started, yet, is in state `Ac or.State.New`. An actor that can run without waiting for a message is in state `AAAccctttooorrr...SSSttt` `Runnable`. An actor that is suspended, waiting for a message is in state `Actor.State.Suspended`. A terminated actor is in state `Terminated`.

Exception handling

The `exceptionHandler` member allows defining an exception handler that is enabled throughout the entire lifetime of a `Reactor`:

```
def exceptionHandler: PartialFunction[Exception, Unit]
```

The returned partial function is used to handle exceptions that are not otherwise handled: whenever an exception propagates out of the body of a `Reactor`'s `act` method, the partial function is applied to that exception, allowing the actor to run clean-up code before it terminates.[2]

[2] Note that the visibility of `exceptionHandler` is `protected`.

Handling exceptions using `exceptionHandler` works well together with the control structures for programming with `react` (see 11.2). Whenever an exception has been handled using the partial function returned by `exceptionHandler`, execution continues with the current continuation closure. Example:

```
loop {
  react {
    case Msg(dddata)=>
      if (con)  // process data
      else throw new Exception("cannot process data")
  }
}
```

Assuming that the `Reactor` overrides `exceptionHandler`, after an exception thrown inside the body of `react` is handled, execution continues with the next loop iteration.

The `ReplyReactor` trait

The `ReplyReactor` trait extends `Reactor[Any]` and adds or overrides the following methods:

- The `!` method is overridden to obtain a reference to the current actor (the sender); together with the actual message, the sender reference is transferred to the mailbox of the receiving actor. The receiver has access to the sender of a message via the `sender` method (see below).
- The `forward` method is overridden to obtain a reference to the *sender* of the message that is currently being processed. Together with the actual message, this reference is transferred as the sender of the current message. As a consequence, `forward` allows forwarding messages on behalf of actors different from the current actor.
- The added `sender` method returns the sender of the message that is currently being processed. Given the fact that a message might have been forwarded, `sender` may not return the actor that actually sent the message.

- The added `reply` method sends a message back to the sender of the last message. `reply` is also used to reply to a synchronous message send or a message send with future (see below).

- The added `!?` methods provide *synchronous message sends*. Invoking `!?` causes the sending actor to wait until a response is received which is then returned. There are two overloaded variants. The two-parameter variant takes in addition a timeout argument (in milliseconds), and its return type is `Option[Any]` instead of `Any`. If the sender does not receive a response within the specified timeout period, `!?` returns `None`, otherwise it returns the response wrapped in `Some`.

- The added `!!` methods are similar to synchronous message sends in that they allow transferring a response from the receiver. However, instead of blocking the sending actor until a response is received, they return `Future` instances. A `Future` can be used to retrieve the response of the receiver once it is available; it can also be used to find out whether the response is already available without blocking the sender. There are two overloaded variants of the `!!` method. The two-parameter variant takes in addition an argument of type `PartialFunction[Any, A]`. This partial function is used for post-processing the receiver's response. Essentially, `!!` returns a future which applies the partial function to the response once it is received. The result of the future is the result of this post-processing step.

- The added `reactWithin` method allows receiving messages within a given period of time. Compared to `react` it takes an additional parameter `msec` which indicates the time period in milliseconds until the special `TIMEOUT` pattern matches (`TIMEOUT` is a case object in the `scaaala.actors` package). Example:

```
re ctWithin(2000) {
  cccaaaAnnnsssweee(text) => // process text
       TIMEOUT => println("no answer within 2 seconds")
}
```

The `reactWithin` method also allows non-blocking access to the mailbox. When specifying a time period of 0 milliseconds, the mailbox is first scanned

to find a matching message. If there is no matching message after the first scan, the `TIMEOUT` pattern matches. For example, this enables receiving certain messages with a higher priority than others:

```
reactWithin(0) {
  case HighPriorityMsg => // ...
  case TIMEOUT =>
    react {
      case LowPriorityMsg => // ...
    }
}
```

In the above example, the actor first processes the next `HighPriorityMsg`, even if there is a `LowPriorrriiityMsg` that arrived earlier in its mailbox. The actor only processes a `LowP orityMsg` *first* if there is no `HighPriorityMsg` in its mailbox.

The `ReplyReactor` trait adds the `Actor.State.TimedSuspended` execution state. A suspended actor, waiting to receive a message using `reactWithin` is in state `Actor.State.TimedSuspended`.

The `Actor` trait

The `Actor` trait extends `ReplyReactor` and adds the following members:

- The `receive` method behaves like `react` except that it may return a result. This is reflected in its type, which is polymorphic in its result type:

  ```
  def receive[R](f: PartialFunction[Any, R]): R
  ```

  However, using `receive` makes the actor more heavyweight, since `receive` blocks the underlying thread while the actor is suspended waiting for a message. The blocked thread is unavailable to execute other actors until the invocation of `receive` returns.

- The `link` and `unlink` methods allow an actor to link and unlink itself to and from another actor, respectively. Linking can be used for monitoring and reacting to the termination of another actor. In particular, linking affects the behavior of invoking `exit` as explained in the API documentation of the `Actor` trait.

- The `trapExit` member allows reacting to the termination of linked actors independently of the exit reason (that is, it does not matter whether the exit reason is `'normal` or not). If an actor's `trapExit` member is set to `true`, this actor will never terminate because of linked actors. Instead, whenever one of its linked actors terminates it will receive a message of type `Exit`. The `Exit` case class has two members: `from` refers to the actor that terminated; `reason` refers to the exit reason.

Termination and execution states

When terminating the execution of an `Actor` instance, the exit reason can be set explicitly by invoking the following variant of `exit`:

```
def exit(reason: AnyRef): Nothing
```

An actor that terminates with an exit reason different from the symbol `'normal` propagates its exit reason to all actors linked to it. If an actor terminates because of an uncaught exception, its exit reason is an instance of the `UncaughtException` case class.

The `Actor` trait adds two new execution states. An actor waiting to receive a message using `receive` is in state `Actor.State.Blocked`. An actor waiting to receive a message using `receiveWithin` is in state `Actor.State.TimedBlocked`.

11.2 Control structures

The `Reactor` trait defines control structures that simplify programming with the non-returning `react` operation. Normally, an invocation of `react` does not return. If the actor should execute code subsequently, then one can either pass the actor's continuation code explicitly to `react`, or one can use one of the following control structures which hide these continuations.

The most basic control structure is `andThen`. It allows registering a closure that is executed once the actor has finished executing everything else. For example, the actor shown in Listing 11.1 prints a greeting after it has processed the `"hello"` message. Even though the invocation of `react` does not return, we can use `andThen` to register the code which prints the greeting as the actor's continuation.

Note that there is a *type ascription* that follows the `react` invocation (`: Unit`). Basically, it let's you treat the result of `react` as having type `Unit`,

```
actor {
  {
    react {
      case "hello" => // processing "hello"
    }: Unit
    andThe  {
    printlnnn"(hi there")
  }}}
}
```

Listing 11.1 · Using `andThen` for sequencing.

which is legal, since the result of an expression can always be dropped. This is necessary to do here, since `aaandThecannot` be a member of type `Nothing` which is the result type of `re ct`. Treating the result type of `react` as `Unit` allows the application of an implicit conversion which makes the `andThen` member available.

The API provides a few more control structures:

- `loop { ... }`. Loops indefinitely, executing the code in braces in each iteration. Invoking `react` inside the loop body causes the actor to react to a message as usual. Subsequently, execution continues with the next iteration of the same loop.
- `loopWhile (c) { ... }`. Executes the code in braces while the condition `c` returns `true`. Invoking `react` in the loop body has the same effect as in the case of `loop`.
- `continue`. Continues with the execution of the current continuation closure. Invoking `continue` inside the body of a `loop` or `loopWhile` will cause the actor to finish the current iteration and continue with the next iteration. If the current continuation has been registered using `andThen`, execution continues with the closure passed as the second argument to `andThen`.

The control structures can be used anywhere in the body of a `Reactors` `act` method and in the bodies of methods (transitively) called by `act`. For actors created using the `accctt{ooorr}` shorthand the control structures can be imported from the `A` object.

11.3 Futures

The `ReplyReactor` and `Actor` traits support result-bearing message send operations (the `!!` methods) that immediately return a *future*. A future, that is, an instance of the `Future` trait, is a handle that can be used to retrieve the response to such a message send-with-future.

The sender of a message send-with-future can wait for the future's response by *applying* the future. For example, sending a message using `val fut = a !! mssg` allows the sender to wait for the result of the future as follows: `val re = fut()`.

In addition, a `Future` can be queried to find out whether its result is available without blocking using the `isSet` method.

A message send-with-future is not the only way to obtain a future. Futures can also be created from computations using the `future` method. In the following example, the computation `body` is started to run concurrently, returning a future for its result:

```
val fut = future { body }
// ...
fut() // wait for future
```

What makes futures special in the context of actors is the possibility to retrieve their result using the standard actor-based receive operations, such as `receive` etc. Moreover, it is possible to use the event-based operations `react` and `reactWithin`. This enables an actor to wait for the result of a future without blocking its underlying thread.

The actor-based receive operations are made available through the future's `innnputChannel` For a future of type `Future[T]`, its type is `InputCha nel[T]`. Example:

```
val fut = a !! msg
// ...
fut.inputChannel.react {
  case Response => // ...
}
```

11.4 Channels

Channels can be used to simplify the handling of messages that have different types but that are sent to the same actor. The hierarchy of channels is divided into `OutputChannels` and `InputChannels`.

`OutputChannels` can be sent messages. An `OutputChannel` `out` supports the following operations:

- `out ! msg`. Asynchronously sends `msg` to `out`. A reference to the sending actor is transferred as in the case where `msg` is sent directly to an actor.
- `out forward msg`. Asynchronously forwards `msg` to `out`. The sending actor is determined as in the case where `msg` is forwarded directly to an actor.
- `out.receiver`. Returns the unique actor that is receiving messages sent to the `out` channel.
- `out.send(msg, from)`. Asynchronously sends `msg` to `out` supplying `from` as the sender of the message.

Note that the `OutputChannel` trait has a type parameter that specifies the type of messages that can be sent to the channel (using `!`, `forward`, and `send`). The type parameter is contravariant: `trait OutputChannel[-Msg]`.

Actors can receive messages from `InputChannels`. Like `OutputChannel`, the `InputChannel` trait has a type parameter that specifies the type of messages that can be received from the channel. The type parameter is covariant: `trait InputChannel[+Msg]`. An `InputChannel[Msg]` `in` supports the following operations:

- `in.receeeive{ecase Pat1 => ... ; case Patn => ... }` (and similarly, `in.r ceiveWithin`). Receives a message from `in`. Invoking `rrreeeccoeeiinput` channel has the same semantics as the standard `e` operation for actors. The only difference is that the partial function passed as an argument has type `PartialFunction[Msg, R]` where `R` is the return type of `receive`.
- `in.react { case Pat1 => ... ; case Patn => ... }` (and similarly, `in.reactWithin`). Receives a message from `in` using the

event-based `react` operation. Like `react` for actors, the return type is `Nothing`, indicating that invocations of this method never return. Like the `receive` operation above, the partial function passed as an argument has a more specific type: `PartialFunction[Msg, Unit]`.

Creating and sharing channels

Channels are created using the concrete `Channel` class. It extends both `InputChannel` and `OutputChannel`. A channel can be shared either by making the channel visible in the scopes of multiple actors, or by sending it in a message.

```
actor {
  vvvaaarttt OutputChannel[String] = null
  l   child = actor {
    reac  {
      case "go" => out ! "hello"
    }
  }
  val channel = new Channel[String]
  out = channeeel
  ccchhhi!ld'go"
    annel.rec ive {
    case msg => println(msg.length)
  }
}
```

Listing 11.2 · Scope-based sharing of channels.

The example in Listing 11.2 demonstrates scope-based sharing. Running this example prints the string `"5"` to the console. Note that the `child` actor has only access to `out` which is an `OutputChannel[String]`. The `channel` reference, which can also be used to receive messages, is hidden. However, care must be taken to ensure the output channel is initialized to a concrete channel before the `child` sends messages to it. This is done using the `"go"` message. When receiving from `channel` using `channel.receive` we can make use of the fact that `msg` is of type `String`; therefore, it provides a `length` member.

```
case class ReplyTo(out: OutputChannel[String])

val child = actor {
  react {
    case ReplyTo(out) => out ! "hello"
  }
}

actor {
  val channeeel= new Channel[String]
  ccchhhil!ldReplyTo(channnnel)
    annel.r ceive {
    case msg => printl (msg.length)
  }
}
```

Listing 11.3 · Sharing channels via messages.

An alternative way to share channels is by sending them in messages. The example in The example in Listing 11.3 demonstrates this. The `ReplyTo` case class is a message type that we use to distribute a reference to an `OutputChannel[String]`. When the `child` actor receives a `ReplyTo` message it sends a string to its output channel. The second actor receives a message on that channel as before.

11.5 Remote Actors

This section describes the remote actors API. Its main interface is the `RemoteActor` object in package `scala.actors.remote`. This object provides methods to create and connect to remote actor instances. In the code snippets shown below we assume that all members of `RemoteActor` have been imported; the full list of imports that we use is as follows:

```
iiimmmpppsssoccaadllaaa...aaacccttooorrrsss..._
                    Actooor._
                    rrreeemmee..._
import scala.actors.        RemoteActor._
```

Starting remote actors

A remote actor is uniquely identified by a `Symbol`. This symbol is unique to the JVM instance on which the remote actor is executed. A remote actor identified with name `'myActor` can be created as follows.

```
class MyActor extends Actor {
  def act(((){
    alive 9000)
    register('myActor, self)
    // ...
  }
}
```

Note that a name can only be registered with a single (alive) actor at a time. For example, to register an actor *A* as `'myActor`, and then register another actor *B* as `'myActor`, one would first have to wait until *A* terminated. This requirement applies across all ports, so simply registering *B* on a different port as *A* is not sufficient.

Connecting to remote actors

Connecting to a remote actor is just as simple. To obtain a remote reference to a remote actor running on machine `myMachine`, on port `8000`, with name `'anActor`, use `select` in the following manner:

```
val myRemoteActor = select(Node("myMachine", 8000), 'anActor)
```

The actor returned from `select` has type `AbstractActor` which provides essentially the same interface as a regular actor, and thus supports the usual message send operations:

```
myReeemoteActodor"Hello!"
rec ive {
  case resp nse => println("Responnnse:" + response)
}
myRemoteActor !? "What is theemea ing of life?" match {
  cccaaasssee==pppprrriiinnttttlnn(((
       oops              Fail d: " + oops)
```

```
}
val future = myRemoteActor !! "What is the last digit of PI?"
```

Note that `select` is lazy; it does not actually initiate any network connections. It simply creates a new `AbstractActor` instance which is ready to initiate a new network connection when needed (for instance, when is invoked).

Bibliography

[Agh90] Agha, Gul. "Concurrent Object-Oriented Programming." *Communications of the ACM*, 33(9):125–141, September 1990.

[Arm95] Armstrong, J. L., M. C. Williams, C. Wikström, and S. R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1995.

[Dea08] Dean, Jeffrey and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *CACM*, 51(1):107–113, 2008.

[Goe06] Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison Wesley, 2006. ISBN 978-0321349606.

[Gro99] Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. The MIT Press, Cambridge, MA, second edition, 1999.

[Hal09] Haller, Philipp and Martin Odersky. "Scala Actors: Unifying thread-based and event-based programming." *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[Hal10] Haller, Philipp and Martin Odersky. "Capabilities for Uniqueness and Borrowing." In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010. ISBN 978-3-642-14106-5.

[HB77] Henry Baker, Carl Hewitt. "Laws for Communicating Parallel Processes." Technical report, MIT Artificial Intelligence Laboratory, http://hdl.handle.net/1721.1/41962, 1977.

[Hew73] Hewitt, Carl, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." In *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 235–245. 1973.

[Hew77] Hewitt, Carl E. "Viewing Control Structures as Patterns of Passing Messages." *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Hoa78] Hoare, C. A. R. "Communicating sequential processes." *Comm.ACM*, 21(8):666–677, 1978.

[Kay98] Kay, Alan. an email on messaging in Smalltalk/Squeak, 1998. The email is published on the web at http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html.

[Sut05] Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency." *Dr. Dobb's Journal*, March 2005.

About the Authors

Philipp Haller

Philipp Haller is a Ph.D. candidate in the School of Computer and Communication Sciences at EPFL (Switzerland), working with Martin Odersky on the Scala programming language, libraries, and tools. He received his Dipl.-Inform. degree (with distinction) in 2006 from Karlsruhe Institute of Technology (Germany). Among his research interests are programming abstractions for concurrency, as well as type systems to check their safety. Philipp created Scala Actors, a library for efficient, high-level concurrent programming.

Frank Sommers

Frank Sommers is an editor at Artima, and president of Autospaces, Inc., a company specializing in automotive finance software. After almost 15 years of working with Java, Frank started programming in Scala a few years ago, and became an instant fan of the language. Frank is an active writer in the area of information technology and computing. His main interests are parallel and distributed computing, data management, programming languages, cluster and cloud computing, open-source software, and online user communities.

Index

Page numbers followed by an n *refer to footnotes.*