

Building reliable software in a multi-core world

Akka Concurrency



artima

Derek Wyatt

Akka Concurrency

PrePrint™ Edition

Thank you for purchasing the PrePrint™ Edition of *Akka Concurrency*.

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.

Akka Concurrency

PrePrintTM Edition

Derek Wyatt

artima
ARTIMA PRESS
WALNUT CREEK, CALIFORNIA

Akka Concurrency

First Edition PrePrint™ Edition Version October 14, 2012

Derek Wyatt is a Software Architect and Developer specializing in large-scale, real-time applications for the World Wide Web.

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

Copyright © 2012 Derek Wyatt. All rights reserved.

First edition published as PrePrint™ eBook 2012
Build date of this impression October 14, 2012
Produced in the United States of America

No part of this publication may be reproduced, modified, distributed, stored in a retrieval system, republished, displayed, or performed, for commercial or noncommercial purposes or for compensation of any kind without prior written permission from Artima, Inc.

This PDF eBook is prepared exclusively for its purchaser. The purchaser of this PrePrint Edition may download, view on-screen, and print it for personal, noncommercial use only, provided that all copies include the following notice in a clearly visible position: "Copyright © 2012 Derek Wyatt. All rights reserved." The purchaser may store one electronic copy and one electronic backup, and may print one copy, for personal, noncommercial use only.

All information and materials in this book are provided "as is" and without warranty of any kind.

The term "Artima" and the Artima logo are trademarks or registered trademarks of Artima, Inc. All other company and/or product names may be trademarks or registered trademarks of their owners.

Overview

Contents	vii
List of Figures	xiii
1. Preface	21
2. Concurrency and Parallelism	33
3. Set Up Akka	58
4. Akka Does Concurrency	61
5. Actors	90
6. Akka Testing	127
7. Systems, Contexts, Paths, and Locations	150
8. Supervision and DeathWatch	174
9. Being Stateful	220
10. Routing Messages	261
11. Dispatchers and Mailboxes	290
12. Coding in the Future	304
13. Networking with IO	343
14. Going Multi-Node with Remote Actors	352
15. Sharing Data with Agents	381
16. Granular Concurrency with Dataflow	395
17. Patterns for Akka Programming	405
18. Antipatterns for Akka Programming	443
19. Growing Your App with Add-On Modules	453
20. Using Akka from Java	461
21. Now that You're an Akka Coder	478
About the Author	480

Contents

Contents	vii
List of Figures	xiii
1 Preface	21
1.1 Concurrent Challenges	22
1.2 Akka Is Concurrency	24
1.3 Concurrency Methodologies	24
1.4 The Akka Concurrency Toolkit	26
1.5 Who You Are	29
1.6 How to Read this Book	30
1.7 What You're Going to Learn	30
2 Concurrency and Parallelism	33
2.1 Parallelism vs. Concurrency	33
2.2 A Critical Look at Shared-State Concurrency	34
2.3 Immutability	44
2.4 Chapter Summary	56
3 Set Up Akka	58
3.1 Scala Setup with SBT	58
4 Akka Does Concurrency	61
4.1 The Actor	61
4.2 The Future	79
4.3 The Other Stuff	85
4.4 You Grabbed the Right Toolkit	89

5 Actors	90
5.1 The Components of an Actor	92
5.2 Properties of an Actor	94
5.3 How to Talk to an Actor	98
5.4 Creating Actors	102
5.5 Actors in the Clouds	103
5.6 Tying It Together	117
5.7 How Message Sending Really Works	119
5.8 The ActorSystem Runs the Show	123
5.9 Chapter Summary	125
6 Akka Testing	127
6.1 Making Changes to SBT	127
6.2 A Bit of Refactoring	128
6.3 Testing the EventSource	129
6.4 The Interaction Between ImplicitSender and testActor	133
6.5 TestKit, ActorSystem, and ScalaTest	134
6.6 Testing the Altimeter	139
6.7 Akka's Other Testing Facilities	145
6.8 About Test Probes and the testActor	146
6.9 Chapter Summary	148
7 Systems, Contexts, Paths, and Locations	150
7.1 The ActorSystem	150
7.2 Actor Paths	152
7.3 Staffing the Plane	154
7.4 The ActorContext	163
7.5 Relating the Path, Context, and System	170
7.6 Chapter Summary	172
8 Supervision and DeathWatch	174
8.1 What Makes Actors Fail?	174
8.2 The Actor Life Cycle	175
8.3 What Is a Supervisor?	178
8.4 Watching for Death	186
8.5 The Plane that Healed Itself	192
8.6 Dead Pilots	211
8.7 Chapter Summary	218

9 Being Stateful	220
9.1 Changing Behaviour	220
9.2 The Stateful Flight Attendant	226
9.3 A Better Flyer	231
9.4 The Naughty Pilot	246
9.5 Some Challenges	255
9.6 Testing FSMs	256
9.7 Testing the Pilot	258
9.8 Chapter Summary	259
10 Routing Messages	261
10.1 Routers Are Not Actors	261
10.2 Akka's Standard Routers	261
10.3 Routers and Children	265
10.4 Routers on a Plane	267
10.5 Magically Appearing Flight Attendants	280
10.6 Sectioning off Flight Attendant Territory	282
10.7 More You Can Do with Routers	288
10.8 Chapter Summary	288
11 Dispatchers and Mailboxes	290
11.1 Dispatchers	290
11.2 Dispatcher Tweaking	294
11.3 Mailboxes	296
11.4 When to Choose a Dispatching Method	301
11.5 Chapter Summary	302
12 Coding in the Future	304
12.1 What Is the Future?	304
12.2 Don't Wait for the Future	306
12.3 Promises and Futures	308
12.4 Side-Effecting	326
12.5 Futures and Actors	328
12.6 Plane Futures	338
12.7 Chapter Summary	341
13 Networking with IO	343
13.1 The Plane's Telnet Server	343

13.2	Iteratees	351
13.3	Chapter Summary	351
14	Going Multi-Node with Remote Actors	352
14.1	Many Actors, Many Stages	352
14.2	Simple Build Tool (SBT)	353
14.3	Remote Airports	354
14.4	Going Remote	359
14.5	Flying to the Airport	362
14.6	Programmatic Remote Deployment	367
14.7	Configured Remote Deployment	370
14.8	Routers Across Multiple Nodes	371
14.9	Serialization	372
14.10	Remote System Events	375
14.11	On the Subject of Lost Messages	378
14.12	Clustering	379
14.13	Chapter Summary	379
15	Sharing Data with Agents	381
15.1	SBT	382
15.2	Agents as Counters	382
15.3	Working with Agents	388
15.4	The API	389
15.5	Transactional Agents	394
15.6	Chapter Summary	394
16	Granular Concurrency with Dataflow	395
16.1	Caveats	396
16.2	With That Said...	397
16.3	Getting Dataflow into the Build	397
16.4	Dataflow Values	398
16.5	Flow	399
16.6	Another Way to Get Instrument Status	403
16.7	When to Use Dataflow	404
16.8	Chapter Summary	404
17	Patterns for Akka Programming	405
17.1	Behavioural Composition	405

17.2	Isolated and Parallel Testing	408
17.3	Strategies for Implementing Request/Response	411
17.4	Mechanisms for Handling Non-Deterministic Bootstrapping	418
17.5	The Circuit Breaker	422
17.6	Breaking Up a Long-Running Algorithm into Multiple Steps	422
17.7	Going Parallel	425
17.8	An Actor EventBus	427
17.9	Message Transformation	428
17.10	Retry Behaviour	429
17.11	Shutting Down When All Actors Complete	441
17.12	Chapter Summary	442
18	Antipatterns for Akka Programming	443
18.1	Mutability in Messages	443
18.2	Loosely Typing Your Messages	444
18.3	Closing over Actor Data	445
18.4	Violating the Single-Responsibility Principle	447
18.5	Inappropriate Relationships	448
18.6	Too Much actorFor()	449
18.7	Not Enough Config	451
18.8	Needless Future Plumbing	452
18.9	Chapter Summary	452
19	Growing Your App with Add-On Modules	453
19.1	Extensions	453
19.2	Working with Software Transactional Memory	454
19.3	ZeroMQ	455
19.4	Microkernel	456
19.5	Camel	456
19.6	Durable Mailboxes	458
19.7	Clustering	458
19.8	HTTP	459
19.9	Monitoring	460
19.10	Chapter Summary	460
20	Using Akka from Java	461
20.1	Immutability	461
20.2	Differences Overall	462

20.3	Glue Classes	462
20.4	akka.japi.Procedure	462
20.5	Messages	465
20.6	The Untyped Actor	466
20.7	Futures	471
20.8	Manipulating Agents	477
20.9	Finite State Machines	477
20.10	Dataflow	477
20.11	Chapter Summary	477
21	Now that You're an Akka Coder	478
21.1	Akka.io	478
21.2	The Mailing List	479
21.3	<i>Have Fun!</i>	479
	About the Author	480

List of Figures

2.1	One way we're used to writing our shared-state concurrency: create a piece of the domain model, give access to it from a couple of different threads, and code with side effects, ensuring a proper rendezvous with locks.	45
2.2	When making changes to our User object, we tend to code like this: we make alterations using a pipeline of commands instead of several modifications to the same object.	46
2.3	The same operations on the User object are done in parallel here. We must recombine the state changes into a final object representation, of course, but we can see that the input to the operations and the output of the operations remain exactly the same as the linear version.	47
2.4	Here's an attempt to get the same type of parallelism from a mutable User object; in this case, the User object is protected by locks now. Even though we've spread the work out to multiple threads, we're still linear in our speed simply because only one thread can work at a time. Not only do we not get parallelism, we are also wasting threads.	48
2.5	A good ol' linked list: our symbol name <code>list</code> points to the head of the list and it always ends in <code>Nil</code>	49
2.6	The recursive nature of <code>List</code> coupled with its immutability lets us quickly create a new list from a pre-existing one by simply dropping elements from it.	49

2.7	We can easily create new Lists from existing Lists by prepending new elements at various locations. Each List is truly its own distinct List—it just happens to be that the implementation can use the underlying storage because the underlying storage is immutable.	51
2.8	The act of appending a 6 to the list results in an entirely new list with a new pointer to the head of the new list. The old list, of course, remains entirely unchanged.	52
2.9	A really bad representation of a map; we haven't attached values to this map just because it's easier to see, but a tree tends to make a pretty decent map implementation.	52
2.10	After removing the E from the original map, we now have two distinct maps. The original is represented by the purple squares and the new one, without E, is represented by the blue ovals with D' at the root.	53
2.11	This shows an immutable calendar-like structure of meetings and we want to add Fred as an attendee to the meeting from 2pm to 3pm on Wednesday. If the structure were mutable, then this would be a piece of cake. Immutability makes this more difficult.	56
4.1	Chances are you've either been the coding intern or you've been the other guy. If you've ever been the coding intern, then you'll be the other guy eventually. It's being the intern that makes the other guy—nobody knows whether the chicken or the egg came first.	62
4.2	Three interns can write three tests faster than one intern can write three tests.	63
4.3	It's quite possible that these interns have chosen different methods for implementing the tests. The second one seems to have chosen a decent toolkit, or didn't sleep, or is high on some sort of amphetamine, or... who cares? He won.	64
4.4	So this intern is pretty smart, or at the very least, sneaky. There's nothing to say he can't do exactly what you're doing. Not only has he done it, he's hidden it by ensuring that his friends return their results to him so that he can send them to you. You're clueless.	65

4.5 Poor Joe. The evil doers were just too much for him. But this isn't a problem for you—you've got Mary! Mary can get the job done.	66
4.6 The message processing for Actors is not unlike the types of message processing that you might be familiar with if you've had any experience with message pumps inside GUI frameworks. This is a simplified view of the Actor constructs in Akka and it doesn't give a very good indication of the power behind it, but that's a good thing for the moment. If I blew your mind now, then you wouldn't have the mental capacity to continue reading.	68
4.7 When we're processing a series of RSS feeds, we can carry the full input and output in the messages themselves. The business logic in charge of downloading the RSS feeds from the series of tubes is held inside the Actor, but the information about what to download and the results of the download travel in the messages between the iterations of the algorithm.	71
4.8 One way of viewing the Actor/Message pair: Only as a full set of Message, Message body, and Actor do we realize the full notion of a function. When the Actor processes the message, it realizes what behaviour it needs to invoke via the message, and uses the message body to drive the behaviour.	73
4.9 The Actor is now an input/output function... mostly.	74
4.10 The simplest case where the initiating entity receives the eventual response from the Actor, thus completing the function analogy.	74
4.11 The Actor that's getting the groceries isn't under any contract to send results back to the original guy making the request. In this case, the original request passed a reference to Betty in the message, which directs the Actor to deliver the groceries to her.	75
4.12 A simple pattern for replicating messages to go to multiple destinations from the source. In this case, we're just sending it to where it was supposed to go originally, as well as streaming it to disk.	76
4.13 A simple representation of 10,000,003 matrices that we want to multiply together into a final result.	80

4.14 Grouping a set of matrices to be multiplied into two groups: Group A can be multiplied together at the same time as group B. Once both results have been obtained, we can multiply the result into a final, single matrix.	80
4.15 When you give work to a set of Actors, they will complete it at non-deterministic intervals. This means that response messages will come back in what is effectively random order.	81
4.16 Akka's Future implementation allows us to take a list of Futures and convert them to a Future that contains a list of their results, and it maintains the same sequence as the original Futures.	82
4.17 Futures can tacitly bind themselves to Actors by representing themselves as the "sender" of the request message. Assuming the Actor responds to the sender, the response will go back to the Future. This entire interplay between the Actor and the Future is non-blocking.	85
4.18 The conceptual difference between Futures and dataflow: Fu- tures are non-blocking and can execute full functions in par- allel, whereas dataflow creates concurrency that's more of an intra-function type of concept. Each algorithm, represented by the individual boxes, is either waiting for (while not blocking a thread) or is populating a piece of data. Between any two points of contention, things run concurrently, but they rendezvous on the shared data.	87
5.1 Shows us the parts of the Actor that concern us as programmers. All of these components together facilitate the execution of mes- sage processing in your Actor's code. Note that everything but the ActorRef and the message is internal. Someone in the out- side world constructs a message, and he can only send it to the ActorRef. He can't send it to the Actor directly.	93
5.2 Programming with Actors is more like hanging out with a bunch of people at the office than it is about sequences and algorithms. Jill's not interested in <i>who</i> gets her coffee for her, just so long as she gets it.	95
5.3 When an Actor has a lot to do, blocking on a response from it can be more painful than you might otherwise be used to.	101

5.4 This is what our Plane currently looks like, from the view of the avionics-driving main, which we depict as the Outside World. Solid lines indicate a physical binding—i.e., the ActorSystem owns the Plane and the Plane owns the Altimeter and Control-Surfaces, whereas the dotted lines indicate a reference relationship for message sending.	118
5.5 Three different representations of how message sending works. Every time a message is sent, the message gets a passenger—a reference to the sender. If ! is called from within an Actor, then that Actor becomes the sender, or it's the Dead Letter Office if you're not inside an Actor. When forwarding, we propagate the original sender with the message in order to implement standard forwarding semantics.	123
6.1 When we test our Altimeter, we are technically testing every one of its components.	128
6.2 Normally, the EventSource Actor is unavailable to our runtime code because Akka hides it behind an ActorRef. The TestKit's TestActorRef gives us access to the Actor's internals so we can poke and prod it directly.	131
7.1 Here, we see the key players in the ActorSystem that interest us most of the time, including three Actors (Dead Letter, System, and User) as well as three other key entities (Scheduler, Event Stream, and Settings). These ActorSystem elements, with the exception of the System Guardian, regularly pop up in Actor programming.	151
7.2 The Plane's hierarchy, including the sub-hierarchy of the Flight Attendants. Note that the path requirements are determined entirely by the parent/child relationships between the Actors.	162
7.3 The structure of the Plane after everyone's in place	171
7.4 How we relate the Actor to its context, its ActorRef, and the path. It's the context that holds the relationships together, and those point to the ActorRefs (not the Actors or the ActorContexts). From the ActorRef, we can retrieve the ActorPath object, which we can interrogate as we please.	172

8.1 Actor supervision is a recursive definition. The Actor system's hierarchical nature provides a simple definition of supervision such that the question of "what to do?" can be asked at any level.	176
8.2 The Actor life cycle, including code examples that can get us to each point as well as certain overrideable callbacks that are available to hook into the life cycle.	177
8.3 Two hooks exist for restart processing in the Actor. It's important to understand that the <code>preRestart()</code> method is executed on the Actor that processed the failing message, whereas the <code>postRestart()</code> method is executed on the freshly instantiated Actor.	182
8.4 We wouldn't necessarily want to restart all of the children when the Actor that failed is a great, great, great, great, great, great, great grandfather to 5,000 descendants.	183
8.5 The Actor that restarts passes through the <i>restart</i> life cycle, but its children are a different story. The children are specifically <i>stopped</i> and re-created. With each Actor restart, the children are stopped and a new generation replaces them. If the Actor has DeathWatch on them, then it will get <code>Terminated()</code> messages for each one on every restart.	188
8.6 Actor restarts are not visible to the outside world and thus all of the guys that have an <code>ActorRef</code> to it are unaffected. However, it's a different story for those that might have references to that Actor's children. Since the children will by default stop and then get re-created, those old references send all messages to the Dead Letter Office.	191
8.7 The outline of the structure we're going for when building our Plane. Actor systems inevitably become a tree in the real world and our Plane is starting to get that way.	193
8.8 The slice of the Plane's hierarchy after <code>startControls()</code> completes.	199
8.9 The slice of the Plane's hierarchy after <code>startPeople()</code> completes.	200

8.10 Risk is generally pushed down the tree. The higher up you go, the less risk you're willing to take on, and the lower you go, the more risky you become. Guys that live at the bottom are the ones you want to send on the brutal missions where they're not guaranteed to survive, but the guys at the top don't get their hands dirty at all.	208
8.11 The simple hierarchy we need in order to test the CoPilot	213
8.12 The AutoPilot's test doesn't need to have any special parenting. Its parent can simply be the test ActorSystem's User Guardian.	217
9.1 The state transitions are easily deduced from the code due to the static usage of the <code>become()</code> method.	224
9.2 Our state stack's condition after a whole load of <code>become()</code> s without a single <code>unbecome()</code> . If you're not careful in your own (more complex) code, you could end up with a stack that eventually blows up. You just need to accumulate more pushes than pops over a long enough period of time and you have the exact same problem.	225
9.3 The FlightAttendant changes its behavioural state when certain messages are received in certain states. One nice effect of knowing you are in a particular state is that you can simply hard-code values rather than having to check what state you're in with an <code>if</code> statement. Here, we illustrate that the response to the <code>Busy_?</code> message is determined by the state in which the FlightAttendant finds itself.	228
9.4 The Pilot receives information about how he feels from the drinking behaviour Actor, which allows him to alter his flying behaviour accordingly.	249
10.1 Messages are routed to the composed Actors in a round-robin fashion.	262
10.2 In this case, the SmallestMailboxRouter will choose Actor 3 as the recipient of Message 1, since its Mailbox is clearly the smallest.	263

10.3 The PassengerSupervisor's structure houses an IsolatedStopSupervisor to manage its Passengers and uses the default supervision strategy for its immediate children. The BroadcastRouter will be created after the IsolatedStopSupervisor is completely instantiated.	275
10.4 The non-blocking, asynchronous algorithm we use to obtain the BroadcastRouter of the children of the PassengerSupervisor's embedded Stop Supervisor.	278
10.5 In this layout for the SectionSpecificAttendantRouter test, the mocked pieces and messages are geared toward getting enough information into the <code>testActor</code> so we can verify what we expect to route where.	286
11.1 The conceptual view of the Event-Based Dispatcher with respect to its Actors and the Mailboxes that contain their messages	292
11.2 The conceptual view of the BalancingDispatcher with respect to its Actors and the Mailbox that contains the messages for all of them	293
11.3 By changing the <code>throughput</code> value in the Dispatcher configuration, we can alter how quickly these Mailboxes will drain.	295
11.4 A simple diagram of the PressureQueue: The Actor holds the “put” lock as long as the pressure algorithm demands. This length of time is a factor of the current queue size. The “take” lock releases immediately so that the queue’s consumer can empty it as quickly as possible.	300
12.1 A simple depiction of using a Future to synchronize an asynchronous API into pre-existing sequential code	305
12.2 Futures are fulfilled by a Promise, which is held by a servant that gives the Future to the requester. This creates the conduit through which the servant and requester communicate.	308
12.3 There's a pretty clear difference between <code>flatMapting</code> pre-existing Futures and creating them on the fly.	312
13.1 The Telnet Server routes all incoming data to the appropriate Sub-Server Actor that has been instantiated to handle a specific client. It also instantiates that Sub-Server Actor when a new client connects, and clears it out when it disconnects.	346

13.2 The alterations that have been made to the Altimeter, HeadingIndicator, and Plane allow for the above message flow. The Plane acts as a relay, forwarding the requests that are then fulfilled to the original sender (the SubServer).	348
14.1 Two nodes in an Akka application. An ActorSystem can span nodes and still maintain the parent/child relationships between them, including Supervision. All ActorRefs appear as ActorRefs, no matter where the Actors may reside, so DeathWatch and standard message passing still apply.	353
14.2 A message transformer converts one message type to another, providing any data transform required during the transformation.	358
14.3 The words RemoteClientLifeCycleEvent and RemoteServerLifeCycleEvent only have meaning when we bind them to a particular node context. A “client” event for the Plane Node is a corresponding “server” event for the Airport Node and vice versa.	376
14.4 Sending the message to the Airport Server connects and starts the Plane Client, while the shutdown of the Airport Server subsequently disconnects and shuts down the Plane Client.	377
15.1 To create more than one bathroom, we’ll use a Round Robin Router and give each bathroom access to the counter Agents (which will be housed inside the Plane).	386
17.1 The multi-stage asynchronous algorithm is essentially the propagation of evolving copies of a single message type. The message has two member lists; the left list contains work to be done, which decreases over time, while the results list on the right increases over time.	423
18.1 Masters send events to Slaves, but it only does this after the Slaves have advertised their existence to the Master. Don’t do it the other way around.	449

Chapter 1

Preface

Somewhere around the year 2005, Herb Sutter¹ told us that “The Free Lunch Is Over.”² Ever since then, we’ve been going to the deli counter with baseball cards, a couple of nickels we earned from our lemonade stand, and whatever else we could haul out of our piggy banks, if only we could get a sandwich. It’s about time we got our hands on a real wad of cash.

I remember the days when it was all about the *megahertz* and nobody was even thinking about the day when it would be all about the *number of cores*. As I write this book, the tech heads are predicting that this is the year when we’ll see quad-core smartphones.³ Even the mobile developer can no longer hide from the multi-core world. Developers should not even consider the idea of writing an application that can’t take advantage of the hardware on which it’s running. But when you’re given a nice piece of hardware with a ton of cores and huge potential for concurrency, what tools do you have to go with it? Threads, locks, mutexes, critical sections, synchronized methods, and all of their brothers, sisters, cousins, and pets... oh my. We’ve certainly learned a lot about concurrency over the last decade or so, and one of the things we’ve learned is that concurrency is still *hard*.

¹<http://herbsutter.com/>

²<http://www.gotw.ca/publications/concurrency-ddj.htm>

³http://www.cnet.com/8301-17918_1-57364255-85/quad-core-smartphones-this-is-their-year/

1.1 Concurrent Challenges

What is it about getting our creations to do many things at once that's so difficult? Is it the processor architecture, the caches, the memory, the bus, the OS constructs, language choice, programming paradigm? When you really look at it, with all the complexity that exists in our field of study, it's amazing we get anything done at all. To survive it, we employ the same trick time and time again: simplification and abstraction. We'll continue that tradition here by breaking down the challenges into two general categories: conceptual and technological.

Modern hardware and software provide us with heaps of awesome that help us solve the problems we face when writing our applications. Unfortunately, that's only half the story. The tools we're given require that we handle several technological challenges that generally come in the form of synchronizing on data in order to tame the concurrency we're writing in the first place. These technological issues are well known and pretty well understood, but what's much less discussed are the cognitive challenges that the tools present.

You and I deal with concurrency every single day. Let's face it, your entire life is a series of interrupts and a collection of requests and responses. You send requests to people and expect responses (at some point), and people are doing the same to you on a constant basis. It's not rocket science, right? You function pretty well, right? Your friends, coworkers, and loved ones manage to get through their daily lives without running into dead locks and memory corruption, right? How on earth do we manage to accomplish this feat without an army of code inspectors checking our every decision before we make it?

These cognitive challenges are subtle and not the easiest to grasp, but the basic notion is pretty simple: the mental energy we expend on modern concurrency models is rather unnatural. What you've learned as a concurrency programmer these days doesn't translate all that well to your daily life, and the converse is also true.

But before we travel too far down that road, let's make a quick summary of what our challenges are.

Technological

- Optimizing the use of threads requires non-blocking APIs, which are rare in the wild and are also historically cumbersome to write.
- Dealing with errors in an asynchronous system can be difficult to manage. Traditional error-handling strategies (exceptions and return codes) do not translate well when inside a concurrent application.
- Controlling access to your data requires constant vigilance. Ensuring that race conditions and deadlocks are eliminated can verge on the impossible.
- Setting up a signaling mechanism between two objects is required in order to eliminate race conditions, and speed up wait times.
- The complexity of most common concurrent applications can increase the cost of refactoring to a fairly high degree. This makes complex concurrent applications accrue technical debt at a much faster rate than their sequential counterparts.

Cognitive

- Organizing objects and functions into a concurrent model can be difficult to visualize.
- Segmenting data structures with respect to concurrent access can be difficult to design.
- Designing application flow such that it minimizes bottlenecks may not be obvious.
- What to do in the face of unexpected failure can generally heap on too much mental load during design and coding. This tends to obscure the purity of the model while, at the same time, compromising the error handling of the application.

These reasons, and many others, have made using the concurrency tools of our past rather challenging. This is not to say that they are bad; in fact, several facilities provided by some libraries (`java.util.concurrent`, for

example) are quite useful and will continue to be useful going forward. However, the challenges they present, both conceptual and technological, leave room for improvement. And considering the low-level importance of concurrency in our lives as software developers, eliminating some of the greater pain points of concurrency development would be a welcome improvement indeed!

1.2 Akka Is Concurrency

Akka is positioned to help solve many of the difficulties with concurrency programming by bringing together a set of complimentary programming styles, backed by a strong toolkit and core paradigm. With Akka, you are armed with a collection of tools that address the technological concerns while, at the same time, smoothing out the issues in the conceptual model.

1.3 Concurrency Methodologies

When you look at the topic of concurrency in the general sense, there really are only two types: shared-state concurrency and message passing for concurrency. Shared-state concurrency has dominated for a long time, but before it did, the idea of using message passing was the mainstay. Multiprocessing was used to carve up work between processes on a machine, and communication between those processes was accomplished using file descriptors on which you could simply read and write. This was message-passing in its most primitive form.

Today, both methods are prevalent in our world. Applications tend to favor the use of shared-state concurrency with threads and locks, whereas larger systems favor message passing at the macro level (unless, of course, a database is being used as a rendezvous). The Internet itself is a massive message passing system and web programmers are becoming much more acquainted with this style of writing apps.

Shared-State Concurrency

To this day, the bulk of our tools rely on shared-state concurrency; i.e., the dysfunctional family described earlier. We make all or most of our data mutable, provide access to it from anywhere (properly encapsulated, of course),

and then spend the rest of our time trying to duct-tape our way into a safe application.

Most successful applications that employ shared-state concurrency grow in a manner that resembles something like:

- Create some object that carries some data.
- Create some functions to operate on that data.
- Realize it's slow. Create threads (or an ExecutorService) and refactor your code so it can run concurrently.
- Find out that you have a ton of race conditions. Protect the data with synchronization primitives.
- Scratch your head about why only 10% of your cores are being utilized. Eventually you blame the synchronization. Refactor again to try and increase concurrency. And so on...

For the purposes of our discussion, shared-state concurrency is all about using threads to make our code "go fast" and then using synchronization primitives to make it "safe."

It's not all a doom-and-gloom story, of course. An incredible number of successful applications take advantage of multi-core hardware; shared-state concurrency has brought us a long way. But with all of the advantages that we've gotten from employing these low-level concurrency tools comes a large number of concurrency headaches. Deadlocks, race conditions, memory corruption, scalability bottlenecks, resource contention, and of course my personal favorite, the good ol' *Heisenbug*.⁴

Message Passing for Concurrency

With shared-state concurrency, we tend to create the problem first, and then fix it up with synchronization primitives later. When a new crack in the wall appears, we grab some sticky goo and shove it in there to fix it up. Message passing looks at the problem from the other side: "What would it look like if we didn't create the problem in the first place?"

⁴Add a `printf` to see what's going on, and the bug disappears...love that one.

For multiprocessing, this is absolutely clear. In an operating system, there's simply no way for one process to muck up another process's internal data, and the only way you'll be able to get any two processes to communicate with each other is via message passing (i.e., reading and writing some sort of file descriptor). If those processes are single threaded, then they don't need any synchronization primitives at all. We are therefore saved from all of the pain that can be caused by trying to control concurrent access to mutable data simply because there is no concurrent access to that mutable data.

Message passing has become ubiquitous in our programming lives due to the Internet itself. No service on the Web is going to survive these days unless it provides a top-notch RESTful interface. And what's REST if not a message passing system?

But clearly this isn't the end of the story since multiprocessing is certainly no golden hammer; we need to write concurrent software inside a single process. Fortunately for us, a long time ago message passing became perfectly viable in single process programming as well and more recently it has become available on the JVM through Akka.

1.4 The Akka Concurrency Toolkit

If we're going to remain productive as programmers and truly scale our programs to the hardware on which they run, while at the same time keeping them reliable and understandable, then we need to stop *coding our concurrency* and just let our code *run concurrently*.

Enter Akka.

The dominant platform for enterprise (and cloud?) software development is still the Java Virtual Machine but what's been missing from Java is a truly abstracted concurrency toolkit. There are some solid concurrency tools in the Java library, and many of those tools are employed by Akka to deliver its solution, but those tools aren't what most of us would dream about when we sit down in front of our favorite editor.⁵ We want to code at a higher level and leave the pain of concurrency programming to someone else. Akka delivers on that need.

The shared-state concurrency tools we have today certainly have at least one thing right: *they stay out of your way*. Ultimately, the control of my application still belongs to me, which is exactly where it should be. Attempts

⁵i.e., Vim (<http://www.vim.org>)

have been made over the years to provide large frameworks that do all of the heavy lifting for you, but these have largely failed to provide us with the agility, speed, and flexibility that modern applications require. We don't need another framework; we need a toolkit full of powerful abstractions we can go to when we've got a concurrency solution that needs modeling. Akka delivers here as well.

A Short Note Regarding Akka and Scala

Akka was designed alongside Scala (and Java), which created some duplicity between Akka and Scala on a conceptual level. Scala has always had Actors and Futures, as has Akka. However, Akka's design and implementation of Actors and Futures has evolved far past that of Scala's implementation.

Because Scala and Akka are so closely aligned, the code in Akka is migrating to Scala. For Scala 2.10 and Akka 2.1, this migration includes the Future implementation, so we'll be talking about Futures in the context of the Akka paradigm. However, from a packaging perspective, we'll be importing code from the `scala` name space.

Akka Tools

Akka brings together a solid set of tools into a new paradigm of concurrent development:

Actors In 1973, Carl Hewitt defined the Actor⁶ and its most popular implementation to date has been in Erlang.⁷ Unfortunately, Erlang itself has not seen the widespread adoption of a mainstream language like Java.

Futures We'll see that the Actor, while a powerful mechanism for concurrency and resiliency, is not a silver bullet. In fact, some problems are just downright *wrong* to be modeled with Actors. Where Actors don't apply, we often find that Futures work incredibly well, and this is in no small part due to the fact that the Future implementation in Akka⁸ is very powerful indeed. Using Futures allows you to create robust pipelines of concurrent code that can easily operate on immutable data safely and without blocking.

⁶There's some decent information at http://en.wikipedia.org/wiki/Carl_Hewitt

⁷<http://www.erlang.se/>

⁸Currently slated to become part of Scala 2.10.

Timers The timer has never ceased to serve us well and Akka doesn't discriminate. Timers play a significant role in Akka.

Callbacks/Closures Shared-state concurrency uses callbacks and closures all the time, and there's nothing wrong with them. However, synchronizing them has always caused us pain. With respect to modeling concurrency, these abstractions are fantastic.

There are some variations on these themes within Akka, but those concepts cover the bulk of what it is that the Akka toolkit provides. And, yes, Akka is providing it for you *on the JVM*, accessible even from Java itself.

Error Handling (a.k.a. Resiliency)

Developers tend to ignore any serious error handling (i.e., those errors that fall outside of the usual ones that are found in unit tests) until after the nick-of-time. This situation isn't helped by the fact that most asynchronous toolkits (or frameworks) don't consider error handling at all; it's entirely your problem. Can you throw an `Exception`? Can anything even catch it? Can you return an error code? To whom? What are they expected to do with it when they get it? Retry? Abort? Resume? Ugh... Far too often, developers are left trying to poorly bolt on some sort of concurrent error-handling mechanism all on their own.

The tools we are used to in imperative programming—Exceptions and return codes—don't help us when we're developing complex concurrent systems. Akka doesn't ignore the problem of error handling, or bolts it on as an afterthought. One of Akka's core philosophies is that errors and failures are *part of the design* of any good application, not things to be avoided. Your application will be affected by real life at some point.

When real life comes around and tries to kick you in the head, what do you do? Often, people try to change real life into some sort of fantasy land where things don't go wrong, or simply ignore the problem in the hope that it won't happen. Akka's different; real life won't stop kicking you in the head – that's just what real life does—so if you can't beat it, join it and just “Let It Crash.”⁹. Some sort of scenario will happen where a small piece of

⁹One of the core philosophies of the Actor paradigm involves embracing failure to the point where we let small parts of the application crash, and trust the Actor system to heal itself. We'll learn more about this later.

your app will take a hit to the family jewels, and that's going to take it down (who could stand up in the face of that?). Akka provides a mechanism where that piece of the application can heal itself and keep on trucking. Welcome to *reliability*.

Non-Blocking by Default

Threads are a precious (and very expensive) resource. When you start getting into programs that have to handle thousands of concurrent operations per second (we don't even need to think about the ones that have to handle *millions* to see this), blocking on a thread is a surefire way to ensure that your app won't scale. Any toolkit that claims to provide a new concurrency development paradigm must provide a way to protect these precious resources. Akka doesn't disappoint, and you'll become quite competent in ensuring that your threads are working their butts off.

1.5 Who You Are

So, who are you? You're passionate about software development, crazy about concurrency, fed up with using the primitive tools of the day (or at least a bit dissatisfied with them), and you're looking for a new way to write your awesome software on the JVM. It's just that simple.

What You Already Need to Know

- You know Scala. While Akka is available with a Java API, which we'll be covering to a certain degree, the bulk of this book will be using Scala. We won't be using any of the really esoteric (and sometimes ultra-cool) aspects of Scala so you don't need to be an expert Scala programmer. However, you should know your way around it pretty well. If you're familiar with Scala but not a Scala programmer, that's probably fine—just have a reference manual handy in case you get lost.¹⁰
- Most real-world Scala applications use the Simple Build Tool (SBT), which we'll be using here. If you don't know it, then a crash course is

¹⁰Of course, the standard *Programming in Scala, 2nd Edition* by Martin Odersky, et al. would work just fine, but if another book works for you, then go for it.

available where SBT is distributed at <https://github.com/harrah/xsbt>.

- You should know the ScalaTest testing framework¹¹ or at least be familiar with standard unit testing frameworks and methodologies.

1.6 How to Read this Book

This book is designed to be a tutorial for those who want to learn Akka and may or may not have previous experience writing concurrent code. It is intended to be read from start to finish but for those of you who already have some extensive experience writing concurrent code, you might want to skim through the early sections of the book.

The beginning couple of chapters take you through a high-level tour of Akka, as well as describe some of the foundational aspects on which Akka is designed, from the perspective of concurrency. These sections help lay the foundation for understanding the paradigm that Akka describes. So, if you're confident in these aspects, I encourage you to skim them, just to be sure. Eventually, we get into the heavy nuts and bolts of Akka, and I hope you'll all join at that time.

This book is not intended to be a reference for the Akka toolkit; the materials that the Akka team have put together already serve that purpose, and serve it well. One hallmark of a solid project is its documentation, which is a wonderful thing to see in the Akka project. They've done a marvelous job of writing high-quality reference documentation as well as the ScalaDoc. You can find all of this on the Akka website: <http://akka.io/docs>.

1.7 What You're Going to Learn

In a nutshell, you're going to learn how to competently write software in the paradigm of Akka. But before we drive headlong into delivering on that promise, I'm going to have to jump onto my soapbox for a second.

<soapbox>

¹¹<http://scalatest.org/>

Did you see the word *paradigm* up there? That's not the first time that's been written in this book, and we aren't even finished with the first chapter yet. It's a pretty serious word.

One of my passions in life is maximizing my own productivity as a programmer, and one of the ways I do this is to use an editor you may have heard of called Vim.¹² A couple of years ago, I spent a short amount of time throwing together a series of screencasts about Vim in order to convey my passion for this particular tool and to hopefully help others become proficient in it. One of the things I yell about in these screencasts is that you should *not* use the mouse or the arrow keys.

Why? Some think it's because I'm just another Vi nut-bar¹³ but that's really not the reason. The reason is because the Vi family of editors delivers a particular paradigm on text manipulation. If you're going to use the editor much like you use any non-mode-based editor, then go and use those editors instead; the learning curve won't be as heavy. If you're going to use a Vi-style editor, then you have to recognize that it's merely a tool. The tool delivers the mechanics that you can use to manipulate text in new and powerful ways. You're not learning the tool so much as you're teaching your brain new ways of thinking. There are tons of different flavors of Vi out there, and they all deliver the tool to help you express your skills. You should be able to use any of them to great effect, because your brain understands how to forge code (or indeed any text) using the paradigm they deliver.

It's exactly the same when it comes to languages. If you're going to code in Python like you'd code in Java, then just code in Java. Python presents you with more and different paradigms than Java does, and if you're going to use Python, you should be using it because of what it allows you to express, not because it has less syntax than Java.

Learning a language or toolkit is easy, learning a paradigm is *hard*. But it's learning new ways of thinking that makes us great at what we do. Applying a new syntax to our old and often

¹²<http://www.vim.org/>

¹³There are many of us, of course.

outdated methods is of little value.

</soapbox>

OK, I'm hoping that wasn't too brutal. The point I'm trying to make with that little speech is to impress upon you that it's important to get more out of this than simply knowing how to write code with Akka. What you really should be getting out of this is much more important than that—if we both do this right, then you're going to learn how to think in ways that you may not have been familiar with before, and that's a *very good thing*.

Enough, already? I think so. Let's get you set up and ready to go, so that we can start learning all about this new paradigm.

Chapter 2

Concurrency and Parallelism

If you could sum up the features of Akka in one word it would be *concurrency*. There are many aspects to concurrency, not least of which is how to deal with the less-than-stellar aspects of real life. For now, let's focus on the main point: doing lots of stuff, doing it all at the same time, doing it safely and quickly.

The first part of this chapter is written for those not battle-hardened from years of writing concurrent code. If you already have a solid understanding of concurrent programming and have chosen to read this book because you merely want to move on to concurrent programming in Akka, then you can skim past these opening bits of concurrency theory. For those of you who haven't suffered through dozens of late nights tracking down elusive Heisenbugs, busting thread deadlocks, opening up synchronization bottlenecks, and writing the odd thread-safe concurrent class, then you might want to stick with me for this section. We're not going to cover so much that you'll feel like a battle-hardened concurrency programmer—I mean, this book does have to end at some point—but it should give you a decent grounding that will make understanding and using Akka much clearer.

2.1 Parallelism vs. Concurrency

These two terms get thrown around a lot and people often use them to mean the same thing. They're not, or at least we're not going to treat them the same. For our purposes, we're going to define them as:

Parallelism The act of modifying a seemingly sequential algorithm into a

set of mutually independent parts that can be run simultaneously, either on multiple cores or multiple machines. For example, you can multiply thousands of matrices sequentially but you can also do this in parallel if you break them up into a hierarchy of multiplications. Later on, in the section on Futures, we'll do exactly that and show how easy it is to do such a thing with Akka.

Concurrency This is all that other stuff, also known as *life*. It's the act of an application, which has many dependent or independent algorithms, running through multiple threads of execution simultaneously. The easiest example is that of a web service, such as Twitter. Twitter is highly event driven, taking in tweets from millions of concurrent users as well as events from its own internal systems. All of this stuff happens concurrently.

These aren't strict definitions and there are as many people that would disagree with them as would agree with them. I define them here in order to clarify what I mean when using the terms, but also to contrast the "purity" of the two situations.

Parallelism is meant to be a highly controlled situation. We code an algorithm with the intention that it runs in parallel (either through a specific piece of code we write, or naturally via a particular language or library's core functionality). Concurrency is the result of *stuff that happens*. We have no idea when clients will make their requests, what they're going to ask, or when they need responses. We just need to be able to deal with life as it comes and whatever it may impact in our application.

For ease of language, we'll assume that when we talk about concurrency, we also include parallelism. However, when we say *parallelism*, we are talking about parallelism plain and simple and do not include concurrency.

Akka has solid solutions for both parallelism and concurrency.

2.2 A Critical Look at Shared-State Concurrency

We've already said that shared-state concurrency can be the less-than-great solution when it comes to writing concurrent software. What is it about treating our data like a children's ball pit and our algorithms like a gaggle of sugar-infused toddlers that makes it hard to scale, debug, and understand?

And why does a system that doesn't look like that require so many lines of code that it's still difficult to debug and understand?

The Product

We'll illustrate some ideas with something that's a pretty common occurrence: the modeling of a User. We'll also model something else that's pretty common: software evolution. When you start writing your app, the requirements are small and you, therefore, build small. As the requirements grow and the audience increases, you pile on the code and the features. The day you need to switch from sequential programming to concurrent programming, all hell breaks loose.

In the Beginning

When you start out, you have a nice little User class that looks about the same as any other User class you'd find in the "Hello World" of User classes:

```
public static class User {  
    private String first = "";  
    private String last = "";  
  
    public String getFirstName() {  
        return this.first;  
    }  
    public void setFirstName(String s) {  
        this.first = s;  
    }  
  
    public String getLastName() {  
        return this.last;  
    }  
    public void setLastName(String s) {  
        this.last = s;  
    }  
}
```

A few months or a year goes by and you get to the point where you have a few threads. Everything's cool until you start to see some weird stuff happening with your output. Every once in a while some names get messed up. How hard could it be, right?

The First Concurrency Round

What's happening is that your concurrency isn't allowing your changes to be visible between threads at the right time, so you toss in some synchronized versions of your methods.

```
public static class User {  
    private String first = "";  
    private String last = "";  
  
    synchronized public String getFirstName() {  
        return this.first;  
    }  
    synchronized public void setFirstName(String s) {  
        this.first = s;  
    }  
  
    synchronized public String getLastName() {  
        return this.last;  
    }  
    synchronized public void setLastName(String s) {  
        this.last = s;  
    }  
}
```

This helps, but someone points out that using volatile would be better, so you do that instead:

```
public static class User {  
    private volatile String first = "";  
    private volatile String last = "";  
  
    public String getFirstName() {  
        return this.first;  
    }  
    public void setFirstName(String s) {  
        this.first = s;  
    }  
  
    public String getLastName() {  
        return this.last;
```

```
    }
    public void setLastName(String s) {
        this.last = s;
    }
}
```

That looks nicer. Now things are cooking!

The Real Problem Shows Up

The visibility of your changes is now awesome, but something new has shown up, and this is the real problem. Every once in a while, in the tradition of the Heisenbug, you retrieve a name that doesn't exist. You've tracked down what you think is the offending line of code:

```
System.out.println(user.getFirstName() + " " + user.getLastName());
```

But every once in a while you see this:

Spider Lantern

You shake your head a bit and have a look through your database for a “Spider Lantern” but you can’t find one. You find “Spider Man” and “Green Lantern,” but not “Spider Lantern”.

You really hope that it’s just some weird interleaving of output on the terminal, but it’s not. You’ve got a bigger problem.

You have code running on a thread that is trying to change someone’s name from “Green Lantern” to “Spider Man,” but the method by which it has to do it is pretty messed up:

```
user.setFirstName("Spider");
user.setLastName("Man");
```

In a concurrent system, there are many CPU cycles between those two lines of code where something can sneak in and grab the user’s first name and last name. It grabs the new first name, “Spider,” and the old last name, “Lantern,” and spits them out. Damn.

The Problem's Solution

There are many ways to solve this problem; the good ones involve changing your API. However, you'd have to change so much code that it just doesn't seem worth it, so you try the cheap way out.

```
import java.util.concurrent.locks.ReentrantLock;  
  
public static class User {  
    private ReentrantLock lock = new ReentrantLock();  
    private String first = "";  
    private String last = "";  
  
    public void lock() {  
        lock.lock();  
    }  
  
    public void unlock() {  
        lock.unlock();  
    }  
  
    public String getFirstName() {  
        return this.first;  
    }  
    public void setFirstName(String s) {  
        try {  
            lock();  
            this.first = s;  
        } finally {  
            unlock();  
        }  
    }  
  
    public String getLastName() {  
        return this.last;  
    }  
    public void setLastName(String s) {  
        try {  
            lock();  
            this.last = s;  
        } finally {  
            unlock();  
        }  
    }  
}
```

```
    }  
}  
}
```

The ol' stand-by revolves around locks. You toss some concurrency locks around the problem and figure that if "getters" try to grab the lock and someone else has it, then you're golden. What you'll do is change the one line `println` to lock the whole guy first:

```
try {  
    user.lock();  
    System.out.println(user.getFirstName() + " " + user.getLastName());  
} finally {  
    user.unlock();  
}
```

And that totally works! Except that it doesn't. It reduces the window of the race condition but it doesn't completely stop the following from happening:

```
// Thread 1  
user.setFirstName("Green");  
  
// Thread 2  
try {  
    user.lock();  
    System.out.println(user.getFirstName() + " " + user.getLastName());  
} finally {  
    user.unlock();  
}  
  
// Thread 1  
user.setLastName("Lantern");
```

If that happens you have the same problem. In order to get around that, you need to lock during setting as well.

BLURGH!

It's just plain ridiculous. To fix this problem, you have to change the API, use some sort of Database (DB) transaction, or something else that's hideous.

And this is just this one issue; race conditions and concurrency issues can show up in *far* more subtle ways than this. We didn't even look at deadlocks, which could have been quite interesting if we had chosen to use two separate locks for this problem. And imagine if this were a library that you had published to the world, and not just made as a convenience for yourself.

If you haven't experienced this before, trust me, it's not the sort of thing you want to waste your time on.

Threads

Now that we've covered the main problem that programmers have had for so long with shared-state concurrency, we can move forward and look at some of the machinery that helps us run our stuff.

In our modern applications, threads are taking care of the heavy lifting required to keep our code running concurrently. Some people seem to think that threads are *cheap*. They're really not. Threads are an incredibly expensive resource to just start spending like you're dealing with Brewster's Millions.¹ Some concurrency frameworks of the past even thought it was reasonable to spin up a new thread for every incoming network request. The rule of thumb with those was to make sure your app didn't get too many incoming network requests. That's pretty silly.

I've also seen people make their apps go "faster" by spinning up multiple threads to do some work in parallel and then kill them when it's time to stop them. If there are 200 incoming requests, they'll happily spin up 10,000 threads to do their work for them, and this can happen on-and-off every couple of seconds! Threads are not meant to be used this way. Really.

This gets worse once those threads start blocking each other with locks, synchronized blocks, or other concurrency mechanisms. The shared-state concurrency model can turn your threading methods into spaghetti really fast.

Thread Pools

If you don't want to be spinning up threads manually, then what's the better option? Well, it's thread pools. Thread pools are important in concurrent programming, and are equally important when programming with Akka, although we don't often use them directly.

¹Yeah, it was a pretty bad movie, but I was a kid when it came out, so it was awesome.

To get any concurrent work done within a single process, you need to have threads. There are several drawbacks to using threads directly:

- They have a fixed life cycle. If you put a `java.lang.Runnable` on a thread instance, then when that `Runnable` completes, the thread dies right along with it and can't be restarted.
- They take time to start up. Creating a thread certainly doesn't have a zero cost when it comes to creation.
- They're certainly not free when it comes to memory usage.
- There are operating system limits on these things; you aren't free to create an infinite number of them.
- You pay a huge cost in the management of threads with respect to *context switching*. A thread needs to run on a processor, and if it isn't currently allocated to one, then the OS needs to remove one from a processor and put another one in its place. Moving all that data around is also expensive.

To eliminate and/or hide all of these problems, we use thread pools. Java has created a set of reusable thread pools for you that have many of the wonderful aspects of thread pools, which developers have created for themselves over the years.

The thread pool creates a managed layer on which your concurrent methods can execute. They ensure that the system is being used efficiently, so long as you're not specifying thread pools of an unreasonable size or amount. Using them helps you avoid creating and destroying threads by yourself all the time, and it ensures that concurrent work gets throttled, to a certain degree.

The Thread Balance

One challenge of managing threads is to ensure that you have enough, but you don't have too many. If you have 2 cores and 10,000 threads, you're probably not doing yourself any favors. The reason for this is due to the context switching that the OS must do on your behalf.

All of the threads you have must run at some point; otherwise, they'll starve for attention from the CPU, which is certainly something that must be avoided. So the OS slices them off sometime. In order to do that, it must

freeze the running state of a given thread, pull it off of the CPU, store it somewhere fun, and then put the new thread in its place for a few microseconds, and then switch it out to make room for the next one.

Maximizing Processor Time

All of this context switching takes time and you want to avoid it as much as possible. If you're multiplying 200,000 matrices together on a machine with 8 cores, then the right decision is definitely *not* to break the work up into 1,000 threads of 200 matrices each. In this situation, you want approximately 8 threads running approximately 25,000 matrix multiplications each. That will minimize your context switching and maximize the amount of time that your application spends on the processor. It's not an exact science due to the fact that a general-purpose computing OS will always be doing more than just dealing with your application, but in this case, the approximation isn't too bad. A couple more threads here might help.

CPU vs. IO

There are really two kinds of work in most applications: that which requires the CPU and that which performs synchronous IO. Asynchronous IO isn't much of a problem because you don't have anything tying up threads while the IO is taking place, so we only concern ourselves with synchronous IO.

Clearly, if you have an application that is 100% bound to the CPU (let's say it's only calculating π), then you can keep the number of threads down to a value that's commensurate with the number of cores. But if you're performing a fair bit of synchronous IO, then what?

IO is a problem because it's *slow*. While the IO is performing, your application isn't busy; it's just waiting for the IO to complete. And while it's waiting, it's tying up a thread in your application. If you've only allocated 8 threads, and they're all doing IO, where's your CPU work going to go?

It's often a good idea to separate your IO from CPU work by creating separate thread pools. The IO pool will be "large" in comparison to the CPU pool since the threads on the IO pool spend most of their time avoiding the CPU. You can then tune the IO pool independently from the CPU pool. In general, this is a real pain, which is why the world is really starting to get serious about asynchronous IO.

Blocking Calls

To make sure that you're not making a ton of blocking calls, your language or your toolkit needs to help you. Back in the days before C++11, we didn't have closures. As such, creating non-blocking code was a big problem. You ended up doing things like this:

```
class MyBusinessLogic { public: // ... stuff ...

    bool ourCallbackFunction(const SomeResult& results) { // do stuff return
        results.ok(); }

    // ... stuff ...

    private: void someCode() { someObject.call(param1, param2,
        bind(&MyBusinessLogic::ourCallbackFunction, this, _1)); } };
```

And that's when it's generally easy. `MyBusinessLogic` calls `someObject`, which allows `someObject` to call back into `MyBusinessLogic`, but what if that's not the end of the story? What if we want more chaining, more decision making, and more delegates? It just gets worse and worse.

Java isn't much better since everything is a noun (i.e., class or instance thereof), even though you can instantiate anonymous classes. C++ at least can have simple verbs (good ol' functions) that we can pass around if needed.

This is why we have so many blocking calls in our code today—doing anything else is just *too damn hard*.

NonBlocking APIs

A few years ago, NodeJS² came on the scene with a single-threaded solution to our blocking call problem. NodeJS's bread and butter is the idea that if all IO is asynchronous, then code in user-land is free to execute and react to IO events. Your user-level code is on a single thread so you don't need any synchronization primitives to protect its data, and with the help of Java Script's closure mechanisms, we get the tools we need to write a ton of non-blocking code (albeit, heavily nested at times).

NodeJS and Akka differ in many ways, but they both make no blocking a huge goal. If you can swap out a Java library that blocks in its IO for one that doesn't, do it.

²<http://www.nodejs.org/>

We won't discuss NodeJS anymore; NodeJS and Akka are trying to solve *very* different problems in the software world and it's simply not reasonable to compare them. I bring up NodeJS at this time since it's gained quite a following, and it may help ground you in the importance of non-blocking IO.

2.3 Immutability

Why all this talk of mutability vs. immutability? When you get into conversations about concurrency, especially with people who have a strong grasp of functional programming, you'll invariably hear about the value of immutable data structures. By now it's obvious, right? All of those synchronization primitives that we've grown to love and hate are there because we need to protect that valuable mutable data! If we didn't have mutable data then we wouldn't have synchronization of operations on that data since the only operation that can be performed is a simple *read*.

If you're not comfortable with immutability, then you're sitting there right now saying that an application isn't useful unless it can also *write*. You probably won't get any arguments from anyone on that, certainly not from me. But just because a data structure is immutable doesn't mean that your program's state can't be altered. Those who have created our immutable data structures are very clever indeed, and have ensured that most modifications of those data structures are fast and deterministic.

Immutability Implies an Altered Programming Model

As far as I can surmise, what I'm about to tell you is some sort of secret. Well, maybe *secret* is too strong of a word, but people don't seem to be making it all that clear either. The data structures that you use in imperative programming tend to be mutable and thus when you start coding concurrency, you tend to opt for shared-state concurrency with mutable data structures; it's just a natural extension to the programming you've always done. People tend to initially agree when someone says, "immutable data structures are better," but then their brain starts to rebel. They imagine all of those data access object (DAO)-like classes they've coded over the years and all of the side-effect-based programming that has worked so well and they can't see how to fit that into an immutable world. There's a reason for that: *it doesn't*.

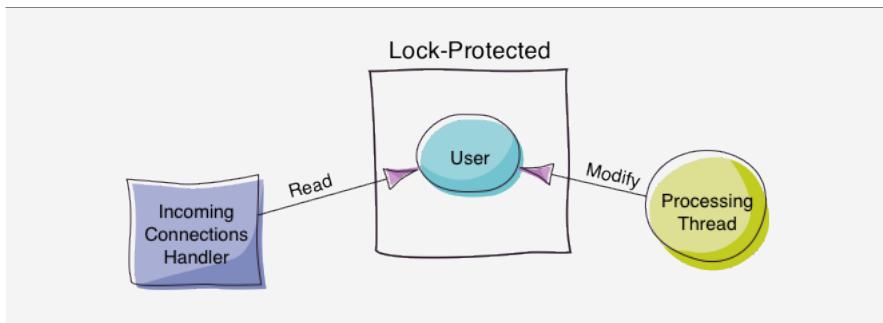


Figure 2.1 · One way we’re used to writing our shared-state concurrency: create a piece of the domain model, give access to it from a couple of different threads, and code with side effects, ensuring a proper rendezvous with locks.

Given what we see in Figure 2.1, how could we possibly take advantage of immutability? The short answer is, you can’t. The longer answer is that you could be clever, make the User immutable, and provide a protected reference to it instead. The reference would be global and you could synchronize that reference, so when you change the reference to a new, altered User, everyone gets updated with the new reference atomically. But that’s not what we mean when we talk about using immutable data structures.

It’s this minor confusion that’s concerning us at the moment. Coding with immutable data structures isn’t just about swapping out the mutable ones in your code for immutable ones and then grabbing a beer. It just doesn’t work that way. Immutability is part of the design of your application more than it is about the code of your application—your algorithms are different.

If we make our User immutable, then we need to constantly work with new copies of the object, which should ideally be stack-based as opposed to objects we put on the heap and share references to. When we have immutable objects such as this, then our code tends to process objects in the style of Figure 2.2.

Immutable objects that create new versions of themselves directly from mutation dovetail very well with the idea of FluentInterfaces,³ which arguably already provide an interface that’s easier to understand. But there’s

³<http://www.martinfowler.com/bliki/FluentInterface.html>

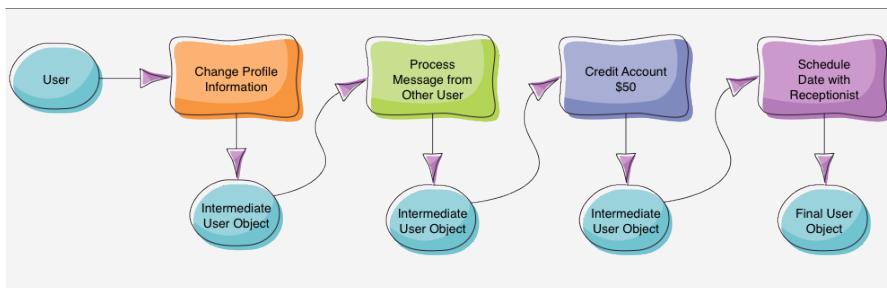


Figure 2.2 · When making changes to our User object, we tend to code like this: we make alterations using a pipeline of commands instead of several modifications to the same object.

another benefit to working this way. Assuming that the pipelined operations of 2.2 are independent, then it's entirely reasonable to parallelize them, as seen in Figure 2.3.

If we try to do the same kind of parallelism with a mutable object, protected by locks, then we get the type of situation we see in Figure 2.4. Even though we've gone to the trouble of scheduling work to be done on multiple threads, we'll still perform this work, in effect, linearly. Not only are we not getting the effect we want, we're also wasting the threads we're consuming in the first place. What's to blame? The User object.⁴

We can try to fix the problems with our shared-state User object in one of two ways:

Lock Less Take the locks out. Doing so, of course, will probably give us all the problems with race conditions and object corruption that the locks were meant to avoid in the first place. So, probably a bad idea.

Lock More Yup. If we put more locks into the User class, then we can be more selective about what gets locked when. In theory, we could open this class up to full parallelism in this situation if we can lock individual bits of data discretely. For example, lock the credit data separately from the profile data.

⁴There's another problem with the shared mutable object as well, which falls into the realm of improper CPU cache use, but that's beyond the scope of this book.

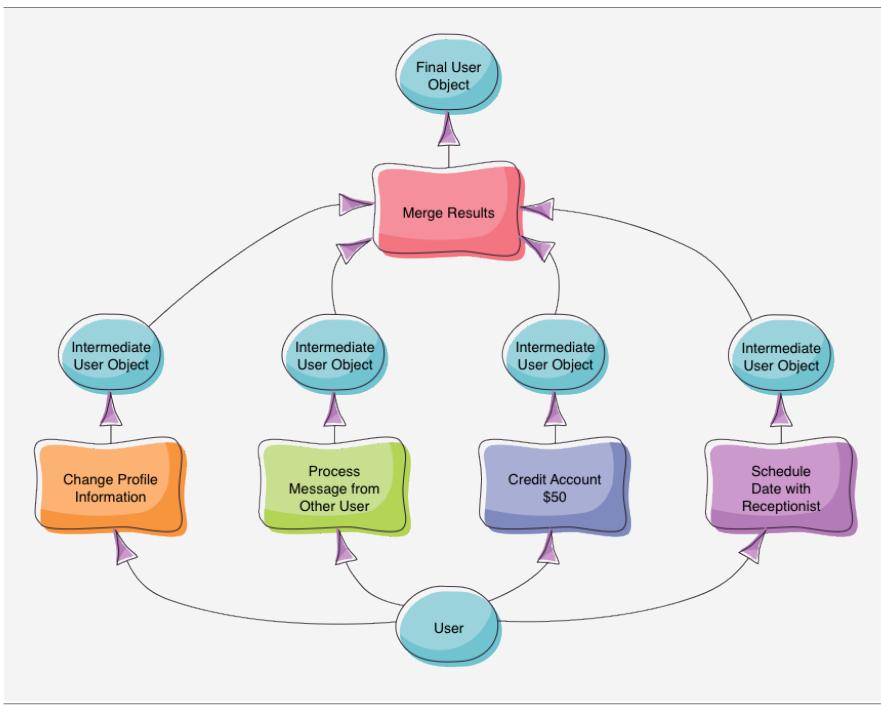


Figure 2.3 · The same operations on the `User` object are done in parallel here. We must recombine the state changes into a final object representation, of course, but we can see that the input to the operations and the output of the operations remain exactly the same as the linear version.

Painful. This is what those guys mean when they say, “immutable is better.” You can go parallel much more naturally with immutable data structures than you can with mutable ones.

But most of those who say it is better are *functional programmers* not imperative programmers. While working with immutable data is certainly possible and not necessarily uncommon when it comes to imperative programming, it’s not as natural and certainly not easy to wrap your head around. The good news is that you don’t have to become a full-fledged functional programmer to be able to take advantage of the power of immutable data structures. Akka is fully capable of blending styles of imperative and functional programming together, which allows you to take advantage of the best of both worlds.

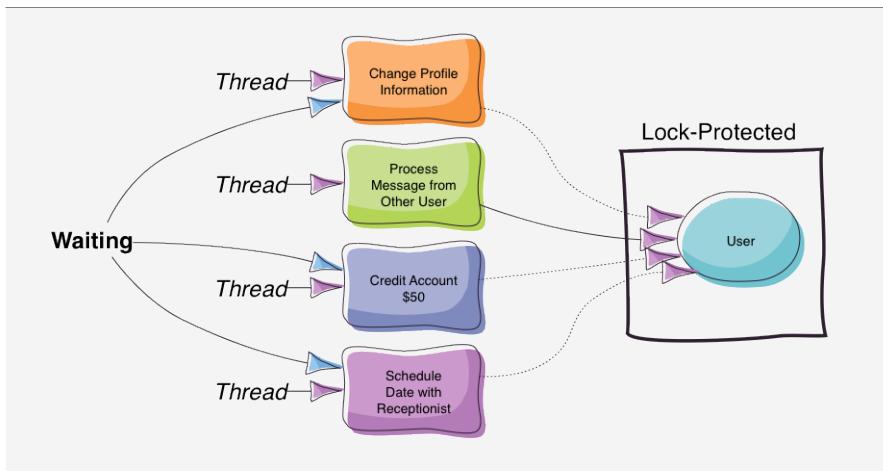


Figure 2.4 · Here's an attempt to get the same type of parallelism from a mutable User object; in this case, the User object is protected by locks now. Even though we've spread the work out to multiple threads, we're still linear in our speed simply because only one thread can work at a time. Not only do we not get parallelism, we are also wasting threads.

Immutable Data Structures

OK, so now that we have you more acquainted with the nature of working immutably, we can start to look at some fundamental ideas with immutable data structures to help you understand more of the programming model that surrounds them. We'll also bust a myth or two at the same time.

The Linked List

The quintessential data type for immutability is the Singly Linked List. It's simple to write, easy to work with, clear to understand, and extremely fast if used appropriately.

```
val list = List(1, 2, 3, 4, 5)
```

If you give the value `list` to someone, then you don't need to protect it. There's no way you can modify anything that `list` points to, either directly or indirectly. However, you can give that reference to as many people as

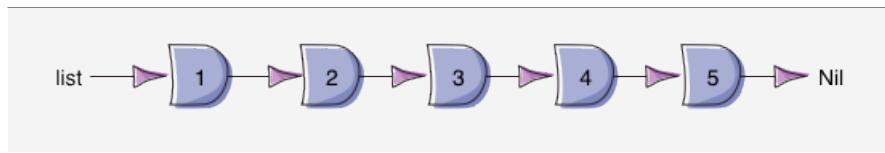


Figure 2.5 · A good ol' linked list: our symbol name `list` points to the head of the list and it always ends in `Nil`.

you like and not worry about it at all. No copies need to be made and no synchronization is required.

What's more is that you can give pieces of that list to other pieces of code and they get the same nice deal:

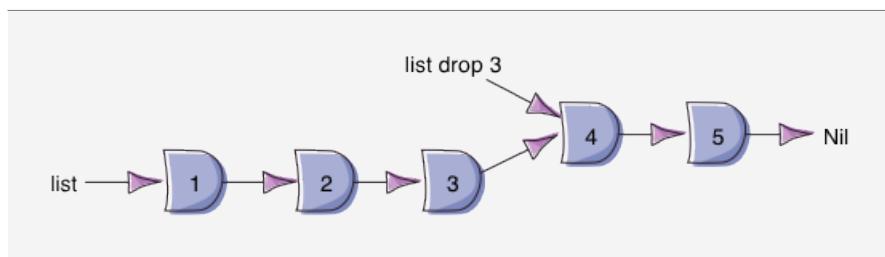


Figure 2.6 · The recursive nature of List coupled with its immutability lets us quickly create a new list from a pre-existing one by simply dropping elements from it.

```
val list = List(1, 2, 3, 4, 5)
Future { workWithEntireList(list) }
Future { workWithTheLastTwo(list drop 3) }
```

Here we spawn two separate pieces of work using Akka's Future object. You don't need to know how it works, just be aware that we're using it to create parallel operations. So now we have two separate functions working in parallel on the exact same immutable, unprotected data structure, but we're viewing it in different ways. Again, we don't need to care what these functions are doing with the shared data structure because there's nothing that they can do to it that will cause any harm.

Modification Can Be Fast

One major concern people have with immutable data structures is that, when you do want to modify them, you have to take a big recursive copy of whatever the data structure represents and then modify your piece. As with most things in life, general statements rarely apply. It's true that often times, given an immutable type, modification can require *some* copying, but copying the whole thing is generally the exception to the rule. If you're finding that you're making full copies a lot, then you're probably using the wrong data type—or you're just doing it wrong.

We can see that the opposite is sometimes true with mutable data. If we have a reference to something that someone else has a reference to, then there are times when we must make a *defensive copy* of that data in order to ensure that someone doesn't change it under our feet. In fact, I've seen many sections of code where people make defensive copies simply because they are finding it difficult to reason what's going on in their code. Maybe they need a defensive copy, maybe they don't, but they do it just to be sure that everything's going to work out alright. When you're immutable, you don't have this problem; your code is always easy to reason about with regards to concurrent access to data.

So, it's certainly no slam dunk that mutable data is faster under modification than its immutable counterpart; not when you take the full breadth of the application space into account.

For instance, *prepend*ing to a list is a fast operation. We can grow this list from any point we choose by simply prepending elements to various places.

```
val list = List(1, 2, 3, 4, 5)
Future {
    6 :: 7 :: (list drop 2)
}
Future {
    9 :: 8 :: (list drop 4)
}
```

The power of immutability in the list and the idiomatic usage of its most performant modification operation (*prepend*) has saved us the cost of copying, both in CPU cycles and memory consumption. We can do this safely in a concurrent environment because the list is immutable.

What about *appending*? Appending sucks. Here's why.

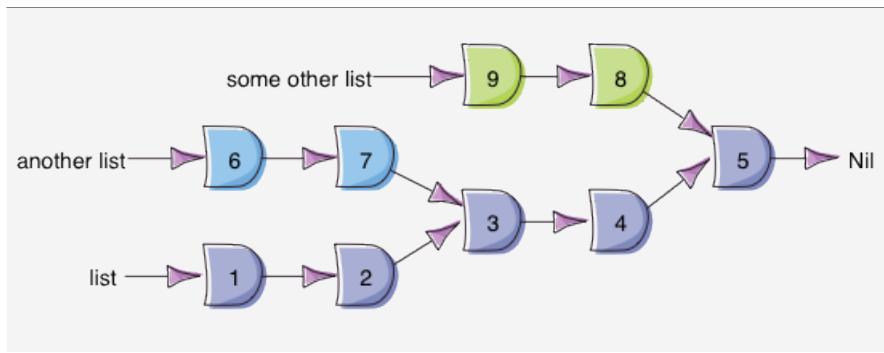


Figure 2.7 · We can easily create new Lists from existing Lists by prepending new elements at various locations. Each List is truly its own distinct List—it just happens to be that the implementation can use the underlying storage because the underlying storage is immutable.

- To append to a list, you need to modify the last element, but you can't because it's immutable.
- So, you make a copy of that last element and modify the copy.
- But now your newly copied element has nobody pointing to it. You need to get its old previous element to point to it, but you can't because it's immutable.
- You need to create a new copy of the previous element in order to point to the new element.
- And so on, all the way back to the head of the list.
- Appending to a List creates an entirely new list.

Programming with immutable data structures is *different* than programming with mutable ones. If you find yourself appending to lists all the time, then you're using the wrong data type—use something else (e.g., a Vector). But if you find yourself running a ton of recursive algorithms over sequences and you want to run those algorithms concurrently over the same sequence, then a list is probably your best friend.

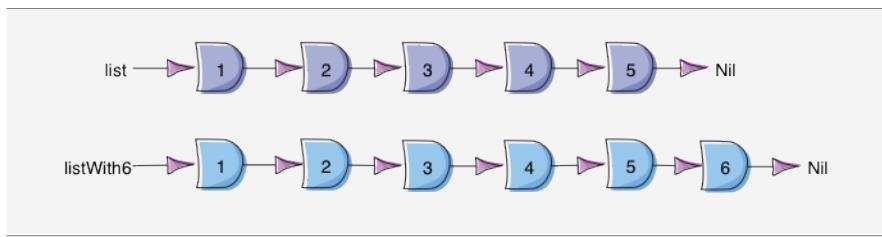


Figure 2.8 · The act of appending a 6 to the list results in an entirely new list with a new pointer to the head of the new list. The old list, of course, remains entirely unchanged.

Immutable Maps

Lists are easy; what about maps? Without going into great detail, I can show you that you can also quickly modify immutable maps. Granted they aren't quite as fast as their mutable counterparts or as easy as prepending on a list, but they're still pretty darn fast. And always remember that we *never* have to make a defensive copy of an immutable map.

The basic notion revolves around cloning-affected data and reusing what you can. Since maps are generally implemented as trees, we can illustrate with a simple tree-like structure, avoiding the whole key/value pair thing you have with maps since that doesn't make much of a difference to the illustration. Let's keep it simple. Have a look at our basic map (tree) in Figure 2.9.

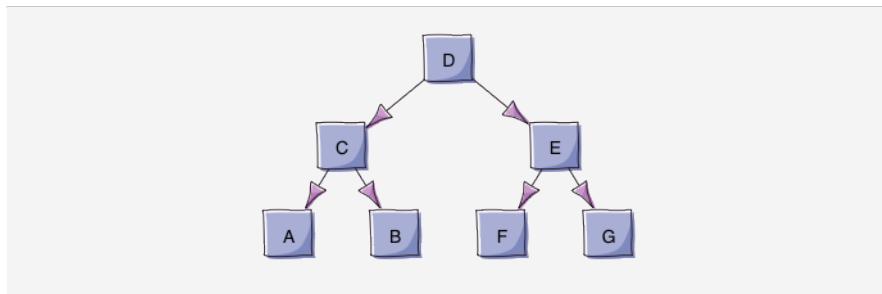


Figure 2.9 · A really bad representation of a map; we haven't attached values to this map just because it's easier to see, but a tree tends to make a pretty decent map implementation.

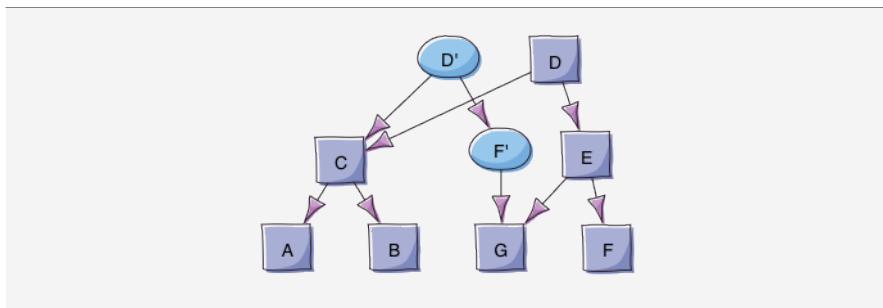


Figure 2.10 · After removing the E from the original map, we now have two distinct maps. The original is represented by the purple squares and the new one, without E, is represented by the blue ovals with D' at the root.

If we want to remove E, then we need to have something that looks like Figure 2.10. Anyone who has a reference to D won't notice a thing. All of the original nodes, indicated as purple boxes, still point at purple boxes so nothing has changed. The person who asked for the modification would have operated originally on D, but the return value from the modification would have been D'. The act of removing E has caused the modification of F and D only, indicated by the blue ovals.

The Speed of Immutability

Is making that map modification as fast as just deleting E from a mutable map? No, it's not. The time to remove E from a mutable collection is $O(\log(n))$ and the time required to remove E from an immutable collection is around $O(2\log(n))$. (Yes, I know that's $O(\log(n))$ but bear with me for a second).

But what about in a concurrent environment? How much does it cost then? Well, that's an interesting question and it depends largely on how your application is designed and how it's working with the data, so we need to speculate just a tad.

If you have one global mutable map and everyone's got a reference to it, then the cost of modifying that might be very high. Why? Synchronization, that's why.⁵ Everyone who wants to have a look at that map is going to

⁵Unless it's a lock-free map, which is cheaper but still has its own costs associated with it.

have to wait while the guy who wants to modify that map makes his modification. That means that threads are tied up, which is costly. Can we put a quantifiable cost on it right now? No, but we know it's there.

If the map is immutable, then clearly we may not have to pay any cost at all. Now that I graciously stuck on the cost of the immutable map modification is starting to look pretty good, right? The accepted cost of an immutable map modification is really $O(\log(n))$ anyway, since we ignore the constant multipliers. However, since the non-quantifiable cost of waiting threads in any given situation can actually be quite significant, I think we can at least argue that the immutable map can be superior in a concurrent situation.

But That's Not Equivalent!

OK, so it's not exactly an apples-to-apples comparison. The mutable version had the virtue that everyone could see the modification. Well, assuming that that's what you want (and it's not a given that it is what you want) then we do have a bit of a problem.

We're back to saying that coding with immutable types is different again. Immutability doesn't generally wash with traditional imperative or OO-style programming because the data we work with tends to stay in a global-ish sort of concept. All of our objects hold on to some piece of data, and references to those objects ensure that they're long lived. Since those objects are long lived, their life cycles demand that their internal data be modified. If they never changed, then people wouldn't see them do much, and what's the point of that? While you don't *have* to code this way in an OO or imperative style, it does turn out that this is most often the case.

When you break out of that mindset, and put the immutable data outside of an object context, then we find that the data gets passed around much more to short-lived functions. A function takes the data, mutates it, and then passes it off to someone else, who possibly mutates it again and returns it to the caller, who returns it to his or her caller, and so on. When the program behaves like this, then we can immediately see how functions can run concurrently.

Memory Consumption

Another question that comes up with immutability is with respect to memory consumption. Mutation implies copying and a lot of orphaned references that eventually head to the reclamation department; i.e., the garbage collector. Don't concern yourself with it. At the very least, don't concern yourself with it until you can see that it's a problem. We won't go into modern garbage collector design (mostly because I'm far from an expert), but the objects that need to be reclaimed are usually in the first generation of the memory manager and are handled very efficiently. There's no pain involved here.

And, again, let's not forget about the memory consumption that's a factor of all those defensive copies you need to make of your mutable data.

Advanced Immutability

Modification under immutability isn't always easy. Sure, we have a map or a vector or a list or some other standard library type that we can use and all of that magical modification is simply done for us. But what about for our own immutable types? For example, look at [Figure 2.11](#).

We just want to add Fred to the meeting that's happening on Wednesday. But if we do that, then we'll be modifying the list of attendees, which is immutable. So, we'll have to return a new list, and we'll have to get the meeting object to point to it, which means we'll need a new meeting object. If we do that, then we'll need to modify the list of meetings to point to it, and that means we'll need to modify Wednesday so that it points to the new list of meetings. Finally, we'll have to modify the Week object to incorporate the new version of Wednesday.

Whew! Fun stuff...

Is there a way to do it? Sure there is, but we're not going to cover it here.⁶ In Scala, you can use case classes with the helpfully generated copy method that can ease your work a bit, but don't stick with it for too long as it'll probably make your life rather hellish in the end. There's a method for doing this that has been encapsulated in Scalaz by implementing a version of the Lens concept.

I highly encourage you to investigate the Lens when you come across your next complex immutable type modification. You can find Scalaz at <https://github.com/scalaz/scalaz> and several good tutorials on Scalaz

⁶Please insert mental smiley face with an evil grin...

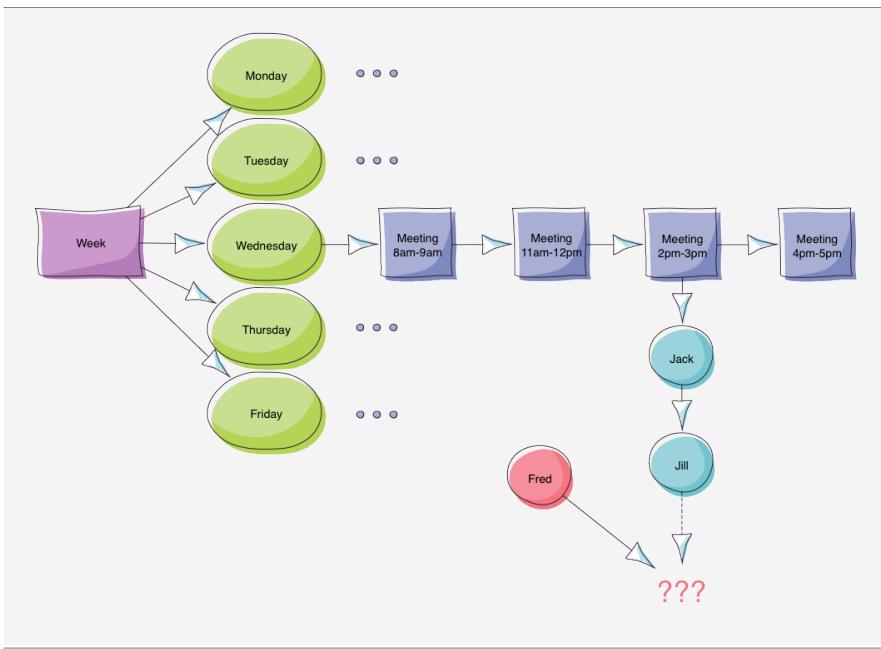


Figure 2.11 · This shows an immutable calendar-like structure of meetings and we want to add Fred as an attendee to the meeting from 2pm to 3pm on Wednesday. If the structure were mutable, then this would be a piece of cake. Immutability makes this more difficult.

and the Lens by typing `scalaz tutorial` and `scalaz lens tutorial` into Duck Duck Go.⁷ I'm also hoping to see several useful concepts such as this covered in the yet-to-be-completed book, *Functional Programming in Scala* written by Rúnar Bjarnason, Paul Chiusano, and Tony Morris (Manning).

2.4 Chapter Summary

That was quite the whirlwind tour of concurrency, parallelism, the pains of shared-state concurrency, and the wonders of immutability. We could have gone on and on, but the goal of all of that is to lend some grounding to much of the topics you'll see in the rest of this book, not to discuss all of

⁷<http://duckduckgo.com/>

the ins and outs of concurrency outside of the Akka context. We'll see a *lot* of asynchronous code and asynchronous design, which is all geared toward attacking the problems we've discussed in this section.

The issues with threads and blocking vs. non-blocking calls (of your own code), along with a whole host of other issues, will become easy in comparison to what we've been doing thus far in our careers. On top of that, it will open up a whole new world of possibilities that you may never have thought of before.

So, let's strap in and get going!

Chapter 3

Set Up Akka

Before we go any further, let's set up so that you're all ready to start coding in Akka.

3.1 Scala Setup with SBT

The Simple Build Tool (SBT) has become a lot easier to work with over the last year or so, and there's a nifty little script that does for SBT, what SBT does for code. I'll show you how I work with SBT, and if you have your own way of doing it, then go for it.

Download the SBT Bootstrapping Script

You can find the script on GitHub that front-ends SBT so that you don't even have to install SBT itself. The script is part of the `sbt-extras` project on GitHub, so you can use `git clone` to pull down the project, but I'll show you a dead simple way. I highly recommend that you put the script somewhere into your PATH and wherever that is, it should be writable directly by you; (i.e., do not put it in `/usr/bin`). I choose to put it in `$HOME/bin`, since I've already put that in my PATH and it's certainly writable by me.

```
> curl https://raw.github.com/paulp/sbt-extras/master/sbt > ~/bin/sbt  
> chmod 755 ~/bin/sbt
```

That's all there is to it. I did say that this had become pretty easy, huh?

Create an SBT Project

Next, we need to create the SBT project. Let's put it into your home directory in a subdirectory called `code`:

```
> mkdir -p ~/code/AkkaInvestigation
> cat <<EOH > ~/code/AkkaInvestigation/build.sbt
name := "AkkaInvestigation"
version := "0.1"
scalaVersion := "\scalaversion"
resolvers += "Typesafe Repository" at "http://repo.typesafe.com/typesafe/releases/"
libraryDependencies ++= Seq(
  "com.typesafe.akka" % "akka-actor" % "\akkaversion"
)
EOH
```

Pull Akka Actors into SBT

Now we'll run `sbt update`, so we can let SBT do its thing:

```
> cd ~/code/AkkaInvestigation
> sbt update
[info] Set current project to AkkaInvestigation
  (in build file:/Users/quinn/code/AkkaInvestigation/)
[info] Updating {file:/Users/quinn/code/AkkaInvestigation/}default-6bd931...
[info] Resolving org.scala-lang\#scala-library;2.9.1 ...
[info] Resolving com.typesafe.akka\#akka-actor;2.0-RC1 ...
[info] downloading http://repo.typesafe.com/typesafe/releases/
  com/typesafe/akka/akka-actor/2.0-RC1/akka-actor-2.0-RC1.jar ...
[info]  [SUCCESSFUL] com.typesafe.akka\#akka-actor;2.0-RC1!akka-actor.jar (9583ms)
[info] Done updating.
[success] Total time: 14 s, completed Feb 16, 2012 5:23:32 AM
```

We're now ready to go! Let's test things out for real by creating simple code that we can compile and run. We'll use an Akka Future just to test and make sure that everything can be compiled and run properly.

Take It for a Spin

Put the following into the `src/main/scala/zzz/akka/investigation/MainAkka.scala`:

```
package zzz.akka.investigation

import java.util.concurrent.Executors
import scala.concurrent.{Await, Future, ExecutionContext}
import scala.concurrent.util.duration._

object MainAkka {
    val pool = Executors.newCachedThreadPool()
    implicit val ec = ExecutionContext.fromExecutorService(pool)
    def main(args: Array[String]) {
        val future = Future { "Fibonacci Numbers" }
        val result = Await.result(future, 1.second)
        println(result)
        pool.shutdown()
    }
}
```

You don't need to understand that code. What matters right now is that it compiles and runs:

```
> sbt run
[info] Set current project to AkkaInvestigation
  (in build file:/Users/quinn/Dropbox/book/AkkaInvestigation/)
[info] Compiling 1 Scala source to
  /Users/quinn/Dropbox/book/AkkaInvestigation/target/scala-2.9.1/classes...
[info] Running zzz.akka.investigation.MainAkka
Fibonacci Numbers
[success] Total time: 3 s, completed Feb 16, 2012 6:46:38 AM
```

If you see the magic phrases, "Fibonacci Numbers" and [success], then go grab a beer and let's get moving.

Chapter 4

Akka Does Concurrency

Akka is a domain-neutral concurrency toolkit designed for the purposes of building scalable, fault-tolerant applications on the JVM. It provides many tools to help you achieve your goals, and in this chapter we'll start to understand how to work with those tools so you can start building your high-quality, highly concurrent applications. At the end of this chapter, you should be in a better position to begin *thinking* in the Akka paradigm.

4.1 The Actor

When we get past what Akka *is* and start looking at what it *contains*, it's the Actor that pops its head up first. The Actor does most of the heavy lifting in our applications due to its flexibility, its location independence, and its fault-tolerant behaviour. But even beyond these features, there's an interesting consequence of the Actor design—it helps make concurrency development more *intuitive*.

Your day-to-day world is full of concurrency. You impose it on yourself as well as the people around you, and they impose it on you. The real-world equivalents of critical sections and locks as well as synchronized methods and data are all naturally handled by yourself and the people in your world. People manage this by literally doing only *one thing at a time*. We like to pretend that we can multi-task, but it's simply not true. Anything meaningful that we do requires that we do just that one thing. We can pause that task and resume it later, switch it out for something else to work on and then return to it, but actually doing more than one thing at a time just isn't in our wheelhouse.

So what if we want to do more than one thing at a time? The answer is pretty obvious: we just use more than one person. There's not much in the world that we've benefited from that wasn't created by a gaggle of talented people.

This is why Actors make our application development more intuitive and our application designs easier to reason about: they're modeled after our day-to-day lives.

Concurrency Through Messaging

If you want a coworker to do something for you (such as write a bunch of tests for your code because you're simply too busy playing NetHack¹ to engage in such trivialities), what do you do? You send the poor sod an email, of course.

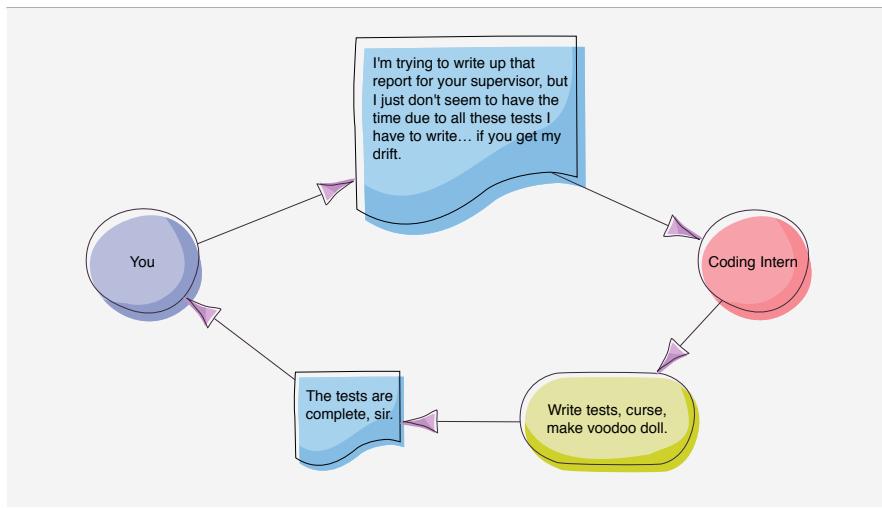


Figure 4.1 · Chances are you've either been the coding intern or you've been the other guy. If you've ever been the coding intern, then you'll be the other guy eventually. It's being the intern that makes the other guy—nobody knows whether the chicken or the egg came first.

It's just that simple. Get the right people in place, have a decent mechanism for shunting messages around (bonus points if they're durable), and

¹<http://www.netHack.org/>

you're good to go. Hell, if you could spawn enough interns you may be able to play NetHack all day, every day, and even get paid to do it.

Actors follow this model. You send an Actor a message that tells it to do something, which it does presumably quickly and well, and then it tells you what it did. You can scale this model out to thousands or millions (or billions?) of Actors and many orders of magnitude more messages and your applications are still reasonable, not to mention huge and fast.

Concurrency Through Delegation

Given what happened in [Figure 4.1](#), it would seem pretty obvious that we can delegate work from one Actor to another, but you can take this simple idea pretty far to achieve your goals. Since interns are just so wonderfully cheap, there's no reason we can't have a ton of interns chained to desks in a dark room somewhere churning out whatever it is they are supposed to churn out.

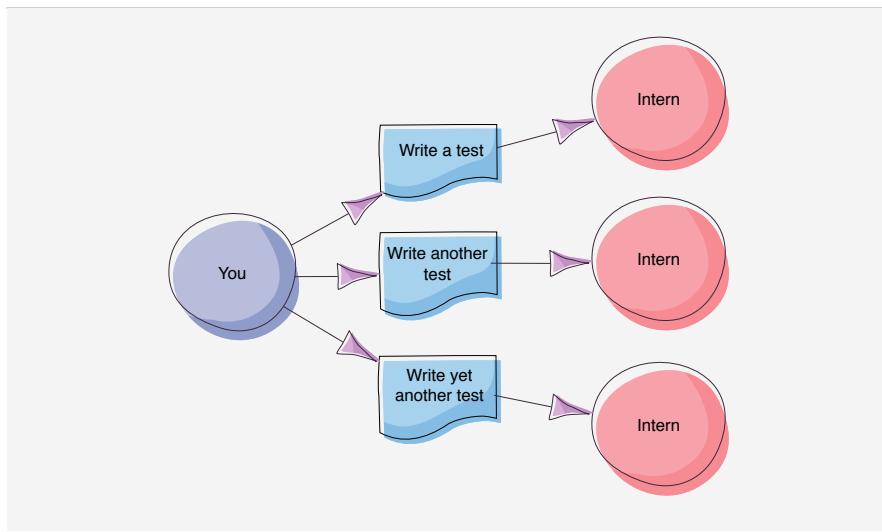


Figure 4.2 · Three interns can write three tests faster than one intern can write three tests.

But [Figure 4.2](#) is a pretty ineffective use of such a cheap resource. It might even be better to have a single goal in mind and set a bunch of interns to the task. They can each do it the exact same way, or they can all use a

different method for achieving the goal. You don't need to care *how* they get it done, just that someone gets it done before the rest. The intern who wins gets a decent report to his or her supervisor, and maybe even a job offer (although, seriously, you're pretty mean) while the other interns get a rather unfavorable letter sent to their supervisors.

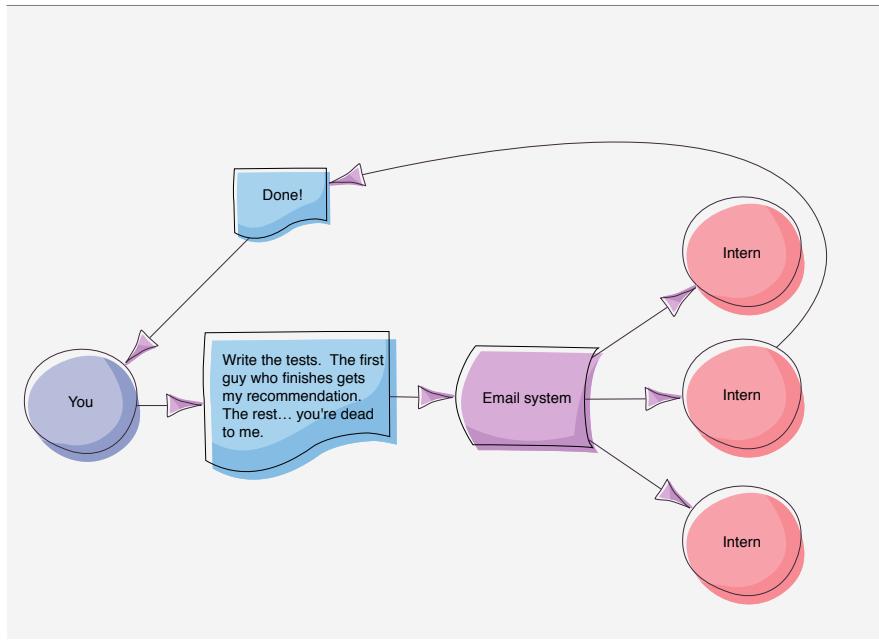


Figure 4.3 · It's quite possible that these interns have chosen different methods for implementing the tests. The second one seems to have chosen a decent toolkit, or didn't sleep, or is high on some sort of amphetamine, or... who cares? He won.

Figure 4.3 gets a particular job done quicker by burning resources with wild abandon. If you've got the people and they're not doing anything else, then why not give them some work to do? Sure, you might throw the results of their efforts right in the trash, but who cares? OK, maybe they'll care, but who cares about that?²

So what if your interns realize what you're doing and one of them decides to learn from your example? If he's got the resources available to him, then

²Jeez, you're *really* mean.

he's probably going to win if he follows your tactics. There's nothing to stop him from doing something like what's in [Figure 4.4](#).

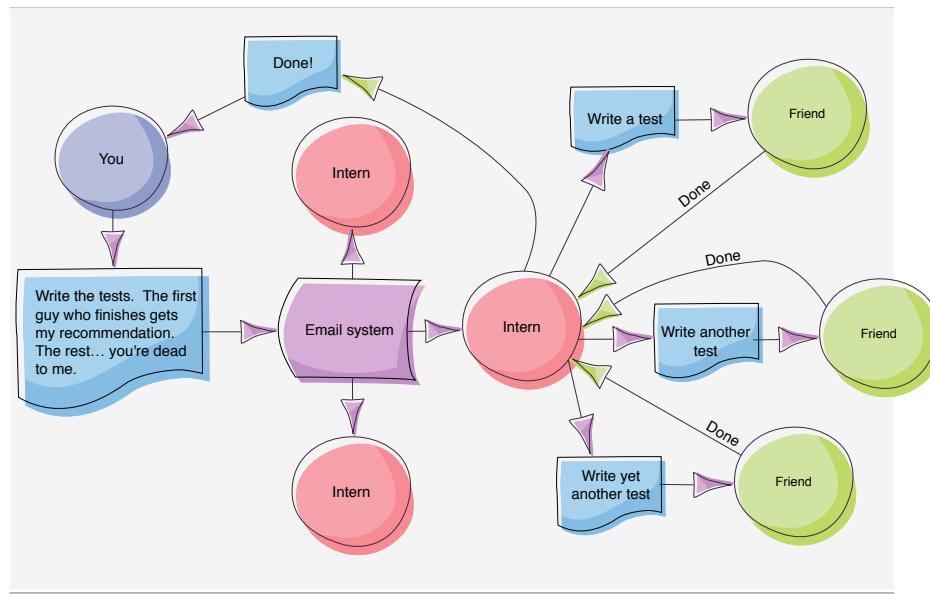


Figure 4.4 · So this intern is pretty smart, or at the very least, sneaky. There's nothing to say he can't do exactly what you're doing. Not only has he done it, he's hidden it by ensuring that his friends return their results to him so that he can send them to you. You're clueless.

Interestingly enough, the guy who won in [Figure 4.4](#) would probably be the guy who you hired, but he'd also be the guy that you fired because his friends would eventually get tired of working for free, and he'd be exposed as the lazy, slack-jawed worker he really is. Too bad for you.

Delegation for Safety

While we're on the subject of delegation, we should probably talk about one of the other advantages it provides: *safety*. When was the last time you heard of a sitting U.S. president heading out on a mission with a Navy Seal team to rescue one of his constituents from a group of terrorists? OK, maybe it's because the guy's seriously out of shape, or couldn't hit the broadside of a barn with a bullet the size of a fist from 10 paces out, but let's assume he's

awesome. He still wouldn't go on that mission. Why not? He's just too damn important. There are times when Actors are too important to go on dangerous missions, and when that's the case, we delegate the mission to someone else.

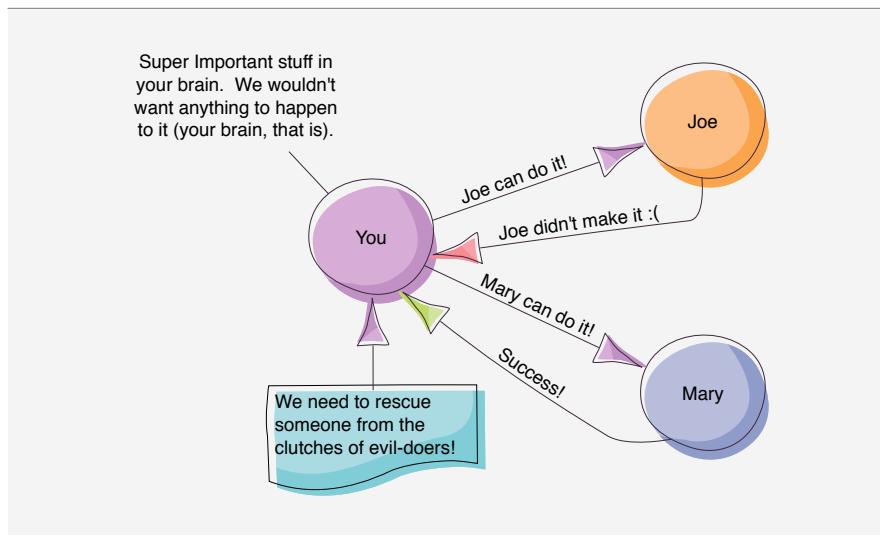


Figure 4.5 · Poor Joe. The evil doers were just too much for him. But this isn't a problem for you—you've got Mary! Mary can get the job done.

You're happy and safe in [Figure 4.5](#) because you can delegate the dangerous work to others. You may be mean, but you're certainly no fool! All of that cool information that you hold—the nuclear launch codes, the itinerary for that policy summit, your spouse's birthday, and all of that other important stuff—is safely locked away in your brain. Unfortunately, Joe didn't make it, but truth be told, his brain was full of quotes from episodes of Family Guy. Cool as that is, it's just not vital stuff.

While We're on the Subject of Death

We weren't explicit with [figure 4.5](#), but let's be perfectly frank about it—Joe died. It's unfortunate, but it happens. An Actor's life isn't always an easy or safe one but the point is that the Actor does have a life and along with it, a *life cycle*. We're going to see much more about the Actor life cycle later, and find ways in which we can hook in to its life cycle, as well as the life cycles

of others. What's interesting at the moment is that there is a life cycle and that it (sort of) matches what we're used to in real life. The people you work with, the interns you are continuously beating on (metaphorically speaking, of course) had to be born at some point, and there will come a day, sooner or later, when they're going to give up their ghost.

But the death of an Actor is nothing to get upset about. Actor death can be a very good thing. In an Akka application, there's always someone looking out for Actors; someone's always got their back. It's not really the fact that they die that is so great, it's the fact that someone is (or many someones are) there to watch it and do something about it that is. There is, at most, one guy around to clean him up, resurrect him, ignore him, and let someone else figure it out or just ignore him altogether. We can literally just pretend that nothing bad actually happened.

When death occurs, there's only one guy who manages to do something with the deceased but there are many guys who can react to that death and take action upon notification of it. Presumably that notification is something along the lines of what we saw in [Figure 4.5](#). The notification in that case was the unfortunate message: *Joe didn't make it*. You were able to understand the implications of that message and send Mary to take care of it. If you had sent her first, Joe would probably still be with us, but hey, you can't always make the solid decisions.

There's nothing wrong with creating Actors for the sole purpose of putting them in harm's way. In fact, it's a very good thing. So don't be afraid of giving birth to an Actor only to have him meet his ultimate demise microseconds later. He's more than happy to give his life in the service of his parent's good.

You also shouldn't be afraid to use death to your advantage. Very often, an Actor can self-terminate when its work is completed and that death can be a signal to anyone watching that the time has come to move on to the next operation.

Doing One Thing at a Time

Actors only do one thing at a time; that's the model of concurrency. If you want to have more than one thing happen simultaneously, then you need to create more than one Actor to do that work. This makes pretty good sense, right? We've been saying all along that Actor programming draws a lot on your day-to-day life experiences. If you want work done faster, put more

people on the job.³

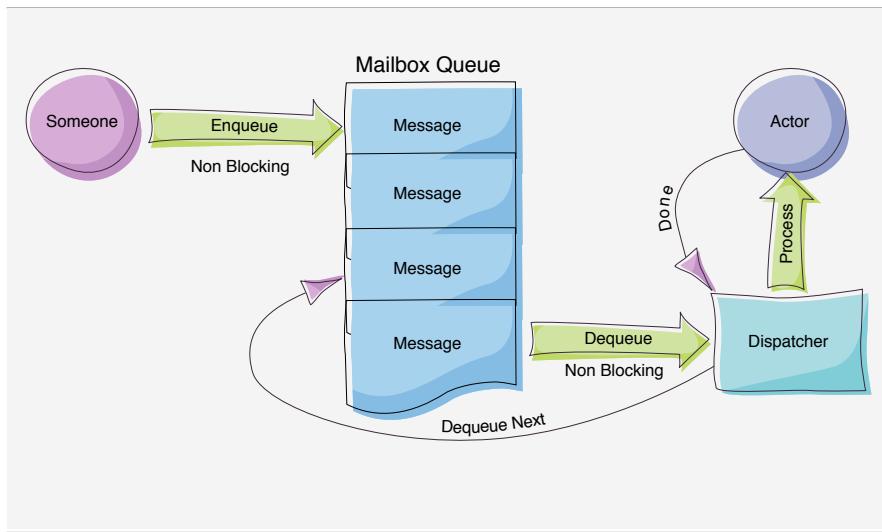


Figure 4.6 · The message processing for Actors is not unlike the types of message processing that you might be familiar with if you've had any experience with message pumps inside GUI frameworks. This is a simplified view of the Actor constructs in Akka and it doesn't give a very good indication of the power behind it, but that's a good thing for the moment. If I blew your mind now, then you wouldn't have the mental capacity to continue reading.

Figure 4.6 provides a taste of what the Actor structure looks like with respect to processing things.

1. Messages come into a mailbox through (unless you want otherwise) a non-blocking *enqueue* operation.
 - This allows the caller to go about his business and doesn't tie up a waiting thread.
2. The enqueue operation wakes up the dispatcher who sees that there's a new message for the Actor to process.

³Those of you who are thinking of the *Mythical Man Month* have earned a cookie, but forget about it. Actors are not bound by such trivialities.

- In the case of [Figure 4.6](#), we can see that the Actor is already processing a message, so there's really nothing for the dispatcher to do in this case, but if it were not processing anything at the moment it would be scheduled for execution on a thread and given the message to process.
3. The dispatcher sends the message to the Actor and the Actor processes it on whatever thread it was put on to do the work.
- During the time when the Actor is processing the message, it's in its own little world.
 - It can't see other messages being queued and it can't be affected by anything else that's happening elsewhere (unless you've done something relatively silly, that is).
 - The Actor is just head-down doing what it needs to do. If it takes a long time, then it's going to tie up that thread for a long time. It's just that simple.
4. Eventually, the Actor will finish processing the message.
- The mere fact that it's complete will signal the dispatcher and it can then pull the next message off the queue and give it to the Actor to start the cycle all over again.

Details of the Akka implementation are subject to change, so that may not be 100% accurate, of course, but the basic notion is correct. The whole point is that it's the messages that matter and the processing of those messages happens one at a time. The act of queueing them and dispatching them is entirely non-blocking by default, which allows threads to be truly dedicated to doing work. Akka does a good job of staying out of your way so that when you have scalability problems or bottlenecks in performance, it's *your fault*. And that's the great news: if it's your fault, then you're in complete control of fixing it.

What's more is that the processing of those messages happens in complete isolation from other work. It's simply not possible for anything to happen that can screw with what the Actor is doing right now (again, unless you do something really silly). You don't need to lock the Actor's private data, you don't have to synchronize a set of internal operations that must be atomic, all you have to do is write your code.

The Message is the Message

Have you ever heard the phrase, “The medium is the message?”⁴ I’m sure it made great sense to Marshall McLuhan when he said it and I’m sure that it resonates with a bunch of other people, but it always seemed pretty silly to me. You know what really makes a good message? A *message*. Thankfully, Actor programming is really all about the message. It’s the *message* that travels from place to place, and it’s the message that carries the really interesting state. For our purposes, it’s also the message that carries the type. A strongly typed message allows us to write code that makes sense to the compiler, and if we can make the compiler happy then we’re probably going to be pretty happy ourselves.

But let’s step back for a second. What does it mean to say that a message carries the interesting state? Aren’t Actors the important mechanism here? Isn’t it Actors that *do things*? Of course it is, but if you remember back to [Section 2.3](#) you might recall that objects that *change* can be a bit unwieldy. If the entire state of a running algorithm is contained inside the messages used to execute that algorithm, then we are free to give that work to any Actor with code that can process that state. What’s more is that the Actor, which may be processing the algorithm at any given moment, isn’t burdened by weird internal data that it has to keep alive during complex message processing.

Aggregating RSS Feeds

To illustrate, let’s say you want to collect data from several RSS feeds, aggregate them into one interesting content feed, and then send them off to somewhere else. What’s more is that you want to make sure that you can scale the problem to multiple threads when you become successful and have to do this for a thousand users simultaneously. You don’t care about making a single user’s requests go quickly, you care about increasing your capacity for the number of users on a given machine and their given requests, so we’re going to do an individual user’s set of requests sequentially.

[Figure 4.7](#) shows us what an algorithm would look like that behaves this way. Note that the messages that travel between different invocations of the Actor have two separate sets of data in them: the list of sites to pull data from, and the results of pulling that data. Initially, the list of sites is “full” (i.e., has N things in it) and the list of results is empty. As the algorithm

⁴http://en.wikipedia.org/wiki/The_medium_is_the_message

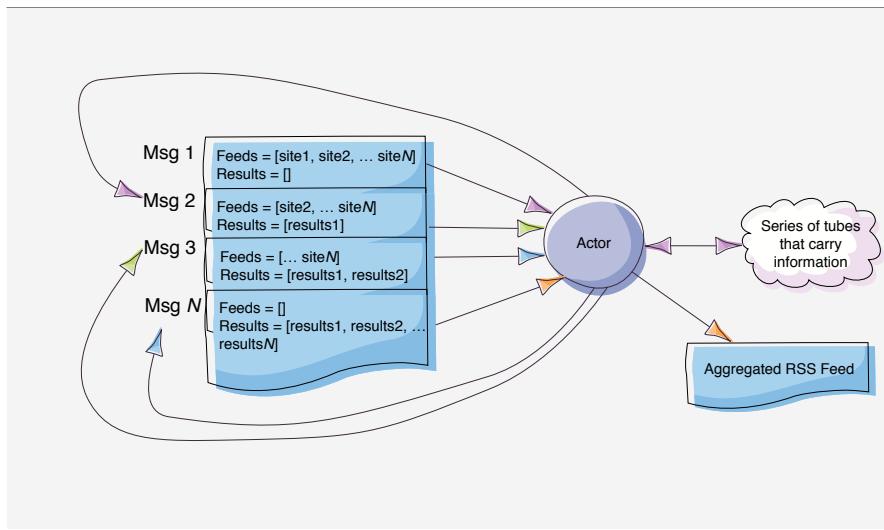


Figure 4.7 · When we're processing a series of RSS feeds, we can carry the full input and output in the messages themselves. The business logic in charge of downloading the RSS feeds from the series of tubes is held inside the Actor, but the information about what to download and the results of the download travel in the messages between the iterations of the algorithm.

progresses, the list of sites to visit becomes smaller and the list of results becomes proportionally larger. Eventually the Actor gets a message where there's nothing to do; the list of sites to visit is empty. When it gets this message, it triggers different behaviour that collects the results into a single aggregated feed and then publishes that forward to someone else (which we don't illustrate).

The fact that we've broken the problem up into individual messages ensures that we give back the executing thread at semi-regular intervals. This keeps the system responsive and lets it handle a greater capacity of users. The Actor that's doing the processing could even manage a bunch of different users for us if we want, because it's clueless about what's happening between invocations; all of the state is held inside the messages themselves.

Message Immutability

The messages that are used in the RSS aggregation algorithm from [Figure 4.7](#) are immutable. This will keep coming up—it came up before in [section 2.3](#) and it will come up again. In order for Akka to be able to do the cool things that it does, and to work quickly and deterministically, it needs you to make sure that your messages are immutable. It's not just a good idea, it's the Akka law. Break it and you break yourself. Don't break yourself.

Strongly Typed Messages, Loosely Typed Endpoints

The fact that the message is so important doesn't diminish the Actor's role, but it does underscore the stark difference between the two. Messages are key to the model of the Actor application but it's the Actors that facilitate messaging.

One of the things that really helps Actors deliver on power is the fact that they're loosely typed; every Actor looks like every other Actor. They may behave differently or may accept a different set of messages, but until someone sends those messages, you'll never know the difference.

Now, before we extol the virtues of the *untyped* Actor, we've got to get something out of the way: Akka has a *typed* Actor as well. We're going to ignore it in this book because, while the typed Actor has its purpose, it's the flexibility of the untyped Actor that drives a lot of power into an Actor program. To further explore this idea, let's look at how the Actor and the messages interact.

An Actor Is Behaviour

One of the ways to view an Actor/Message pair is to see them together as loosely equivalent to a function. [Figure 4.8](#) shows one side of how you can picture this; the Actor contains the behaviour that is driven forward by the message. The message is the symbol we use to describe the particular behaviour that the Actor will execute (such as "Buy from the Grocery Store"), and in order to execute that behaviour, the Actor will probably need some data (although not necessarily). This data is held in the body of the message.

This decoupling of behaviour from the invocation definition is also not unlike a polymorphic function call. An interface can declare the method signature but you can use any number of implementations of that interface to implement the method signature in whatever manner is reasonable for those

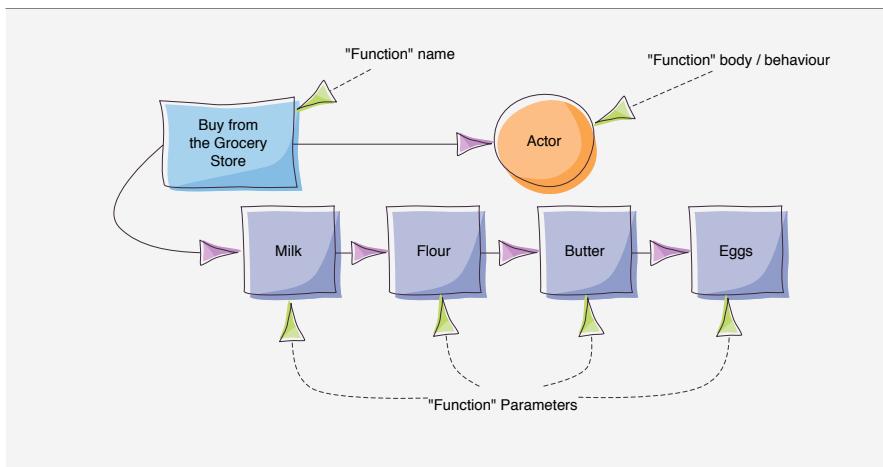


Figure 4.8 · One way of viewing the Actor/Message pair: Only as a full set of Message, Message body, and Actor do we realize the full notion of a function. When the Actor processes the message, it realizes what behaviour it needs to invoke via the message, and uses the message body to drive the behaviour.

implementations. However, with an untyped Actor you have more flexibility due to the fact that the Actor does not need to implement the strongly typed interface. The Actor must only be able to process a message and that message is the strongly typed entity.

Now, as we said above, this is only part of the story. We can't just throw the word *function* around as though its meaning were so easy to tailor to our needs. Functions, in most people's definition, evaluate their input data to output data. Actors almost by definition have side effects. To truly view them as functions instead of void procedures, we need to complete the picture. The next step is realizing that Actors can send messages as well. Figure 4.9 shows the obviousness of that idea.

We're really stretching the analogy now. We can think of the Actor as returning the new message, if that helps you wrap your head around some of the concepts of Actors. However, we must recognize that it only really works as an analogy when the entity that receives the returned message is the same one that sent the request, as in Figure 4.10.

In truth, the Actor isn't necessarily *returning* the message; it's really just *sending* the message to some other entity, which is probably another

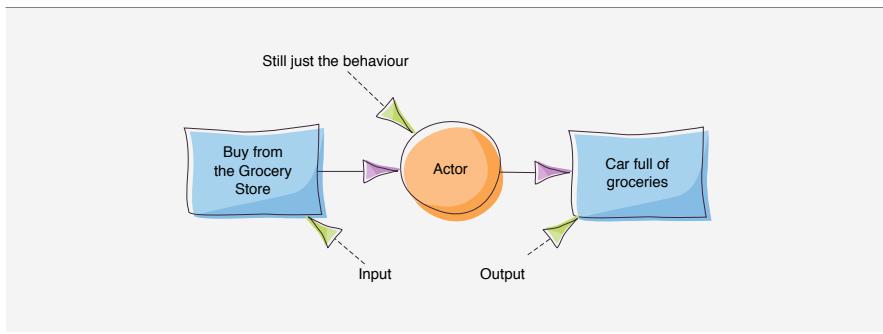


Figure 4.9 · The Actor is now an input/output function... mostly.

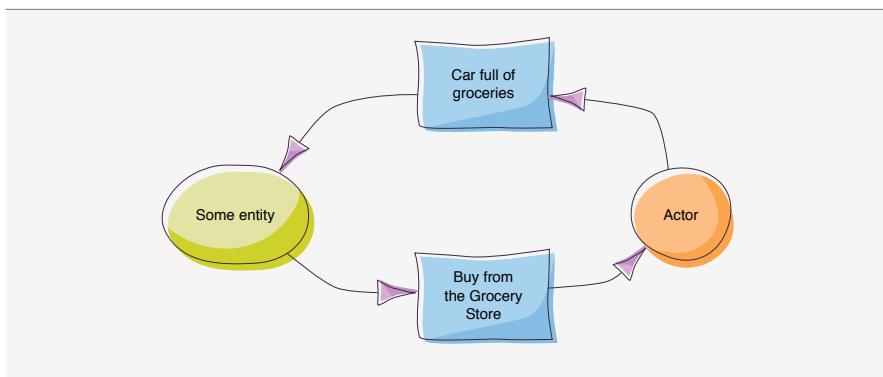


Figure 4.10 · The simplest case where the initiating entity receives the eventual response from the Actor, thus completing the function analogy.

Actor. The entity to which he's sending it may be the initial Actor that made the request or it might be something else entirely. The Actor itself doesn't really need to know anything about who sent what or who he's sending things to. All of this plumbing can be set up on-the-fly by anyone who interacts with the Actor. For example, let's have someone tell an Actor to get some groceries, but to deliver them to someone else, as depicted in [Figure 4.11](#).

Because we've spent most of our programming lives writing functions, it's important to try and draw a parallel to them. To a certain extent, there is a relationship there, but it breaks down fairly quickly, as you can see. An Actor is *behaviour* and we can wire up that behaviour however we see fit. This wiring can be simple, as in the case of [Figure 4.9](#) or it can be far

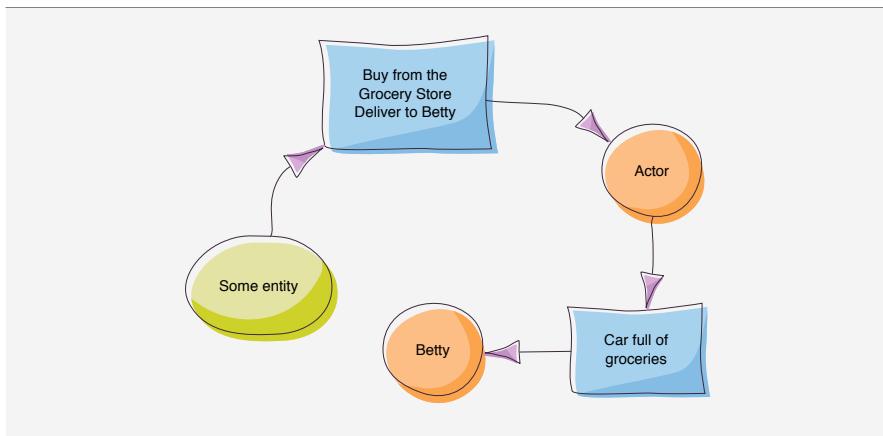


Figure 4.11 · The Actor that's getting the groceries isn't under any contract to send results back to the original guy making the request. In this case, the original request passed a reference to Betty in the message, which directs the Actor to deliver the groceries to her.

more complicated than anything we've seen thus far. Not only that, it can be entirely determined at runtime. You can dynamically create new Actors to handle work that wasn't able to be statically constructed in your editor. This is part of the Actor paradigm; we need to get your brain to move beyond the analogy of the function and start thinking in terms of Actors. That's part of what this chapter's all about.

Add Behaviour by Adding Actors

One of the excellent things you can do with Actors is to add behaviour to an algorithm by inserting Actors into the message flow. For example, let's say you've got a system that distributes a bunch of events to Actors and you want to start recording those events to disk. Rather than mixing behaviour into a single class or inheriting functionality, in the style of OO, we have a different alternative. With untyped Actors, you can get away with putting a *tee*⁵ in between the source and destination Actor, as depicted in Figure 4.12.

This sort of thing happens all the time in Actor programming. When a problem presents itself, you tend to solve it by way of creating more Actors

⁵As in the good ol' Unix *tee* program.

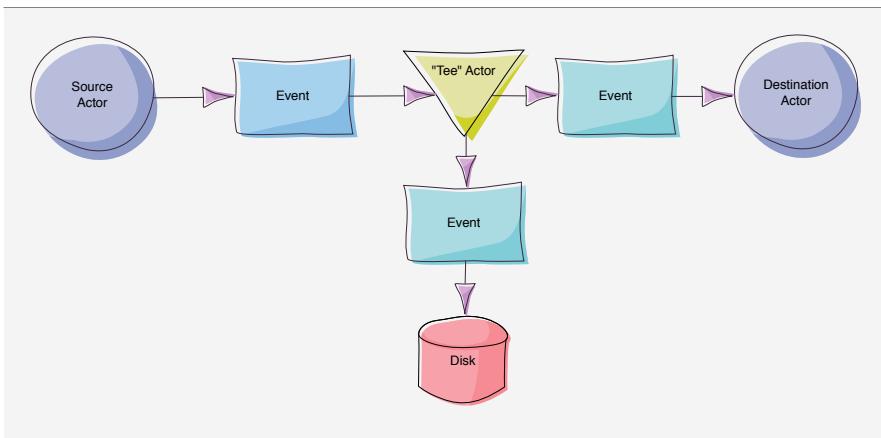


Figure 4.12 · A simple pattern for replicating messages to go to multiple destinations from the source. In this case, we're just sending it to where it was supposed to go originally, as well as streaming it to disk.

with discrete behaviour than by adding functionality to existing Actors. It's the fact that the Actors are untyped and that the real information is contained within the messages that makes this sort of flexibility possible.

Rather than modifying N classes or functions by putting in a callout (e.g., to a logging function) or refactoring to a new and very specific class hierarchy, it may be quite natural to slide a new Actor into the message flow to let it intercept certain messages, reroute them, duplicate them, transform them, or whatever else is required by your situation.

The separation of typed-ness between the strongly typed message and the loosely typed Actor brings power to your designs and your code.

Don't Be Scared

You're a type junky. I get it. I'm a type junky too. One of the major reasons I write in Scala is because it gives me a strong type system, and that lets me know that my programs are sane when the compiler spits them out. How can type junkies live in the untyped world of Actors and still manage to sleep at night?

Web servers are untyped as well, and when we write a web service we're sending messages to an untyped endpoint. This doesn't make us cringe because there are so few of them. I only have a few URLs that I code against so

it's easy to keep it straight in my head and I can be quite sure that messages aren't going to the wrong spots. But when you have an Actor system, you don't have a few endpoints, you have tens to thousands to *millions*. Millions of untyped endpoints can give type junkies the shakes.

Don't sweat it. I have no numbers or theory to convince you that no sweat should be shed over this lack of type safety; all I can say is that I've never sent the wrong message to an Actor. This is probably due to the fact that Actor programs are so easy to reason about; when things are clear, confusion doesn't exist, and it's confusion that makes us mess stuff up.

But, even if we do send the wrong message to an Actor from time to time, it's going to be worth it; so worth it, you won't even think about it. If you're a type junkie, let it go. You'll still use the type system for a ton of stuff and it will be the sweet safety net that it's always been. But when you leave it behind for this one type of object, that will free you up to do some incredible things.

Reactive Programming

Actor programming is *reactive programming*. Another way to say this is that it's *event-driven programming*. Event programming has been with us for a long time, but it's arguably never been epitomized as much as with Actor programming. The reason for this is that Actors naturally sit there just waiting for something to happen (i.e., waiting for a message).⁶ It's not the act of *sending* a message that's important; it's the act of *receiving* one that really matters.

There are two major reasons for this:

1. People like to think in terms of timing. They want to know how long it takes for something to happen after a message is sent.
 - This is a very natural expectation. But in Actor programming, you have to put this into context.
 - What does it mean for the message to be sent in the first place?
 - Is it in the Actor's mailbox? Is it on a queue ready to be sent to the mailbox? Is it traversing a network, and is there a store-and-

⁶OK, they don't "wait" in the traditional sense; that would tie up threads needlessly and that would be downright dumb.

forward system that it's been handed off to? Is the queuing of a message a synchronous or asynchronous function?

- Once it's in the mailbox, what does that mean? Is it one of 20,000 other messages waiting to be processed, or would the mailbox be empty otherwise? Is it in a priority mailbox and is it so low that it's going to be trumped for the next little while?

Clearly, the act of sending something isn't really all that deterministic. So when you start trying to put bounds or meaning on it with respect to timing, things get very murky very quickly.

2. People also like to attach significance to the sending of the message much like they would a function call.

- If we say `Math.exp(-5.0)`, then the act of invoking that function has meaning. The code that underlies the `exp` function is executed on the current thread. Dead simple.
- But, due to all of the reasons discussed above, we can't say the same about queueing a message in an Actor's mailbox.

The act of sending a message is important, since without it nothing would happen, but it's the reception of that message that carries true meaning in Actor programming. When the Actor pulls that message out of its mailbox and begins processing it, then it has truly *received* that message. It's at this time when meaning is applied in the sense of execution.

These reasons illustrate why *reception* is the important part of message passing in an Actor system, but it doesn't make the reactive programming argument completely solid.

Well, you won't get a completely solid argument for it, since nothing is black and white in our complex world of software development, at least nothing at this level of complexity. What's important right now is that you start thinking along those lines, especially if you're not used to it. It's perfectly reasonable to code your Actors to react to events that occur in the system, which is something that isn't necessarily common in standard OO code (for example). It can be as simple as the difference between these two statements:

- Turn the car left.

- The steering wheel on the car has turned to the left.

In the first, someone issued a *command* or a directive that says to do something. In the second, someone posted an *event* that indicates a change to the state of the world. This change to the state of the world would result in the car turning to the left (we hope), which may cause another change to the state of the world, and so forth.

The difference between the two is subtle, but important. Actor programming isn't just about a set of tools, but about *thinking* differently about how you design and write your software. While you aren't going to spend all of your time writing reactionary code, there is some great potential for improving your designs by thinking in a more reactionary style in many cases.

4.2 The Future

In the early days of Akka, the Actor was the true headliner of the production, and the Future was mostly there to support the Actor. As time progressed, the Akka team built out the Future concept more and more, and now in 2.x the Future has come into its own. It has grown up into a full-fledged paradigm of concurrent programming that helps you solve tons of interesting problems with speed and grace.

Unlike the Actor, the Future should be much more familiar to most, so we'll be blasting past it a bit quicker than we did the Actor. But fear not, these are the early stages only; we'll be covering much more of the Future in later chapters.

Contrasting with the Actor

The Actor is not a silver bullet. There are many times when the problem with which you're faced isn't solved well with Actors. One of the easiest examples I find is the idea of multiplying a bunch of matrices together. It looks like what we've got in [Figure 4.13](#).

We would like to parallelize this computation in order to saturate all of our cores and/or all of our machines. To break the problem up, we can group the multiplications, evaluate them in parallel, and then multiply the results together to get one final matrix. [Figure 4.14](#) shows us a specific case of two groups of matrices, but we can generalize the idea to as many groups as we need.

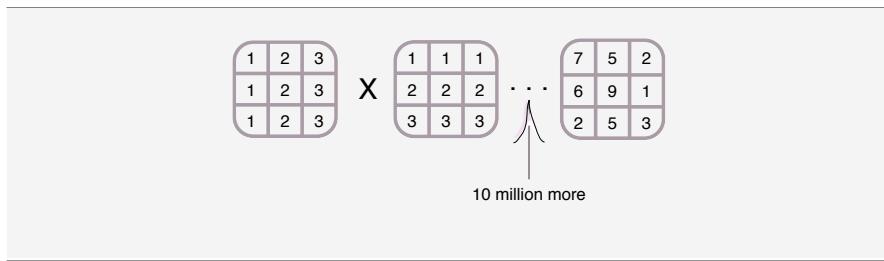


Figure 4.13 · A simple representation of 10,000,003 matrices that we want to multiply together into a final result.

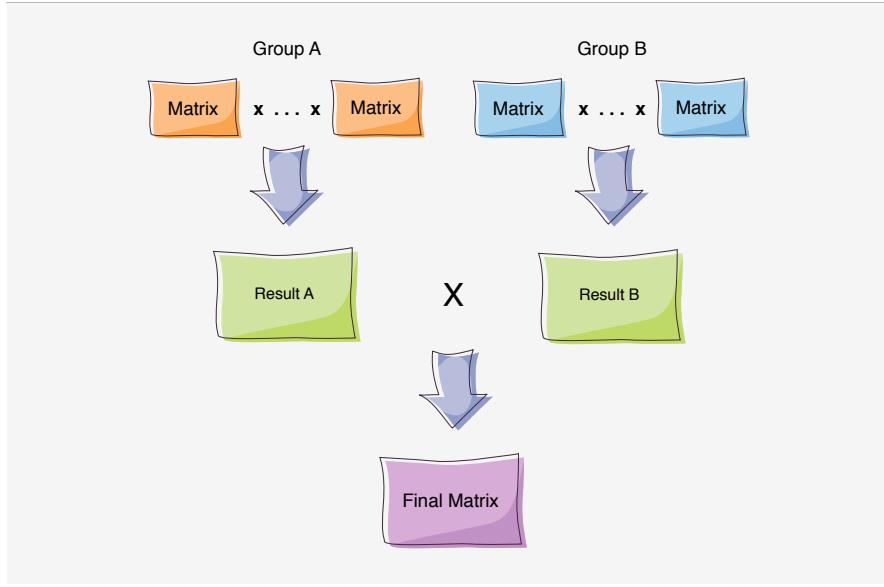


Figure 4.14 · Grouping a set of matrices to be multiplied into two groups: Group A can be multiplied together at the same time as group B. Once both results have been obtained, we can multiply the result into a final, single matrix.

There's a subtlety to Figure 4.14 that might not be obvious to you if you've never done this before. When multiplying matrices together, *order matters*. It's not the same as multiplying N numbers together, which you can do in any order you'd like ($5 \times 2 \times 7$ is the same as $7 \times 5 \times 2$). The dimensions of the matrices have to line up properly, and if you start shuffling the order

around, then you're going to find that the dimensions won't line up anymore or, if they do, you're not going to get the right answer.

The challenge here isn't the grouping and multiplying together of those groups since their ordering is already set for us. What's harder is taking the results and keeping them in the right order. You must multiply $A_{result} \times B_{result} \times C_{result}$... and so on. If we model this problem with Actors, then keeping the results in the right sequence is non-trivial. It's not brutal, but it's a pain. [Figure 4.15](#) shows the core of why it's a problem.

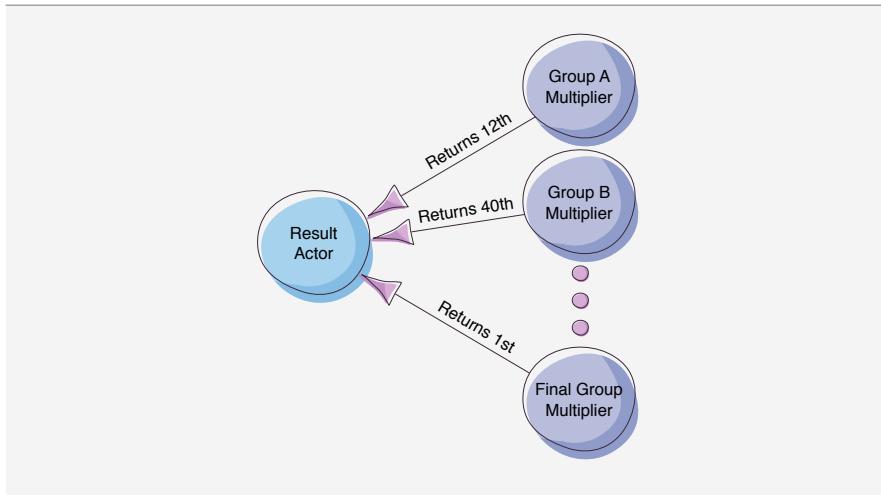


Figure 4.15 · When you give work to a set of Actors, they will complete it at non-deterministic intervals. This means that response messages will come back in what is effectively random order.

You can have a single Actor receive all of the matrices to be multiplied; it can then group them and spawn new Actors to multiply the groups. As each group completes, it can send the result back to the original Actor and it can store that result while it waits for the rest. But it can't just store it without thinking about where it needs to go. So you end up having to pass a group to an Actor and give it some sort of sequence number as well. When the result comes back, it must return the same sequence number so that the original Actor can slot it into the right spot. In addition, as each result comes in, the Actor must check to see if the latest result is actually the last result and, if so, it can then multiply the results together and then pass the final result off to someone else.

Whew! That's a lot of work. It's certainly doable, but it's way more of a bother than you'd like. Fortunately, the Akka Future implementation makes this problem much easier for us.

Futures Are Great at Being Context-Free

One thing that Futures are great at is accelerating “raw computation,” which is why we’ve started by looking at matrix multiplication. The information required to multiply N matrices together is simply the matrices themselves and their ordering. We don’t need anything from a disk, or the network or a user or anything of that sort. All we have to do is just plow through N matrices, multiplying them together. If you want to parallelize a very deterministic algorithm, Futures are the way to go.

So, how would Futures help us solve the matrix multiplication problem better than Actors? They solve the two biggest problems we have: maintaining the sequence and knowing when everything’s done (see [Figure 4.16](#)).

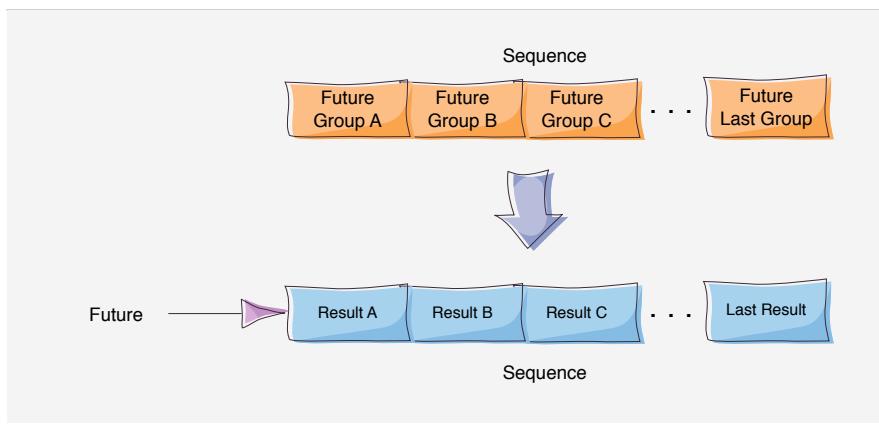


Figure 4.16 · Akka’s Future implementation allows us to take a list of Futures and convert them to a Future that contains a list of their results, and it maintains the same sequence as the original Futures.

All we need to do is transform our list of matrices into a list of a groups of matrices, and then transform that into a list of Futures that compute the multiplications.

```
val list = ... list of matrices ...
```

```
val grouped = list.grouped(5000) // 5000, just for fun
val futures = grouped.map { m => Future { ... multiply them ... } }
```

We've obviously left out some details, which aren't really important for us right now. The bottom line is that we've converted our list of matrices to a list of Futures and the only Akka-like thing in that code snippet is the construction of the Future with a closure that multiplies the group.

Now we need to collect things, which was the same problem we had to solve with the Actor-based approach. We don't have the sequencing problem since the list of Futures is in the same order as the groups, but how do we know when all of the Futures complete? We're not going to go into any detail about what you'll see because we're not ready for it, but the simplicity of it should get you thinking in the right mode.

```
val results = Future.sequence(futures)
// results is now a Future whose value is the list of resulting group
// multiplications.
val finalResult = ... multiply the last list of matrices together ...
```

Again, we've left some details out, but that's the bulk of it. Not bad for half a dozen lines of code, eh?

Futures Compose, Actors Don't

Actors are great at many things, as we've seen, and what we've seen is merely a glimpse into their potential. But one of the things that Actors don't do well is *compose*.

The fact that Actors don't compose is rather significant, and if you're a devotee of functional programming, or you've worked with OO patterns such as the Decorator⁷ or Chain of Responsibility,⁸ then you understand that significance.⁹ Functional composition, in particular, gives us a level of expressiveness that brings a large amount of power and flexibility to our daily coding. What if we could bring that level of expressiveness to our daily coding while at the same time mixing in concurrency? If the picture of a

⁷http://en.wikipedia.org/wiki/Decorator_pattern

⁸http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

⁹I do apologize for lumping the OO composition in the same league as functional composition. While I'm quite aware that they don't belong together, for the sake of establishing familiarity I hope you'll forgive me.

Tyrannosaurus Rex slam-dunking a basketball during the final moments of an inter-galactic game of hoops against the backdrop of 1.5 billion simultaneous supernovas just popped into your head, then you're getting the idea.

The Futures implementation in Akka allows us to set up sequential pipelines of code that run asynchronously to other pipelines, but also allows us to create an awesome interplay of parallel and sequential pipelines that run together, are still very easy to reason about, are concise, and still very functional.

Futures Work with Actors

Futures are designed to work with Actors. The converse isn't really true, but that's simply because there's no reason for it to be. A long time ago, Akka used to have a whole bunch of ways to send a message to an Actor. The Actor itself had three methods declared on it: `!`, `!!`, and `!!!`. No, that's not a stutter. The different methods signified that the call could be non-blocking, blocking, or Future-based, respectively. This was a decent model for learning how to write the API, and the Akka team learned a lot from it; they learned that it wasn't great. Since then things have been changed, and only the non-blocking version is used. The Actor itself doesn't know anything about Futures.

The Actor and the Future bind together using an external pattern and the Future is the one that understands what an Actor is (for all intents and purposes). It's really as simple as [Figure 4.17](#).

One of the many things that this allows is the continuation of the pipelining concept. A Future can be used to coordinate responses, and then *pipe* that response message to another Actor instance. And of course all of the usual transformations you'd like to apply to the resulting message can be applied before piping it to that Actor. The amount of flexibility provided by the Future-to-Actor relationship creates a partnership in the Akka toolkit that is greater than the sum of its parts.

Thinking in the Future

As with everything else in Akka, using Futures isn't just about tossing another tool in the chest, it's about allowing you to think about your code differently. We don't need to worry about "running this in parallel" or "waiting for that to complete" or "putting that other thing on a thread." We don't need

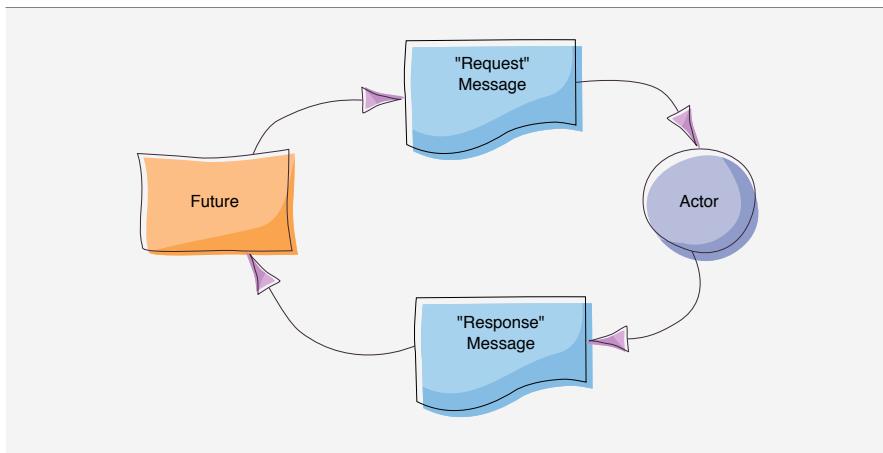


Figure 4.17 · Futures can tacitly bind themselves to Actors by representing themselves as the “sender” of the request message. Assuming the Actor responds to the sender, the response will go back to the Future. This entire interplay between the Actor and the Future is non-blocking.

to worry about building out the future by hand (i.e., by creating work to go on threads, or any of that nonsense). We simply construct our algorithms and let the Future happen for itself.

For example, one thing we might do in our day-to-day may be to create a couple of queues for different threads to use for communication. One side may pull work out from their queue while the other side polls, or otherwise waits, on their own queue for results. With Futures, we don’t need to think in that manner anymore. We would create a Future to do the work, and then compose another Future on top of it to process the results. Everything can be done from the call site directly and we don’t concern ourselves with queues, messages, protocols, or even threads. The Future just unfolds as we’ve defined, without our having to construct any scaffolding to realize that Future.

4.3 The Other Stuff

There are many other tools in the Akka tool chest but most of them dovetail with either the Actor or the Future, so you’ve been introduced to the most important concepts you need in order to understand the rest.

So what's the rest?

The EventBus

The EventBus is a nifty little Pub/Sub abstraction that evolved out from an internal Akka implementation that the team thought the world might just make some decent use out of. You're going to find out that they were right.

As we work with messages and events, the idea of distributing certain types of event classes to various types of endpoints just naturally becomes desirable. This happens on a micro-level all the way up to a macro level. You might want to have your Actor send certain messages to a few friends that have an interest in what it has to say, or you might want to broadcast a small amount of events across your entire super-computing cluster of Actors that spans the entire Northern Hemisphere.

The Scheduler

Concurrent programming, especially coupled with the concepts of events, has always needed timed or future-based events. Akka provides you with a scheduler that executes functions at timed intervals or single operations at some point in the future. It even provides a special construct for sending messages to specific Actors in the future.

Not much is alien to us in the world of the scheduler so you should be pretty familiar with the concept. We'll see it and use it extensively so if you're not familiar with it now, you will be.

Dataflow Concurrency

Dataflow concurrency builds on Futures and allows you to look at your application's concurrency from the point of view of the data that it uses.

Instead of creating your application as a set of operations that happen in parallel, you can think of it more as algorithms that operate on data. At some point, a piece of data acquires a value, which allows other parts of the application that are waiting on that data to move forward. It operates more like an application that's using locks and condition variables than one that's using Futures, except that it's much more deterministic and it doesn't block threads. [Figure 4.18](#) shows the difference.

With Futures, our goal is to run functions concurrently with other functions and rendezvous on the results of those functions if we need to. When

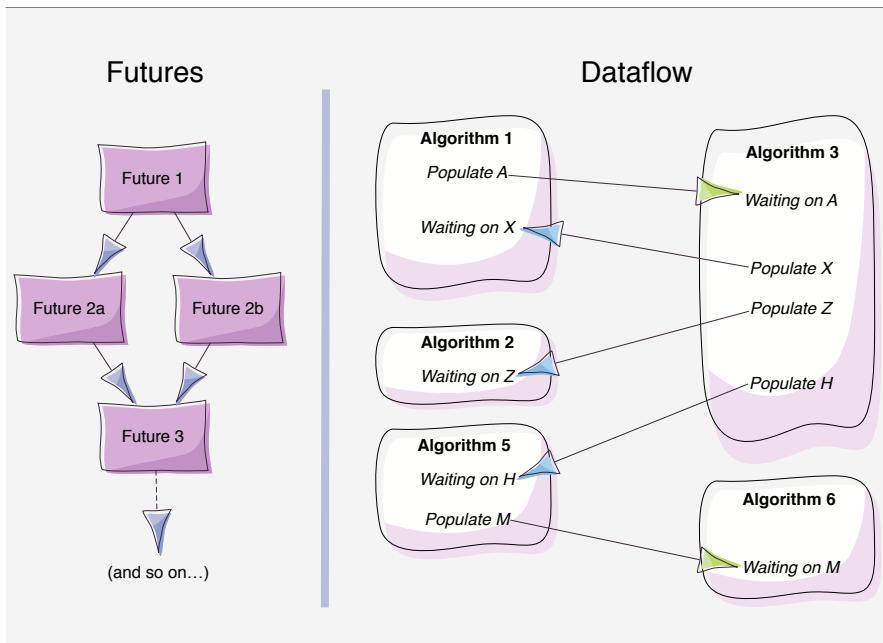


Figure 4.18 · The conceptual difference between Futures and dataflow: Futures are non-blocking and can execute full functions in parallel, whereas dataflow creates concurrency that's more of an intra-function type of concept. Each algorithm, represented by the individual boxes, is either waiting for (while not blocking a thread) or is populating a piece of data. Between any two points of contention, things run concurrently, but they rendezvous on the shared data.

we employ dataflow, we're getting concurrency more intrusively than that. Pieces of our functions run in parallel with pieces of other functions and they rendezvous on any shared data on which they might be working.

"So, they're sharing mutable data? Isn't that a bad thing?" Well, it's not actually mutable in the traditional sense, so the sharing isn't quite the same as we're used to in shared-state concurrency. These aren't *variables* but are *values* and are thus immutable. The only difference between dataflow values and standard values is that dataflow values exist in a future context, whereas standard (non-lazy) values are, effectively, set at the time of access.

"But those (hidden) locks and condition variables are a bad thing, right?" They would be if the data were more promiscuous than it is, but it isn't.

We'll be getting into dataflow concurrency later, but any notions you might have about it bringing back the paradigm of shared-state concurrency that we're (somewhat) trying to leave behind shouldn't bother you. The nice thing about dataflow concurrency is that, while things could go horribly wrong (e.g., you could get a deadlock), they're *guaranteed to go wrong all the time*. So you're not stuck looking for Heisenbugs in production because the first time you run the code, it's going to go bad on you.

Message Routing

Passing messages to untyped endpoints provides you with a ton of flexibility, and one of those points of flexibility is embodied in the routing feature of Akka. You can send messages anywhere you'd like, of course, but what good is that? Well, if you think back to Figures 4.2, 4.3, and 4.4 you might recall that sending the same message to multiple endpoints can get us greater levels of concurrency and Figure 4.5 tells us that we can use routing to get us some safety.

Akka provides routing right down to the configuration level of your application. We can use routing to make our applications faster, more scalable, more fault tolerant, and a lot more flexible. The fact that Actors can only do one thing at a time will never be a problem for us.

Agents

Agents are inspired by the feature with the same name in Clojure¹⁰ and might look a bit like the atomic classes that are part of the `java.util.concurrent.atomic` package, but they're much more than that. Agents are effectively Actors and thus provide the same single-threaded guarantees that Actors provide, but without the need to send messages to them in order to obtain their values.

You can use Agents to provide deterministic locations in memory that are guaranteed to be safe places to store and read data that can change across entities. Agents can be waited on, while other entities play with them and can also participate in transactions, which make them much more interesting than the atomic family of classes that exist in the `java.util.concurrent.atomic` package.

¹⁰<http://clojure.org/>

And Others...

You've become acquainted with the core philosophies and classes that Akka provides. As you'll see as you continue to read, Akka provides more tools that we can use, including non-blocking IO, interaction with Akka deployments on remote hosts, distributed transactions, finite-state-machines, fault-tolerance, performance tuning, and others.

4.4 You Grabbed the Right Toolkit

In summary, welcome aboard! You've just received a whirlwind tour of the high points of Akka and should have some clue as to why it will be the awesome toolkit that you've heard about. When it comes to building highly concurrent and fault-tolerant applications on the JVM, Akka is a solid choice.

As we progress, you'll learn how to apply the tools we've already discussed to your application design and development. You'll also start seeing *a lot* more code than we've seen thus far that will help establish a set of patterns for coding in Akka. Later on, we'll establish a set of anti-patterns, because there certainly are a fair number of those. Like any decent power tool, if you point it straight at your eye and then run forward, bad things will happen. There are great ways to use Akka and there are also the power-tool-to-the-eye ways, and we're going to favor the former.

You've learned a ton so far, and you should feel pretty awesome about that, but before you run out into the street naked declaring your superiority over the mere machine that you sit in front of, let's cover some more of the nuts and bolts.

Flip the page and let's go...

Chapter 5

Actors

It's time to start concurrency programming with Actors. We'll begin by exploring the most basic mechanics of Actor construction and operation, so you can get a feel for things and how most of the work gets done. Using that example, we'll explore more about what Actors are and how they work, building up use cases and understanding so that you're armed with heaps of awesomeness that you can employ when solving your coding and design problems.

What follows is a single Actor definition of a terrible Shakespearean Actor and a poor sod who has to talk to him. The Shakespearean Actor is a true Akka Actor and the poor sod is main. In these early stages, I'll show you some fairly atypical elements of Actor programming, but they'll help keep things familiar.

```
package zzz.akka.investigation

// All that's needed for now are three components from Akka
import akka.actor.{Actor, Props, ActorSystem}

// Our Actor
class BadShakespeareanActor extends Actor {
    // The 'Business Logic'
    def receive = {
        case "Good Morning" =>
            println("Him: Forsooth 'tis the 'morn, but mourneth for thou doest I do!")
        case "You're terrible" =>
            println("Him: Yup")
    }
}
```

```
}

object BadShakespeareanMain {
    val system = ActorSystem("BadShakespearean")
    val actor = system.actorOf(Props[BadShakespeareanActor])
    // We'll use this utility method to talk with our Actor
    def send(msg: String) {
        println("Me: " + msg)
        actor ! msg
        Thread.sleep(100)
    }
    // And our driver
    def main(args: Array[String]) {
        send("Good Morning")
        send("You're terrible")
        system.shutdown()
    }
}
```

We've already set up all of the dependencies and build infrastructure using SBT, so you should be able to place this code and give it a go. Put the content into `src/main/scala/zzz/akka/investigation/BadShakespeareanActor.scala`, type `sbt run`, and you should see this:

```
Me: Good Morning
Him: Forsooth 'tis the 'morn but, mourneth for thou doest I do!
Me: You're terrible
Him: Yup
```

Just a couple of guys having a painful chat. I said earlier that we'd see things that were atypical of Actor programming, and this example certainly has some of them, but they tend to be items that we use all the time, so they should be easy to spot. There are three main things:

- We generally don't use raw `Strings` to communicate with Actors due to the fact that it's not type safe. No big surprise there.
- Our Actor doesn't *do* anything and that's pretty boring. A `println` doesn't count as something.

- `Thread.sleep()` is the big one. If your code has a *sleep* in it, then you're tying up an execution thread for some amount of time and we don't do that in Akka. In other words, we're doing it wrong.

In this chapter, you'll learn how to write code that's agile, decoupled, fast, non-blocking, type safe, and highly communicative. What we've seen thus far falls somewhat short of those goals, but it is also quite illustrative of some basic properties of Actors. Let's explore the mechanics behind what we've seen and start working on things that actually *are* typical of Actor programming in Akka.

5.1 The Components of an Actor

An Actor isn't just the Actor itself. Akka uses several components that work together to deliver the Actor experience as well as decouple the Actor from the other aspects of the code on which it relies. This componentization allows for heaps of coolness that we'll continue to discover, but some of it should become clear immediately. First, let's take a better look at what we get when we instantiate an Actor. This won't be a picture-perfect example of the concrete Akka class diagram. If you want to see that, then feel free to download the source from Github.¹ What you'll see here are the bits and pieces that concern you as the developer, since these are the components that you have the power to alter directly. You can reconfigure and/or modify certain components to suit the particular needs of your application at a whim.

Figure 5.1 shows us what we need to know about how Akka has implemented Actors for us. The actual Akka components (as opposed to those that it merely makes use of from other packages) are:

- Actor
- Mailbox
- Dispatcher
- ActorRef

¹All the source is available at <https://github.com/akka/akka> inside the akka-actor/src/main/scala/akka subtree.

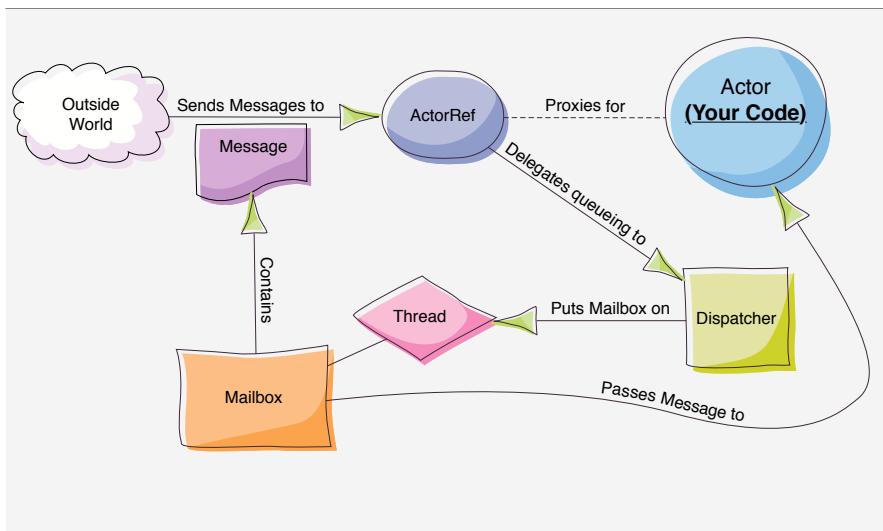


Figure 5.1 · Shows us the parts of the Actor that concern us as programmers. All of these components together facilitate the execution of message processing in your Actor's code. Note that everything but the ActorRef and the message is internal. Someone in the outside world constructs a message, and he can only send it to the ActorRef. He can't send it to the Actor directly.

When you send a message to an Actor, you're only sending it to its ActorRef; you never get to interact with the Actor in any direct way. The ActorRef will then contact the Dispatcher and use it to queue the message onto the Actor's Mailbox. Once that's done, the Dispatcher will put the Mailbox onto a thread and when the Mailbox executes, it will dequeue one or more messages and send them to *your* Actor's receive method for processing. But it's important to remember that once the message has been put in the Mailbox, the caller is free to do what it wishes. The only blocking that occurs from the caller's perspective is the act of enqueueing the message. After that, all of the extra work and processing is done on a separate thread.

These components work together to provide the Actor experience. Each component serves a particular purpose that you can leverage and alter depending on the problem you're currently trying to solve. All of the decoupling that these components provide enable a rich set of functionality in Akka's Actor implementation. We'll explore this structure as we proceed, so you might want to bookmark this page for reference.

5.2 Properties of an Actor

There are a few key properties of Actors that distinguish them in our designs. We'll cover a few of them now and continue to recognize them as we move forward.

Actors Are Alive

There's a reason we chose the `BadShakespeareanActor` as our first Actor. Actors are *live objects* and thus they are best at modeling things that are “active.” While you can use an Actor to model a block of wood, that may not be the best way to do it, since wood doesn't really do much on its own. Perhaps the following Actor definition might be appropriate:

```
class Wood extends Actor {  
    def receive = {  
        case _ =>  
            throw new Exception("Wood can't hear you.")  
    }  
}
```

If we don't model wood with Actors, then what do we model? It wouldn't be appropriate to say “everything else,” but there are so many things that you *can* model with Actors that we may as well say it anyway. Everything else.

Actor programming is less about steps in an algorithm than it is about having a near infinite number of people to whom you can assign tasks. Thinking about Actor programming in this way is extremely useful because people are alive and active. I can send an email to Bob and he'll *react* to that email, perform some work, and eventually, if I need him to, either respond to me personally or he'll delegate that task to someone else.

With that analogy, we get a glimpse into the core benefit of eliminating strong typing for the Actor itself. If I am coded to work with Bob, then Bob will have to deal with me personally. However, if I'm happy to work with anyone², then I can get my results from anyone whom Bob sees fit to assign the work.

Figure 5.2 makes this even more obvious than it may already be.

- All Jill wants is some coffee.

²And let's face it, we all put that on our resumes, right?

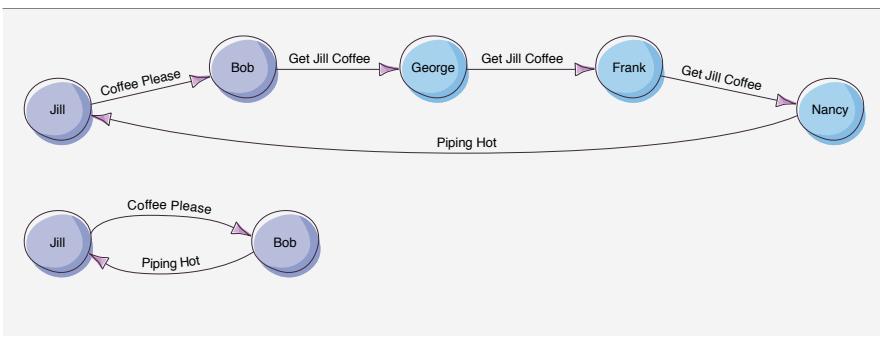


Figure 5.2 · Programming with Actors is more like hanging out with a bunch of people at the office than it is about sequences and algorithms. Jill's not interested in *who* gets her coffee for her, just so long as she gets it.

- Or, all your Actor wants is a row from a database.
- She asks Bob to get her some and Bob delivers.
 - Or, your Actor sends a message to the Database Actor, and that Actor responds with the row.

Well, let's say Bob would love to do it, but his appendix just burst and he really needs to tend to that first. Should Jill just *wait* for her coffee? Jill's a very important person and Bob knows not to trouble her with trivialities, so he decides to pass this work off to someone else while he tends to this other business. It makes no difference to Jill.

It's the same thing with Actor programming. Talking to a Database Actor yesterday is no guarantee that you're talking to a Database Actor today. The designer could have decided that the Database Actor was overloaded and we could obtain better throughput if we balanced that load across a pool of Database Actors. Here's the cool thing: that Database Actor could be swapped out for the load balancer without you having to know. What's possibly even more cool is that the load balancer would have no idea what you're talking about, or what a Database is—all it knows is how to find the best Actor to service your request.

Now *that's* a decoupled design.

One Thing at a Time

If you've ever watched M*A*S*H³, you might recognize this quote:

“I do one thing at a time, I do it *very* well, and then I move on.”

That was Major Charles Emerson Winchester III describing his surgical technique. Actors would say the same thing, except we might want to change it just a tad:

“Actors do one thing at a time, they do it *very* well, and then they *quickly* move on.”

There's only one way to get an Actor to do anything—send it a message—and there's only one way it will do anything with that message—process it in its `receive` method. If you put these two aspects together, you get an Actor that works through its problems completely isolated from external influence. In other words, even though you're working in a highly concurrent environment, your Actor can't be adversely affected by that concurrency.

Have you ever printed output to a terminal or log file, which worked great when you had a single thread, but got all garbled when multiple threads tried to do the same thing? Were you to do this with an Actor, you'd get sequencing by default:

```
class PrintingActor extends Actor {  
    def receive = {  
        case msg => println msg  
    }  
}
```

The `PrintingActor` can never print more than one message at a time, so it can't get garbled. There are, however, some caveats when it comes to writing well-behaved Actors that do one thing at a time, and do it quickly.

1. It's up to you to do things quickly. If you're going to calculate π to 8 bazillion decimal places without ever leaving the `receive` method, then don't expect your Actor to be all that responsive.

³An awesome T.V. show from the 1970s.

2. You can always create weird ways of breaking out of this model (e.g., start modifying some global variables, sharing your state with the world, and so on...). Don't.

Actors Live in a Fortress

In case it's not incredibly obvious from the last section, you can't mess with an Actor from the outside. Any behaviour and data inside of an Actor is encased in a fortress made of 5-foot-thick Adamantium⁴. You *can't* mess with what's inside that fortress. Sure, there's a pretty responsible dude sitting in the guard tower who will answer the phone when anyone calls, but there's no way you're getting in there yourself.

There's a very good reason for this, of course. The Actor paradigm protects your code from yourself on those days when your brain isn't firing on all cylinders, or you have someone hacking away at your masterpiece. If it were possible for anyone and their grandmother to waltz inside the fortress and start redecorating, then the paradigm would basically be shot to hell. The fortress makes concurrency programming *reasonable*, and it's the ability to easily reason about our Actor applications that makes them so formidable.

The fortress will start to be a royal pain when we start to look at unit testing your Actors, but the Akka team realized this and Akka 2.x has a pretty solid answer to this problem. Under very controlled conditions, we can enter the fortress and flip on the light switch directly in order to make sure it works. You don't have to rely on the dude in the guard tower when you're testing.

You Can Always Find Your Actors

With all of these living things running around your system (and if you're a cloud developer, you could easily have hundreds upon hundreds of millions of them), how are you going to find them? Akka's got you covered here. We won't go into depth with it at the moment, but rest assured that when you create an Actor it has a globally unique identity and you can locate it.

You can get a single Actor by name, a group by type, or Actors within a hierarchy of deployment. Akka is organized in a whole set of hierarchies in a recursive structure, so essentially an Actor structure looks the same no

⁴<http://en.wikipedia.org/wiki/Adamantium>

matter where you might find yourself within it. It's almost turtles all the way down.⁵

But again, this is pretty much the same as working with people. You can find people by name, email address, geographical location, position in the organization, etc....

5.3 How to Talk to an Actor

Now, I've known a lot of actors in my life⁶ and I can tell you for a fact⁷ that talking to actors is not easy.⁸ Fortunately, Akka Actors are incredibly easy to talk to and will accept any kind of message you want to send their way.

Actor Messages

Actor messages derive from `scala.Any`, so they can be anything at all. The magic of Scala's *pattern matching* lets us concisely deal with messages within Actors. Let's look at some code:

```
package zzz.akka.investigation

import akka.actor.Actor

case class Gamma(g: String)
case class Beta(b: String, g: Gamma)
case class Alpha(b1: Beta, b2: Beta)

class MyActor extends Actor {
    def receive = {
        // Literal String match
        case "Hello" =>
            println("Hi")
        // Literal Int match
        case 42 =>
            println("I don't know the question. Go ask the Earth Mark II.")
        // Matches any string at all
    }
}
```

⁵http://en.wikipedia.org/wiki/Turtles_all_the_way_down

⁶This is not true.

⁷This is not a fact.

⁸But I really wouldn't know.

```
case s: String =>
    println("You sent me a string: " + s)

// Match a more complex case class structure
case Alpha(Beta(beta1, Gamma(gamma1)), Beta(beta2, Gamma(gamma2))) =>
    println("beta1: %s, beta2: %s, gamma1: %s, gamma2: %s".format(
        beta1, beta2, gamma1, gamma2))

// Catch all. Matches any message type
case _ =>
    println("Huh?")
```

{}

That's pretty flexible. You can simply use basic raw data types (e.g., `String`) or you can build up any type of complex data structure you like either with `case` classes or with your own hand-rolled classes. When we conversed with our `BadShakespeareanActor`, we used raw `Strings`, which was not such a great idea. It isn't in most situations when the best type of message is usually a `case` class.

In the rest of this book, we'll favor `case` classes and `case` objects over most other alternatives, since they give us so much for free, are type-safe, and are very easy to work with.

Sending Messages

Messages are one half of the equation; delivering them is the other half. Sending a message to an Actor facilitates a mechanism of concurrency. For example, let's say we do this:

```
object MySequencedObject {
    def doSomething(withThis: String) {
        // ... do something ...
    }
}

MySequencedObject.doSomething("With this")
println("Hi Jaime!")
```

We won't see "Hi Jaime!" until `doSomething()` is finished doing whatever it is it will do with "With this". By sending a message to an Actor,

we issue the request immediately, but the execution of that request is not dependent on our current thread of execution.

```
case class DoSomething(withThis: String)
class MyConcurrentObject extends Actor {
  def receive = {
    case DoSomething(withThis) =>
      // ... do something ...
  }
}

system.actorOf(Props[MyConcurrentObject]) ! DoSomething("With this")
println("Hi Jaime!")
```

Now we've given more than one thread some work to do. "Hi Jaime" may or may not be printed out before the Actor gets to work; it's all non-deterministic because it's now concurrent.

Tell Syntax

This is all facilitated by the *tell* syntax, denoted by `!`. The `!` method is an asynchronous message pass that puts the message in the Actor's Mailbox and returns immediately, safe in the knowledge that the message is in the queue. While you can get a reply to a message sent via tell, the tell syntax does not concern itself with replies. *Telling* Actors what to do or telling them about events that have occurred is the most scalable way to write code in Akka. If nobody's blocking a thread while waiting on the response to a message, then the hardware resources are fully at the application's disposal.

Ask Syntax

There's another way to send messages to Actors and, while this is a great way to do it, we won't get into it yet. This relies on the other axis of Akka concurrency: Futures, which we saw earlier but aren't ready to start coding with yet. The syntax for using the `ask` method, however, is quite natural:

```
// Required since the "?" method is not a direct
// method on Actors but is "pimpled" on after the fact
import akka.pattern.ask

val question = actor ? 42
```

The question will come back to us in the form of a Future on which we can operate later. You don't know a lot about Akka Futures at this point, but it might be helpful to understand a bit about how it works. The Future that's returned has a counterpart called a *Promise*, which Akka abstracts away from the Actor. When the Actor replies to the message, it actually *completes* the Promise and the Future becomes satisfied.

You can block the calling thread with this Future, but you should never block on an Actor request without some pretty serious thought first. Blocking a thread while waiting for an Actor to do something with your request is generally much more painful than blocking a thread while you wait for something to happen on a traditional concurrent object (e.g., pushing a new value into a concurrent queue). [Figure 5.3](#) shows us what we saw in [Figure 4.6](#), but it's much more explicit about a blocking request.

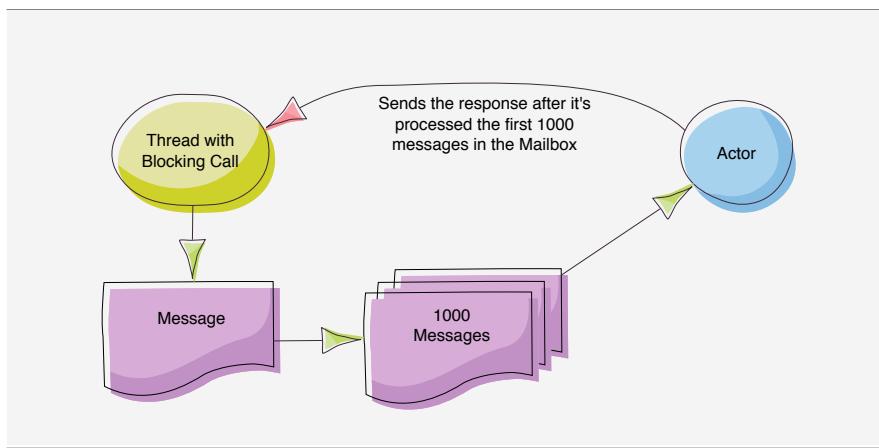


Figure 5.3 · When an Actor has a lot to do, blocking on a response from it can be more painful than you might otherwise be used to.

As discussed, Actors do *one thing at a time*. It's not like 50 guys holding a read lock, all waiting for the guy holding the write lock to complete. Once the guy with the write lock lets go, all 50 guys can go nuts reading the results at the same time. With Actors, if you've got 1000 guys in the queue ahead of you, then you have to wait until they get processed before it's your turn. The Actor model doesn't have the same concept of readers vs. writers. Essentially, it assumes everyone is a writer so everyone's got to wait their turn.

Note

Making a blocking request/response call on an Actor has much more serious implications than a standard blocking function call with a return value. Use the ask syntax as liberally as you'd like, but be very wary of blocking on the Future value.

By and large, this is a very good thing, but it has serious implications if you want to do something silly, like block on a Future result. The cool thing is that you very rarely (if ever) need to block on a Future result and writing the non-blocking equivalent is quite easy most of the time.

Just to be clear, this doesn't mean that using the ask syntax is a bad thing. Far from it. Ask syntax provides you with a different binding context that lets you operate on the logic in a manner that can apply much better to your current situation if an Actor context doesn't do the job very well. It does this while still being entirely asynchronous.

5.4 Creating Actors

We've covered quite a bit about Actors to this point but before we move on to using them in some real application development, we should touch on Actor creation. Akka gives you a lot from behind the scenes when it comes to Actors – stuff that you'll be quite amazed by as we proceed—but doing that requires that we give it a bit of control over life cycles. One of the most important parts of any life cycle is the birth, so let's see how to create Actors with Akka:

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

// Assume we've got 'MyActor'
class MyActor extends Actor { ... }

// We need a 'system' of Actors
val system = ActorSystem("MyActors")

// 'Props' gives us a way to modify certain
// aspects of an Actor's structure
val actorProperties = Props[MyActor]

// And finally we pass the properties to the
// actorOf factory method
val actor = system.actorOf(actorProperties)
```

We'll be covering the ActorSystem later, but for the moment we can simply think of it as the root of a collection of Actors. The Props class lets us modify some of the structure that surrounds the Actor, such as its execution context, which we'll be seeing more of as the book progresses. Once we have those things in place, we can pass the properties to the system and get a reference to our Actor. Now the magical stuff in the Akka toolkit has a proper hook that we can use to manipulate our Actor in all of the ways that we want.

There's a fair bit more to creating Actors that we'll see, but we've had enough dry theory for now. Let's start building an Airplane.

5.5 Actors in the Clouds

For the bulk of this book, we'll stick with a common theme so that we can continually create and improve a single application. Since I don't know anything about Airplanes, I figured it would make a great thing to build. It also doesn't hurt to add the cheesy relationship between Airplanes and the fact that Akka makes a great toolkit for Cloud Computing.⁹

We can build control surfaces, instruments, passengers, pilots, computers, flight attendants, weather... all kinds of different cases for Actors to concurrently start messing around with each other.

Let's start by building the Altimeter.

How High Are You?

The Altimeter will be a pretty important piece of gear in our Plane. It will tell us how high we are, figure out if we're stalling, help control how quickly we can ascend, and inform any Actor that's interested how high we are at any given moment.

Now, what we're building here isn't exactly physics-approved, so I wouldn't recommend ripping it off to build your next video game. However, it will certainly send and receive messages enough to keep our Plane flying.

Let's start with the Altimeter's companion object. The companion object can, of course, be used for anything, but it tends to make a good spot for construction code and messages that are bound to the Actor we're defining.

```
package zzz.akka.avionics
```

⁹Please send groans to @derekwyatt on Twitter

```
// Imports to help us create Actors, plus logging
import akka.actor.{Props, Actor, ActorSystem, ActorLogging}
// The duration package object extends Ints with some timing functionality
import scala.concurrent.util.duration._
// The Scheduler needs an execution context - we'll just use the global one
import scala.concurrent.ExecutionContext.Implicits.global

object Altimeter {
    // Sent to the Altimeter to inform it about a rate-of-climb changes
    case class RateChange(amount: Float)
}
```

The Altimeter doesn't have a big interface at this point; it can only accept messages that tell it when our rate of climb has changed. Up next is the definition of our Actor:

```
class Altimeter extends Actor with ActorLogging {
    import Altimeter._

    // The maximum ceiling of our plane in 'feet'
    val ceiling = 43000

    // The maximum rate of climb for our plane in 'feet per minute'
    val maxRateOfClimb = 5000

    // The varying rate of climb depending on the movement of the stick
    var rateOfClimb: Float = 0

    // Our current altitude
    var altitude: Double = 0

    // As time passes, we need to change the altitude based on the time passed.
    // The lastTick allows us to figure out how much time has passed
    var lastTick = System.currentTimeMillis

    // We need to periodically update our altitude. This scheduled message send
    // will tell us when to do that
    val ticker = context.system.scheduler.schedule(100.millis, 100.millis,
        self, Tick)

    // An internal message we send to ourselves to tell us to update our
    // altitude
    case object Tick

    def receive = {
```

```
// Our rate of climb has changed
case RateChange(amount) =>
    // Keep the value of rateOfClimb within [-1, 1]
    rateOfClimb = amount.min(1.0f).max(-1.0f) * maxRateOfClimb
    log.info(s"Altimeter changed rate of climb to $rateOfClimb.")

    // Calculate a new altitude
    case Tick =>
        val tick = System.currentTimeMillis
        altitude = altitude + ((tick - lastTick) / 60000.0) * rateOfClimb
        lastTick = tick
    }

    // Kill our ticker when we stop
    override def postStop(): Unit = ticker.cancel
}
```

The Altimeter Actor is definitely a live object because it does stuff. At the moment, it really just reacts to changes in the rate of climb. Someone (hopefully some sort of pilot) can change the rate of climb and the Altimeter will start calculating altitude changes based on this new information.

We get some feedback about what has occurred through the use of the logger, which we've mixed in via with ActorLogging. This logger is a standard log4j-style logger and it can be configured through the configuration system, which you'll meet in due course. For the moment, we'll use the INFO level of logging since it is enabled by default.

You can see that there is some mutable data in there: `rateOfClimb`, `altitude`, and `lastTick`. The `altitude` variable really represents why the Altimeter exists, since its main purpose is to calculate altitude changes, and the others are support variables. Due to the fact that the Actor works in isolation, all of this mutable state can exist without any concurrency protection.

We also see an example of *reactive programming*. We could have coded the Altimeter to accept a command, such as `ChangeRateOfClimb` to which it would respond with some sort of success or failure message, perhaps. Rather than do that, we've chosen to take it from the other direction; the rate change has already happened. The pilot (presumably) has pulled back on the stick, the elevator has responded, and the Plane has changed its pitch. What the Altimeter needs to do is react to that event and modify the altitude accordingly.

If you're not used to thinking in a reactive style, then it might not seem all that interesting. It's subtle and sometimes its merely a semantic difference, but it's semantics that often make the difference between one algorithm and another. When we design in a reactive style, we open up several possibilities that allow pieces of code to be much more *autonomous* than they might be otherwise. Various aspects of your application merely generate events and other aspects of your application react to those events. The alternative is to couple those aspects together such that they rely on each other to get work done. And when a lot of your code relies on a lot of your other code to get things done, your app becomes harder to reason about, scale, and evolve.

This isn't to say that reactive programming is superior to imperative programming. Our discipline is far too complex to say that any given style is a silver bullet. What we're saying is that reactive programming provides a model of design that fits very well into places where the imperative style does not.

Note

Keep all of your tools sharp. The golden hammer fallacy is even more dangerous in concurrency programming than it is in sequential programming.

The Scheduler

The Altimeter can receive messages from outside sources telling it that the rate of climb has changed, but it also accepts a message from the Scheduler. The Scheduler is technically an external source as well, but the reference we have to the instance that the Altimeter created is inside the fortress, so it's very reasonable to think of it as an internal source. The Scheduler is a simple and effective mechanism that Akka provides for doing stuff at some point (or regular points) in the future. Here, we use it to send a message to self every 100 milliseconds.

Inside an Actor, we obtain the Scheduler from `context.system`. We'll cover `context` and `system` soon.

The Scheduler can execute arbitrary code in the future as well:

```
scheduler.schedule(100 millis, 100 millis) {  
    println("100 milliseconds has passed")  
}
```

However, we don't really want to be running code like this when dealing with an Actor's internal guts. Imagine if instead of sending Tick to the Altimeter, we just performed some work instead:

```
scheduler.schedule(100 millis, 100 millis) {  
    val tick = System.currentTimeMillis  
    altitude = altitude + ((tick - lastTick) / 60000.0) * rateOfClimb  
    lastTick = tick  
}
```

With that block of code, we just broke into the fortress. Now, I know I said that you simply can't do that... I lied, sorta. We didn't break into the fortress directly—it was an inside job. Someone inside the fortress gave the keys away to a nefarious evil-doer who later walked in and set the place on fire. You can break into the fortress this way, and it's actually not that hard to do—it's so easy, in fact, that you can do it by mistake. All you have to do is create a closure that closes over the wrong thing, and then run that in some sort of asynchronous context (e.g., a Future or a scheduled task).

If the Altimeter is processing a RateChange message at the same time we've got this Scheduler running, what happens? *The plane crashes, and everyone aboard dies in a raging inferno.* Do you really want that? Of course you don't. When this sort of fortress-breaking occurs, it's pretty much always a rookie mistake. In the early stages, you might find yourself making this blunder, but with experience comes the natural coding habits that ensure you never do this.

Note

Don't subvert the Actor programming paradigm. Any mutable Actor data must only be accessed (that means reading too) in the Actor's receive method.

This is precisely why we have the Scheduler send the Actor a Tick message instead of having the Scheduler do the Actor's work for it. By sending a message, we're using the standard Actor programming method and staying within the paradigm. The message will be processed when its time comes.

Controlling the Plane

So who gets to send the Altimeter these RateChange messages? We'll give that responsibility to the control surfaces. For now, we'll model the control

yoke and only its forward and backward motions that change the rate of the Plane's climb.

```
package zzz.akka.avionics

import akka.actor.{Actor, ActorRef}

// The ControlSurfaces object carries messages for controlling the plane
object ControlSurfaces {

    // amount is a value between -1 and 1. The altimeter ensures that any
    // value outside that range is truncated to be within it.

    case class StickBack(amount: Float)
    case class StickForward(amount: Float)

}

// Pass in the Altimeter as an ActorRef so that we can send messages to it
class ControlSurfaces(altimeter: ActorRef) extends Actor {

    import ControlSurfaces.~
    import Altimeter.~

    def receive = {

        // Pilot pulled the stick back by a certain amount, and we inform
        // the Altimeter that we're climbing
        case StickBack(amount) =>
            altimeter ! RateChange(amount)

        // Pilot pushes the stick forward and we inform the Altimeter that
        // we're descending
        case StickForward(amount) =>
            altimeter ! RateChange(-1 * amount)
    }
}
```

Again, we've decided to make an Actor represent our entity. The ControlSurfaces Actor is a live object that we can give to various other entities and let them manipulate. We could imagine a pilot doing that sort of thing.

In order for the ControlSurfaces Actor to send changes to the Altimeter, it needs to have a reference to that Altimeter, so we pass that in during the construction of the ControlSurfaces Actor. We could theoretically look it up, given Akka's facilities for doing that, but it would be bad form to do so. Substitutability and composability demand that we give the Altimeter to the ControlSurfaces Actor. It should be no surprise at this point that the

Altimeter is an ActorRef. Since the Altimeter is an Actor and we can't send messages to Actors, but only to ActorRefs, it would be pointless to give an Altimeter or an Actor to the ControlSurfaces Actor.

In fact, Akka makes it impossible to even construct an Actor, or any derivation thereof, so even if we declared that the ControlSurfaces object should take an Altimeter or an Actor, we would never be able to actually pass it one. Don't believe me? Let's look at what happens when you try to do this with Akka:

```
scala> import akka.actor.Actor
import akka.actor.Actor

scala> class A extends Actor { def receive = { case _ => } }
defined class A

scala> new A
akka.actor.ActorInitializationException:
You cannot create an instance of [A] explicitly using the
constructor (new).

You have to use one of the factory methods to create a new
actor. Either use:
'val actor = context.actorOf(Props[MyActor])'
(to create a supervised child actor from within an actor), or
'val actor = system.actorOf(Props(new MyActor(..)))'
(to create a top level actor from the ActorSystem)
```

See? Actors have some smarts in their constructors that ensure you can't construct them outside of Akka's control. So, not only would it be silly to pass an Actor (or a derivation thereof) around, it's actually not even possible.

The Plane

Now that we can tell how high we are and we can change how high we are, we can build the thing that goes higher. The Plane will create the Altimeter and the ControlSurfaces as well as provide access when needed.

Again, we'll start with the companion object:

```
package zzz.akka.avionics

import akka.actor.{Props, Actor, ActorLogging}

object Plane {
```

```
// Returns the control surface to the Actor that asks for them
case object GiveMeControl
}
```

Since the Plane is the entity that holds the controls, someone will need to ask the Plane to get those controls; hence, the GiveMeControl message.

```
// We want the Plane to own the Altimeter and we're going to do that
// by passing in a specific factory we can use to build the Altimeter
class Plane extends Actor with ActorLogging {
    import Altimeter._
    import Plane._

    val altimeter = context.actorOf(Props[Altimeter])
    val controls = context.actorOf(Props(new ControlSurfaces(altimeter)))
    def receive = {
        case GiveMeControl =>
            log.info("Plane giving control.")
            sender ! controls
    }
}
```

By creating the Altimeter first, we get the ActorRef to it that we can then pass to the ControlSurfaces constructor. Since we can't just type new Altimeter, we use a special version of the actorOf(), which is accessible from inside an Actor. Using context.actorOf() creates the ActorRef and ties it to the current Actor as a *child*. This hierarchical relationship is a huge part of Actor programming in Akka and we'll be covering it in great depth later.

We also see our first real Actor message response. When asked for the controls, the Plane returns a reference to the ControlSurfaces that it created at construction time. Since we can send anything in a message, a reference to an Actor is a perfectly reasonable message to send.

Alright, let's try it out:

```
package zzz.akka.avionics
import akka.actor.{Props, Actor, ActorRef, ActorSystem}
import akka.pattern.ask
import scala.concurrent.Await
```

```
import akka.util.Timeout
import scala.concurrent.util.duration._
import scala.concurrent.ExecutionContext.Implicits.global

object Avionics {
    // needed for '?' below
    implicit val timeout = Timeout(5.seconds)
    val system = ActorSystem("PlaneSimulation")
    val plane = system.actorOf(Props[Plane], "Plane")

    def main(args: Array[String]) {
        // Grab the controls
        val control = Await.result(
            (plane ? Plane.GiveMeControl).mapTo[ActorRef],
            5.seconds)
        // Takeoff!
        system.scheduler.scheduleOnce(200.millis) {
            control ! ControlSurfaces.StickBack(1f)
        }
        // Level out
        system.scheduler.scheduleOnce(1.seconds) {
            control ! ControlSurfaces.StickBack(0f)
        }
        // Climb
        system.scheduler.scheduleOnce(3.seconds) {
            control ! ControlSurfaces.StickBack(0.5f)
        }
        // Level out
        system.scheduler.scheduleOnce(4.seconds) {
            control ! ControlSurfaces.StickBack(0f)
        }
        // Shut down
        system.scheduler.scheduleOnce(5.seconds) {
            system.shutdown()
        }
    }
}
```

When we run this code, we end up seeing a decent amount of log output

telling us about changes in the rate of climb.

```
[INFO] ... Plane giving control.  
[INFO] ... Altimeter changed rate of climb to 5000.000000.  
[INFO] ... Altimeter changed rate of climb to 0.000000.  
[INFO] ... Altimeter changed rate of climb to 2500.000000.  
[INFO] ... Altimeter changed rate of climb to 0.000000.
```

For brevity, some of the logging information has been removed and replaced with “...”. Akka will output much more interesting information about which Actors are logging the information using some of the Actor attributes that we have yet to encounter.

The `main()` method of the `Avionics` object actually completes very quickly, as most of the work is *scheduled* immediately and then executed later.

First, we use a facility of the Akka Futures implementation called `Await.result` that will block on the response from the Plane to the `GiveMeControl` message. I said earlier that when talking about the `ask` syntax, blocking on responses from Actors isn’t a great idea, but for this little test driver, it’s very convenient.

We first *ask* the Plane for the controls, which returns a Future. Unfortunately, the Future only knows about the type that is returned as an Any, due to the fact that messages between Actors are of type Any. As a result of that loss of typing, we must use the Future’s `mapTo` facility to coerce it down to the type we’re expecting. Don’t worry; we’ll be covering all of this stuff in detail later. If the Plane doesn’t give us that result within 5 seconds, then a timeout exception will occur, but for us 5 seconds is way more than enough.

Once we have that control, we set up a whole bunch of commands for the Scheduler to execute in the future. Akka keeps the system running until the system is shut down, so if we didn’t shut it down manually 5 seconds after the reception of the controls, then our app would never exit. Shutdown is the purpose of the last scheduled command.

But this is somewhat incomplete at this stage, wouldn’t you say? The pilot (for now, he’s `main`) gets control, can manipulate those controls, and the Altimeter senses what’s happening, but shouldn’t the Altimeter tell someone about it? I think so. Let’s implement a rudimentary event listener using Actors on the Altimeter with which the Plane can register.

Getting Updates from the Altimeter

Since our Actor is untyped, it makes for a nice generalized endpoint to receive events. We can make a trait that encapsulates this notion and mix it into the Altimeter.

```
package zzz.akka.avionics

import akka.actor.{Actor, ActorRef}

object EventSource {
    // Messages used by listeners to register and unregister themselves
    case class RegisterListener(listener: ActorRef)
    case class UnregisterListener(listener: ActorRef)
}

trait EventSource { this: Actor =>
    import EventSource._

    // We're going to use a Vector but many structures would be adequate
    var listeners = Vector.empty[ActorRef]

    // Sends the event to all of our listeners
    def sendEvent[T](event: T): Unit = listeners foreach { _ ! event }

    // We create a specific partial function to handle the messages for
    // our event listener. Anything that mixes in our trait will need to
    // compose this receiver
    def eventSourceReceive: Receive = {
        case RegisterListener(listener) =>
            listeners = listeners :+ listener
        case UnregisterListener(listener) =>
            listeners = listeners filter { _ != listener }
    }
}
```

We’re pretty familiar with most of what’s here already. There’s some Scala self-typing with `this: Actor =>` and some type specification in the `sendEvent` method, but the interesting part with respect to Akka is the `eventSourceReceive` method. No Actor definition is valid without a `receive` method, but it can also only have one. If an Actor mixes in traits that are also Actor-like, then we probably need to compose the `receive` method from

many different partial functions. We provide one of the pieces of the final receive method here with `eventSourceReceive`.

Modifying the Altimeter

Now that we have the `EventSource`, we can mix it into the `Altimeter`. Let's look at the changes that need to be made to the `Altimeter` to make this happen.

First, add the message to the `Altimeter` object:

```
// Sent by the Altimeter at regular intervals
case class AltitudeUpdate(altitude: Double)
```

Then we need to change the definition of the `Altimeter` Actor:

```
class Altimeter extends Actor with ActorLogging with EventSource {
    ...
}
```

Now we need to compose the `receive` method, as already indicated. We must take what we already have defined as `receive` and change its name so that we can compose the final `receive` method out of the respective parts.

```
def altimeterReceive: Receive = {
    ... contents of old receive method ...
}

def receive = eventSourceReceiveorElse altimeterReceive
```

Note that for `eventSourceReceive` and `altimeterReceive` we needed to be specific about the return type. As a derivation of `Actor`, the `receive` method is abstract and has a return type associated with it already. However, our other methods don't get that for free, so we use the convenience type already defined on the `Actor`:

```
type Receive = PartialFunction[Any, Unit]
```

We're almost done with the `Altimeter`. We've done all of our plumbing, but we haven't actually sent any events yet. Let's modify the handler for the `Tick` message:

```
case Tick =>
  ... as before ...
  sendEvent(AltitudeUpdate(altitude))
```

Fantastic! Our Altimeter is now a solid source of altitude information for anyone who wants it.

Modifying the Plane

We could build and run this as before and everything would still work, but we haven't traveled the final mile yet. The Plane wants to get updates about the altitude. When the Plane starts up, we must register with the Altimeter; the best place to do this is in a life-cycle hook that the Actor provides:

```
import EventSource._

override def preStart() {
  altimeter ! RegisterListener(self)
}
```

Akka will call this method before the Plane starts up, which will ensure that our Plane is appropriately hooked into the Altimeter to receive updates about altitude.

The astute reader (yeah, I'm looking at you) might be wondering about the `UnregisterListener` message, and why we aren't worried about unregistering the Plane from the Altimeter. This will become clear as we continue, but there are several reasons why we don't need to worry about it:

- Technically, we could do it. Akka provides another life-cycle hook called `postStop()` that we could use to send the `UnregisterListener` message to the Altimeter.
- There's really no point. Since we went through the trouble of creating the Altimeter inside the Plane, we've made the Altimeter's life dependent on the Plane's life. When the Plane dies, so does the Altimeter, so sending the message to unregister the Plane isn't really all that useful.
- Even if the Altimeter lived beyond the Plane, and we didn't clear the registration with the Altimeter (and if you think that this would be bad form, I'd agree with you), Akka still has us covered. Messages sent to

recipients that no longer exist still have a deterministic destination: the Dead Letter Office. We'll be covering the Dead Letter Office soon—for now, the usual definition should suffice. It's simply where letters go when the recipient can't be found.

You don't need to really understand this stuff yet; these concepts will become as clear as a mud-free river pretty soon.

Lastly, we need to put in a message handler in the Plane's receive method:

```
def receive = {
    case AltitudeUpdate(altitude) =>
        log.info(s"Altitude is now: $altitude")
    ... and as before ...
}
```

And we're done.

Flying the Plane

If we run it now, using our same driving main method, we will see the output that follows:

```
[INFO] ... Plane giving control.
[INFO] ... Altitude is now: 0.000000
[INFO] ... Altimeter changed rate of climb to 5000.000000.
[INFO] ... Altitude is now: 16.666667
[INFO] ... Altitude is now: 33.333333
[INFO] ... Altitude is now: 50.000000
[INFO] ... Altitude is now: 66.583333
[INFO] ... Altimeter changed rate of climb to 0.000000.
[INFO] ... Altitude is now: 66.583333
```

```
[INFO] ... Altitude is now: 66.583333
[INFO] ... Altitude is now: 66.583333
[INFO] ... Altitude is now: 66.583333
[INFO] ... Altimeter changed rate of climb to 2500.000000.
[INFO] ... Altitude is now: 74.916667
[INFO] ... Altitude is now: 83.208333
[INFO] ... Altitude is now: 91.583333
[INFO] ... Altitude is now: 99.875000
[INFO] ... Altitude is now: 108.208333
[INFO] ... Altimeter changed rate of climb to 0.000000.
[INFO] ... Altitude is now: 108.208333
```

Congratulations! You've got a Plane.

5.6 Tying It Together

At this point, you might have a lot of moving parts flying around in your head that lack a bit of cohesion and it's always helpful to pull it back together into a clear picture.

Figure 5.4 shows us the important bits of our Plane simulation thus far. That picture underscores one of the reasons why Actor programming is so powerful. Often, we'll draw diagrams like that on a grease board in order to help others, and even ourselves understand the structure and behaviour of our application designs, but they tend to be illustrative devices only; the actual nuts and bolts of the software usually don't have direct mappings from the diagrams we draw.

In our case here, the entities we see have actual live counterparts in the application and the messages written on those arrowed lines are real messages. In other words, we can draw direct parallels from the pictures we see to the code we write, and when someone sees our pictures, they can relate what they see to our code as well. By looking at Figure 5.4, you know that there is an Actor called *Plane* and that that Actor will accept *AltitudeUpdate* and *GiveMeControl* messages.

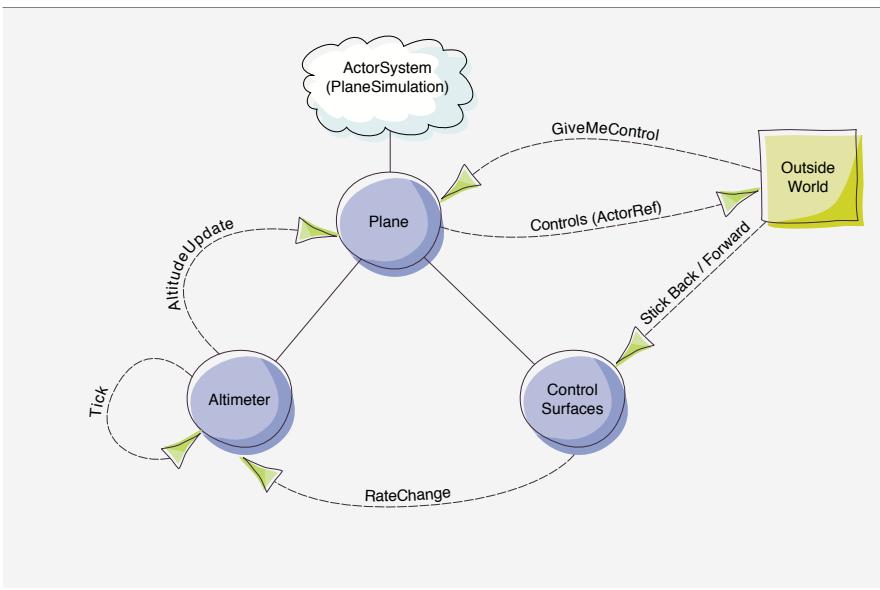


Figure 5.4 · This is what our Plane currently looks like, from the view of the avionics-driving main, which we depict as the Outside World. Solid lines indicate a physical binding—i.e., the ActorSystem owns the Plane and the Plane owns the Altimeter and ControlSurfaces, whereas the dotted lines indicate a reference relationship for message sending.

We don't need to hunt through the code in order to understand the application to a fairly high degree because hunting through the code won't reveal much more to us. Sure, there will be some subtle nuances here and there, some clever and not-so-clever tricks, just like any other mountain of code, but those will be exceptional cases that you couldn't draw a picture of anyway.

Note

This is part of the power of the Actor programming paradigm—the power to understand and convey your application's design and implementation at highly abstract levels, without being so abstract that they're useless descriptions.

5.7 How Message Sending Really Works

It's time to pull the curtain back a little bit so that you have a better understanding of how the Actor model helps you send and route messages. We've already covered the Actor's basic entity layout back in [Section 5.1](#), but there's more to it than that when we start working with more than one Actor at a time.

One of the most illuminating things we can see is the true definition of the method for the tell syntax, which we will pull from the Scala implementation of the ActorRef, which is called `ScalaActorRef`:

```
def ! (message: Any)(implicit sender: ActorRef = null): Unit
```

What can we see from this declaration?

1. `!` is a side-effect function. We know it does something, but it returns `Unit`, so whatever it does is a side effect of its execution. This isn't a surprise. We know that Actors don't compose and that tell syntax enqueues messages.
2. As promised, a message can be anything we like since it can be a subtype of `Any`.
3. Every time we call `!`, there is an `implicit ActorRef` added to the call, if one exists.

That last point is the one in which we're really interested. Through the magic of Scala's `implicits`, we get a nice clean syntax for sending messages that lets us ignore the plumbing that sets up the reference to the message's sender. The receiving Actor will get a reference to the sender, provided that the sender is actually an Actor (as opposed to `main()` in some of our previous encounters with `!`).

If we were to look at the ScalaDoc for `akka.actor.Actor`, we would see a nifty little member value there:

```
implicit val self: ActorRef
```

When you're inside an Actor, you have an `implicit` in scope that can satisfy the second curried parameter of the `!` method, and this is how the sender is populated when the message is sent. When you send your message,

Akka puts your current Actor's ActorRef and the message itself inside an envelope and that's what actually gets delivered to the target Actor.

You can also see from the declaration that a default value applies to the `implicit` should nothing be in scope to satisfy it – `null`. There's no big surprise to this... if a value can't be found for it then it's presumed that no such value exists and `null` is the only reasonable thing to put in there. `null` is, effectively, a sentinel value that Akka uses to understand that the sender's context is outside of its influence. The message still gets there, but it's truly a one-way message since there's nobody to receive any response.

So, if you're inside an Actor and want to `null` out the sender, then you can easily specify the sender explicitly:

```
someActorSomewhere.!("This is a message")(null)
```

But that's pretty ugly. In this case, it's much nicer to use the `tell` function, which is more generally defined on the `ActorRef` as opposed to the `ScalaActorRef`:

```
// tell is defined as
def tell (msg: Any, sender: ActorRef): Unit
def tell (msg: Any): Unit

// so we can do this
someActorSomewhere tell "This is a message"
```

Accessing the Sender

Inside the receiving Actor, we can get the reference to the sender using an aptly named method:

```
def sender: ActorRef
```

Look at that again. See it? That's a *method* not a *value*. Not only that, it's a method that's defined without parens. This might lead some to believe that it's equivalent to a value, but that's not the case. If you really wanted to be dogmatic about it, you could argue that it must be defined as `sender()` since that would be a better indicator that its return value is dependent on some sort of internal state, which can change from moment to moment. But it's much more pleasing without the parens, don't ya think?

This sender method gives access to the sender that hitched a ride on the incoming message that the Actor is currently processing. But remember that because it's a method and one that can change its return value (conceivably with every single incoming message), you have to treat it with care.

For example, something like this would be a bad idea:

```
case SomeMessage =>
    context.system.scheduleOnce(5 seconds) {
        sender ! DelayedResponse
    }
```

It's quite likely that the value returned from the call to `sender` 5 seconds from now won't be the value you were hoping for. To make this work out the way you'd like, you need to freeze the value from `sender` in a `val`:

```
case SomeMessage =>
    val requestor = sender
    context.system.scheduleOnce(5 seconds) {
        requestor ! DelayedResponse
    }
```

Tip

Using `sender` inside of a closure is a textbook way of giving away the keys to the fortress. It's easy to do, and the only thing you can do to prevent it is to not do it. In short order, you're going to recognize this sort of mistake very easily and you'll avoid it without trouble.

Null Senders

So what happens when the sender is `null`? The quick answer is that we don't get a dreaded `NullPointerException` if we try to send a message to it. Akka uses `null` only as a sentinel value to let the toolkit know that there's no hitchhiker to add to the message. When this situation occurs, Akka attaches a default sender called the *Dead Letter Office*, which is a single Actor instance per `ActorSystem` and can be accessed directly from the `ActorSystem` via the `deadLetters` method.

This means that you can always use the result from the `sender` method with confidence. It may be that the receiver of your response message is

dead or never existed in the first place, but in either case the message goes to a deterministic endpoint: the Dead Letter Office.

Forwarding

Message *forwarding* is another method of sending a message to an Actor. It works such that the one doing the forwarding is invisible to the receiver. If *A* sends to *B* and *B* forwards that message to *C*, then *C* sees the sender of the message as *A*, not *B*. It should now be fairly obvious how forwarding works. When an Actor forwards a message from itself to another ActorRef, it's really the sender that's most important. For example, the equivalent of a call to `forward` is most definitely not:

```
case msg @ SomeMessage =>
    someOtherActor ! msg
```

When the receiving Actor, `someOtherActor`, accesses the sender in order to know who to respond to, the value he'll get is the one that made the call to `!`, not the original sender, and that's not the semantic of forwarding a message. When you forward a message, you're handing it off to someone else, and it's supposed to look like you were never involved. This is the same as forwarding a phone call. The caller is passed off to someone else, and the guy who did the passing isn't involved in the conversation anymore. Therefore, that `forward` only needs to preserve the original sender in order to make the forward a success.

```
def forward (message: Any)(implicit context: ActorContext): Unit
```

The implicit parameter here is a bit confusing. You would expect that the `implicit` would be an `ActorRef`, but the implicit `ActorRef` in the Actor is already known to be `self`, which won't work here since it's the reference to `self` that we're trying to avoid. It turns out that the `ActorContext` holds enough information that we can extract the original sender and thus have a nice interface (i.e., the `implicit` parameter) and deterministic behaviour. The `ActorContext` will be covered in more detail in later chapters.

We aren't giving too much away about Akka internals if we show the implementation of `forward` because it's only doing what we would guess that it's doing anyway:

```
def forward(message: Any)(implicit context: ActorContext) =  
    tell(message, context.sender)
```

These different ways of sending are summarized in Figure 5.5.

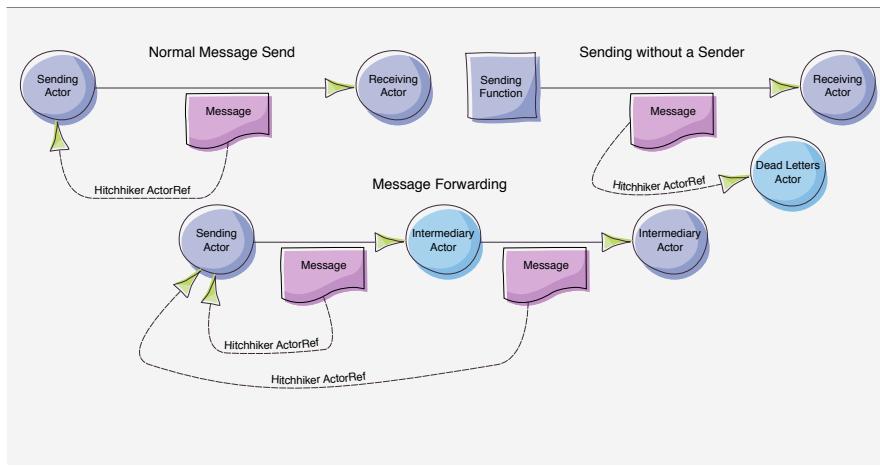


Figure 5.5 · Three different representations of how message sending works. Every time a message is sent, the message gets a passenger—a reference to the sender. If `!` is called from within an Actor, then that Actor becomes the sender, or it's the Dead Letter Office if you're not inside an Actor. When forwarding, we propagate the original sender with the message in order to implement standard forwarding semantics.

5.8 The ActorSystem Runs the Show

It's time to look at something we've seen several times now but we haven't really explained: the `ActorSystem`. We won't cover all of it right now because there's temporal relevancy¹⁰ that must be considered, but certain aspects of the `ActorSystem` are important to understand before we go much further.

It's absolutely impossible to create an Actor on its own; someone else needs to "own" it. This requirement ensures that Actors form a hierarchical structure – or a tree. There are some practical reasons why the tree should be

¹⁰I never thought I'd have a use for those two words in the same sentence. Keen.

“decently” formed, but theoretically there’s no reason why you can’t make a tall, skinny tree or a fat, bushy tree. Now, while it would be cool to say that it’s turtles all the way down again, the hierarchy terminates at the ActorSystem. The ActorSystem is special because it is always the root of its own hierarchy and it can never appear anywhere else in the tree but the root.

The ActorSystem is where the turtles stop because it’s not an Actor; it’s a *house* for Actors. Not only do your Actors live in that house, so do many other Actors that Akka uses for its own purposes. Beyond that, it provides access to other facilities and functions such as:

- System configuration. When you need access to the configuration of Akka, and of your own application, you can get it from the ActorSystem.
- The default Scheduler, which we’ve seen earlier, is also available via the ActorSystem.
- The entire event stream of your application is also available. You can see everything from this access point.
- The Dead Letter Office is available, which means you can hook things up to it directly, if you’d like.
- It’s the ActorSystem from which we can obtain references to currently running Actors in its hierarchy via a set of functions called `actorFor()`.
- You can get the uptime of your app from here as well.
- There are functions that let you shut the system down, as well as stop individual Actors.

There are several other goodies in there, and we’ll look at most of them as we proceed, but at this point a couple of things should be clear:

1. The ActorSystem is definitely not an Actor—it’s special.
2. While you can make more than one of them, there will be orders-of-magnitude fewer ActorSystems than there will be Actors in your application.

This second point has the most impact on your life, with respect to Actors. The ActorSystem will be the root of the hierarchy, and thus the hierarchy is real; you must compose your Actor application as a hierarchy of Actor instances.

Note

The imposition of a hierarchy on your Actor application is the most significant design requirement that Akka puts in front of you.

That requirement carries a naming consistency and uniqueness along with it. ActorSystems must have a globally unique name in your application, and the Actors that live within it must also have a unique name at any given level of your hierarchy.

We'll see an important practical impact of the ActorSystem on your code when we start looking at tests.

5.9 Chapter Summary

We've covered a lot of ground in this chapter and we've learned a lot about programming with Actors in Akka. Armed with the knowledge you now have, you could conceivably go and write very simple, highly concurrent applications with Akka. Of course, I don't recommend you do that just yet, but you can with what you know. Let's summarize what we've learned:

- Actor creation from both inside and outside of an Actor with the `actorOf` factory method and the `Props` class
- Types of messages that we can use and have a solid pattern for defining and using those messages
- Sending messages in a non-blocking manner and how to program *reactively*
- Logging information by mixing in the `ActorLogging` trait
- Composing Actor behaviour through traits and partial functions
- Scheduling actions in the future using the `Scheduler`
- Futures, enough to be able to synchronize simple behaviours if need be

- How Actor references travel back and forth between Actors during message passing
- The ActorSystem, from an introductory level
- What it means to write software in the new paradigm that Akka has given us

Of course, this is just the beginning, but we're well on our way to programming with Akka. By now, you should be feeling more comfortable about thinking in the Akka programming style, and that's really what it's about. Akka provides you with state-of-the-art tools and techniques, but perhaps the greatest benefit is how it enables you to think about concurrent applications in a new way. Learning the tools and techniques is a vital step, but getting comfortable *thinking* about concurrent programming in the way that Akka provides is where the real benefit lies.

Chapter 6

Akka Testing

No code is valid unless you can prove it to be and this is no less true of code written in Akka. Unfortunately, Akka programs, just like all other programs, will do exactly what you tell them to do rather than what you want them to do. So, we test!

You can test Akka with any testing framework, but we'll use ScalaTest.¹ This is just a personal preference, so you can use whatever you like.

6.1 Making Changes to SBT

Along with ScalaTest, we'll include Akka's `testkit` module, which gives us a load of nifty tools to help us test what we've written. We won't be covering all of the test variants here, but rest assured that Akka has you covered for everything from surgically targeted unit testing to simulating multi-node testing by spooling up any number of JVMs for you to launch and deterministically run your tests.

Let's modify our SBT dependencies so they now look like this:

```
libraryDependencies ++= Seq(  
    "org.scalatest" %% "scalatest" % "1.9-2.10.0-M6-B21" % "test",  
    "com.typesafe.akka" % "akka-testkit" % "2.1"  
    "com.typesafe.akka" % "akka-actor" % "2.1"  
)
```

Now execute an `sbt update` and you're good to go.

¹<http://www.scalatest.org/> by Bill Venners, et al.

6.2 A Bit of Refactoring

You may have noted that some of the code we wrote earlier isn't quite as testable as we might like. [Figure 6.1](#) shows the subset of code we intend to test. In particular, we're talking about the relationship between the Altimeter and the EventSource. Let's refactor these pieces of code so that they're a bit more decoupled and easier to test.

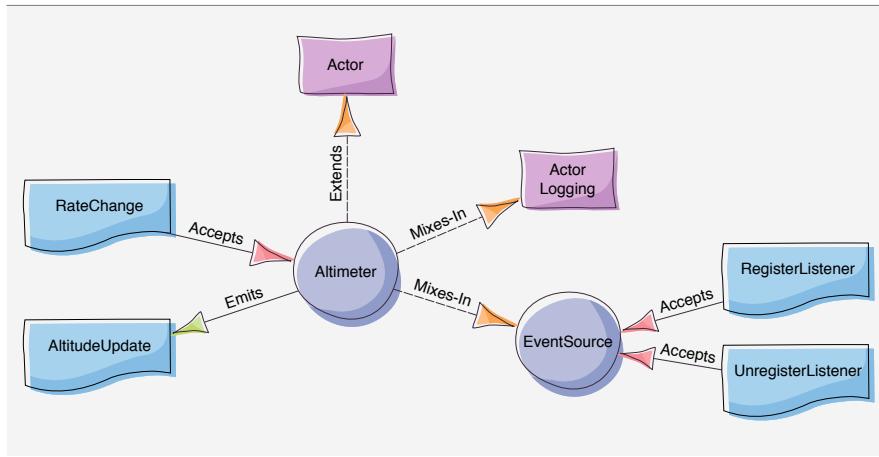


Figure 6.1 · When we test our Altimeter, we are technically testing every one of its components.

The target line of code we're interested in fixing is:

```
class Altimeter extends Actor with ActorLogging with EventSource {
```

We can see here that we've tightly coupled the Altimeter to the EventSource, which isn't ideal for the purposes of our testing. Our goal with this refactoring exercise is to remove this hard-wired dependency. Doing this sort of thing is always a matter of choice and one could argue that this case doesn't warrant it, but if I didn't do it, you wouldn't see it.

Slicing Up the Altimeter

The first thing to do is to abstract the EventSource into a trait that we can extend as needed.

```
trait EventSource {  
    def sendEvent[T](event: T): Unit  
    def eventSourceReceive: Actor.Receive  
}  
  
trait ProductionEventSource extends EventSource { this: Actor =>  
    // Original contents of EventSource here  
}
```

Next, we need to change the definition of the Altimeter so that it self-types to the EventSource, and we need to alter the factory method so that we construct the Altimeter with the ProductionEventSource.

```
class Altimeter extends Actor with ActorLogging { this: EventSource =>  
    // Original contents of Altimeter here  
}  
  
object Altimeter {  
    // Content as before. We're changing the factory method.  
    def apply() = new Altimeter with ProductionEventSource  
}
```

Done. Now we have our class working exactly the way it was before, but we've got a hook that we can use to change the dependencies for testing.

6.3 Testing the EventSource

Our EventSource trait was designed to be used to send stuff to those who want it. Akka will give us several key components that help us make sure it does what it's supposed to do.

```
package zzz.akka.avionics  
  
import akka.actor.{Props, Actor, ActorSystem}  
import akka.testkit.{TestKit, TestActorRef, ImplicitSender}  
import org.scalatest.{WordSpec, BeforeAndAfterAll}  
import org.scalatest.matchers.MustMatchers  
  
// We can't test a "trait" very easily, so we're going to create a specific  
// EventSource derivation that conforms to the requirements of the trait so  
// that we can test the production code.  
class TestEventSource extends Actor with ProductionEventSource {
```

```
def receive = eventSourceReceive
}

// "class"Spec is a decent convention we'll be following
class EventSourceSpec extends TestKit(ActorSystem("EventSourceSpec"))
    with WordSpec
    with MustMatchers
    with BeforeAndAfterAll {

import EventSource._

override def afterAll() { system.shutdown() }

"EventSource" should {
    "allow us to register a listener" in {
        val real = TestActorRef[TestEventSource].underlyingActor
        real.receive(RegisterListener(testActor))
        real.listeners must contain (testActor)
    }

    "allow us to unregister a listener" in {
        val real = TestActorRef[TestEventSource].underlyingActor
        real.receive(RegisterListener(testActor))
        real.receive(UnregisterListener(testActor))
        real.listeners.size must be (0)
    }

    "send the event to our test actor" in {
        val testA = TestActorRef[TestEventSource]
        testA ! RegisterListener(testActor)
        testA.underlyingActor.sendEvent("Fibonacci")
        expectMsg("Fibonacci")
    }
}
}
```

Akka Adds-On to Test Frameworks

This special `import` statement in our test is important:

```
import akka.testkit.{TestKit, TestActorRef, ImplicitSender}
```

Each of these components provides us with specific helpers that we can use to write our tests:

- **TestKit:** Gives us the basic framework we need to work with Actors; this includes access to the ActorSystem as well as helper methods for dealing with responses from Actors under test. There are also some primitives for dealing with time in general. Note that at the end of all of the tests we need to `shutdown()` the ActorSystem.
- **TestActorRef:** Gives us access to the underlying Actor we have written. Everything that's publicly accessible on our Actor is now available to our test. Don't be a smart alec and think you should use this in real life. Respect the Fortress.
- **ImplicitSender:** This nice part of the kit lets us receive responses to messages that we may send to our Actor under test directly in our test code. Functions, such as `expectMsg()` and `expectMsgPF()`, are enabled by `ImplicitSender`.

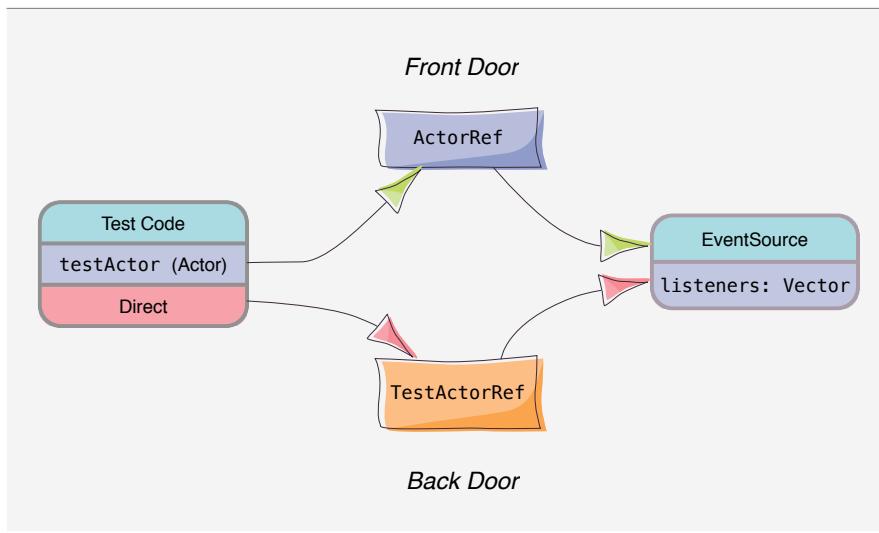


Figure 6.2 · Normally, the EventSource Actor is unavailable to our runtime code because Akka hides it behind an `ActorRef`. The TestKit's `TestActorRef` gives us access to the Actor's internals so we can poke and prod it directly.

Figure 6.2 shows us the macro view of what's happening in most of these tests. Normally, Akka ensures that nothing can access your Actor's internals by hiding things behind a location-neutral, type-independent `ActorRef`. This

is a huge benefit to coding in the Actor paradigm and is key to helping you deliver scalable and reliable applications, but when it comes to unit testing, it can really get in your way.

To remove that barrier, Akka provides you with the `TestActorRef` that gives you a magic key to the back door of the Fortress, so you can just waltz inside and do pretty much anything you like. Now, we already know this is a terrible idea, but Akka is pragmatic about testing and provides you with what you need to get things done. It's up to you to make sure that you use this particular super power for good and not evil.

We can do this in a testing situation because we are entirely in control of the environment. Rather than running in the real world, where any mischievous monkey can come along at any moment and chew the socks right off your feet, our Actor is running in an environment where the monkey is our favorite pet. It still chews the socks right off your feet, but it only does it when we say so.

The `ImplicitSender` is what allows your test code to react directly to messages that are sent from your code under test using the methods supplied in the `TestKit` (`expectMsg()` and friends). But in order for the code under test to be able to send messages to your testing code, the `TestKit` supplies the `testActor`. When you need to inject an `ActorRef` into code that's under test, you can supply the `testActor`, which will route messages back to the `TestKit`, enabling the use of methods like `expectMsg()`.

The plumbing and functionality of the `TestKit` is really quite powerful. And the beauty of the untyped Actor gives you the kind of flexibility that you probably always wanted in your tests but never actually had; the untyped Actor provides a hook that makes it a very nice, and very natural mock object.

The EventSource Tests

The `EventSource` is so wonderfully simple that we can really get down to the unit level and test it directly using the facilities that Akka provides in the `TestActorRef`. Our first two tests can poke at the internals of our `EventSource` directly, calling the `receive` method and poking at our list of listeners directly. This is as comfortable as any unit test we're used to writing.

The last test is really the functional test of the `EventSource` and the best way to test it is to use asynchronous concepts. We can poke at the `EventSource` directly, but the internals of it will ensure that asynchronous

code is used (i.e., the `sendEvent()` method will bang on `ActorRefs`), so we're stuck with a small bit of non-determinism. Here, we use the `expectMsg("Fibonacci")` method, which is part of `TestKit` to ensure that the `EventSource` sends us a message within an appropriate amount of time.²

```
"send the event to our test actor" in {
    val testA = TestActorRef[TestEventSource]
    testA ! RegisterListener(testActor)
    testA.underlyingActor.sendEvent("Fibonacci")
    expectMsg("Fibonacci")
}
```

The code on the second line provides a test reference to our Actor so that we can go through the front door and the back door to test our `EventSource`. We then go through the front door to register our `testActor` (we could have used the back door but this gives us some variety) and then through the back door again to invoke the behaviour we want to test. The advantage here is that we can go directly for the `sendEvent()` method right from the test. If we didn't have access to the back door, then we would have had to do more work in our `TestEventSource` subclass in order to invoke the behaviour. Yes, that's right... Akka's awesome.

The `expectMsg` assertion is really interesting, mostly in its simplicity. It's the fact that we have an `ImplicitSender` and `TestKit` mixed into our test that allows us to have this simplicity. This one line of code will use our `testActor`'s `receive` method to receive messages from our `EventSource`. It will expect to receive a `String` with the value of "Fibonacci" within a default timeout threshold. Failure to meet any of those conditions will fail the test but, assuming that the `EventSource` does what it's supposed to, the test will pass as quickly as possible.

6.4 The Interaction Between `ImplicitSender` and `testActor`

When we were writing our Actors, we learned about the interplay between messages, Actors, and senders. When a message gets sent, the sending Actor

²We just let Akka use the default value for the timeout on `expectMsg` rather than specifying our own.

hitches a ride along with the message so that the receiver can know who sent it and can then reply to it.

The TestKit defines a member called `testActor`, which is a full-fledged `ActorRef` that we can send to various components any time they need an `ActorRef` to talk to. This ensures that the `testActor` can easily be a primary player in any test that requires separate Actors. But if someone's going to send messages to the `testActor`, then it's reasonable to ask how the test itself will see those messages.

Since our test specification class isn't an Actor, there is no implicit sender that Akka can stick on to the traveling message. This is where the Implicit-Sender comes in; the definition of `ImplicitSender` is so simple we might as well have a look at it:

```
trait ImplicitSender { this: TestKit =>
    implicit def self = testActor
}
```

This is one of the places where the power of Scala's implicits and the elegance of Akka's design really shines. The implicit is so flexible, and Akka's choice of where to use it is so appropriate, that we can now make tests that *naturally* fit into the Akka paradigm with a simple trait mixin.

Now that we have all of this plumbing in place, calls like the following become easy:

```
testA ! RegisterListener(testActor)
...
expectMsg("Fibonacci")
```

Not only do we have the `testActor` available to send to the `EventSource`, we also have the `testActor` as the sender of the message, just in case there's a response that we'll need to assert. The magic of the `testActor` and of the `TestKit` class ensure that messages routed to the `testActor` are easily accessible from our tests and can be asserted in a variety of ways.

6.5 TestKit, ActorSystem, and ScalaTest

In the last chapter, we saw the `ActorSystem` and learned that, in order to write an Actor application, we must have at least one instance of an `ActorSystem`.

An Actor test specification is of course no different from any other Actor application in this regard. In order for the TestKit to do what it does, it will need to have an ActorSystem and we must supply that ActorSystem during the TestKit's construction, as we have already seen:

```
class EventSourceSpec extends TestKit(ActorSystem("EventSourceSpec"))
```

When we construct our test specification in this way, we create a single instance of the ActorSystem for all of our specification's tests. If the tests are completely independent of one another (as is the case in our tests thus far), then we have an ideal situation. The ActorSystem only needs to construct once, and our tests run quickly and well. We may even be able to run them in parallel and still have completely deterministic results.

However, there are times when the existence of a single ActorSystem across all tests in a given specification can be a problem. That single context may require that all of our tests run sequentially, since the single instance is effectively “shared state” across all of our tests. For example, we discussed that ActorSystems demand *uniqueness* in naming Actors. Knowing that, we can see that running these tests in parallel would yield unreliable results:

```
class MyActorSpec extends TestKit(ActorSystem("MyActorSpec"))
  with WordSpec
  with MustMatchers
  with BeforeAndAfterAll
  with ParallelTestExecution {

  override def afterAll() { system.shutdown() }

  def makeActor(): ActorRef = system.actorOf(Props[MyActor], "MyActor")

  "My Actor" should {
    "throw an exception if it's constructed with the wrong name" {
      evaluating {
        val a = system.actorOf(Props[MyActor]) // use a generated name
      } must produce [Exception]
    }
    "construct without exception" {
      val a = makeActor()
      // The throw will cause the test to fail
    }
    "respond with a Pong to a Ping" {
```

```
    val a = makeActor()
    a ! Ping
    expectMsg(Pong)
}
}
```

If we were to run these in parallel (using ScalaTest's `ParallelTestExecution` trait), then the presence of the second and third test would cause an `InvalidActorNameException` to be thrown from one or the other test, causing it to fail.

We would have to run these tests sequentially in order to give them an environment in which they can run safely. But this wouldn't work either because we aren't shutting down the Actors that we create between tests. To handle this problem, we can change the above code to the following:

```
// Add BeforeAndAfterEach as well as remove the parallelism
class MyActorSpec extends TestKit(ActorSystem("MyActorSpec"))
    with WordSpec
    with MustMatchers
    with BeforeAndAfterAll
    with BeforeAndAfterEach {
    override def afterAll() { system.shutdown() }
    def makeActor(): ActorRef = system.actorOf(Props[MyActor], "MyActor")
    override def afterEach() {
        system.stop(/* Actor reference here */)
    }
    ...
}
```

In other words, between each test we could shut down the Actor that we want to recreate. This doesn't work though because the `stop()` function is asynchronous; the next test will start way before the `stop()` has had time to complete. Akka includes a helper function called `gracefulStop()` that you might think to employ here. You can call `gracefulStop()`, which returns a Future on which you can await completion. Without getting into too much detail, it would look like this:

```
// Await on the result, giving timeouts for the gracefulStop as well as
```

```
// the timeout on the Future that's running
Await.result(gracefulStop(/* actor reference */, 5 seconds)(system),
             6 seconds)
```

We don't get into too much detail because this doesn't work either. The reason that it doesn't do what we want is because it's not intended to do what we want. `gracefulStop()` only ensures that the Actor's life-cycle functions have run and thus its `postStop()` callback has executed, which isn't the same as saying that the `ActorSystem` has had a chance to reap the identifier.

So what do you do when you need to isolate your tests from one another and the `ActorSystem`'s causing the lack of isolation? When we run up against these situations, `ScalaTest` provides us with a great solution: we push the `TestKit` down.

Test Isolation

To get true isolation between our tests, we'll create a helper class that will handle the `ActorSystem` for us:

```
import akka.actor.ActorSystem
import akka.testkit.{TestKit, ImplicitSender}
import scala.util.Random

class ActorSys(name: String) extends TestKit(ActorSystem(name))
  with ImplicitSender
  with DelayedInit {
  def this() = this(s"TestSystem${Random.nextInt(5)}")
  def shutdown(): Unit = system.shutdown()
  def delayedInit(f: => Unit): Unit = {
    try {
      f
    } finally {
      shutdown()
    }
  }
}
```

This helper gives us a few features that we need:

- It mixes in all of the Akka behaviour we want in our tests: the TestKit and the ImplicitSender.
- It hides the ActorSystem from us and gives us a simple constructor that takes no parameters. The ActorSystem that we'll create will have a random name. This shouldn't make it difficult to debug any test errors, but if you don't like it, then you can use the one-parameter constructor.
- It moves our test code out of the constructor and into the `delayedInit` function so that we can wrap it. We now no longer need to worry about shutting down the ActorSystem after all of the tests are complete. In this context, there's only one test and we can shut the ActorSystem down after that test completes.

To use it, we have to construct a new one in which our test will live. We can alter the parallel execution example from before and get true isolated parallelism:

```
class MyActorSpec extends WordSpec
    with MustMatchers
    with ParallelTestExecution {
  def makeActor(): ActorRef = system.actorOf(Props[MyActor], "MyActor")
  "My Actor" should {
    "throw when made with the wrong name" in new ActorSys {
      evaluating {
        val a = system.actorOf(Props[MyActor]) // use a generated name
      } must produce [Exception]
    }
    "construct without exception" in new ActorSys {
      val a = makeActor()
      // The throw will cause the test to fail
    }
    "respond with a Pong to a Ping" in new ActorSys {
      val a = makeActor()
      a ! Ping
      expectMsg(Pong)
    }
  }
}
```

}

This has simplified our lives a fair bit! We don't need to have the BeforeAndAfterEach or the BeforeAndAfterAll included, and our specification itself is no longer a TestKit or an ImplicitSender. On top of that, we've gained full isolation between tests and the ability to run our tests completely in parallel. Since the ActorSys is a class, it can be derived from and specialized further. The ActorSys would make a perfectly fine spot for the definition of a shared fixture, if it applies. For example, instead of having mutable data at the spec level, you could put it in the ActorSys and let the constructor initialize the data appropriately.

There couldn't possibly be a down side to such awesomeness, right? Of course there is...there *always* is. It's speed. We've potentially gained some speed by enabling parallelism, but this should only manifest when you have individual tests that run long. The real overhead here is in the ActorSystem's construction. While we hardly ever notice this construction due to the fact that it's usually only done once or twice, it isn't as trivial as creating an Actor. Now we're creating many of them all at the same time, so the visibility of the expense increases. You're going to notice it.

So, we need to make decisions about when it should apply. If you don't need to apply this high level of isolation, then you might not want to do so. There's nothing technically wrong with doing it when you don't need to, but you might see a slight speed decrease in your tests, and that isn't something we should just lightly ignore.

I will be using this isolation where appropriate and avoid it where we don't need it. You might want to do the same.

6.6 Testing the Altimeter

When we're testing code, it's almost as important to be *fast* as it is to be *correct*. We need our tests to run as quickly as possible to ensure that our edit-compile-test cycle is as short as it can be. This is tricky when it comes to certain types of tests with Actors because those tests may require that you go through the front door. Concurrency often means things like sleeps and timeouts. Tests should succeed as quickly as possible, so if there's a sleep in the path of success then something's wrong.

Akka provides several functions in the TestKit that help us keep the path of success free of unwarranted sleeps. We've already seen the expectMsg()

method from the toolkit. Most of the other functions that it provides follow the same concept as `expectMsg()`, but there are a couple of them that provide some extra nifty behaviour. We'll take a look at another one, but also look at a non-Akka-provided mechanism that makes it really easy to get these tests running quickly.

```
package zzz.akka.avionics

import akka.actor.{Actor, ActorSystem, Props}
import akka.testkit.{TestKit, TestActorRef, ImplicitSender}
import scala.concurrent.util.duration._
import java.util.concurrent.{CountDownLatch, TimeUnit}
import org.scalatest.{WordSpec, BeforeAndAfterAll}
import org.scalatest.matchers.MustMatchers

object EventSourceSpy {

    // The latch gives us fast feedback when something happens
    val latch = new CountDownLatch(1)

}

// Our special derivation of EventSource gives us the hooks into concurrency
trait EventSourceSpy extends EventSource {

    def sendEvent[T](event: T): Unit = EventSourceSpy.latch.countDown()

    // We don't care about processing the messages that EventSource usually
    // processes so we simply don't worry about them.
    def eventSourceReceive = { case "" => }

}

class AltimeterSpec extends TestKit(ActorSystem("AltimeterSpec"))

    with ImplicitSender
    with WordSpec
    with MustMatchers
    with BeforeAndAfterAll {

    import Altimeter._

    override def afterAll() { system.shutdown() }

    // The slicedAltimeter constructs our Altimeter with the EventSourceSpy
    def slicedAltimeter = new Altimeter with EventSourceSpy

    // This is a helper method that will give us an ActorRef and our plain
    // ol' Altimeter that we can work with directly.
    def actor() = {
        val a = TestActorRef[Altimeter](Props(slicedAltimeter))
```

```
(a, a.underlyingActor)
}

"Altimeter" should {
  "record rate of climb changes" in {
    val (_, real) = actor()
    real.receive(RateChange(1f))
    real.rateOfClimb must be (real.maxRateOfClimb)
  }
  "keep rate of climb changes within bounds" in {
    val (_, real) = actor()
    real.receive(RateChange(2f))
    real.rateOfClimb must be (real.maxRateOfClimb)
  }
  "calculate altitude changes" in {
    val ref = system.actorOf(Props[Altimeter]())
    ref ! EventSource.RegisterListener(testActor)
    ref ! RateChange(1f)
    fishForMessage() {
      case AltitudeUpdate(altitude) if (altitude) == 0f => false
      case AltitudeUpdate(altitude) => true
    }
  }
  "send events" in {
    val (ref, _) = actor()
    EventSourceSpy.latch.await(1, TimeUnit.SECONDS) must be (true)
  }
}
}
```

Fishing for Results

The "calculate altitude changes" test uses a cool method for succeeding a test as long as a certain message shows up before the timeout, even when other messages might show up first.

```
"calculate altitude changes" in {
  val ref = system.actorOf(Props[Altimeter]())
```

```
ref ! EventSource.RegisterListener(testActor)
ref ! RateChange(1f)
fishForMessage() {
    case AltitudeUpdate(altitude) if (altitude) == 0f => false
    case AltitudeUpdate(altitude) => true
}
```

We have two `!` asynchronous messages that go to the Altimeter and we're guaranteed that they'll go in that natural order so we know that `RegisterListener` is processed before `RateChange`, but that's all we're guaranteed. Do you remember what's happening internally in the Altimeter? There's a scheduler that's sending a `Tick` message to it at regular intervals, right? As a result of that, either of these message sequences in the Actor's mailbox is possible:

```
RegisterListener(testActor)
RateChange(1f)
Tick
```

... or ...

```
Tick
RegisterListener(testActor)
Tick
Tick
RateChange(1f)
Tick
```

The first example is easy. Were that to happen, then asserting that the altitude was calculated (i.e., it's no longer zero) would be a piece of cake, but what if the second example occurs? It will look like the altitude hasn't changed! But that's not our Actor's fault; it hasn't had the chance to try yet.

This is where the argument to `fishForMessage()` comes in. The `fishForMessage()` call will run the passed in partial function repeatedly as long as it returns `false` up until the (default) timeout. If it never returns `true`, then the test ultimately fails, but if it ever returns `true`, then `fishForMessage()` succeeds and the test succeeds. This allows us to handle the second example with grace and dignity. It also ensures that, assuming our test succeeds, that it succeeds as quickly as it possibly can.

Countdown to Success

Another common mechanism for locking down concurrency in our tests uses something that Akka doesn't provide at all: the standard `java.util.concurrent.CountDownLatch` class. Using it means we generally need to be able to hook into the code somehow. Either we're injecting some client code that can bridge to our test via the standard API (e.g., registering a standard event handler), or we derive from the system under test and hook a latch in there, or some other variant. Here, we take advantage of the refactoring we did earlier that lets us slide in a new `EventSource`.

Since the `EventSource` is already tested, we don't need to care that it sends events; we *do* need to care whether or not our `Altimeter` publishes events to the `EventSource`. So, we first create the `EventSourceSpy` and give it a `CountDownLatch` we can hook to and then use that latch to solidify our test.

Let's first look at the content of the `EventSourceSpy`.

```
trait EventSourceSpy extends EventSource {  
    def sendEvent[T](event: T): Unit = EventSourceSpy.latch.countDown()  
    // We don't care about processing the messages that EventSource usually  
    // processes so we simply don't worry about them.  
    def eventSourceReceive = { case "" => }  
}
```

In the `sendEvent` method we trigger the latch. The test code will sit on the latch waiting for `sendEvent()` to open it. When the `Altimeter`'s Tick happens, it will call `sendEvent()` and the latch will open, which will let our test succeed. It's not a really in-depth test, but it does assert that the `Altimeter` does call the `sendEvent()` function, and since we know that the `EventSource` works, we know that the act of calling `sendEvent()` should do what we want in the end.

The next important bit to understand doesn't have much to do with testing. Remember the requirements that were put in place to use the `EventSource`? Whoever mixed it in would have to compose the `receive` method by mixing the `eventSourceReceive` with its own behaviour, and we did this in [Section 5.5](#). Were we not to define `eventSourceReceive` here, we would get a compile error because the `Altimeter` requires it in order to compose `receive`.

That's all well and good, but what body should we give our mocked `eventSourceReceive`? This brings us to an aside into receive composition. That actually depends on:

- If `receive = altimeterReceive` or `Else eventSourceReceive`, then it doesn't really matter what we do since we don't plan on sending anything but Altimeter messages to our Altimeter.
- If `receive = eventSourceReceive` or `Else altimeterReceive`, then we need to make sure that whatever messages we send to the Altimeter won't be matched by `eventSourceReceive`.

This is why we defined `eventSourceReceive` to be `case "" =>`. We know we won't be sending an empty string to either the Altimeter or the EventSource, so we can be sure that it will never matter. Had we defined it as `case _ =>`, then it would have matched anything and the Altimeter would never see a single message.

And Back to Our Countdown...

Now let's take another look at "send events" so we can close this up:

```
"send events" in {  
    val (ref, _) = actor()  
    EventSourceSpy.latch.await(1, TimeUnit.SECONDS) must be (true)  
}
```

We don't need to register anything as a listener because our EventSource isn't a real event source anyway; it's already hooked up and ready to trip our latch. All we need to do is wait. It doesn't matter what the altitude is; we only care that the altitude is sent as an event, so we'll await on the latch. If the latch is tripped before we await, then it will return true; otherwise, we'll block the current thread until it opens or there is a timeout. If the timeout trips, then the return value is false and our test fails. Otherwise, it returns true and we have a successful result that finished quickly.

Now, we could have written more elaborate code that tests for certain altitudes and proper calculation, but that's not what we're focusing on in these tests; we're interested in proper messaging and communication between our players. You might want to spend time writing a few more tests in order to flex your new Akka testing muscles.

6.7 Akka's Other Testing Facilities

We've already seen the major aspects of Akka's testing facilities:

- Back-door access to the Fortress
- Injection into the message flow via `testActor` and `ImplicitSender`

The second one is where we get to use the fancy stuff, like `expectMsg()` and `fishForMessage()`. As stated earlier, there are many more functions that fall into this category of assert functions and while you can read about those in the Akka ScalaDoc, let's list a few of the more practical ones:

- `expectMsgPF()`: This highly useful variant of `expectMsg()` allows you to attach a partial function to the assertion. Effectively, this lets us write a simple receive block for incoming messages. You can then deconstruct the message and evaluate its constituent parts much like we did with the `fishForMessage()` block in the `AltimeterSpec`.
- Other `expectMsg*`() functions: We won't exhaustively describe all of the other functions that look like `expectMsg*()`, but there are quite a number of them. You can expect messages of a particular class or type. You can assert that all of a given list of messages are received, that any of a given set of messages is received, that a certain number are received, that no messages are received, and more. See the `TestKit` ScalaDoc for more information about these assertions.
- `ignoreMsg()`: If there are several messages that the Actor sends under test and they don't concern the body of the test (in fact, they are really quite a pain to deal with), then you can use this function to ignore them. Your tests will see any messages that match the partial function. Note that this hook lives for the `TestKit`'s lifetime, so ignored messages will remain ignored until you call `ignoreNoMsg()`, which will clear out the partial function.
- `receiveWhile()`: The `fishForMessage()` function uses `receiveWhile()` in order to do its business. With `receiveWhile()`, you can test for messages and then exit the assertion when you've found what you are (or aren't) looking for. For example:

```
val altitudes = receiveWhile() {
    case AltitudeUpdate(alt) if alt > 0f => alt
}
altitudes.sum must be < (altitudes.size * maxCeiling)
```

This would keep receiving `AltitudeUpdates` as long as they're greater than 0, and then make a weak check to ensure that the altitude messages (generally) stayed below the maximum ceiling of the plane. If the plane doesn't stop climbing in the time that `receiveWhile` is running, then we consider that a failure as well.

- And don't forget the facilities that `ScalaTest` already provides. We used the `evaluating ... must produce` functionality earlier and there are many other facilities that come in handy with Akka testing as much as they do with non-Akka testing.

Other functions exist, but we won't be covering them unless we need them. I highly encourage you to rummage through the `TestKit` ScalaDoc and learn all that you want about the facilities it provides.

6.8 About Test Probes and the `testActor`

Concurrency is a problem. With respect to Actors, and tests, it can manifest between tests. For example, let's say you have the following:

```
// An annoying Actor that just keeps screaming at us
class AnnoyingActor(snooper: ActorRef) extends Actor {
    override def preStart() {
        self ! 'send
    }

    def receive = {
        case 'send =>
            snooper ! "Hello!!!"
            self ! 'send
    }
}

// A nice Actor that just says Hi once
```

```
class NiceActor(snooper: ActorRef) extends Actor {  
    override def preStart() {  
        snooper ! "Hi"  
    }  
  
    def receive = {  
        case _ =>  
    }  
}  
  
...  
  
"The AnnoyingActor" should {  
    "say Hello!!!" in {  
        val a = system.actorOf(Props(new AnnoyingActor(testActor)))  
        expectMsg("Hello!!!")  
        system.stop(a)  
    }  
}  
  
"The NiceActor" should {  
    "say Hi" in {  
        val a = system.actorOf(Props(new NiceActor(testActor)))  
        expectMsg("Hi")  
        system.stop(a)  
    }  
}
```

There's a problem here. That `AnnoyingActor` eventually stops but it doesn't necessarily stop before the call to `expectMsg("Hi")` in the second test. This can cause the second test to fail because, while it's expecting "Hi" it may get "Hello!!!". This is simply because the `testActor` is the sole entity that is receiving messages from tests. Most of the time this isn't a problem because you don't have annoying Actors that you're testing, but once in a while it can be a problem.

You can solve the problem by using some test isolation, as described in [Section 6.5](#) but there's another alternative.

Test Probes

Test probes help solve this problem, as well as give you an infinite number of unique test Actors that you can use in your test code. A TestProbe has most of the functionality you need from the TestKit wrapped up in one object. Using a TestProbe we could solve the earlier problem like this:

```
import akka.testkit.TestProbe

"The AnnoyingActor" should {
    "say Hello!!!" in {
        val p = TestProbe()
        val a = system.actorOf(Props(new AnnoyingActor(p.ref)))
        // We're expecting the message on the unique TestProbe,
        // not the general testActor that the TestKit provides
        p.expectMsg("Hello!!!")
        system.stop(a)
    }
}
```

Of course, that's just a single use of a TestProbe. You can use them for all sorts of interesting work whenever your code under test requires a number of different Actors in its operation. By using the tell syntax, you can pass references to TestProbes whenever you like.

```
someActor.tell("Message", probe.ref)
```

It's just that easy. Remember that you're not limited to the Implicit-Sender and single testActor when you're writing your tests, and try to employ TestProbes whenever it simplifies your problem.

6.9 Chapter Summary

We've just put a very important topic under our belts. Testing closes the loop on programming, ensuring that what we've written not only works but is also *usable*. It also gives us a sense of comfort, knowing that we have the ability to tie up any loose ends in our code that might have been nagging us, keeping any cognitive debt to a minimum. As with any other code, you should have your Akka programs tested thoroughly as you go.

Let's summarize the skills we've acquired in this chapter:

- Deterministic unit testing of Actors using the `TestActorRef` instantiation
- Fast detection of success using facilities of `TestKit`, such as `expectMsg()` and `fishForMessage()`
- Techniques for locking down non-determinism using facilities provided by Akka (mentioned above) and those not provided by Akka (e.g., `CountDownLatch`)
- Coding for testability by factoring our code into pieces using standard Scala techniques
- A powerful abstraction for isolating our tests from one another and from the singular `ActorSystem` that would have otherwise been contained within the entire test specification

Akka provides many more testing goodies for when your code becomes more complex. Simply put, Akka's testing framework scales to meet your needs. As you code, you'll need to consider how you'll verify that code, and the techniques you now understand will take you a long way before you have to start looking at the richer aspects of Akka testing.

Chapter 7

Systems, Contexts, Paths, and Locations

One of the things that differentiates Actor programming from other types of programming is *structure*. Of course, all programs must be structured in some way, but the Actor’s “live” nature and the fact that it can spawn new Actors on demand make its type of structuring rather unique. Not only are the Actors “physically” built into a structure, the interaction between them through messaging also provides a certain amount of structure to your application, which you can leverage. In this chapter, we’ll go deeper into the Actor paradigm, building out more structure as we describe the facilities in Akka that make working within this structure natural and powerful.

7.1 The ActorSystem

We’ve seen the ActorSystem several times now; it plays a very important and strong role in the Actor implementation as well as forms the root of the Actor structure.

Figure 7.1 shows us the key parts of the ActorSystem that interest us for the moment:

Dead Letter Office We’ve seen this already, but now it’s clear that the Dead Letter Office is an Actor, structurally no different from any other Actor we’ve seen. Any time a message is destined for an Actor that either doesn’t exist or is not running, it goes to the Dead Letter Office.

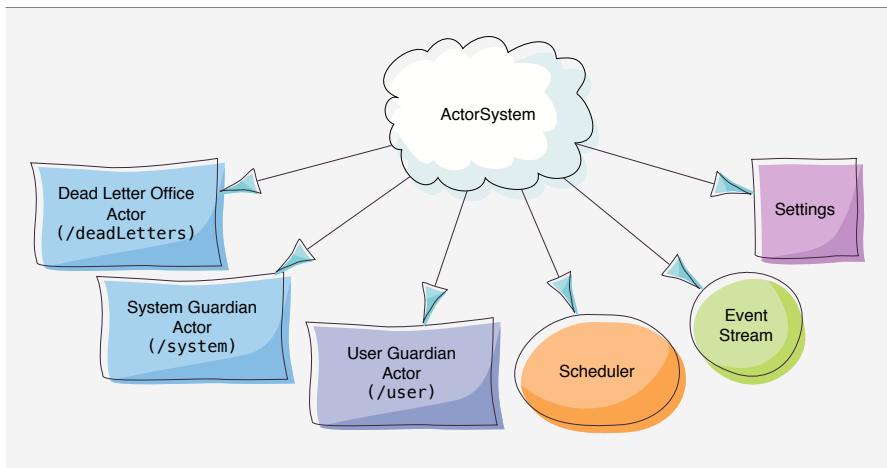


Figure 7.1 · Here, we see the key players in the ActorSystem that interest us most of the time, including three Actors (Dead Letter, System, and User) as well as three other key entities (Scheduler, Event Stream, and Settings). These ActorSystem elements, with the exception of the System Guardian, regularly pop up in Actor programming.

User Guardian Actor We know that no Actor we create can exist without a parent - *something* has to own it. The User Guardian Actor is the parent of all Actors we create from the ActorSystem.

System Guardian Actor For internal Actors that Akka creates in order to assist you, there is the System Guardian Actor, which serves the same purpose as the User Guardian Actor, but for “system” Actors.

Scheduler We’ve met the Scheduler before and, while you could always instantiate one for yourself, the default one lives as an ActorSystem child.

Event Stream We’ve never seen the Event Stream in its bare form before, but we use it every time we write a log message. The Event Stream has more uses than just logging and we’ll look at it very soon.

Settings Akka uses a new configuration system that is useful for configuring Akka and your application. You can access it from the ActorSystem.

The part of the ActorSystem with which we've interacted most has been the User Guardian. The Guardian watches over its children (i.e., the Actors we create from the ActorSystem) and takes care of them. This core responsibility of the Guardian is pervasive across the entire Actor paradigm in Akka and it will be the subject of a future chapter. The reason I mention it here is because the restriction that Akka places on your Actor's structure – that of an imposed hierarchy—is significant. I want to assure you that, while the imposition may be significant, you are definitely getting something for your money—a key thing being *resiliency*. But before we cover that important topic, we first need to lay more of a foundation on working with the Actor paradigm.

Attaching to the User Guardian

When we create the Plane, we're creating it as the User Guardian's child. As you might recall, we do this:

```
val system = ActorSystem("PlaneSimulation")
```

When we create the ActorSystem's Plane, it becomes the User Guardian's child. The User Guardian is now watching over our Plane to ensure that nothing bad happens to it (again, I'll go into detail about what that really means in a subsequent chapter). This parent-child relationship is reflected in the Plane's *path*. It looks like this:

```
/user/Plane
```

The path indicates the names of the Actors we would need to traverse were we to walk from one Actor to the next. All we would have to do here is start from the User Guardian and step onto the Plane, but obviously, as we increase the size of our ActorSystem's hierarchy we will have farther to walk.

7.2 Actor Paths

The path is not actually as simple as a Unix-style path, but is really a standard URL. The Akka team chose this format due to their Actor model's enormous

flexibility, for ease of definition, and for future growth of the toolkit.¹ Given that the path is a URL, we can expand the Plane path to be more correct:

```
akka://PlaneSimulation/user/Plane
```

Since this is a URL, we can also include more locating information within it. Of course, since our Plane currently lives on only one machine and we're only dealing with a single ActorSystem, the location information won't be of any use to us, but this extra information will be a good thing when we start working with remote machines.

```
akka://PlaneSimulation@{host}:{port}/user/Plane
```

Here, we can specify the ActorSystem, followed by a host and port and then the path to the Actor in which we're interested. Again, we won't be using this right now, but it will become important later.

That's all I'll say about paths for the moment. In the rest of this chapter, we'll create Actor hierarchies and rely on the path to help us find things within the structure, so we're not done yet...just done for now.

Accessing the Actor Path

The Actor's path is a property of the ActorRef for that Actor. This means that you can get the path for an Actor even though you can't reference that Actor directly. We'll see more about how the path relates to various other aspects of the Actor's properties later. If you want to see the path of a given ActorRef, you can access it like this:

```
val a = system.actorOf(Props(new Actor {
    def receive = {
        case _ =>
    }
}), "AnonymousActor")
// prints "/user/AnonymousActor"
println(a.path)
```

¹Akka is currently working on the Clustering feature, which will require you to use the URL to specify Actors in a Cluster. The URL format will remain the same, but the content will be different in order to accommodate the new feature.

```
// prints "AnonymousActor"
println(a.path.name)
```

7.3 Staffing the Plane

Without a crew, our Plane doesn't have much purpose. We'll rectify that by adding some people to our Plane who know how to run things. There'll be a Pilot, CoPilot, and a set of flight attendants with a leader.

The Flight Attendants

Let's start by creating some flight attendants in the form of a hierarchy, with a lead flight attendant and a collection of subordinate flight attendants. All of the people on our Plane will have names, including the flight attendants. To add a twist, we'll supply everyone's name using the configuration system that ships with Akka.

To play nice with SBT, we'll put our configuration file inside of `src/main/resources` and it will be called `reference.conf`. The content we need for the moment will include pilots and flight attendants:

```
zzz {
  akka {
    avionics {
      flightcrew {
        pilotName = "Harry"
        copilotName = "Joan"
        leadAttendantName = "Gizelle"
        attendantNames = [
          "Sally",
          "Jimmy",
          "Mary",
          "Wilhelm",
          "Joseph",
          "Danielle",
          "Marcia",
          "Stewart",
          "Martin",
```

```
        "Michelle",
        "Jaime"
    ]
}
}
}
}
```

Akka uses the HOCON² configuration specification, which is a pragmatic and simple-to-use configuration system; you're gonna love it.

An Aside into Configuration

The configuration system may be unlike anything you've used before, since it departs from some traditional wisdom when it comes to defaults. Configurations systems often allow you to specify defaults in your code, like this:

```
val myvalue = configuration.get("some.configuration.variable",
                                "defaultValue")
```

HOCON does not support this. Relax, it's a good thing. Default values don't belong in code, especially the default values for "big" software. If you have a solid configuration system, then it will be used for tuning your software.³ Nine times out of nine point zero zero one, the software authors have no idea what values are appropriate for defaults when it comes to tuning and performance (among many other things). What tends to happen is a developer ends up doing something like:

```
val myvalue = configuration.get("some.configuration.variable", 7)
// Sure, 7 seems good... I looked at my watch and it had a '7' on
// it, so I used it... it'll be just fine.
```

Later, once people actually understand the nature of the written program, the default values start to take shape. More often than not, the people who start to really understand what the default values are supposed to be are not the programmers who are writing the code; they're testers, performance engineers, integrators, convenience store workers, and literally anyone else.

²<https://github.com/typesafehub/config/blob/master/HOCON.md>

³Right? *RIGHT???*

So, if the default values are in the code, then one of two things ends up happening:

1. Bug reports get filed against the code, programmers get their backs up about it and insult the person who filed it, and eventually their managers make them change it.
2. The bug reports don't get filed, and the people who know what it should be just provide a configuration override for nearly every configuration parameter in the code.

Number 1 makes work where it need not be. Number 2 ensures that the default values are irrelevant, and the programmers have no idea how the product is really being used. By not supplying the functionality to put default values in the code, HOCON eliminates the issue altogether. It doesn't take the power of the default away from the developer (she can still supply it in the reference configuration file), it just puts the power of the default value in the place where it has the most power, *outside of the code*.

The last argument I hear from people on this philosophy is, "But if the value isn't in the configuration file and I try to dereference it, my code will crash!" There's only one answer to that: "Yup."

See, fantastic no?

Back to Our Flight Crew

The configuration file now contains several names for the employees in our flight crew, which gives us the data we need to start creating some flight attendants. In the spirit of starting simple, we'll have some pretty boring flight attendants; they'll just respond to drink requests at some point in the future that we can govern using an Akka Duration specifier.

The Flight Attendant

```
package zzz.akka.avionics

import akka.actor.Actor
import scala.concurrent.util.duration._
import scala.concurrent.ExecutionContext.Implicits.global

// This trait allows us to create different Flight Attendants with different
// levels of responsiveness.
```

```
trait AttendantResponsiveness {  
    val maxResponseTimeMS: Int  
    def responseDuration = scala.util.Random.nextInt(maxResponseTimeMS).millis  
}  
  
object FlightAttendant {  
    case class GetDrink(drinkname: String)  
    case class Drink(drinkname: String)  
    // By default we will make attendants that respond within 5 minutes  
    def apply() = new FlightAttendant with AttendantResponsiveness {  
        val maxResponseTimeMS = 300000  
    }  
}  
  
class FlightAttendant extends Actor { this: AttendantResponsiveness =>  
    import FlightAttendant._  
    def receive = {  
        case GetDrink(drinkname) =>  
            // We don't respond right away, but use the scheduler to ensure  
            // we do eventually  
            context.system.scheduler.scheduleOnce(responseDuration, sender,  
                Drink(drinkname))  
    }  
}
```

When we were looking at Actors before, we saw that *closing over the sender* is not something we want to do, simply because it's a method that returns the current value. If we close over it, then the value returned when the closure executes will probably not be what it was when we made the closure. Just to be clear, that's not what's happening when we schedule the Drink response. The second parameter of `scheduleOnce()` takes a proper `ActorRef`, not a *by-name* parameter, so the sender is frozen at the point of scheduling.

By now, the contents of the `FlightAttendant` should be old news to you. The interesting part is when we create the boss, because when hasn't middle management been incredibly interesting?

Testing Our Flight Attendant

Did you happen to think about testing this beast at all? Sure you did! Every time you look at our code, you’re thinking about how it gets tested—I know you are.⁴

There’s a problem with testing our `FlightAttendant` at this point, or at least there would have been if we hadn’t planned for it; the response time between a drink request and a drink response could be up to *5 minutes*. One of the reasons we created the `AttendantResponsiveness` trait was in order to make the `FlightAttendant` easier to test. It’s really easy to create a derivation of this trait so that we can define the `maxResponseTimeMS` to an incredibly low value, say 1. Let’s do it:

```
package zzz.akka.avionics

import akka.actor.{Props, ActorSystem}
import akka.testkit.{TestKit, TestActorRef, ImplicitSender}
import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers

object TestFlightAttendant {
    def apply() = new FlightAttendant with AttendantResponsiveness {
        val maxResponseTimeMS = 1
    }
}

class FlightAttendantSpec extends TestKit(ActorSystem("FlightAttendantSpec"))
    with ImplicitSender
    with WordSpec
    with MustMatchers {

    import FlightAttendant._

    "FlightAttendant" should {
        "get a drink when asked" in {
            val a = TestActorRef(Props[TestFlightAttendant]())
            a ! GetDrink("Soda")
            expectMsg(Drink("Soda"))
        }
    }
}
```

⁴What do you mean you’re not? We had a *deal*.

We've been able to slide in our "fast" implementation of the `AttendantResponsiveness` trait using a standard factory method. But there's a problem: if you tried that out, you'd find that it's not all that fast. It's definitely not a one millisecond test—far from it. In fact, it will run in 100 milliseconds, three orders of magnitude slower than we'd like. Go ahead, give it a try and see. To really get a decent illustration of this, you might want to create 10 or 20 variations of the "get a drink when asked" tests in order to expand the delays.

So why does it run so slow? To answer this question, we need to learn more about the Scheduler's implementation choices.

The Scheduler and the HashedWheelTimer

The authors of Akka are really concerned about performance and scalability, which is great because it means we're allowed to focus on better things. Worrying about the Scheduler is just standard operating procedure for these guys. Everything in software is a trade-off and when it comes to timers, the Akka team have opted to use a `HashedWheelTimer` in their Scheduler implementation. It trades off timer resolution and microsecond accuracy for low overhead. You can read up on the details of the `HashedWheelTimer` at <http://www.jboss.org/netty/>. The bottom line for us, from a practical perspective, is that the Scheduler has a default resolution of only 100 milliseconds. This is why our test runs slow.

Now, before you freak out and say you *must have* millisecond (i.e., one millisecond) resolution, just breathe. You have to ask yourself why you need it. If your app will be busy doing a billion things per second, then you'll want a low overhead timer, which is why you should leave its default behaviour just the way it is. Read up on the `HashedWheelTimer` very carefully before you commit to trading off on something else.

So, are we screwed? Of course not. We can modify the configuration for our test in order to ensure that we get a successful result incredibly fast.

We just have to change the `ActorSystem`'s construction within the test spec:

```
import com.typesafe.config.ConfigFactory  
  
class FlightAttendantSpec extends TestKit(ActorSystem("FlightAttendantSpec",  
    ConfigFactory.parseString("akka.scheduler.tick-duration = 1ms"))
```

And that's it. We've used a different flavor of `ActorSystem` creation to pass in a piece of configuration that we can mix into the overall configuration.

We're overriding the default value of the tick duration for the Scheduler so that our test runs fast. There's no need to care about the efficiency problem with respect to what we're trading off because we're just running a simple test.

Now we can run 100 tests in the same time it used to take to run one. This keeps our development cycle efficient and that's a very good thing.

The Lead Flight Attendant

The LeadFlightAttendant won't do much more than simple management of the flight attendants under its care. The subordinates themselves are the ones who will do all the real work on this Plane. (This is not a comment on middle-management in general, of course.)

```
package zzz.akka.avionics

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

// The Lead is going to construct its own subordinates.
// We'll have a policy to vary that
trait AttendantCreationPolicy {
    val numberOfWorkers: Int = 8
    def createAttendant: Actor = FlightAttendant()
}

// We'll also provide a mechanism for altering how we create the
// LeadFlightAttendant
trait LeadFlightAttendantProvider {
    def newFlightAttendant: Actor = LeadFlightAttendant()
}

object LeadFlightAttendant {
    case object GetFlightAttendant
    case class Attendant(a: ActorRef)
    def apply() = new LeadFlightAttendant with AttendantCreationPolicy
}

class LeadFlightAttendant extends Actor { this: AttendantCreationPolicy =>
    import LeadFlightAttendant._

    // After we've successfully spooled up the LeadFlightAttendant, we're
    // going to have it create all of its subordinates
    override def preStart() {
```

```
import scala.collection.JavaConverters._  
val attendantNames = context.system.settings.config.getStringList(  
    "zzz.akka.avionics.flightcrew.attendantNames").asScala  
attendantNames take numberOfAttendants foreach { i =>  
    // We create the actors within our context such that they are  
    // children of this Actor  
    context.actorOf(Props(createAttendant), i)  
}  
}  
// 'children' is an Iterable. This method returns a random one  
def randomAttendant(): ActorRef = {  
    context.children.take(  
        scala.util.Random.nextInt(numberOfAttendants) + 1).last  
}  
def receive = {  
    case GetFlightAttendant =>  
        sender ! Attendant(randomAttendant())  
    case m =>  
        randomAttendant() forward m  
}  
}
```

Figure 7.2 shows the layout of our hierarchy. You can verify the existence of this hierarchy by looking at Akka itself. Let's write a short program that will show what Akka has constructed for us by looking at the debug output of the Actor life cycle.

```
object FlightAttendantPathChecker {  
    def main(args: Array[String]) {  
        val system = akka.actor.ActorSystem("PlaneSimulation")  
        val lead = system.actorOf(Props(  
            new LeadFlightAttendant with AttendantCreationPolicy),  
            "LeadFlightAttendant")  
        Thread.sleep(2000)  
        system.shutdown()  
    }  
}
```

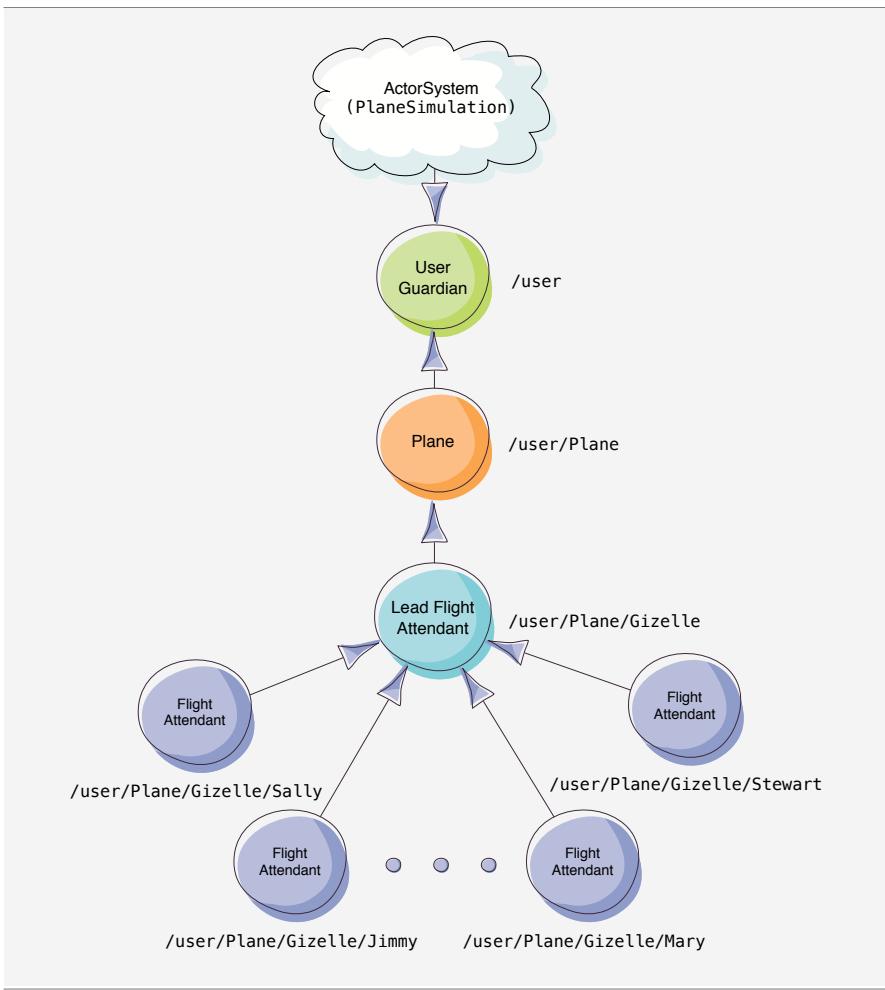


Figure 7.2 · The Plane’s hierarchy, including the sub-hierarchy of the Flight Attendants. Note that the path requirements are determined entirely by the parent/child relationships between the Actors.

If you put that code into the same source file as the one containing the `LeadFlightAttendant`, then you can run it from SBT using `sbt run`. However, before we do that, we’ll need to turn on some special debugging in Akka, which will let us see the life-cycle output. If you modify the `application.conf` file that we’re using to configure Akka by adding

the following, then we'll be able to see what we're looking for:

```
akka {  
    loglevel = DEBUG  
    actor {  
        debug {  
            lifecycle = on  
        }  
    }  
}
```

So when we run it, we get:

```
[akka://PlaneSimulation/user/LeadFlightAttendant] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Jimmy] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Sally] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Wilhelm] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Joseph] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Mary] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Marcia] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Stewart] started  
[akka://PlaneSimulation/user/LeadFlightAttendant/Danielle] started
```

When we turn on life-cycle debugging, Akka will log a lot of information about what's happening to our Actors' lives. It's not the kind of thing you'd want to turn on regularly, but keep it in mind if you think you might be doing something like killing off an Actor prematurely.

7.4 The ActorContext

Every Actor has a context member that helps it do a lot of its work. The context is one of the things that decouples your internal Actor logic from the rest of Akka that's managing it. One thing it does is protect Akka from your code, and the other packages that you might be using. When they go bad on you and your Actor turns into a pile of scrambled bits, Akka is isolated from it and can keep things running.

We've seen the ActorContext already, when we've been accessing its ActorSystem and when we've been using its actorOf() method(s), but there's

a lot more to it. Let's look at what it brings us, including some of the functionality we've already seen:

Actor Creation Just like the `actorOf()` methods that exist on the `ActorSystem`, we have the ability to create Actors from the context as well. The difference here, of course, is that these newly created Actors are children of the current Actor instead the User Guardian.

System Access The `ActorSystem` at the root of this Actor's hierarchy is accessible as well, which lets us access goodies like the `Scheduler` and the `Settings` that we've seen previously.

Relationship Access The context knows who our parent is, who all of our children are, and gives us the ability to find other Actors in the `ActorSystem`, including grandparents, brothers, sisters, cousins, uncles to 5th degree, and so on.

State When accessing `self` or `sender` from the Actor, we're actually getting that information from the `ActorContext`; the Actor merely gives us convenient access to them. There are other types of state as well, which are carried in the context that holds information vital to our Actor. We'll see these in future chapters, but they include things such as the Actor's current behavioural state, its `Dispatcher`, and a list of other Actors whose life cycles the Actor is interested in.

Useful Functionality The `ActorContext` also provides some useful functionality that we'll be invoking a fair bit as Actor programmers. We can stop Actors (including ourselves), change our internal behaviour (i.e., change the implementation of the `receive` method), and insert ourselves into the event flow of other Actors' life cycles.

The `ActorContext` is aptly named. One of the major things that differentiates one Actor from another is the *context* in which it lives. The `ActorContext` provides that context and keeps it fresh, so that our Actor has most things it needs to do its job.

Let's start putting more hierarchy into our Plane and see how we can get some relationships going.

Pilots

It's probably about time that we put some brains behind the controls of our Plane. We have some names for them in our `reference.conf` configuration file, so it makes sense to turn them into real live objects.

The relationships we'll put together for our pilots will be a bit different than we had for our flight attendants. The flight attendants are direct children of the lead that created them, but the pilots will discover each other when we've reached a stable point after creation.

To cement the pilot relationships, we'll need to look them up using the Actor paths that Akka creates for us. We can find Actors using the `ActorContext` or the `ActorSystem`, but in the case of the pilots, we'll use the `ActorContext`.

Looking Up Actors

The `ActorContext` provides us with the `actorFor()` set of functions; there are three of them:

`actorFor(path: Iterable[String]): ActorRef` Allows us to use an iterable collection to look up our Actors. For example, `List("/user", "/Plane", "/Cont`

`actorFor(path: String): ActorRef` Allows us to look up Actors using a familiar string. For example, `"/user/Plane/ControlSurfaces"`.

`actorFor(path: ActorPath): ActorRef` The `ActorPath` is a nice abstraction of the path concept, which provides some behaviour for comparison, extracting parts of the path, etc. Semantically, it's not really any different from the other two.

All of those versions of the function are essentially the same. What's interesting to note is the *return type*. If we ask the following...

```
val doesNotExist = context.actorFor("/user/this/actor/does/not/exist")
```

...then we'll get an `ActorRef` back; it won't throw an exception. Why won't it throw an exception? And if it won't throw an exception, then what on earth will it return to us?

The Reality of Concurrency

Let's tackle the first question by asking a question. If we ask for this...

```
val exists = context.actorFor("/user/this/actor/really/does/exist")
```

...then what will we get back? That's a silly question. Of course, we'll get back a valid ActorRef for the existing Actor, right?

It turns out that it's not a silly question. When we look up that Actor, we're doing it in a concurrent system, and *concurrency changes the rules*. Concurrency presents us with a near infinite set of timelines for our application, and in many of them the response from `actorFor()` can easily return us a non-existent ActorRef for something that we *think* should be there.

Due to the real-life aspects of concurrency, Akka has made a conscious decision to return an ActorRef for all `actorFor()` requests, regardless of whether or not it can find the Actor instance for the request; it will never throw an exception.

And now we can answer the second question: The returned ActorRef, when its corresponding Actor cannot be found, is the Dead Letter Office. We can now add on to the above code with a deterministic assertion:

```
val doesNotExist = context.actorFor("/user/this/actor/does/not/exist")
if (doesNotExist == context.system.deadLetters) {
    println("Yup, it really doesn't exist")
} else {
    println("Whoah. Someone created something that I thought didn't" +
        "exist while I was checking to see if it didn't exist")
}
```

Optional ActorRefs

One other thing you could ask is, “Why not return `Option[ActorRef]`?” Well, the answer to this question is exactly the same as the answer to why an exception doesn't get thrown: *concurrency*. To the point, the following piece of code is entirely possible:

```
// Hypothetical, assuming that actorFor() returns an Option[ActorRef]
val exists: Option[ActorRef] =
```

```
context.actorFor("/user/this/actor/really/does/exist")
if (exists.isEmpty)
  println("Whoah. The thing I thought would exist, no longer does.")
```

The thing that you thought would exist might have died before `actorFor()` could look it up; either it died of its own accord, someone killed it, or it never existed in the first place. You can't know which one.

Terminated ActorRefs

Another thing to understand is that, just because you have an `ActorRef` and that `ActorRef` may actually point at the Actor you think it should, it still may not be of any use to you because it's been terminated. For example:

```
val exists = context.actorFor("/user/this/actor/really/does/exist")
if (exists.path.name == "exist" && exists.isTerminated)
  println("Damn. The 'exist' Actor died before I could talk to it.")
```

actorFor() and Concurrency Conclusions

All of this discussion around `actorFor()` should have driven home the idea that there are aspects of concurrency that you simply can't avoid. *Stuff happens* outside of the current context in which you're thinking and that's real life—you simply just have to deal with it.

However, it shouldn't concern you that these things can happen; all of this discussion has been meant to familiarize you with the API. Akka has several facilities to make sure that, while these things might occur, they would occur because *you want them to*. The Actor model, coupled with the model of resiliency, ensures that your application is actually easy to reason about, even in the face of concurrency that dictates your returned Actors may not be running, or even exist.

The Pilot

Given that we now understand the issues surrounding the `actorFor()` we can start using it a bit.

```
package zzz.akka.avionics
```

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

object Pilots {
    case object ReadyToGo
    case object RelinquishControl
}

class Pilot extends Actor {
    import Pilots.~
    import Plane.~

    var controls: ActorRef = context.system.deadLetters
    var copilot: ActorRef = context.system.deadLetters
    var autopilot: ActorRef = context.system.deadLetters
    val copilotName = context.system.settings.config.getString(
        "zzz.akka.avionics.flightcrew.copilotName")
}

def receive = {
    case ReadyToGo =>
        context.parent ! Plane.GiveMeControl
        copilot = context.actorFor("../" + copilotName)
        autopilot = context.actorFor("../AutoPilot")
    case Controls(controlSurfaces) =>
        controls = controlSurfaces
}
}
```

The Pilot needs to be able to work with other Actors quite intimately. It takes advantage of the fact that it knows it is the child of the Plane and can thus ask the Plane directly for the controls (the `context.parent` call). We also see that it can ask the Actor's context for its siblings exactly as we would look for sibling directories in our operating system's file system (the `context.actorFor(...)` calls).

The CoPilot

The CoPilot is very similar to the Pilot, except that he has no interest in grabbing the Plane's controls when it's ready to go.

```
class CoPilot extends Actor {
    import Pilots.~
    var controls: ActorRef = context.system.deadLetters
```

```
var pilot: ActorRef = context.system.deadLetters
var autopilot: ActorRef = context.system.deadLetters
val pilotName = context.system.settings.config.getString(
    "zzz.akka.avionics.flightcrew.pilotName")
def receive = {
    case ReadyToGo =>
        pilot = context.actorFor("../" + pilotName)
        autopilot = context.actorFor("../AutoPilot")
    }
}
```

There's really not much new here. We have our members on lines 3-5 that, much like the Pilot, are set as Options. The only real difference is that the controls will remain None much longer than they will for the Pilot. We're still using the configuration and the Actor hierarchy to know where our respective pieces are, and the relationships are set up once the Plane tells us that we're ReadyToGo.

Creating Pilots

As we've already discovered, Actor programs depend highly on their structure and, as such, that structure can start to become rather complex. By and large, this isn't really a problem—in fact, it's a very good thing—but testing becomes more interesting. Aside from checking correctness, our tests should be small, targeted, and fast. Small is the challenge when it comes to complex Actor structures, because it means we have to start minimizing that structure as much we can, as well as design our code such that it doesn't care too much about the full structure, if possible.

One thing we'll see a fair bit of is the delegation of *creation* to mixable traits; pilots are no exception here. We've already seen it with the LeadFlightAttendant, so let's continue this pattern by defining the PilotProvider to give us a mixable delegation for creating pilots:

```
trait PilotProvider {
    def pilot: Actor = new Pilot
    def copilot: Actor = new CoPilot
    def autopilot: Actor = new AutoPilot
}
```

Putting Everyone in the Plane

Now that we have our flight crew, we need to get them instantiated properly, which the Plane will do for us. Let's add a couple of lines to the construction of our Plane, right after the construction of the controls:

```
val controls = context.actorOf(Props(new ControlSurfaces(altimeter)))
val config = context.system.settings.config
val pilot = context.actorOf(Props[Pilot],
    config.getString("zzz.akka.avionics.flightcrew.pilotName"))
val copilot = context.actorOf(Props[CoPilot],
    config.getString("zzz.akka.avionics.flightcrew.copilotName"))
val autopilot = context.actorOf(Props[AutoPilot], "AutoPilot")
val flightAttendant = context.actorOf(Props(LeadFlightAttendant()),
    config.getString("zzz.akka.avionics.flightcrew.leadAttendantName"))

override def preStart() {
    // Register ourselves with the Altimeter to receive updates on our altitude
    altimeter ! EventSource.RegisterListener(self)
    List(pilot, copilot) foreach { _ ! Pilots.ReadyToGo }
}
```

Not bad at all. Our Plane is starting to shape up now that it has people in it who can actually do some stuff. We'll start using these guys really soon; for the moment, let's just look at what we have. [Figure 7.3](#) shows us the structure of the Actors we just created, how they relate to one another, and the paths associated with each of them.

7.5 Relating the Path, Context, and System

There's a wonderful diagram in the Akka reference documentation that I've almost entirely ripped off for the purposes of illustration, which you can see in [Figure 7.4](#).

The diagram shows:

- How the Actor relates to its ActorContext through the `context` member
- That the ActorContext holds the understanding of the relationships between children and parents, and also holds the reference to `self`.

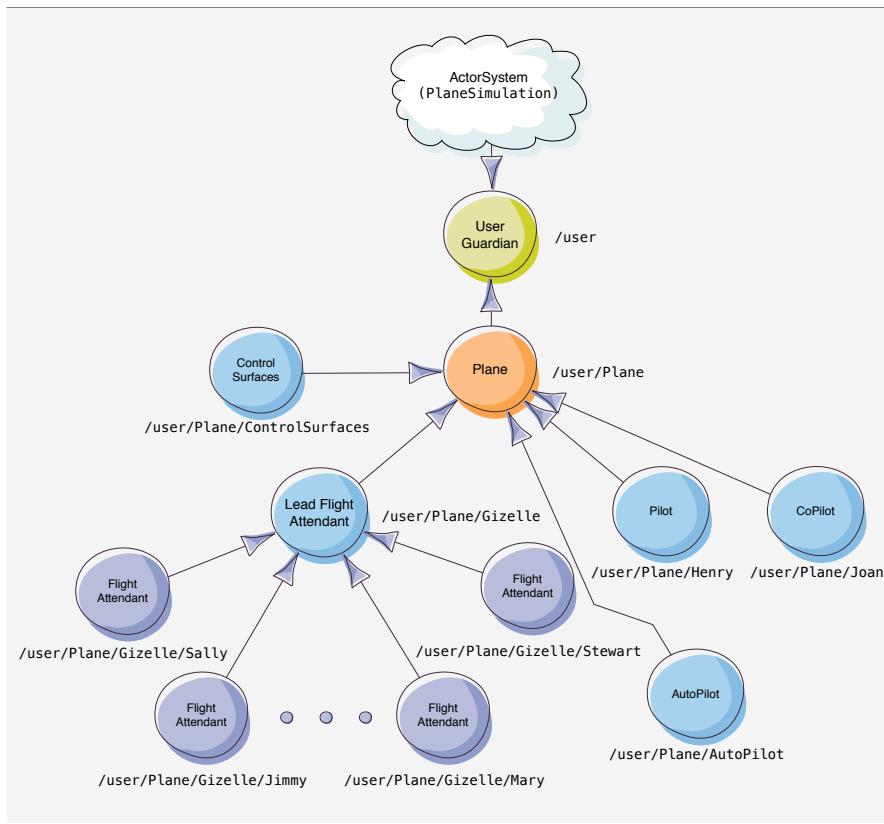


Figure 7.3 · The structure of the Plane after everyone's in place

- That we use `self` to access our own `ActorRef`, which is what is actually visible to the outside world, and use it to see the `ActorPath`.
- That we use `ActorPath` to access the parental relationships, but only in a “textual” manner.

The diagram should serve as a helpful reminder for knowing how you can traverse the hierarchy at any given point, and how you can relate certain parts of the Actor design to others. For example, if you have an `ActorPath`, then you can get the `ActorRef` using the `ActorSystem`, and if you have an `ActorContext`, you can get the children and parent `ActorRef` directly. How-

ever, if you want the ActorPath of any of those, then you'll need to get that through the ActorRefs.

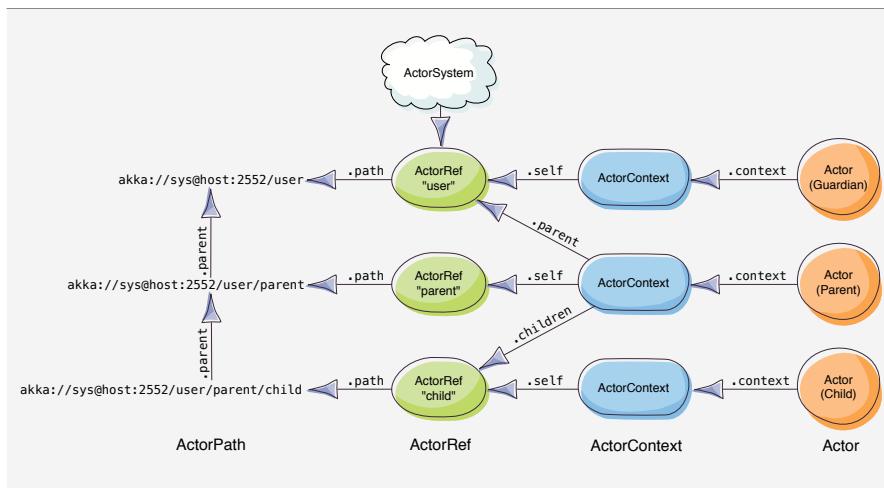


Figure 7.4 · How we relate the Actor to its context, its ActorRef, and the path. It's the context that holds the relationships together, and those point to the ActorRefs (not the Actors or the ActorContexts). From the ActorRef, we can retrieve the ActorPath object, which we can interrogate as we please.

7.6 Chapter Summary

This chapter gave us a pretty good view into how Akka organizes Actors. You'll find that Actor programs are very much defined by how they are physically structured, both to help define the algorithm you're designing and to create a degree of fault tolerance for which Actors are justly famous.

Specifically, you've increased your skills in the following areas:

- We learned a bit about the mechanics of the configuration system, including how to add our own custom configurations to our application.
- We saw how Akka creates Actor hierarchies automatically for us by creating Actors using the `context.actorOf()` factory method.
- We now understand the Guardian Actor and learned that its job is to provide a safeguard for its children.

- We can look up Actors using relative paths just like we would in a file system using `context.actorFor()`.
- We learned a bit more technical information regarding the Scheduler such that we can have some realistic expectations of it as well as tailor it for our tests.
- There's also a neat trick for specifying configuration parameters at runtime that we saw while playing around with that Scheduler.
- We also learned how Akka deals with the real world in a pragmatic sense. The Akka designers don't pretend that they can give you stable results to certain operations in a concurrent world. These are the issues that come up regardless of any technology and/or toolkit; this is the stuff that just happens.

What we now know will help us understand all of the neat aspects of how Akka helps us create fault-tolerant systems. We're going to need it too. This Plane's got some problems and in order to fix them, we're going to let it heal itself.

Chapter 8

Supervision and DeathWatch

You're 35,000 feet in the air, crammed into an aluminum tube full of highly explosive fuel, with food that people in hospitals would reject and that guy in seat 23C muttering to himself in some incoherent babble-speak with a twitchy left eye. What could possibly go wrong?¹

As you know, Actor programs contain anywhere from a few to a few hundred million live objects running around doing what they do best. You also know that they aren't just running around like a bunch of crazy chickens; they're slotted into a very specific and very ordered hierarchy. And because you're a citizen of our planet, you know about this thing called *real life* and are well aware that given enough time, enough code, and the fact that life is generally a problem, you are guaranteed that things are going to go wrong.

This is where Akka and the Actor model come to your rescue. Failures and just plain *weird stuff* are modeled into your application so that, even under strange circumstances, your application still runs; you could even say that it can heal itself when it gets injured.

In this chapter, we'll start ensuring that our Plane is really solid. Not only will we put the controls and instruments in our self-healing scheme, but we'll put the people in it under the same sort of rules. Although, we'll make sure that some of our people aren't quite superhuman... before this chapter's out, somebody's gonna have to die.

8.1 What Makes Actors Fail?

```
throw new Exception("Say hello to my little friend!")
```

¹Seriously, I have to go on a plane next week and now I don't want to.

In a nutshell, that's it. Actors fail because something inside the receive method throws an `Exception`. Either your own code has thrown something (i.e., you specifically throw) or there's a library you're using somewhere that throws; it doesn't matter. Something throws and it causes your Actor to toss his cookies.

We all know what an exception throw does to our usual notion of application flow, but what does it mean to a living Actor?

Actor Paramedics

Every Actor has a parent and, as such, every Actor has a *Supervisor*. The supervisor decides what should be done when it sees a child get injured due to an `Exception`. The parent sees the `Exception`, matches it against a partial function, and then decides what should be done to the affected Actor in order to deal with the problem. The Actor system's hierarchical nature makes this decision-making recursive, so that at any point in the hierarchy we can ask the same question and get a simple result. [Figure 8.1](#) shows how the recursive definition applies to supervision.

Because the Actor's implementation is decoupled from Akka's internal logic, your code's failure cannot adversely affect Akka itself. When your code fails, that merely invokes logic in Akka that performs some sort of operation to deal with the `Exception`. Akka can deal with your quarantined code in several ways, from the default-handling mechanisms to intricate fault-handling strategies of your own devising.

8.2 The Actor Life Cycle

It's hard to talk about supervision, death, and resurrection unless we understand more about the Actor life cycle. We need to know what happens to an Actor from birth to death and subsequent resurrection.

[Figure 8.2](#) represents the states through which an Actor can go. These are not the internal details of Akka's implementation, although they certainly have direct mappings to said implementation, but rather depict the points in which we as programmers are interested. The code we write will influence the life cycle by driving it to particular states. For example:

- We can explicitly create and start an Actor by constructing it using the factory methods we've seen before; i.e., `context.actorOf(...)` and

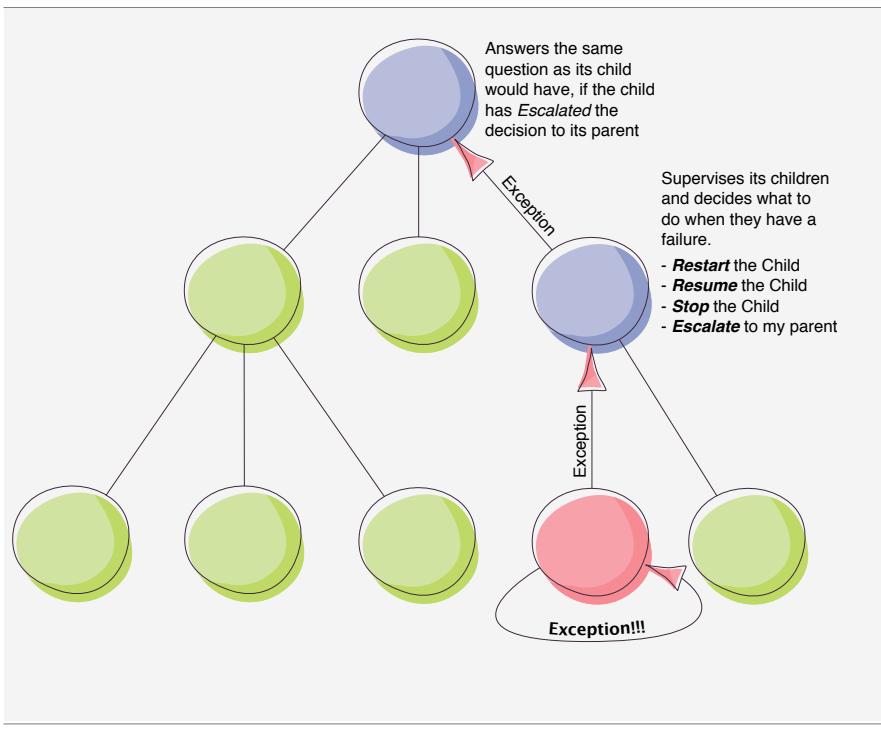


Figure 8.1 · Actor supervision is a recursive definition. The Actor system's hierarchical nature provides a simple definition of supervision such that the question of "what to do?" can be asked at any level.

```
system.actorOf(...).
```

- We can explicitly stop the Actor in several different ways but one of the most explicit ways is to call `context.stop(...)` or `system.stop(...)`, where the "..." is a placeholder for a valid `ActorRef`.
- If our code throws an Exception, then that immediately kicks in the supervisory behaviour. Depending on what we decide to do, our Actor could be restarted, resumed, or stopped.

In addition to influencing the states of the Actor's life cycle on a macro scale (i.e., *start*, *stop*, and *restart*), we can also get some hooks into that life cycle, which we can use to perform certain activities at the right time.

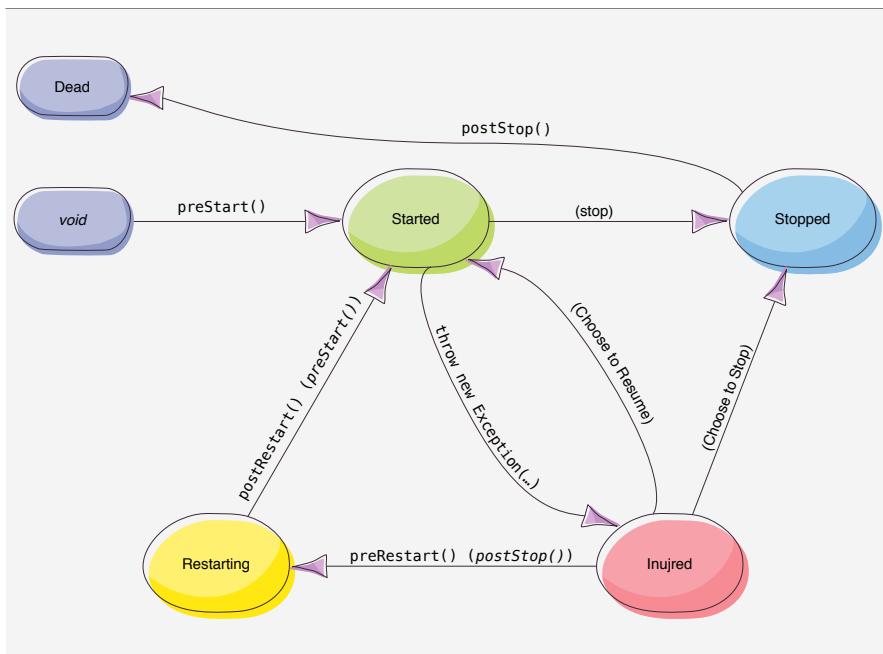


Figure 8.2 · The Actor life cycle, including code examples that can get us to each point as well as certain overrideable callbacks that are available to hook into the life cycle.

We'll be covering the restart cases in a moment, but let's just have a quick look at the *start* and *stop* hooks. They're really quite simple:

```
class MyActor extends Actor {  
    override def preStart(): Unit = {  
        // Perform any initialization setup here  
        // Often this is a good spot to send yourself a message  
        // such as: self ! Initialize  
    }  
  
    override def postStop(): Unit = {  
        // Perform any cleanup here. The message pump is shut down  
        // so any message you send to yourself will only go to the  
        // dead letter office, but if you'd like to clean up any  
        // resources, such as Database sessions, now's the time to  
    }  
}
```

```
// do it.  
}  
  
def receive = {  
    // Do your usual processing here. For example:  
    case Initialize =>  
        // Call your own post start initialization function here  
        postStartInitialization()  
    }  
}
```

8.3 What Is a Supervisor?

A Supervisor is the aspect of an Actor that takes care of its children. A piece of code governs this aspect and specifies the decisions it needs to make when children get injured. This code is created as a value called `supervisorStrategy`, which you can override from the default to do whatever you'd like. The default implementation looks like this:

```
final val supervisorStrategy = OneForOneStrategy() {  
    case _: ActorInitializationException  => Stop  
    case _: ActorKilledException          => Stop  
    case _: Exception                   => Restart  
    case _                            => Escalate  
}
```

There are two key pieces of information in that strategy:

1. `OneForOneStrategy()` declares that children will be dealt with on a one-for-one basis; the decision made regarding an Actor's failure will apply only to that one failed Actor. This is in contrast to the `AllForOneStrategy()`, which applies the decision regarding a single Actor's failure to all children.
2. The code block. The specified block of code is known as the *Decider*. The Decider is a `PartialFunction[Throwable, Directive]`, where Directive has four derivations: Stop, Restart, Resume, and Escalate. If you specify Escalate, then you're getting that recursive decision passing that we saw in [Figure 8.1](#).

When you override the `supervisorStrategy`, you must make choices about these two aspects in order to be complete. First, you must decide whether or not you'll apply the decision to one or all of your children and you must then decide what you'll do depending on the Exception that was thrown.

Reasoning About Fault-Tolerance

Keeping an Actor application fault-tolerant and reliable involves a very intimate relationship between an Actor's behaviour and its parent's supervisor strategy.

Note

The amount of application logic that an Actor can support is directly proportional to the fault-handling strategy that you can employ when any of that logic fails.

When you're writing your Actor application, you'll always have two goals in mind:

- The application logic that gets the job done.
- What happens when that logic fails to do its job.

One of the key aspects to keeping these two forces in balance revolves around complexity. If your Actor is doing too much and becomes too complex, then reasoning about its fault-handling strategy becomes problematic. When you reach this point, it's time to refactor.

For example, let's say your Actor has the following, where a failure when handling `Process(activity)` requires an Actor restart, and a failure when handling `Evaluate(expression)` requires the Actor to resume:

```
def receive = {
    case Process(activity) =>
        // May throw IllegalAccessException
        someLibrary processActivity(activity)
    case Evaluate(expression) =>
        // May also throw IllegalAccessException
        someOtherLibrary evaluate(expression)
}
```

Now you have something difficult to reason about, because the tool you have at your disposal looks like this:

```
override val supervisorStrategy = OneForOneStrategy() {  
    case _: IllegalAccessException => ??????  
}
```

By the end of this chapter, you'll understand what you should do in situations where your logic becomes difficult to reason about when faults could occur. For now, keep in mind that there will come times when your application logic tips the balance of reasonable fault-tolerance, which will require you to rethink and refactor your Actor structure.

Decider Directives

The output of your Decider is a Directive that tells the supervisor logic what it should do with your child or children, depending on whether or not you chose a `OneForOneStrategy` or an `AllForOneStrategy`. You have four choices available to you:

Stop You've decided to stop the Actor that threw the exception. For example, if the Actor throws an `OperationCompletedByAnotherActor` exception, then this might be an entirely reasonable thing to do. There would be no point in getting the child to do anything but stop.

Resume In this case, you've decided that there's nothing special to this problem. The Actor can simply resume its operation as though nothing bad had ever happened.

Escalate Your supervisor doesn't have enough information to make a decision about what to do. It passes the buck to its parent, in the hopes that its parent has a grander vision of the problem space and can decide what should be done. Note that the parent isn't deciding what to do with its children's children, but what to do with its own children. This means that the Actor that passed the Escalate will potentially restart as a consequence.

Restart This is the decision that is the most desirable, but is also the most difficult to reason about for most of us. The idea of a restart is to construct a fresh instance of the Actor and let it process the next message.

The reason for choosing this option is when you think that there's some inconsistent internal state of the Actor, which is causing the problem and it can no longer be trusted. Since everything (presumably) worked just fine before the Actor entered this unfortunate state, a fresh instance should be much more trustworthy.

What Happens to the Current Message?

This is a very good and common question: What happens to the message that was being processed when the Actor failed?

The simple answer is that, by default, it disappears. Since the message was (most likely) the cause of the problem in the first place, and computers being the rather deterministic machines that they are, then it's reasonable to assume that trying to process it again will cause the same problem to occur. Because of this, the message is removed from the Mailbox and processing begins at the next message.

The more complicated answer is that it depends.

- If you're going to Stop the Actor, then it really doesn't matter what happened to the message, does it? The entire Mailbox is about to be sent to the Dead Letter Office anyway, so what's one more message?
- If you're going to Resume the Actor, then again the message disappears. You've chosen to resume the Actor's operation, which means you're going to ignore what happened. Because you're ignoring it, you don't get any access to deal with the thing that happened.
- If you're going to Restart, then you have some life-cycle hooks.

Restarting provides the hooks you might want in order to participate in the restart life cycle. Using `preRestart()` and `postRestart()`, you can gain access to the exception that caused the failure and to the message that was being processed during that failure. Look at [Figure 8.3](#) and you'll see that in `preRestart()` we have access to the message and the exception, but since the Actor is in its "failed" state, we also have access to any of the internal state that existed at the time of the failure, including the *sender* of the message.

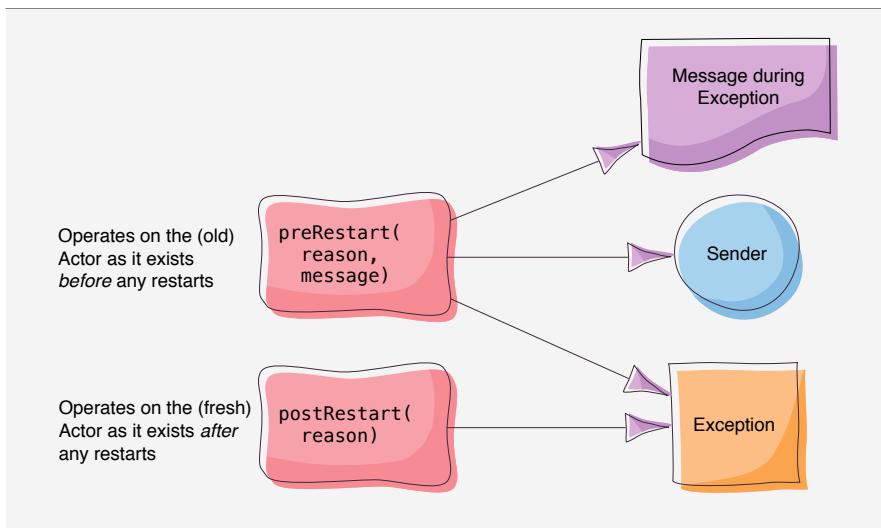


Figure 8.3 · Two hooks exist for restart processing in the Actor. It's important to understand that the `preRestart()` method is executed on the Actor that processed the failing message, whereas the `postRestart()` method is executed on the freshly instantiated Actor.

Won't Someone Please Think of the Children?

Restarting an Actor implies that the entire Actor restarts, which would certainly imply that the entire Actor hierarchy that it represents restarts. Sometimes this is good and sometimes it's not so good.

It's pretty simple to guess when restarting the children is generally a bad thing to do; it's depicted in Figure 8.4. If the Actor being restarted is high up in the Actor hierarchy, and its children restart, then the restarts will propagate all the way down the tree.

You can also imagine a scenario where restarting all of your children might be just what the doctor ordered. For example, the Actor tree is representative of the Unix file system, where each Actor holds the inode of a particular file. If the restart of the root node implies the fail-over to a backup file system, then you may need to refresh all of the known inodes.

However, I wouldn't exactly call that the usual case. The usual case is much more likely to be that the root Actor of a particular part of the hierarchy will be important and simple enough that even in the face of a restart, its

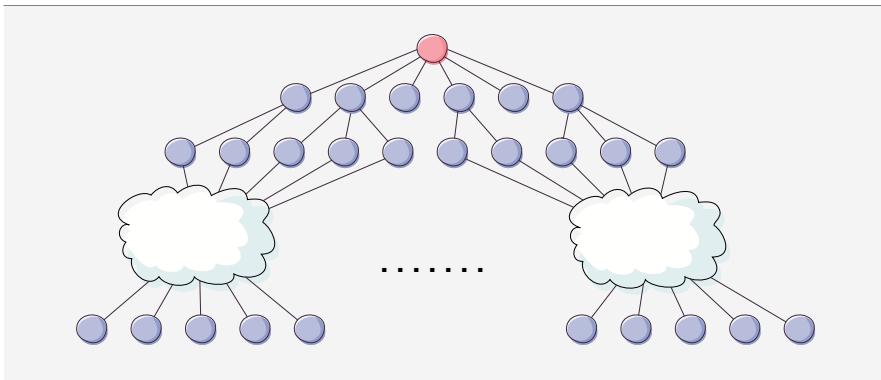


Figure 8.4 · We wouldn't necessarily want to restart all of the children when the Actor that failed is a great, great, great, great, great, great, great grandfather to 5,000 descendants.

children should not restart.

However (again), there's something that needs to be understood about children and restarts. Remember that the purpose of restarting an Actor is to clear out any bad state it may have accumulated. Well, children are part of its state, and it could be that the ultimate reason for the Actor having problems is because of some problem child. You know, that kid in school who insists on beating up that other kid who likes computers and comic books...oh, never mind.

The point is that the default behaviour of restarting children is really a proper default. It's just a default behaviour that becomes less desirable the higher up the hierarchy you travel.

What About that User Guardian?

If we create a child of the User Guardian via `system.actorOf()`, what supervisor strategy gets instantiated? Well, by default, it's the default, which shouldn't be much of a surprise. That means that if the User Guardian's children throw an Exception, they're going to restart. If that's not what you want, then you can modify it by a change to the configuration.

If you set `akka.actor.guardian-supervisor-strategy` to the fully qualified class name of an instance of `akka.actor.SupervisorStrategyConfigurator`, then that configurator will be used to create the User Guardian's supervisor

strategy. Here's how it's done:

```
package zzz.akka
import akka.actor.SupervisorStrategyConfigurator
import akka.actor.SupervisorStrategy._
import akka.actor.OneForOneStrategy

class UserGuardianStrategyConfigurator
    extends SupervisorStrategyConfigurator {
    def create(): SupervisorStrategy = {
        OneForOneStrategy() {
            case _ => Resume
        }
    }
}
```

Now that we have that all defined, we can modify the configuration file to point to it. When the User Guardian gets instantiated, it will use our new configurator to create the OneForOneStrategy we've just defined.

```
akka {
    actor {
        guardian-supervisor-strategy = zzz.akka.UserGuardianStrategyConfigurator
    }
}
```

We've now changed the User Guardian's default strategy such that it won't restart children when they throw Exceptions; they'll now be resumed.

The preRestart() and postRestart() Methods

The Actor's `preRestart()` and `postRestart()` hooks are really quite important because a lot of the default behaviour that you get in the Actor life cycle is governed by the default implementations of these methods. As a result, overriding them can cause adverse affects if you effectively remove the behaviour that they were giving you for free.

Let's first look at what the default implementations of these methods look like:

```
def preRestart(reason: Throwable, message: Option[Any]) {  
    context.children foreach context.stop  
    postStop()  
}  
  
def postRestart(reason: Throwable) {  
    preStart()  
}
```

The reason that this is interesting is because under normal (non-restart) circumstances there's nothing we can or need to do in order to ensure that `preStart()` and `postStop()` are invoked. With `preRestart()` and `postRestart()`, we have control over these other life-cycle methods. In fact, it's up to us to ensure that `preStart()` and `postStop()` get called if that's what we want.

What's also important to note is that it's the `preRestart()` method that is in charge of stopping the children. So, if we don't want our children to stop, then it's under our control to make sure that doesn't happen.

The default implementation of Actor restart involves the following steps:

1. Suspend the Actor's processing.
2. Suspend the processing of all of the Actor's children.
3. Call `preRestart()` on the failed Actor instance.
 - Terminates all children. This is not a blocking operation, but Akka does guarantee that your Actor won't start up again until all its children have stopped.
 - Calls `postStop()` on the failed Actor instance.
4. Constructs a new instance of the failed Actor using the originally provided factory method.
 - Note that this is a standard construction of your Actor code.
 - If you're creating children in your `preStart()` or the constructor, then they will get created at this time.
5. The `postRestart()` method will be called on this new instance.
6. The new Actor is then put back on the queue to process incoming messages.

Restarting Forever

Of course, we need to consider something when we talk about restarting Actors. What happens when restarting simply won't work? For example, assume we have the following Decider:

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

If the database is down for a day, then we'll spin on this restart strategy like it's going out of style. It's very easy to make the argument that a restart strategy that does this is of little value; at some point, someone's going to need to *know* that the database is down.

To deal with this, the `OneForOneStrategy` and `AllForOneStrategy` accept a restart threshold on its constructor that says just how many restarts, within a specific range of time, it's willing to tolerate. For example:

```
// For the Duration DSL import akka.util.duration._  
override val supervisorStrategy = OneForOneStrategy(5, 1.minute)
```

Akka will keep track of how many restarts have been called within the range of time specified, and as soon as it goes over the threshold the Actor will stop and its `postStop()` method will be called.

This ensures that the Actor will not restart *ad infinitum*, but it doesn't really help us take corrective action when the Actor finally dies. At this time, we come to *DeathWatch*.

8.4 Watching for Death

An Actor can restart as many times and as often as you dictate via the supervision strategy. All of this restarting is completely invisible to the outside world; this includes the Actor that's doing the supervising; in fact, that's actually the whole point. But there will probably come a time when the Actor that's restarting eventually has to give up its ghost and pass on to the ultimate end. This is where *DeathWatch* comes in.

Any Actor can *watch* any other Actor for death using the `ActorContext`'s `watch()` and `unwatch()` methods. When an Actor that you're currently

watching dies, you'll get a message that tells you so. It's almost as simple as that. We'll be putting this into practice soon, but let's look at a simple example now:

```
class MyActor extends Actor {  
    override val supervisionStrategy = OneForOneStrategy(5, 1 minute) {  
        case _ => Restart  
    }  
    override def preStart() {  
        context.watch(context.actorOf(Props[SomeOtherActor]))  
    }  
    def receive = {  
        case Terminated(deadActor) =>  
            println(deadActor.path.name + " has died")  
    }  
}
```

If the child Actor (the instance of `SomeOtherActor`) finally terminates, then the `Terminated` message will be processed, and the dead Actor's `ActorRef` will be included in the message. This enables you to take more interesting corrective action than a restart; you can terminate the app, instantiate a different type of Actor in the child's place, back off on re-creating the child Actor (for example, when trying to re-attach to a database server), and so on.

Restarts, Watches, `Terminated()`, and Children

Earlier we looked at the restart life cycle and learned that `preRestart()` will stop children. When you re-create a fresh Actor, the children that you spawn during creation will get created again. But what does that mean for the children's life cycle? It's at this point where the life cycle of the Actor being restarted and the life cycles of its children diverge. [Figure 8.5](#) illustrates.

The restart life cycle is local to the Actor undergoing restart, and that restart life cycle is not propagated to its children. If the Actor has DeathWatch on its children, then the mere fact that it is specifically telling them to `stop()` will cause a `Terminated()` message to go into its mailbox, which will be processed once the fresh instance starts processing messages again. However, it's the *old generation* that stopped, not the current generation.

This leads to the following tricky code:

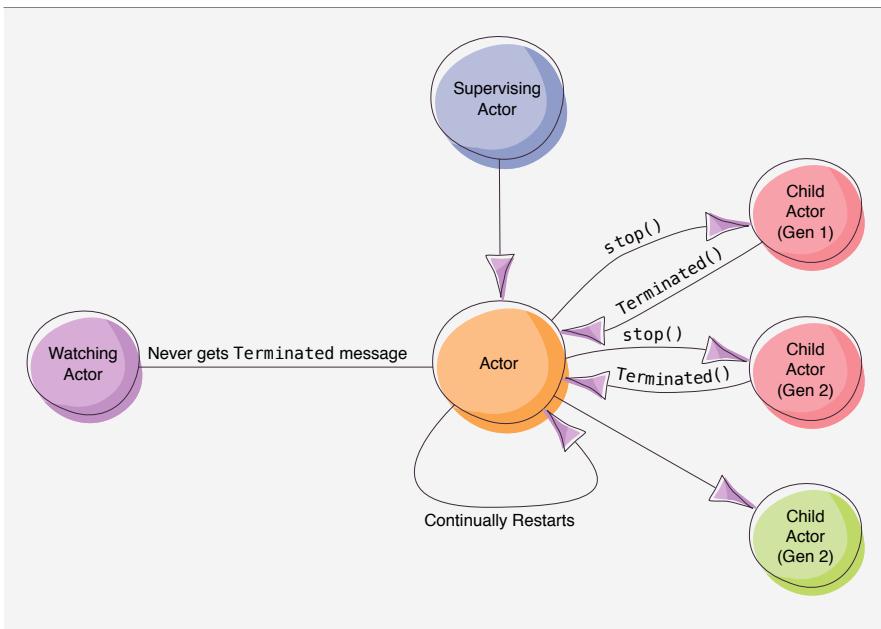


Figure 8.5 · The Actor that restarts passes through the *restart* life cycle, but its children are a different story. The children are specifically *stopped* and re-created. With each Actor restart, the children are stopped and a new generation replaces them. If the Actor has DeathWatch on them, then it will get *Terminated()* messages for each one on every restart.

```
class MyActor extends Actor {
    // Create the child when constructed, whether for the first time
    // or for the 1,000th restart
    context.watch(context.actorOf(Props[SomeChild]))

    def receive = {
        case Terminated(child) =>
            // re-create the failed child <- BUG
    }
}
```

The `Terminated(child)` message will come from one of two situations:

- The child finally gave up its ghost due to exceptions.

- The child was specifically stopped during this Actor's restart.

The above code was written in the spirit of the first case, not the second. When the second case presents itself to this piece of code, the terminated child has already been replaced by MyActor's constructor, so re-creating it is a very bad idea. Every time we restart, we create yet another child.

Managing Your Children During Restarts

While you're probably the author or are intimately involved with all of the authors² of your Actor application, you have to realize that your Actor's supervisor logic is not necessarily up to you, since its the *parent* that defines that logic. If you exhibit this "Actor leakage" as part of your Actor's implementation, it may never manifest so long as the supervisor never restarts it (e.g., it could choose to stop or resume). The day it decides to start restarting it, bad things will happen.

Assuming that...

1. Your Actor is involved in a restart scenario.
2. It has children.
3. It's monitoring those children for death.
4. It restarts them manually when they finally terminate.

...then you need to be careful about the life cycle of those children by effectively taking them out of your own restart piece of the life cycle.

```
class MyActor extends Actor {  
    def initialize() {  
        // Do your initialization here  
    }  
  
    override def preStart() {  
        initialize()  
        // Start your children here  
    }  
  
    override def preRestart(reason: Throwable,
```

²No, I don't mean it like that... geez.

```
message: Option[Any]) {  
    // The default behaviour was to stop the children  
    // here but we don't want to do that  
    // We still want to postStop() however.  
    postStop()  
}  
  
override def postRestart(reason: Throwable) {  
    // The default behaviour was to call preStart()  
    // but we don't want to do that, since that's  
    // where children get started  
    initialize()  
}  
  
def receive = {  
    case Terminated(child) =>  
        // re-create the failed child. Now it's OK,  
        // since the only reason we can get this message  
        // is because the child really died without  
        // our help  
    }  
}
```

We have now removed the children from this Actor's restart life cycle. The old behaviour of `preRestart()` was to stop the children, which we've now removed. We've also ensured that when `MyActor` is truly *started* (i.e., not *restarted*) that its children get created, but when it is restarted that they don't. We've accomplished this by factoring the non-child logic out into its own method.

Tip

When creating children for your Actor, ensure that you do so from inside the `preStart()` override. You can defer other types of initialization to the constructor or some sort of `initialize()` method, depending on your needs, but **do not** create children from inside the constructor since this gives you no opportunity to control their life cycles.

A consequence of this style of child management under this special case is that the original intention of stopping children during restart is now out of play. If the Actor's children represent bad state, then they should be restarted, but in this case they won't be. Make sure you have a deep understanding of

your hierarchies and potential situations under restart.

Beware of External References to Your Children!

There's one last thing you need to understand regarding your children's life-cycle changes. If your restart stops children and re-creates them, then what do you think happens to the ActorRef? Sure, it changes; the old ActorRef will be heading straight to the Dead Letter Office. That's just fine so long as no code has references to those children, but what if you've got code that holds on to those references? [Figure 8.6](#) makes this clear.

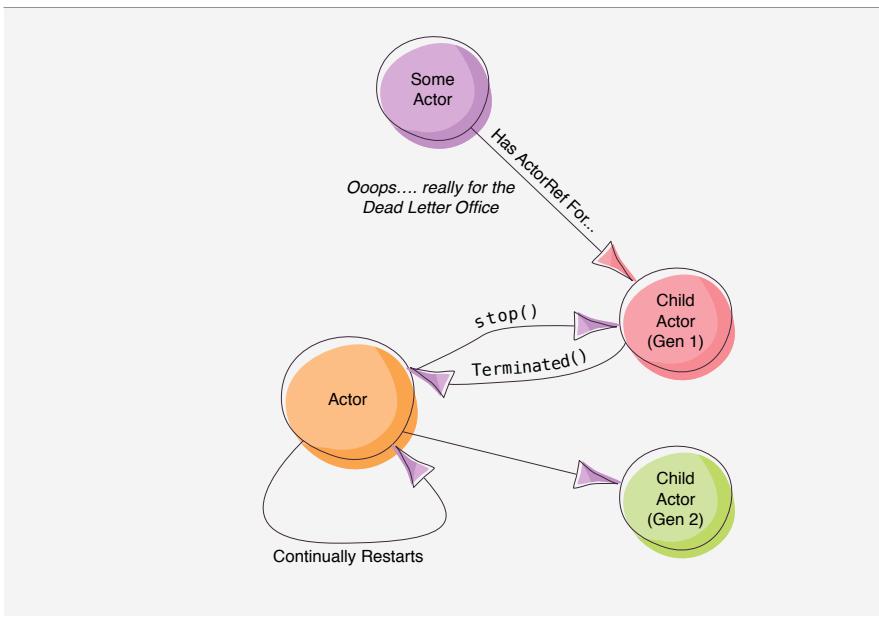


Figure 8.6 · Actor restarts are not visible to the outside world and thus all of the guys that have an ActorRef to it are unaffected. However, it's a different story for those that might have references to that Actor's children. Since the children will by default stop and then get re-created, those old references send all messages to the Dead Letter Office.

So what do you do? DeathWatch, of course. If the Actor that holds the reference to the child puts a DeathWatch on it, then it will be notified when it dies. What it does at that point is dependent on what you decide as a programmer, so we're not going to cover it now. You'll make whatever

the right choice is for your application eventually, armed with the knowledge we'll accumulate as we go forward.

8.5 The Plane that Healed Itself

Armed with all of this knowledge about life cycles, supervision, and death watch, it's time to make our Plane into a mythical super machine that can heal itself when things go wrong. To do this, we'll reorganize things a little and we'll supply a bit of self-typing to allow our Plane to be constructed in a more flexible manner. This will become more important as we drive toward testing what we've written.

The Goal

In the last chapter, we learned about the hierarchical path structure of the Actor system and how we can use the `actorFor()` methods to look up Actors within that system; we even used that mechanism for finding various Actors in our Plane. As we move forward, we're also moving further into the real world of Actor programming with Akka, and in the real world, things change. That's not just a metaphorical statement—we're actually going to change the structure of our Plane, and in doing so, we're going to break some of that hierarchy that we set up before.

[Figure 8.7](#) shows the structure that we're driving toward with our Plane at this stage. Contrast this with [Figure 7.3](#), which was rather shallow in nature. With this new layout, we recognize that different Actors require different types of supervision and we facilitate that by inserting dedicated supervisory nodes into the structure. We do this because, as we know, any given Actor can only implement a single supervision strategy and the power of supervision comes from simplicity at any given level. If we have one node in the tree doing too much, we'll put our system's resiliency at risk. Our goal is to define small pockets of reason in the hierarchy that react well under failure.

Our supervision strategy for the Plane as a whole will be:

- The Plane itself will be supervised by the User Guardian, which employs a `OneForOneStrategy` that restarts children (i.e., our Plane) when a failure occurs.

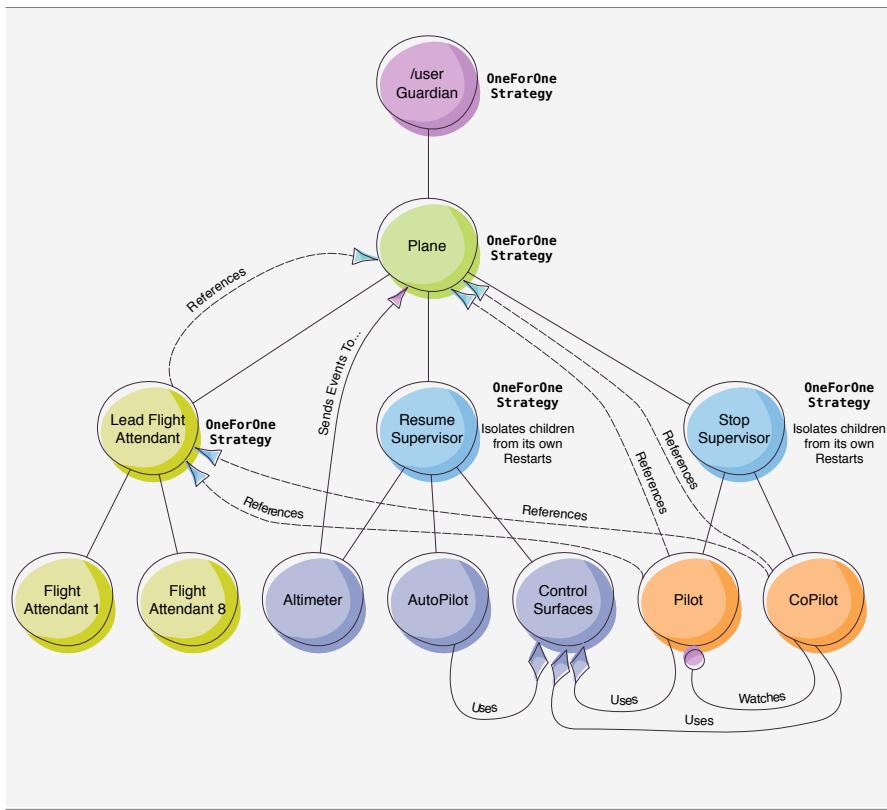


Figure 8.7 · The outline of the structure we’re going for when building our Plane. Actor systems inevitably become a tree in the real world and our Plane is starting to get that way.

- The LeadFlightAttendant will be supervised by the Plane’s OneForOneStrategy, which will Restart the LeadFlightAttendant on failure. The LeadFlightAttendant will, in turn, stop all of its children (i.e., the FlightAttendants) and recreate them.
- The Instruments and Controls will be treated differently; we don’t want the Plane’s Restart handler to affect these Actors. To manage this, we’ll put in a dedicated Resume Supervisor, which will isolate its children from any effects of its own Restart from the Plane’s supervision strategy.

- The Pilots will be different than the rest. When a Pilot dies, he's dead. To facilitate that, we'll create a dedicated Stop Supervisor, which will also isolate its children from any effects of its own Restart from the Plane's supervision strategy.

We'll need some help, and it will come in the form of a generalized Supervisor Actor, whose sole job is to supervise its children. We'll also have this supervisor keep its children isolated from its own restart behaviour, assuming its parent initiates such a restart.

An IsolatedLifeCycle Supervisor

The IsolatedLifeCycleSupervisor will provide us with a supervisor's base functionality that lets children survive the supervisor's own restarts, as well as provide some extra plumbing to make that happen.

```
object IsolatedLifeCycleSupervisor {  
    // Messages we use in case we want people to be able to wait for  
    // us to finish starting  
    case object WaitForStart  
    case object Started  
}  
  
trait IsolatedLifeCycleSupervisor extends Actor {  
    import IsolatedLifeCycleSupervisor._  
    def receive = {  
        // Signify that we've started  
        case WaitForStart =>  
            sender ! Started  
        // We don't handle anything else, but we give a decent  
        // error message stating the error  
        case m =>  
            throw new Exception(s"Don't call ${self.path.name} directly ($m).")  
    }  
    // To be implemented by subclass  
    def childStarter(): Unit  
        // Only start the children when we're started  
    final override def preStart() { childStarter() }  
    // Don't call preStart(), which would be the default behaviour
```

```
final override def postRestart(reason: Throwable) { }

// Don't stop the children, which would be the default behaviour
final override def preRestart(reason: Throwable, message: Option[Any]) { }
}
```

There are some things to note here:

- The `receive` method is implemented for us. If *any* message comes into our supervisor (other than a request to be told when it's finished starting), then we'll throw an exception. The supervisor is fault-tolerance plumbing and there's no reason for anyone to talk to it. We assume that if someone does talk to it directly, then that's a bug.
- We've ensured that most existing life-cycle methods cannot be altered by derived types. This ensures that our isolation remains intact. The job of creating children is left to the derivations, but outside of `preStart()` so that the isolator can control the life cycles on its own.

Given the `IsolatedLifeCycleSupervisor`, we have a class from which we can derive simple supervisors tailored for the specific needs of the Actors within our Plane.

Creating the Supervisors

We'll go pretty deep here and create some types that will really help us generate supervisors quickly and easily. By defining some traits and some abstract classes, we can simplify instantiation of the supervisors a fair bit, thus making our code easier to read and understand.

```
trait SupervisionStrategyFactory {
    def makeStrategy(maxNrRetries: Int,
                    withinTimeRange: Duration)(decider: Decider): SupervisorStrategy
}

trait OneForOneStrategyFactory extends SupervisionStrategyFactory {
    def makeStrategy(maxNrRetries: Int,
                    withinTimeRange: Duration)(decider: Decider): SupervisorStrategy =
        OneForOneStrategy(maxNrRetries, withinTimeRange)(decider)
}
```

```
trait AllForOneStrategyFactory extends SupervisionStrategyFactory {  
    def makeStrategy(maxNrRetries: Int,  
                    withinTimeRange: Duration)(decider: Decider): SupervisorStrategy =  
        AllForOneStrategy(maxNrRetries, withinTimeRange)(decider)  
}
```

Above we have a factory declaration and two definitions that we can use to abstract away the concrete concepts of the OneForOneStrategy and the AllForOneStrategy, respectively. We will be mixing one of these factories into our supervisors as needed.

Given these factories, we can now define the specific instances of the IsolatedLifeCycleSupervisor that we declared earlier:

```
abstract class IsolatedResumeSupervisor(  
    maxNrRetries: Int = -1, withinTimeRange: Duration = Duration.Inf)  
    extends IsolatedLifeCycleSupervisor {  
    this: SupervisionStrategyFactory =>  
  
    override val supervisorStrategy = makeStrategy(maxNrRetries, withinTimeRange) {  
        case _: ActorInitializationException => Stop  
        case _: ActorKilledException => Stop  
        case _: Exception => Resume  
        case _ => Escalate  
    }  
}
```

The Resume Supervisor declares the Decider to be one that resumes operation of the children in the case of any Exception that is not ActorInitializationException or ActorKilledException. It also does not know what type of strategy will be in place, delegating this to a future trait mixin.

```
abstract class IsolatedStopSupervisor(  
    maxNrRetries: Int = -1, withinTimeRange: Duration = Duration.Inf)  
    extends IsolatedLifeCycleSupervisor {  
    this: SupervisionStrategyFactory =>  
  
    override val supervisorStrategy = makeStrategy(maxNrRetries, withinTimeRange) {  
        case _: ActorInitializationException => Stop  
        case _: ActorKilledException => Stop  
        case _: Exception => Stop  
    }
```

```
    case _ => Escalate
  }
}
```

The Stop Supervisor is nearly identical to the Resume Supervisor, with the obvious difference that it stops children instead of resumes them.

Now that we have these helpers in place, we can start refactoring our Plane's structure to fit the goal.

Our Healing Plane

Let's break this down into pieces.

Construction Components

The Actor program's hierarchical structure is a constant presence in your design. It's a powerful structure, to be sure, but it's also an imposing structure that can limit your options, if you use it inappropriately. We want to take advantage of the hierarchy where it can help us, but we also want to ensure that we don't get so wrapped up in it that refactoring becomes more of a challenge than we'd like down the road. This will be a recurring theme as we move forward.

To begin, we need to apply a simple refactoring that allows us to provide construction of Plane members using a variant of the Cake pattern. This will allow us to escape the confines of the hierarchy that would otherwise be hard-coded.

```
trait PilotProvider {
  def newPilot: Actor = new Pilot
  def newCoPilot: Actor = new CoPilot
  def newAutopilot: Actor = new AutoPilot
}
```

Here, we see the trait that will provide us with different types of Pilots when we need them. This trait can be overridden with derivations used in testing, or other cases, which keeps our design flexible. The Plane can now take advantage of this provider, along with similar refactorings that we've applied to the LeadFlightAttendant and the Altimeter:

```
class Plane extends Actor with ActorLogging {  
    this: AltimeterProvider  
        with PilotProvider  
        with LeadFlightAttendantProvider =>
```

The refactorings to be applied to the LeadFlightAttendant and the Altimeter are left as exercises for you, should you choose to accept them.

Building the Basic Hierarchy

Now that we've decoupled the construction of the Plane's elements from the elements themselves, we can revisit the creation of them and the insertion of Supervisors into the hierarchy.

```
def startControls() {  
    val controls = actorOf(Props(new IsolatedResumeSupervisor with OneForOneStrategyFactory {  
        def childStarter() {  
            val alt = context.actorOf(Props(newAltimeter), "Altimeter")  
            context.actorOf(Props(newAutopilot), "AutoPilot")  
            context.actorOf(Props(new ControlSurfaces(alt)), "ControlSurfaces")  
        }  
    }), "Controls")  
    Await.result(controls ? WaitForStart, 1.second)  
}
```

The `startControls()` method is in charge of creating the `IsolatedResumeSupervisor`, mixing in the `OneForOneStrategyFactory` and then defining the required `childStarter()` method, demanded by the `IsolatedLifeCycleSupervisor` we created at the start. Once this method completes, we have the layout depicted in [Figure 8.8](#).

And now the method that starts our People:

```
def startPeople() {  
    val people = actorOf(Props(new IsolatedStopSupervisor with OneForOneStrategyFactory {  
        def childStarter() {  
            context.actorOf(Props(newPilot), pilotName)  
            context.actorOf(Props(newCoPilot), copilotName)  
        }  
    }), "Pilots")
```

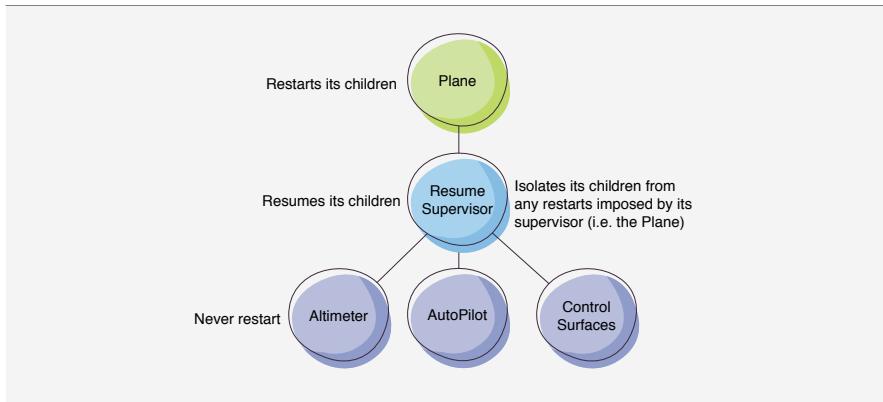


Figure 8.8 · The slice of the Plane’s hierarchy after `startControls()` completes.

```

// Use the default strategy here, which restarts indefinitely
actorOf(Props(newFlightAttendant), attendantName)
Await.result(people ? WaitForStart, 1.second)
}
  
```

Note the slight difference here: we’re starting the `IsolatedStopSupervisor` and adding the Pilots to it, but we want the `LeadFlightAttendant` supervised directly by the `Plane`, so we simply add it to the `Plane`’s list of direct children as we had before. Once this method completes, we have the layout depicted in [Figure 8.9](#).

There! You now have the hierarchy you’re interested in, which provides us with a level of resiliency on which we can build.

But...there’s a problem.

Things Change

The introduction of our supervision nodes has broken some stuff. Look at what the Pilot does when he receives the `ReadyToGo` message from the `Plane`:

```

case ReadyToGo =>
  context.parent ! Plane.GiveMeControl
  copilot = context.actorFor("../" + copilotName)
  autopilot = context.actorFor("../AutoPilot")
  
```

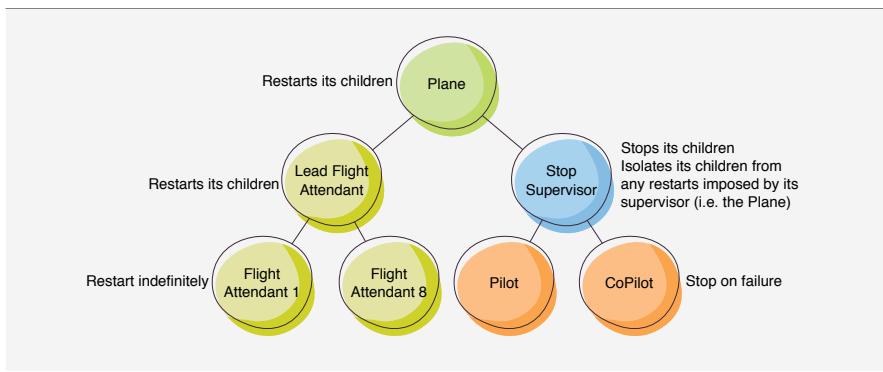


Figure 8.9 · The slice of the Plane’s hierarchy after `startPeople()` completes.

There’s a fair bit there that depended on the original structure of the Actor hierarchy, before we started inserting things and moving them around, and most of that won’t work now. Who’s the parent of the Pilot? Of course, it’s the IsolatedStopSupervisor not the Plane, and if we were to send it the `Plane.GiveMeControl` message, it would throw an exception.

This is where the lack of typing on the Actor comes to bite us. The `context.parent` and `context.actorFor()` return `ActorRefs`, not Planes or CoPilots or any other specific class that we might define, so the compiler won’t be able to catch this misstep. This is *still a good thing* in general, but in situations like these we need to be vigilant about our structure.

Note

It’s best not to rely on an Actor’s hierarchical structure.

That’s not meant to be a hard-and-fast rule, but merely a guideline. Actor programming, just like any other kind of programming with a flexible toolkit such as Akka, presents you with many varied situations. You may find that relying on the Actor’s hierarchical structure is the most amazing choice imaginable in some cases, and in other cases, it paints you into a corner. However, as a general rule, we’ll find that keeping the hierarchy invisible leads to code that is more resilient in the face of change than it would be otherwise.

Note

It's more reasonable for an Actor to depend on the structure beneath it, since it is more influential in that structure than it is in the structure above it.

The structure *above* any Actor is completely outside of its control and it's more brittle for it to make any assumptions about it. The structure below is something that it can generally depend on much better, as is the case with our Plane. The Plane instantiates the IsolatedStopSupervisor and also specifies its direct children. While the children of any given Actor could conceivably alter that structure in a way that exposes brittleness in the Actors above it, this is much less likely and much more in your control as the coder of the Actor in question.

Bypassing the Structure

To reduce any ripple effect of changes to the Actor system's structure, we can employ a couple of facilities. The first is a comfortable friend, *The Dependency Injector*, and the other simply falls into the category of *Message Passing*.

Dependency injection works well when we have one-way dependencies and a uniform method of construction. For example, it's easy to pass the notion of the "parent" to a "child" on that child's construction. It doesn't work so well when you've got a circular dependency between Actors or a construction pattern that simply doesn't lend itself well to a deterministic assignment (for example, it's the Actor itself that decides what it needs, not someone on the outside).

With message passing, we can do things such as:

- Ask a known Actor to do something for us, which it will delegate to an Actor that it already knows, such as the Plane delegating to the LeadFlightAttendant. Essentially, we present the Plane as a facade on top of the LeadFlightAttendant and hide the LeadFlightAttendant from the world.
- Request a reference to an existing Actor, such as asking the Plane for the ControlSurfaces.
- Get a reference to an Actor without asking. We don't have an example of this at the moment, so you might imagine the connection of a client

to your application via WebSocket where that WebSocket is represented by an Actor. Thus, your Actor may receive a `WebSocketConnected(webSocketActor)` message indicating the event has occurred.

Injecting into the Pilot

There are four elements of our Plane that we can specify nicely within the Pilot's constructor:

```
class Pilot(plane: ActorRef,  
           autopilot: ActorRef,  
           var controls: ActorRef,  
           altimeter: ActorRef) extends Actor {
```

This alteration to the Pilot constructor is simple dependency injection. Dependency injection, a very common idiom, is easy to do, read, and understand. Use it when you can.

- Passing in the Plane eliminates the need to access `context.parent` directly, which frees up the Plane to impose any intermediaries between itself and the Pilot that it sees fit.
- The Altimeter and the AutoPilot won't change during the Plane's life cycle and can thus be passed in as well.
- We keep the ControlSurfaces as a `var`, since the Pilot could give up control of the Plane at a later time, but it's the Pilot that's in charge of the Plane initially, so we let the Plane give him control at startup.

Given this, we can now make our changes to the Plane, which will now construct the Pilot correctly:

```
// Helps us look up Actors within the "Controls" Supervisor  
def actorForControls(name: String) = actorFor("Controls/" + name)  
  
def startPeople() {  
    val plane = self  
    // Note how we depend on the Actor structure beneath  
    // us here by using actorFor(). This should be  
    // resilient to change, since we'll probably be the
```

```
// ones making the changes
val controls = actorForControls("ControlSurfaces")
val autopilot = actorForControls("AutoPilot")
val altimeter = actorForControls("Altimeter")
val people = actorOf(Props(new IsolatedStopSupervisor
                           with OneForOneStrategyFactory {
    def childStarter() {
        context.actorOf(Props(newCoPilot(plane, autopilot, altimeter)),
                      copilotName)
        context.actorOf(Props(newPilot(plane, autopilot,
                                      controls, altimeter)),
                      pilotName)
    }
}), "Pilots")
// Use the default strategy here, which
// restarts indefinitely
actorOf(Props(newFlightAttendant), attendantName)
Await.result(people ? WaitForStart, 1.second)
})
```

Of course, this requires changes to the definition of the PilotProvider that provides the newPilot() method. It no longer takes zero arguments and we need to compensate for that, but those changes are simple enough that we won't include them here.

Great! Now our Pilot is isolated from the Plane's structure; except for access to its sibling Actor, the CoPilot as can be seen by the code that still remains in the ReadyToGo handler:

```
case ReadyToGo =>
    copilot = context.actorFor("../" + copilotName)
```

We'll leave this the way it is. Why? There are a few things to consider here:

- The Plane could give the CoPilot to the Pilot at construction time, but the CoPilot needs access to the Pilot as well; we have a circular relationship here. So we can do it for the Pilot, but not the CoPilot. You can make the argument that symmetry is better.

- We could have the Plane send the CoPilot to the Pilot in a later message. For example, we could actually *replace* the ReadyToGo message with a HereIsYourCoPilot(...) message. You could argue that such a message creates a more brittle relationship between the Plane and its Pilots, i.e., the Plane now *knows* that the Pilot wants this reference.
- The Pilot could ask the Plane for the reference to the CoPilot. This would arguably be better, but it puts more work into the Pilot that already exists in Akka itself. Akka already has a lookup facility, which we're using here, and while it's more brittle to do it this way than to use the "stable" Plane reference, this is still not bad.
- Is it really a huge problem that the Pilot expects its CoPilot to be a sibling? This is a judgment call that you'll have to make, which will depend on your circumstances. In this case, we'll say that it's not a big deal. If it proves to be a problem later, we can always refactor.

Starting the Plane

The last thing we have to do is start up the Plane. This will require some fairly obvious changes to the preStart() method.

```
// Helps us look up Actors within the "Pilots" Supervisor
def actorForPilots(name: String) = actorFor("Pilots/" + name)

override def preStart() {
    // Get our children going. Order is important here.
    startControls()
    startPeople()
    // Bootstrap the system
    actorForControls("Altimeter") ! EventSource.RegisterListener(self)
    actorForPilots(pilotName) ! Pilots.ReadyToGo
    actorForPilots(copilotName) ! Pilots.ReadyToGo
}
```

Our Plane is now complete. The structure is set and the children are supervised the way we want them to be. Our refactorings have provided a structure that is both resilient to failure and resilient to change. If you're accustomed to doing some sort of happy dance, now would be the time.

One Strategy

The existence of the `IsolatedLifeCycleSupervisor` and the two separate instances we created in the `Plane` underscores the important point that there is only one strategy that can be applied to a supervisor's children. As usual, there is a very solid reason for this.

Note

The key to a solid fault-tolerant system that naturally handles failure is simplicity. You need to strive for simple Actors and simple supervisors. Generally, if you have the former, you can make do with the defaults for the latter.

What You Don't Know

I'm sure you noticed that you're not getting any information about the Actor in the strategy assessment block, right? Akka only tells you *what* happened not *to whom it happened*. So when you're making your choices about what to do, you have to do it without the knowledge of who threw the Exception.

Are you thinking *clever things*, dear reader? You are, aren't you? I can tell.³ You're thinking back to the refactoring we did when we introduced a level of supervision to our `Plane` and you're remembering the power of pattern matching. The flexibility of pattern matching and the coolness of the supervisor strategy has you thinking along these lines:

“Hold on there! Why couldn't I just keep all of the Actors as direct children of the `Plane` and then use some clever pattern matching to figure out what happened and to whom? Wouldn't that be simpler, not to mention cooler?”

In a word, “No.” In a few words, “Go stand in the corner.” Such thinking takes you away from the Akka paradigm and down that path where bugs lie. Creepy, crawly bugs with hairy legs, fangs, and bad attitudes.

Clever (a.k.a. Bad) Pattern Matching

We could solve this problem by throwing specially worded exceptions for the two scenarios:

³The natives of the village from which I hail believe that I'm a witch.

```
class Altimeter extends Actor {  
    ...  
    def receive = {  
        ...  
        case Tick =>  
            try {  
                // do some math  
            } catch {  
                case e: ArithmeticException =>  
                    throw new ArithmeticException(  
                        "Altimeter: " + e.getMessage)  
            }  
    }  
}  
  
class FlightAttendant extends Actor {  
    ...  
    def receive = {  
        ...  
        case GetDrink(...) =>  
            try {  
                // do some math  
            } catch {  
                case e: ArithmeticException =>  
                    throw new ArithmeticException(  
                        "FlightAttendant: " + e.getMessage)  
            }  
    }  
}
```

There. We could have created specific exceptions for each case, but we're doing this "on the cheap" here; were we going the "expensive" route, we would have been happy putting in the layer of Supervisors. So, we'll just create the exceptions again, but prefix with a string.

Now we can be clever.

```
class SomeActor extends Actor {  
    override val supervisorStrategy = OneForOneStrategy(  
        maxNrOfRetries = 3,
```

```
        withinTimeRange = 30 seconds) {  
    // Resume operation of the Actor when it can't do math  
    case e: ArithmeticException =>  
        if (a.getMessage.startsWith("Altimeter"))  
            Restart  
        else  
            Resume  
    }  
    ...  
}
```

Awesome, no? No, it's not awesome. This sort of thing won't scale as you get more complex structures and hierarchies. First, we've had to toss try/catch blocks in our Actors, which is exactly what we'd like to avoid. As we modify our code and more possible exceptions exist, what happens when we miss a try/catch? Yeah, bad things. The same can be said for our if statement in the Decider; we have an else there, so what happens when we see an unexpected ArithmeticException that's prefixed with "RestartMe!!"?

But more than that, this becomes more and more difficult to *reason about*, and that's a very bad thing. Akka's design makes systems concurrent, reliable, and *reasonable*. The "one strategy" concept is a good one because it ensures that your applications are reasonable and that people other than yourself can reason about them too.⁴

The Error Kernel Pattern

We're about to move on from Supervision, but we can't do that without illustrating one of its most important patterns, *The Error Kernel*. When Actors carry any private data, which is important to presenting the Actor to the outside world (e.g., it's more than just an opportunistic cache variable), then that data needs to be protected.

In a nutshell, this means that we don't want that Actor to restart. The Actor that holds the precious data is protected such that any risky operations are relegated to a slave Actor who, if restarted, only causes good things to happen.

⁴The road to incomprehensible code is paved with the pink slips of clever programmers.

The Error Kernel Pattern implies pushing levels of risk further down the tree as illustrated in [Figure 8.10](#).

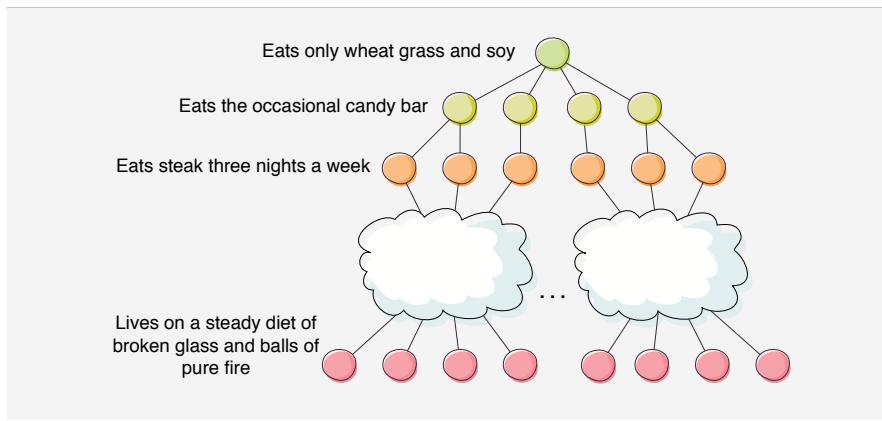


Figure 8.10 · Risk is generally pushed down the tree. The higher up you go, the less risk you're willing to take on, and the lower you go, the more risky you become. Guys that live at the bottom are the ones you want to send on the brutal missions where they're not guaranteed to survive, but the guys at the top don't get their hands dirty at all.

The Plane is our ultimate Error Kernel. If the Plane were to restart, then that would basically mean that the entire application would restart, which would be pretty weird indeed! We need to avoid this.

The Risky Altimeter

One classic component in our Plane that we need to keep safe is the Altimeter; this is because it's carrying mutable data that is critical to its behaviour. If the Altimeter ever restarted from scratch, we'd see some very strange behaviour:

- The current altitude would suddenly be 0.
- The current rate of climb would also suddenly be 0.
- The tick timing would be somewhat out-of-sync with reality, but that might not be all that visible.

- There would be no registered listeners. Nobody would get any altitude events.

Now, we've protected ourselves from these problems in two ways:

1. We've put the Altimeter under a Resume Supervisor, so it can't be restarted anyway.
2. We've made sure that nothing weird will happen inside of its running code by keeping things simple.

But what if we hadn't done these things? What if we put the Altimeter under the default supervision, which would restart it on failure? What if we also made a pretty brutal error in the math, such that an `ArithmaticException` would be thrown?

Right now we're calculating the current altitude with:

```
altitude = altitude + ((tick - lastTick) / 60000.0) * rateOfClimb
```

The only risky value in there will be `rateOfClimb`, since it is the only variable that the outside can specify. Let's say we changed the calculation to something irretrievably silly:

```
roc = (rateOfClimb * rateOfClimb) / rateOfClimb
altitude = altitude + ((tick - lastTick) / 60000.0) * roc
```

Now, every time someone levels the rate of climb (i.e., specifying 0.0f), the calculation of `roc` will throw an `ArithmaticException`, which would cause the Altimeter to restart when placed under a supervisor that performs restarts. Not good.

One solution to this problem is to delegate the risky behaviour to another child Actor, which is either devoid of any mutable data itself or whose mutable data is irrelevant. We could do this by providing a child Actor in the Altimeter along with using some messages to talk to it:

```
case class CalculateAltitude(lastTick: Long, tick: Long, roc: Double)
case class AltitudeCalculated(altitude: Double)

val altitudeCalculator = context.actorOf(Props(new Actor {
  def receive = {
```

```
    case CalculateAltitude(lastTick, tick, roc) =>
      if (roc == 0) throw new Exception("Divide by zero")
      val alt = ((tick - lastTick) / 60000.0) * (roc * roc) / roc
      sender ! AltitudeCalculated(alt)
    }
}, "AltitudeCalculator")
```

We can then modify the Altimeter itself to delegate the work to the calculator.

```
def receive = {
  case CalculateAltitude(lastTick, tick, roc) =>
    if (roc == 0) throw new Exception("Divide by zero")
    val alt = ((tick - lastTick) / 60000.0) * (roc * roc) / roc
    sender ! AltitudeCalculated(alt)
}
```

Note that we aren't storing anything in the altitudeCalculator. The data is still stored in the Altimeter as it always was; we've just moved the actual calculation down to an Actor that can fail just fine.

Note

The Error Kernel Pattern keeps data closer to the root of the hierarchy, moving behaviour (especially “risky” behaviour) down to the leaves.

It's also quite subtle! The alterations made to receive's partial function are rather crucial. Note how the lastTick is updated; we assume success and thus update it immediately after sending the request to the calculator. Had we updated this after receiving the response, it may not get updated for *hours* due to exceptions being thrown from the calculator. This means that once the calculator could finally calculate something, it would be applied over a period of those hours, not the last 100 milliseconds. We need to state this again:

Note

Reliable and reasonable supervision strategies require simplicity. The hierarchical nature of the Actor model allows you to create simple structures at any level you need. If you need to simplify, then you have all the power you need to do that at your fingertips.

Failure to keep your Actors simple and easy to reason about will eventually lead to pain and frustration when the use cases start piling up.

8.6 Dead Pilots

Hey, it happens. You're having a great day, the sun is shining, and the wind is whipping by at a few hundred miles an hour, and *boom!*, your pilot's life of drinking, smoking, and a high cholesterol diet catches up with him and he ups and dies.

Assuming you're a copilot, someone's probably going to expect you to do something about it, right? Our copilot certainly can do something about it, but she has to know when. Enter *DeathWatch*.

DeathWatch is distinct from supervision in that supervision has an active effect on the Actor that it's supervising. In *DeathWatch*, we're interested in knowing when an Actor ceases to be and becomes an *ex-Actor*. This is a perfect mechanism for our copilot to use in order to be notified if/when her superior croaks.

Let's set this up when the copilot is given the signal that everything is `ReadyToGo`:

```
case ReadyToGo =>
    pilot = context.actorFor("../" + pilotName)
    context.watch(pilot)
```

If the Pilot dies, we now know that the CoPilot will get a `Terminated` message, which we can receive in the usual way. When the CoPilot sees the Pilot die, she needs to take over control of the Plane:

```
case Terminated(_) =>
    // Pilot died
    plane ! GiveMeControl
```

In this case, we don't care what the `ActorRef` is that was specified in the `Terminated` message because we're only watching one guy, but if we were watching multiple Actors, then we'd need to deal with the parameter that the `Terminated` message specified.

It's really as simple as that. Akka allows any Actor to put a *DeathWatch* on any other Actor, regardless of where it is in the hierarchy. This is

clearly different from supervision, and it certainly needs to be. Many Actors throughout the hierarchy may be interested in the life cycle of those Actors on which it depends but they aren't in charge of supervising their life cycles.

Testing DeathWatch

So, how do we test that the CoPilot does the right thing once the Pilot finally kicks the bucket? Our Actor system's hierarchical nature, coupled with the fact that Actors are essentially untyped, lets us test the behaviour of our CoPilot with relative ease.

There are two ways we could test this:

1. Go in through the back door and simply call `coPilot.receive(Terminated(_))` and ensure that it emits the `GiveMeControl` message.
2. Go in through the front door; set up a hierarchy and run the CoPilot through its paces from establishing DeathWatch to responding to the `Terminated` event.

This is a choice that you'll have to make for yourself, when it comes to testing your own code. My personal choice would be to go through the front door since I'm interested in the whole thing hanging together properly.

To do this, we need to set up a hierarchy that the CoPilot will be happy with—namely, it needs a sibling Pilot—and then put the CoPilot into the right state.

This is one of those times where our choice to let the Actors assume a particular hierarchical relationship (that of siblings, in this case) affects how our tests are built. Had we passed in the Pilot, or let the CoPilot ask the Plane directly for the Pilot, we could structure our test differently. However, we chose to use the Actor hierarchy directly, and thus we must satisfy the CoPilot with that hierarchy. [Figure 8.11](#) shows what we're going to do.

We don't need the Plane, instruments, or controls; we just need enough to keep the CoPilot happy in what it needs to do for us.

To write this, we'll need a fair bit of help from Akka and ScalaTest:

```
import akka.actor.{ActorSystem, Actor, ActorRef, Props}
import scala.concurrent.Await
import akka.pattern.ask
import akka.testkit.{TestKit, ImplicitSender}
```

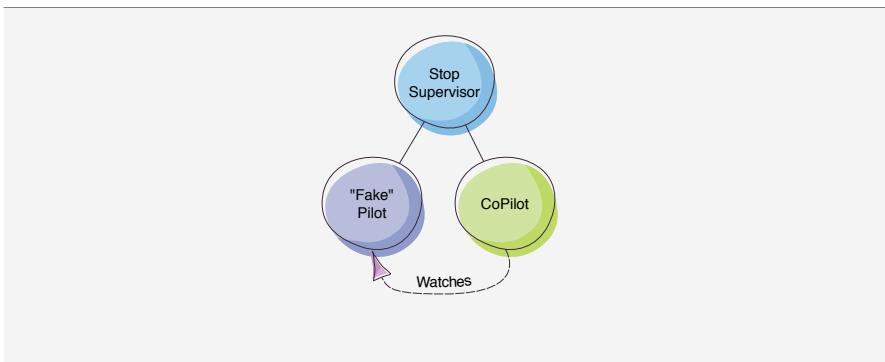


Figure 8.11 · The simple hierarchy we need in order to test the CoPilot

```
import scala.concurrent.util.duration._  
import akka.util.Timeout  
import com.typesafe.config.ConfigFactory  
import org.scalatest.WordSpec  
import org.scalatest.matchers.MustMatchers
```

The “Fake” Pilot is what we’ll create to stand in for the real thing, and its sole purpose will be to die. The fact that we’ll have the IsolatedStopSupervisor as its parent ensures that its death will be immediate.

```
class FakePilot extends Actor {  
    override def receive = {  
        case _ =>  
            throw new Exception("This exception is expected.")  
    }  
}
```

We’ll also need a couple of ActorRefs around to keep the CoPilot’s constructor happy (i.e., the AutoPilot and Altimeter instances). For this, we’ll use a simple NilActor:

```
class NilActor extends Actor {  
    def receive = {  
        case _ =>  
    }  
}
```

Next, we'll need a bit of configuration. The CoPilot uses the configuration system to look up the name of the Pilot so that it can do its call to `actorFor()`. To keep things isolated to the test environment, we'll slide a specific configuration into the test's ActorSystem using this configuration.

```
object PilotsSpec {
    val copilotName = "Mary"
    val pilotName = "Mark"
    val configStr = s"""
        zzz.akka.avionics.flightcrew.copilotName = "$copilotName"
        zzz.akka.avionics.flightcrew.pilotName = "$pilotName"""""
}
```

We can now construct the test specification by including the appropriate framework information and specifying our configuration.

```
class PilotsSpec extends TestKit(ActorSystem("PilotsSpec",
    ConfigFactory.parseString(PilotsSpec.configStr)))
    with ImplicitSender
    with WordSpec
    with MustMatchers {
    import PilotsSpec._
    import Plane._

    ...
}
```

It will make things clearer if we have a few simple helper methods and values:

```
// Helper to make the NilActor easier to create
def nilActor = system.actorOf(Props[NilActor])

// These paths are going to prove useful
val pilotPath = s"/user/TestPilots/$pilotName"
val copilotPath = s"/user/TestPilots/$copilotName"
```

Now we need to construct the hierarchy of [Figure 8.11](#) with the Supervisor, the fake Pilot, and the CoPilot.

```
// Helper function to construct the hierarchy we need
// and ensure that the children are good to go by the
```

```
// time we're done
def pilotsReadyToGo(): ActorRef = {
    // The 'ask' below needs a timeout value
    implicit val askTimeout = Timeout(4.seconds)

    // Much like the creation we're using in the Plane
    val a = system.actorOf(Props(new IsolatedStopSupervisor
        with OneForOneStrategyFactory {

        def childStarter() {
            context.actorOf(Props[FakePilot], pilotName)
            context.actorOf(Props(new CoPilot(testActor, nilActor,
                nilActor)), copilotName)
        }
    )), "TestPilots")

    // Wait for the mailboxes to be up and running for the children
    Await.result(a ? IsolatedLifeCycleSupervisor.WaitForStart, 3.seconds)

    // Tell the CoPilot that it's ready to go
    system.actorFor(copilotPath) ! Pilots.ReadyToGo
    a
})
```

Note that we're using the `IsolatedLifeCycleSupervisor`'s `WaitForStart` message here. The asynchronous startup of the Supervisor and its children is too fast for us to start working with it immediately. We need to be sure that the Mailboxes are ready to go before we start killing things off. Using the `Await.result()` call allows us to block the current thread for a short time until things are ready to go. It's also important to note how we've constructed the CoPilot. The first parameter to its constructor is what it will use as the Plane, and we want the test to be able to be that Plane, so we pass in the `testActor`, which coupled with the declaration of the `ImplicitSender` ensures that we'll see the `GiveMeControl` message. The other two parameters are `NilActors`, since we don't care about those.

We're now ready to actually run the test, which includes making the call to `pilotsReadyToGo()`, killing the fake Pilot, and ensuring that the CoPilot asks the Plane for the controls:

```
// The Test code
"CoPilot" should {
```

```
"take control when the Pilot dies" in {
    pilotsReadyToGo()
    // Kill the Pilot
    system.actorFor(pilotPath) ! "throw"
    // Since the test class is the "Plane" we can
    // expect to see this request
    expectMsg(GiveMeControl)
    // The girl who sent it had better be Mary
    lastSender must be (system.actorFor(copilotPath))
}
}
```

And we're done! If you run this test, you should see that the CoPilot does indeed take control of the Plane when she sees the Pilot die, saving the passengers from death by a blazing aluminum inferno. Sweet.

Extra Credit

What if the CoPilot dies? It would be nice if the AutoPilot had a heart monitor attached to the CoPilot and would take over control of the Plane when it happened to notice that the CoPilot was dead, wouldn't it?

You have all the knowledge you need at this point to do exactly that, but it's not exactly as straightforward as we've seen to this point. To accomplish this feat, you must:

- Refactor the AutoPilot's constructor so that it accepts the Plane as a parameter. This isolates it from knowing the structure above it, as we've seen with the Pilot's refactoring already.
 - This will require altering the PilotProvider as well as the instantiation call in the Plane itself.
- Obtaining the reference to the CoPilot in the AutoPilot itself. This is more involved than we've seen already.
 1. The AutoPilot is not a sibling of the CoPilot.
 2. The CoPilot is also instantiated *after* the AutoPilot, so you can't pass it into the AutoPilot's constructor.

3. You could use `context.actorFor("../Pilots/<copilot's name>")`, but that's ill-advised due to the brittleness of the hierarchical knowledge.
 4. You could have the Plane give the CoPilot to the AutoPilot after the CoPilot has been constructed, which might be just fine.
 5. Or you can send the `ReadyToGo` message to the AutoPilot just as you did with the Pilot's, and then let the AutoPilot ask the Plane for the CoPilot's reference.
 - Of course, this requires the Plane to expose a new request/response message pair, such as `RequestCoPilot(...)` and `CoPilotReference(...)`, but this is arguably the most robust solution.
- Putting the DeathWatch on the CoPilot.
 - Asking the Plane for the controls when it sees the CoPilot die.

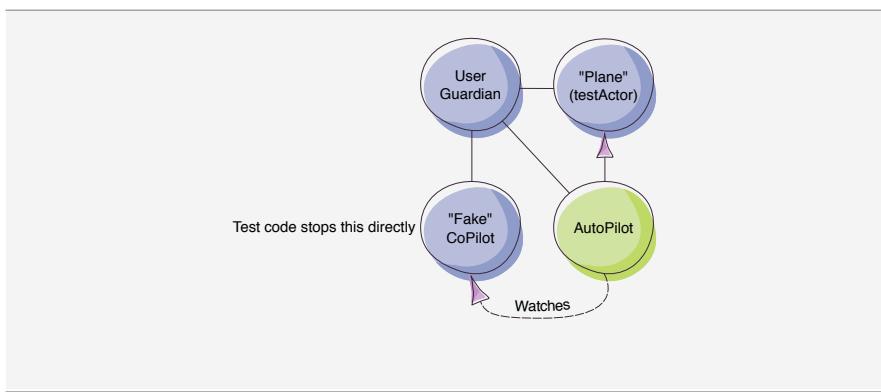


Figure 8.12 · The AutoPilot's test doesn't need to have any special parenting. Its parent can simply be the test ActorSystem's User Guardian.

To ensure that you've done everything right, you're definitely going to want to test this beast. Assuming you've gone for [option 5](#) above, you'll have to set up some sort of Actor structure, but the structure need not be as restrictive as it was with the CoPilot's test, due to the fact that the AutoPilot doesn't need the CoPilot to be a sibling. [Figure 8.12](#) shows you the idea behind what you could do here.

The AutoPilot simply needs to be constructed such that it thinks the `testActor` is the Plane, and when the AutoPilot then asks for the reference to the CoPilot, your test code can send it a fake CoPilot, which you will then immediately stop (via `system.stop(fakeCoPilot)`). At that point, the rest of the test should write itself, much like it was written to test the CoPilot.

8.7 Chapter Summary

Supervision and DeathWatch are core aspects of the Actor paradigm in Akka and provide the foundation for building reliable, fault-tolerant applications. These concepts seem simple initially, but certain nuances of hierarchy and complexity require you to think through your strategy with care. We will continue to explore these topics as we progress, so you'll be getting more experience with them and your comfort level will increase.

So far we've covered some pretty decent ground. You've learned the basics of how to:

- Set up supervision hierarchies.
- View the existing hierarchies with respect to the `ActorSystem` and the paths.
- Specify strategies for supervision.
- Understand the recursive nature of how supervision operates.
- Establish a DeathWatch on another Actor that is not a direct child or parent of the Actor holding the watch.
- React to the termination of a watched Actor for the purposes of taking corrective action in the application to help heal it.
- Use some important techniques for composing your Actor hierarchies while planning for structural resiliency in the face of change.

These sorts of concepts form the foundation for applications that boast incredible levels of availability. When your system incorporates unexpected behaviour into its design, it then has a much more deterministic set of behaviours in the face of that aberrant behaviour. When this is stretched out

across physical machines in an Actor network, your application's fault tolerance as a whole becomes formidable.

From a design perspective, with respect to this new paradigm, the key message you need to gain from this chapter is *simplicity*. Keep the complexity level of your Actors to the absolute minimum, because it's this simplicity that enables a *naturally* fault-tolerant system; one that can heal itself under many circumstances and can be *reasoned about* without having to understand the whole world first. If you've grasped that idea and have recognized the power behind the system's hierarchical structure, then you're well on your way to becoming a solid Actor programmer.

Chapter 9

Being Stateful

If you're a coffee drinker, then it's quite possible that you have a very personal understanding of behavioural state. I once dated a girl whose mother was the most serious caffeine addict I'd ever seen. Every trip in the car required a set of scheduled stops to Canada's staple coffee shop, so she could load up on an extra-large double double. Failure to do so resulted in a state that was... well, that girl and I didn't go out for very long.

From an Actor perspective, if she didn't get a `Coffee(size, coffeeType)` message on a regular basis, she would become(`intolerable`). Akka lets you change an Actor's running behavioural state during message processing, which can change the way it reacts to future messages in any way you see fit.

In this chapter, we'll explore the Actor features that provide us with this ability and apply them to our Plane. We'll explore the ins and outs of the Actor's internal `become()` and `unbecome()` methods as well as the special-purpose Finite State Machine (FSM) Actor, both of which are geared toward making your Actors *stateful*.

9.1 Changing Behaviour

When we talk about being stateful, we're not talking about *data*. Our industry (or at least Enterprise and Cloud industry) has muddled the term *state* over the years and now it appears to simply mean *data*, depending on who you talk to and how much coffee they've had. This isn't too surprising since the contextual data possessed by an entity and its behaviour tend to be tightly coupled, but I'd like to ensure that our focus is on the *behaviour* side of state-

fulness, which is supported by data, if need be.

The primary way that Akka lets us modify our behaviour is by simply using the ActorContext's become() and unbecome() methods:

```
def expectHello: Receive = {  
    case Hello(greeting) =>  
        sender ! Hello(greeting + " to you too!")  
        context.become(expectGoodbye)  
    case Goodbye(_) =>  
        sender ! "Huh? Who are you?"  
}  
  
def expectGoodbye: Receive = {  
    case Hello(_) =>  
        sender ! "We've already done that."  
    case Goodbye(_) =>  
        sender ! Goodbye("So long, dood!")  
        context.become(expectHello)  
}  
  
def receive = expectHello
```

By exercising the code, we can see what it does:

```
// prints: Huh? Who are you?  
println(Await.result(actor ? Goodbye("So long"), 1.second))  
  
// prints: Hello(Hithere to you too!)  
println(Await.result(actor ? Hello("Hithere"), 1.second))  
  
// prints: We've already done that.  
println(Await.result(actor ? Hello("Hithere again"), 1.second))  
  
// prints: Goodbye(So long, dood!)  
println(Await.result(actor ? Goodbye("So long"), 1.second))
```

Depending on the Actor's current behavioural state, the Hello or Goodbye message may trigger a state change, which we affect by replacing the instance of the receive partial function using the ActorContext's become() method. Again, this is all very thread-safe because we're all warm and cozy inside the Actor's single-threaded world. When the next message comes in for the receive body to process, our new behaviour will be observed.

Note

It doesn't matter at what point in the message handler you invoke `become()`, since all it does is replace the instance of `receive`. The behaviour is realized when the next message is processed, not during the processing of the current message. We simply use a convention where the call to `become()` is at, or near, the end of the handler, but there are as many reasons why you wouldn't do it this way as why you would.

Stacking States

Every time we `become()`, we have the option of pushing our current behaviour onto the context stack. This is what allows us to `unbecome()` later. By default, the `become()` method does not push your current state into the stack, thus making a subsequent `unbecome()` impossible.

It turns out that the cases that require the use of `unbecome()` can actually be quite rare, and we can bucket them into three general use cases:

- You're just shifting back and forth between a couple of states; e.g., `databaseOnline()` and `databaseOffline()`. However, this can be very problematic and better implemented without `unbecome()`, since doing so requires that you understand what state you were in before. For example, if you `unbecome()` from the `databaseOffline()` state, then it's presumed that there was a previous state; that is, `databaseOnline()`. What if that's not the case? If you want the state to be `databaseOnline()`, then just `become()` that.
- You have a decently complex system whereby the current state could have been reached from several different previous states. You need to move back to the previous state, but you don't know what it is, so moving "forward" to that previous state is very problematic. The simplest thing to do here is to `unbecome()`.
- You've added data to your states. In this case, the previous state has data locked up in it and the only way to retrieve it is to move back to that state, e.g., `pendingCommit(commitData)` moves to `committing(commitData, transactionId)`. Assuming the result of attempting to push the commit forward in the `committing` state fails, the `unbecome()` can trigger a retry to a subsequent `committing` state with a fresh `transactionId`.

The state stack can be very useful, especially in the latter two cases and you should feel empowered to use it as liberally as you like. With that said, using the state stack can make things more difficult to reason about in your code because of the simple fact that `become()` can be a very static declaration (although it need not be, of course), whereby `unbecome()` is entirely dynamic in nature.

The static nature of `become()` is obvious in the following code:

```
def needsBootstrapping: Receive = {
    case Bootstrap(...) =>
        become(bootstrapped())
}

def bootstrapped: Receive = {
    case PrepareForTransfer(metadata) =>
        become(transferReady(metadata))
}

def transferReady(metadata: Map[String, String]): Receive = {
    case Bytes(bytes) =>
        become(transferInProgress(Vector(bytes)))
}

def transferInProgress(metadata: Map[String, String],
                      data: Vector[Vector[Byte]]): Receive = {
    case Bytes(bytes) =>
        become(transferInProgress(metadata, data :+ bytes))
    case EOF =>
        destination ! NewDataArrived(metadata, data)
        become(bootstrapped())
}
```

Figure 9.1 depicts the nature of the state transitions from above. The state transitions are clear and obvious because the code is written to statically move between them. It would be less obvious were we to store the next state in a variable and then `become()` that variable's value. Generally speaking, there's never a reason to be this confusing in your code, so don't do it.

When we use `unbecome()`, of course, there's no such option. You can't statically “`unbecome`” to a specific state; you must “`unbecome`” to the state that is currently on top of the stack, which is only known at runtime (assuming there is such a state to which you can `unbecome()`).

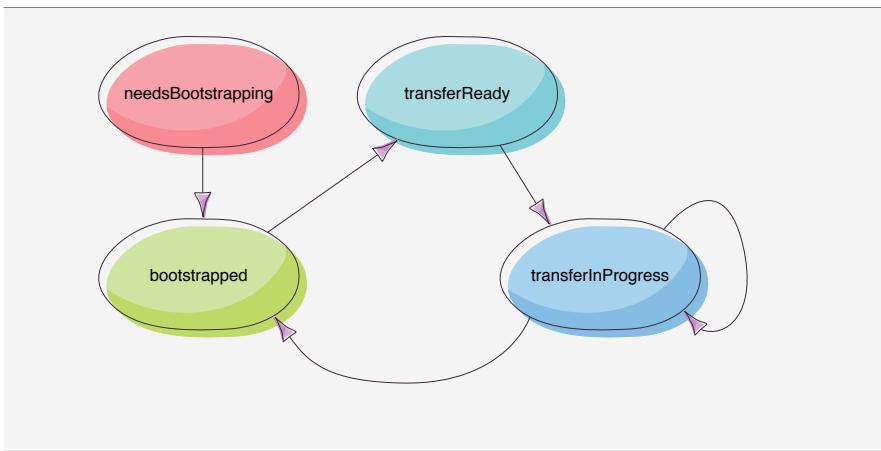


Figure 9.1 · The state transitions are easily deduced from the code due to the static usage of the `become()` method.

If we don't need to use the state stack, then we can read our code as constantly moving *forward* through its states. This is simple because the next state is always visible when using `become(next state)`. When we use `unbecome()`, the code can't tell us where it will go because that information is deduced at runtime.

The fact that there's a stack involved means that you also have another data structure that you need to take care of. This means that you're performing a series of pushes, and if the number of pushes is greater than the number of pops over a long period of time, then you have a problem. For example, if we change the Hello/Goodbye code to:

```
def expectHello: Receive = {
    case Hello(greeting) =>
        // ...
        context.become(expectGoodbye, discardOld = false)
}
def expectGoodbye: Receive = {
    case Goodbye(_) =>
        // ...
        context.become(expectHello, discardOld = false)
}
```

```
def receive = expectHello
```

Then we get the problem depicted in [Figure 9.2](#). This particular bug manifestation is stupefyingly obvious, but the problem can crop up in much more subtle ways as your code complexity increases. If you have several points where you stack the old behaviour during a push and don't have enough paired `unbecome()`s, then you'll slowly start filling up this stack, and that will be a problem.

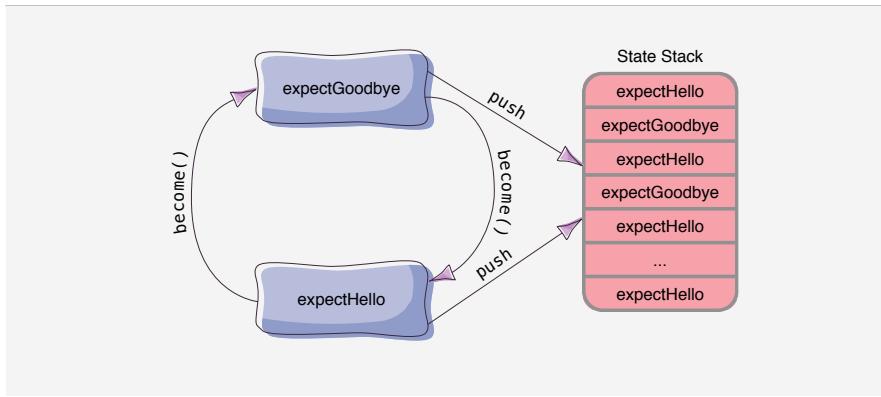


Figure 9.2 · Our state stack's condition after a whole load of `become()`s without a single `unbecome()`. If you're not careful in your own (more complex) code, you could end up with a stack that eventually blows up. You just need to accumulate more pushes than pops over a long enough period of time and you have the exact same problem.

None of this discussion should discourage you from using the state stack; if the stack works for your situation then you should, by all means, use it. But above all else, we want simplicity in our Actors and when using the stack negates that simplicity, you might want to opt for some other mechanism, if possible.

And thus, we return to our recurring theme: in general, our Akka code should be easy to reason about. One way we can achieve this is by continually moving *forward* through our states by using `become()` and avoiding the stack. When this method becomes unreasonable and using the stack increases the code's clarity and understandability, then you should definitely employ it.

9.2 The Stateful Flight Attendant

Our `FlightAttendant` has been pretty simple so far, and we'd like to change that by giving it a few more messages to process and also give it some state so that it's doing something practical.

When the `FlightAttendant` is currently getting a drink for a passenger, it doesn't make much sense to get another one, considering that there are more than a few `FlightAttendants` available. We'll add a couple of different behavioural message handlers to our `FlightAttendant` that facilitate this, as well as provide a mechanism for dealing with passenger emergencies.

We begin by defining a few more messages in the `FlightAttendant`'s companion object.

```
object FlightAttendant {  
    ...  
    case class Assist(passenger: ActorRef)  
    case object Busy_?  
    case object Yes  
    case object No  
    ...  
}
```

We'll be using these newly defined messages in the changes we'll make to the `FlightAttendant`'s message handling structure below.

```
class FlightAttendant extends Actor { this: AttendantResponsiveness =>  
    import FlightAttendant._  
  
    // An internal message we can use to signal that drink  
    // delivery can take place  
    case class DeliverDrink(drink: Drink)  
  
    // Stores our timer, which is an instance of 'Cancellable'  
    var pendingDelivery: Option[Cancellable] = None  
  
    // Just makes scheduling a delivery a bit simpler  
    def scheduleDelivery(drinkname: String): Cancellable = {  
        context.system.scheduler.scheduleOnce(responseDuration,  
            self,  
            DeliverDrink(Drink(drinkname)))  
    }  
}
```

```
// If we have an injured passenger, then we need to immediately assist
// them, by giving them the 'secret' Magic Healing Potion that's
// available on all flights in and out of Xanadu
def assistInjuredPassenger: Receive = {
    case Assist(passenger) =>
        // It's an emergency... stop what we're doing and assist NOW
        pendingDelivery foreach { _.cancel() }
        pendingDelivery = None
        passenger ! Drink("Magic Healing Potion")
}

// This general handler is responsible for servicing drink requests
// when we're not busy servicing an existing request
def handleDrinkRequests: Receive = {
    case GetDrink(drinkname) =>
        pendingDelivery = Some(scheduleDelivery(drinkname))
        // Become something new
        context.become(assistInjuredPassengerorElse
            handleSpecificPerson(sender))

    case Busy_? =>
        sender ! No
}

// When we are already busy getting a drink for someone then we
// move to this state
def handleSpecificPerson(person: ActorRef): Receive = {
    case GetDrink(drinkname) if sender == person =>
        pendingDelivery foreach { _.cancel() }
        pendingDelivery = Some(scheduleDelivery(drinkname))
        // The only time we can get the DeliverDrink message is when we're in
        // this state
    case DeliverDrink(drink) =>
        person ! drink
        pendingDelivery = None
        // Become something new
        context.become(assistInjuredPassengerorElse handleDrinkRequests)
        // If we get another drink request when we're already handling one
        // then we punt that back to our parent (the LeadFlightAttendant)
    case m: GetDrink =>
```

```

    context.parent forward m
  case Busy_? =>
    sender ! Yes
}
// Set up the initial handler
def receive = assistInjuredPassenger orElse handleDrinkRequests
}

```

The goal of our new FlightAttendant is to be more stateful about its current work. We use functional composition and `become()` to change the FlightAttendant's behaviour, depending on whether or not it's currently serving someone.

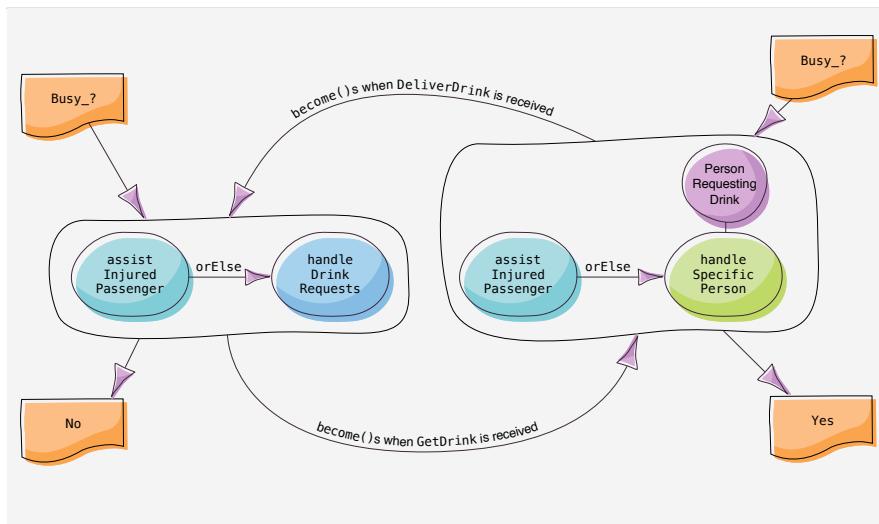


Figure 9.3 · The FlightAttendant changes its behavioural state when certain messages are received in certain states. One nice effect of knowing you are in a particular state is that you can simply hard-code values rather than having to check what state you're in with an `if` statement. Here, we illustrate that the response to the `Busy_?` message is determined by the state in which the FlightAttendant finds itself.

Figure 9.3 illustrates the transition that occurs between these states. When it becomes the state on the right, the person with which the FlightAttendant

is currently working is bound to the receive partial that has been instantiated (`handleSpecificPerson`).

The two states created are composed of the `assistInjuredPassenger` partial and either the `handleDrinkRequests` partial or the `handleSpecificPerson` partial using Scala's `orElse` combinator. Whatever message isn't handled by the `assistInjuredPassenger` partial is passed on to the next partial. Because we intend to exercise the `assistInjuredPassenger` partial no matter what, we compose it in both states and let the other vary.

An important thing to realize is the lack of `if` statements in this code. We would often have some sort of variable data in our classes that we would use to understand what state we're in, but with Akka's behavioural swapping, we don't need to do that. For example, we don't need code like this:

```
var drinkPerson: Option[ActorRef] = None  
...  
case GetDrink(drinkname) =>  
    if (drinkPerson.isDefined)  
        // already handling someone  
    else {  
        drinkPerson = Some(sender)  
        // etc...  
    }  
...  
case DeliverDrink(drink) =>  
    drinkPerson foreach { _ ! drink }  
    drinkPerson = None
```

That kind of coding can get old *really* fast. The beautiful thing here is that we don't need to worry about it at all. The interesting event triggers a state change and all of that state-specific code goes into that new state. It makes writing stateful code easier than chewing gum on a Wednesday afternoon.¹

On Composition

The fact that we're using the `orElse` function combinator to combine our behaviours is pretty standard practice, but it does come with some caveats due to the ordering of message processing.

¹Historically, this is the easiest day on which to chew gum.

Watch Out for the Eclipse

When you're combining your receivers, you need to make sure that it's all reachable.

```
def behaviourA: Receive = {  
    case m =>  
        println(m)  
}  
def behaviourB: Receive = {  
    case MessageForB(payload) =>  
        doSomethingAwesomeWith payload  
}  
// Oops!  
def receive = behaviourA orElse behaviourB
```

Using `orElse` to combine your behaviours is a very powerful mechanism, but in the above code, we've got a problem. Scala is great about checking unreachable code in the following case, since it knows that the first match will always succeed, resulting in the second never being reached:

```
def behaviour: Receive = {  
    case m =>  
        println(m)  
    case MessageForB(payload) =>  
        doSomethingAwesomeWith payload  
}  
  
// error: unreachable code  
// case MessageForB(payload) =>  
//     doSomethingAwesomeWith payload
```

Scala cannot realize the same error in this case:

```
def receive = behaviourA orElse behaviourB
```

It will just silently let this construct, resulting in `behaviourB` never being invoked. So, when you're combining your message handlers with `orElse`, ensure that you're not eclipsing your own code.

Watch the Complexity

There's another practice that we also need to be aware of when we combine behaviours in an Actor: keep it simple. It's quite possible to lose your way and start putting too much responsibility into an Actor and you wind up with constructs, such as:

```
become(protocolBehaviourStart orElse
        bluntHttpHandler orElse
        databaseSelectionHandler orElse
        fileSystemLocker)

// later
become(protocolBehaviourSecondStage orElse // changed
        bluntHttpHandler orElse
        databaseSelectionHandler orElse
        fileSystemLocker)

// and later still...
become(protocolBehaviourThirdStage orElse // changed
        activeHttpHandler orElse          // changed
        databaseSelected orElse          // changed
        fileSystemLocker)

// and so on...
```

The more behaviour you add to your Actor and the more aspects you have for each behaviour, the more permutations you have and eventually it just gets *oogie*. The single-responsibility principle applies to Actors just as much as it applies to everything else. If things get to be too much of a problem, you need to refactor your code by breaking out your responsibilities into multiple Actors.

There are times, however, when the inclusion of multiple traits that carry behaviour as well as certain behavioural states in the Actor itself require a requisite amount of complexity. Later, we'll address this issue when we introduce a collection of Akka coding patterns.

9.3 A Better Flyer

Up until now we haven't had a great mechanism for actually *flying* this Plane. The reason we've overlooked this is because we haven't had the right tool

available. Akka's Finite State Machine (FSM) is that tool, and we'll employ it in this section. It will involve some refactoring of our previous code, since we'll move what little behaviour we had for flying into a more robust class that many aspects of our application can use, instead of the ad-hoc mechanism we currently have in the Plane.

Enhancing the Controls

Most planes do more than just go up and down; they tend to be able to go left and right as well. We'll enhance our Plane with the ability to change our heading using the stick so that we have both elevator as well as aileron control. The rudder won't be of any interest to us, but feel free to model it on your own if you'd like.

A Heading Indicator

Just as the Altimeter provides us with our height, we'll need something that tells us into what direction we're headed; we'll model that with the HeadingIndicator. There won't be too much here that's surprising, so let's just plunk it down, starting with the companion object:

```
object HeadingIndicator {  
    // Indicates that something has changed how fast  
    // we're changing direction  
    case class BankChange(amount: Float)  
    // The event published by the HeadingIndicator to  
    // listeners that want to know where we're headed  
    case class HeadingUpdate(heading: Float)  
}
```

You might have noticed that we used the message name "BankChange" instead of "RateChange," as we have on the Altimeter. We did this for good reason and it's something that many might get tripped up by, so if you're getting weary at the moment, go grab a coffee... I'll wait.

You'll recall that the partial function definition for the receive method is `PartialFunction[Any, Unit]`, which should give you a pretty decent hint as to the subtlety. Because the input to the receive method is of type Any, all type checking is lost. Akka gives us the flexibility of sending *any* message to an Actor, which is awesome, but it means we have to be a bit

more diligent about what we send. Neither Akka nor Scala will have any problem with this code:

```
import Altimeter._  
altimeter ! RateChange(0.5f)  
headingIndicator ! RateChange(0.5f)
```

What you'll find is that the HeadingIndicator fails to handle the RateChange message, which will be indicated in the event stream and eventually the log. It will all *look* really good to us because it's a RateChange message; we have it defined on the HeadingIndicator companion object, all is good! The subtle bit we'll miss, in the many more lines of code than the simple illustration above, is that we failed to import HeadingIndicator._. If we do that, then the compiler will give us an error about the conflicting RateChange types, but it can't give us that error if we don't do the import.

To avoid this subtlety, I find it best practice to name things as uniquely as I can, within reason. If I have any concern that some classes or message handlers might intersect on some common messages, then I ensure that there is reasonable uniqueness in those names. Often, due to imports and other scoping issues, we can find these hidden land mines that would normally be visible if they weren't being passed to a function that accepts Any.

OK, got that? Alright, now that we have the basic messages we need, we can create the corresponding Actor:

```
trait HeadingIndicator extends Actor with ActorLogging { this: EventSource =>  
    import HeadingIndicator._  
    import context._  
  
    // Internal message we use to recalculate our heading  
    case object Tick  
  
    // Maximum degrees-per-second that our plane can move  
    val maxDegPerSec = 5  
  
    // Our timer that schedules our updates  
    val ticker = system.scheduler.schedule(100.millis, 100.millis,  
                                           self, Tick)  
  
    // The last tick which we can use to calculate our changes  
    var lastTick: Long = System.currentTimeMillis  
  
    // The current rate of our bank
```

```
var rateOfBank = 0f
// Holds our current direction
var heading = 0f

def headingIndicatorReceive: Receive = {
    // Keeps the rate of change within [-1, 1]
    case BankChange(amount) =>
        rateOfBank = amount.min(1.0f).max(-1.0f)

        // Calculates our heading delta based on the current rate of change,
        // the time delta from our last calculation, and the max degrees
        // per second
    case Tick =>
        val tick = System.currentTimeMillis
        val timeDelta = (tick - lastTick) / 1000f
        val degs = rateOfBank * maxDegPerSec
        heading = (heading + (360 + (timeDelta * degs))) % 360
        lastTick = tick
        // Send the HeadingUpdate event to our listeners
        sendEvent(HeadingUpdate(heading))
    }
}

// Remember that we're mixing in the EventSource and thus have to
// compose our receive partial function accordingly
def receive = eventSourceReceiveorElse headingIndicatorReceive

// Don't forget to cancel our timer when we shut down
override def postStop(): Unit = ticker.cancel
}
```

As promised, there's not a lot that's special there. If you can code the Altimeter, you can code the HeadingIndicator. Of course, we would need to make changes to the Plane as well in order to get instantiated and supervised, but I'm going to leave this exercise for you—you have all the skill you need to hook that up.

Now that we can change our heading, we need the controls to make it happen. This will involve a couple of simple changes to the ControlSurfaces class.

Adding Banking to the Controls

It really only makes sense for one guy to control the Plane at a time, right? We've set the stage for this ability in the past by making people go to the Plane in order to get control. It hasn't been really well done yet, since there's nothing really stopping a bunch of people asking for control at the same time; the Plane will happily give it to all of them. If we want to enforce this rule, how do we do it?

One cool thing about having all of these live objects running around is that they all have a unique identity. This unique identity comes in several different forms:

- The local Actor path, e.g., /user/Plane/Controls/Altimeter
- The host-specific Actor path, e.g., akka://System@host:port/user/Plane/Controls/
- The ActorRef. When you're in a single JVM, the ActorRef will do just fine. We can depend on it because the ActorRef won't change during Actor restarts; it's a stable value.

We're inside a single JVM for our Plane, so we can easily use the ActorRef to identify our Actors and we'll use the identifiers to ensure that only one entity at a time can fly the Plane.

First, let's make the requisite changes to the companion object, where we'll hold our message definitions:

```
object ControlSurfaces {  
    case class StickBack(amount: Float)  
    case class StickForward(amount: Float)  
  
    // Add these messages  
    case class StickLeft(amount: Float)  
    case class StickRight(amount: Float)  
    case class HasControl(somePilot: ActorRef)  
}
```

The StickLeft and StickRight messages are obvious; the HasControl message will become clear once we see the definition of the actual ControlSurfaces class:

```
class ControlSurfaces(plane: ActorRef,  
                      altimeter: ActorRef,  
                      heading: ActorRef) extends Actor {  
  
    import ControlSurfaces._  
    import Altimeter._  
    import HeadingIndicator._  
  
    // Instantiate the receive method by saying that the ControlSurfaces  
    // are controlled by the dead letter office. Effectively, this says  
    // that nothing's currently in control  
    def receive = controlledBy(context.system.deadLetters)  
  
    // As control is transferred between different entities, we will  
    // change the instantiated receive function with new variants. This  
    // closure ensures that only the assigned pilot can control the plane  
    def controlledBy(somePilot: ActorRef): Receive = {  
        case StickBack(amount) if sender == somePilot =>  
            altimeter ! RateChange(amount)  
        case StickForward(amount) if sender == somePilot =>  
            altimeter ! RateChange(-1 * amount)  
        case StickLeft(amount) if sender == somePilot =>  
            heading ! BankChange(-1 * amount)  
        case StickRight(amount) if sender == somePilot =>  
            heading ! BankChange(amount)  
        // Only the plane can tell us who's currently in control  
        case HasControl(entity) if sender == plane =>  
            // Become a new instance, where the entity, which the plane told  
            // us about, is now the entity that controls the plane  
            context.become(controlledBy(entity))  
    }  
}
```

Here, we use the ActorRef as identity in two key places; we use it to ensure that only the Plane can dictate who is currently in control, and also to ensure that the entity in control is the only one sending control change events.

We've also introduced a mechanism for passing state data to the behavioural state by creating an indirection on the receive function. Instead of simply specifying everything in receive, we've created another method

that can become() the receive function. To instantiate it, we need to pass in the ActorRef for the entity that takes control, which is closed over and used in the body of the newly instantiated receive.

Why would we create a closure instead of just using a var in the class? In this case, there's not a huge reason to do so, but there are times when using a closure is the better option:

- You really hate seeing the word var.
- You want to have many variants of a particular behaviour based on a variable's value, but you don't want to have multiple different versions of that variable.
- You want a subclass to have greater power of configurability over the behaviour as defined in the parent.

It's up to you how you want to handle the data that is tied to your behavioural state. Just remember that you always have the option of creating a closure, which can easily be the best answer to your problem.

Flying Behaviour

The HeadingIndicator is in place, the ControlSurfaces have been updated to ensure that we can actually turn left and right, so now it's time to hook a brain up to it that provides the actual flying behaviour. As stated, we'll use the FSM here.

The Companion Object

```
object FlyingBehaviour {  
    import ControlSurfaces._  
  
    // The states governing behavioural transitions  
    sealed trait State  
    case object Idle extends State  
    case object Flying extends State  
    case object PreparingToFly extends State
```

The above three states are defined through which our FSM can move. Next, we'll look at the data we can use.

```
// Helper classes to hold course data
case class CourseTarget(altitude: Double, heading: Float,
                        byMillis: Long)
case class CourseStatus(altitude: Double, heading: Float,
                        headingSinceMS: Long, altitudeSinceMS: Long)

// We're going to allow the FSM to vary the behaviour that calculates the
// control changes using this function definition
type Calculator = (CourseTarget, CourseStatus) => Any

// The Data that our FlyingBehaviour can hold
sealed trait Data
case object Uninitialized extends Data

// This is the 'real' data. We're going to stay entirely immutable and,
// in doing so, we're going to encapsulate all of the changing state
// data inside this class
case class FlightData(controls: ActorRef,
                      elevCalc: Calculator,
                      bankCalc: Calculator,
                      target: CourseTarget,
                      status: CourseStatus) extends Data
```

It's the FlightData that will be the most interesting. Note how we've parameterized some behaviour in the data. The elevCalc and bankCalc are functions that we'll use to calculate the amount of control change to apply to the ControlSurfaces. We'll have some fun changing these later.

Note

If you'll be sending function closures around, it's best to do this inside a single JVM. When you start to work with remote Actors (something we'll be getting into later), you'll have to figure out how to serialize and deserialize them. The way we're structuring our system here, the FlyingBehaviour FSM will be directly tied to the component making the changes on the same JVM, so this is not our concern.

We also need to define a single external message that other Actors can send to the FlyingBehaviour to make it fly the Plane.

```
// Someone can tell the FlyingBehaviour to fly
case class Fly(target: CourseTarget)
```

The last thing we'll put into the companion is some helper code that we'll use to make some calculations.

```
def currentMS = System.currentTimeMillis

// Calculates the amount of elevator change we need to make and returns it
def calcElevator(target: CourseTarget, status: CourseStatus): Any = {
    val alt = (target.altitude - status.altitude).toFloat
    val dur = target.currentTimeMillis - status.altitudeSinceMS
    if (alt < 0) StickForward((alt / dur) * -1)
    else StickBack(alt / dur)
}

// Calculates the amount of bank change we need to make and returns it
def calcAilerons(target: CourseTarget, status: CourseStatus): Any = {
    import scala.math.{abs, signum}
    val diff = target.heading - status.heading
    val dur = target.currentTimeMillis - status.headingSinceMS
    val amount = if (abs(diff) < 180) diff
                 else signum(diff) * (abs(diff) - 360f)
    if (amount > 0) StickRight(amount / dur)
    else StickLeft((amount / dur) * -1)
}
```

Being able to put these methods in the companion object is one of the great aspects of having a system that has no internal data. These methods don't rely on any internal data inside the FlyingBehaviour class because there is no internal data there. The FSM makes it very easy for us to write code like this.

The FlyingBehaviour Class

The class itself is a fair bit more complex, so we'll go a little slower on that one, starting with the constructor.

```
class FlyingBehaviour(plane: ActorRef,
                      heading: ActorRef,
                      altimeter: ActorRef) extends Actor
  with FSM[FlyingBehaviour.State, FlyingBehaviour.Data] {
  import FSM._
```

```
import FlyingBehaviour._  
import Pilots._  
import Plane._  
import Altimeter._  
import HeadingIndicator._  
import EventSource._  
  
case object Adjust
```

Some things to note:

- The class mixes in the FSM, which requires type parameters for the state and data parameters. Note that the FSM itself is not an Actor, but its self-typing constraints require that you mix it into a class that eventually extends Actor.
- It also requires the Altimeter and HeadingIndicator Actors to be passed in as well, which is now our standard practice when it comes to specifying static dependencies between Actors.
- We've also defined an internal message, Adjust, which we will use to tell the FlyingBehaviour that we need to adjust the Plane's altitude and heading.

Starting State and Data

Given all of this set up, we can now start to implement the code that the FSM needs to do its job. Everything we're about to see is contained in the FSM Domain Specific Language (DSL), invoked from the FlyingBehaviour's constructor for the purposes of setting up the FSM at construction time.

```
// Sets up the initial values for state and data in the FSM  
startWith(Idle, Uninitialized)
```

Initially, we are doing nothing (Idle) and have absolutely no data associated with that idleness (Uninitialized). Now, of course, Uninitialized is real data, but we think of it as nothing in particular.

Being Idle

The next step is to define some behaviours for particular states. These are a series of `when()` calls that are essentially constructors for partial functions that become receive blocks. Because the FSM is an Actor, our state definitions are easily used as receive partial functions, which is exactly what they are.

```
when(Idle) {  
    case Event(Fly(target), _) =>  
        goto(PreparingToFly) using FlightData(context.system.deadLetters,  
                                         calcElevator,  
                                         calcAilerons,  
                                         target,  
                                         CourseStatus(-1, -1, 0, 0))  
}
```

The DSL is really quite easy to read. When we're Idle, we recognize Fly messages, and when we get one we'll simply go to the PreparingToFly state, using the defined FlightData as our data. And that's all we need to do to specify what happens when we're Idle.

Transitioning to Flying

Once we start flying, we'll need something that is responsible for updating our heading and altitude at periodic intervals. This is not really any different than the timers we created for the Altimeter and the HeadingIndicator, except that the FSM gives us a nicer framework surrounding those timers.

As we transition from PreparingToFly to Flying, we are going to turn on a timer that sends us an Adjust message periodically, which will give the FlyingBehaviour time slices for which it can recalculate and adjust the controls of the Plane.

```
onTransition {  
    case PreparingToFly -> Flying =>  
        setTimer("Adjustment", Adjust, 200.milliseconds, repeat = true)  
}
```

That's it. As we transition from the PreparingToFly to Flying state, the FSM will automatically instantiate this timer for us and we'll start getting

the Adjust message in the Flying state. We never have to worry about getting that message outside of that state.

The FSM lets you create as many timers as you'd like, which allows you great flexibility in sending any number of interesting messages to either the FSM you've implemented, or anyone else for that matter.

Transitioning from the Flying State

When we leave the Flying state, there needs to be some clean up. Here, we need to cancel the existing Adjustment timer that we created when we transitioned into the Flying state.

```
onTransition {  
    case Flying -> _ =>  
        cancelTimer("Adjustment")  
}
```

Note that we don't care what state we move to from Flying; we only care that we're moving out of Flying to something else.

Transitioning into the Idle State

When we transition back into the Idle state, we need to clear out the registrations we made to the instruments that we created earlier.

```
onTransition {  
    case _ -> Idle =>  
        heading ! UnregisterListener(self)  
        altimeter ! UnregisterListener(self)  
}
```

Handling the Unhandled

In every state, there's the potential for a message to come in that the state is not defined to handle. This literally can be *anything*, such as 5. The FSM defines the special state definition, `whenUnhandled`, that allows us to define a partial function that can catch these unhandled messages before they go to the event stream (and the logger). One message that can come in at *any*

time is when the Plane tells us to RelinquishControl. Since this can happen at any time and in any state, we will put the handler in the whenUnhandled definition.

```
whenUnhandled {  
    case Event(RelinquishControl, _) =>  
        goto(Idle)  
}
```

Initialization

The last thing we need to do in our class is initialize the FSM. This is just a bit of housekeeping that the FSM trait imposes on us.

```
...  
    initialize  
}  
// Class complete
```

The Whole Thing

Let's bring it all together into one place so you can see it in all its glory.

```
class FlyingBehaviour(plane: ActorRef,  
                      heading: ActorRef,  
                      altimeter: ActorRef) extends Actor  
    with FSM[FlyingBehaviour.State, FlyingBehaviour.Data] {  
    import FSM._  
    import FlyingBehaviour._  
    import Pilots._  
    import Plane._  
    import Altimeter._  
    import HeadingIndicator._  
    import EventSource._  
  
    case object Adjust  
  
    // Sets up the initial values for state and data in the FSM  
    startWith(Idle, Uninitialized)
```

```
// Adjusts the plane's heading and altitude according to calculations
// It also returns the new FlightData to be passed to the next state
def adjust(flightData: FlightData): FlightData = {
    val FlightData(c, elevCalc, bankCalc, t, s) = flightData
    c ! elevCalc(t, s)
    c ! bankCalc(t, s)
    flightData
}

when(Idle) {
    case Event(Fly(target), _) =>
        goto(PreparingToFly) using FlightData(context.system.deadLetters,
                                                calcElevator,
                                                calcAilerons,
                                                target,
                                                CourseStatus(-1, -1, 0, 0))
}

onTransition {
    case Idle -> PreparingToFly =>
        plane ! GiveMeControl
        heading ! RegisterListener(self)
        altimeter ! RegisterListener(self)
}

def prepComplete(data: Data) = {
    data match {
        case FlightData(_, _, _, _, s) =>
            if (c != context.system.deadLetters &&
                s.heading != -1f && s.altitude != -1f)
                true
            else
                false
        case _ =>
            false
    }
}

when (PreparingToFly)(transform {
    case Event(HeadingUpdate(head), d: FlightData) =>
        stay using d.copy(status = d.status.copy(heading = head,
```

```
headingSinceMS = currentMS))

case Event(AltitudeUpdate(alt), d: FlightData) =>
    stay using d.copy(status = d.status.copy(altitude = alt,
                                              altitudeSinceMS = currentMS))

case Event(Controls(ctrls), d: FlightData) =>
    stay using d.copy(controls = ctrls)

case Event(StateTimeout, _) =>
    plane ! LostControl
    goto (Idle)

} using {
    case s if prepComplete(s.stateData) =>
        s.copy(stateName = Flying)
})
```

when(Flying) {

```
case Event(Adjust, flightData: FlightData) =>
    stay using adjust(flightData)

case Event(AltitudeUpdate(alt), d: FlightData) =>
    stay using d.copy(status = d.status.copy(altitude = alt,
                                              altitudeSinceMS = currentMS))

case Event(HeadingUpdate(head), d: FlightData) =>
    stay using d.copy(status = d.status.copy(heading = head,
                                              headingSinceMS = currentMS))
}
```

onTransition {

```
case PreparingToFly -> Flying =>
    setTimer("Adjustment", Adjust, 200.milliseconds, repeat = true)
}

onTransition {
```

case Flying -> _ =>

```
cancelTimer("Adjustment")
}

onTransition {
```

case _ -> Idle =>

```
heading ! UnregisterListener(self)
altimeter ! UnregisterListener(self)
}
```

```
whenUnhandled {  
    case Event(RelinquishControl, _) =>  
        goto(Idle)  
    }  
  
    initialize  
}
```

Now, let's put it to use.

9.4 The Naughty Pilot

Here's a nightmare scenario: your Pilot is a drinker. Unfortunately, our Pilot has managed to sneak a flask of whiskey on board and whenever the CoPilot isn't looking, he takes a short pull from the container. Eventually, he starts doing some pretty nasty stuff and puts everyone in danger.

The model for this will require three separate Actors, since there are three separate bits of responsibility here:

- Flying the Plane: This is the job of our new FlyingBehaviour FSM.
- Being the Pilot: We'll let the Pilot Actor continue doing this.
- Drinking: It's really too cumbersome to put this behaviour into the Pilot himself; it's a lot easier to let some other Actor be in charge of the drinking and let the Pilot *react* to the events coming from this other Actor.

Let's define the new DrinkingBehaviour, starting with the companion object first.

```
object DrinkingBehaviour {  
    // Internal message that increase / decrease the blood alcohol level  
    case class LevelChanged(level: Float)  
  
    // Outbound messages to tell their person how we're feeling  
    case object FeelingSober  
    case object FeelingTipsy  
    case object FeelingLikeZaphod  
  
    // Factory method to instantiate it with the production timer resolution
```

```
def apply(drinker: ActorRef) = new DrinkingBehaviour(drinker)
    with DrinkingResolution
}
```

You'll note the `factor` method, which hides the fact that there's a `DrinkingResolution` trait mixed into the `DrinkingBehaviour`. We're including this to ease testing, as usual. Our behaviour will have some timers in it, and the resolution of those timers should be easily modifiable in order to make tests run faster. Let's look at the `DrinkingResolution` now:

```
trait DrinkingResolution {
    import scala.util.Random
    def initialSobering: Duration = 1.second
    def soberingInterval: Duration = 1.second
    def drinkInterval(): Duration = Random.nextInt(300).seconds
}
```

Not terribly exciting. It just gives us easy access to slide in new timer resolutions when we need to; the Actor itself is a bit more interesting:

```
class DrinkingBehaviour(drinker: ActorRef) extends Actor {
    this: DrinkingResolution =>
    import DrinkingBehaviour._

    // Stores the current blood alcohol level
    var currentLevel = 0f
    // Just provides shorter access to the scheduler
    val scheduler = context.system.scheduler
    // As time passes our Pilot sobers up. This scheduler keeps that happening
    val sobering = scheduler.schedule(initialSobering,
        soberingInterval,
        self, LevelChanged(-0.0001f))

    // Don't forget to stop your timer when the Actor shuts down
    override def postStop() {
        sobering.cancel()
    }
    // We've got to start the ball rolling with a single drink
    override def preStart() {
        drink()
    }
}
```

```
}

// The call to drink() is going to schedule a single event to self that
// will increase the blood alcohol level by a small amount. It's OK if
// we don't cancel this one - only one message is going to the Dead
// Letter Office

def drink() = scheduler.scheduleOnce(drinkInterval(),
                                      self, LevelChanged(0.005f))

def receive = {

    case LevelChanged(amount) =>
        currentLevel = (currentLevel + amount).max(0f)
        // Tell our drinker how we're feeling. It gets more exciting when
        // we start feeling like Zaphod himself, but at that point he stops
        // drinking and lets the sobering timer make him feel better.
        drinker ! (if (currentLevel <= 0.01) {
            drink()
            FeelingSober
        } else if (currentLevel <= 0.03) {
            drink()
            FeelingTipsy
        }
        else FeelingLikeZaphod)
    }
}
```

As time passes, the sobering timer will lower our Pilot’s blood alcohol level, which will battle with his drinking behaviour. So long as he’s not too drunk, he’ll take another pull from the flask and increase his blood alcohol level again.

If anything bad happens to this Actor, it will restart and go back to its original state (since that’s the default behaviour of the Pilot’s supervision strategy, and we’re not going to change that), which means he’s going to be sober.

Adding the Naughty Behaviour

We need to hook up the Pilot to its DrinkingBehaviour and its FlyingBehaviour so that we can realize what’s in [Figure 9.4](#). The events that come

in from the DrinkingBehaviour need to be translated into new calculation behaviours sent to the FlyingBehaviour Actor.

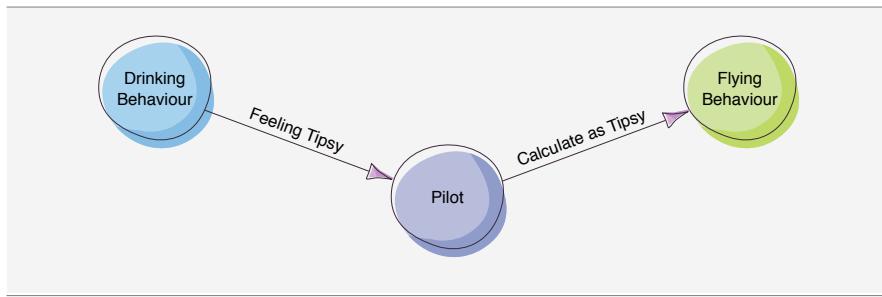


Figure 9.4 · The Pilot receives information about how he feels from the drinking behaviour Actor, which allows him to alter his flying behaviour accordingly.

We don't want to keep sending the same calculation functions to the FlyingBehaviour every time we receive an event from the DrinkingBehaviour, since it will send redundant messages to the Pilot (i.e., FeelingSober, FeelingSober, FeelingSober, etc.). To keep with the theme of the chapter, we'll do this with behavioural state changes.

Let's start by looking at the companion object, and the control calculation functions it defines:

```
object Pilot {  
    import FlyingBehaviour._  
    import ControlSurfaces._  
  
    // Calculates the elevator changes when we're a bit tipsy  
    val tipsyCalcElevator: Calculator = { (target, status) =>  
        val msg = calcElevator(target, status)  
        msg match {  
            case StickForward(amt) => StickForward(amt * 1.03f)  
            case StickBack(amt) => StickBack(amt * 1.03f)  
            case m => m  
        }  
    }  
  
    // Calculates the aileron changes when we're a bit tipsy  
    val tipsyCalcAilerons: Calculator = { (target, status) =>
```

```
val msg = calcAilerons(target, status)
msg match {
  case StickLeft(amt) => StickLeft(amt * 1.03f)
  case StickRight(amt) => StickRight(amt * 1.03f)
  case m => m
}

// Calculates the elevator changes when we're totally out of it
val zaphodCalcElevator: Calculator = { (target, status) =>
  val msg = calcElevator(target, status)
  msg match {
    case StickForward(amt) => StickBack(1f)
    case StickBack(amt) => StickForward(1f)
    case m => m
  }
}

// Calculates the aileron changes when we're totally out of it
val zaphodCalcAilerons: Calculator = { (target, status) =>
  val msg = calcAilerons(target, status)
  msg match {
    case StickLeft(amt) => StickRight(1f)
    case StickRight(amt) => StickLeft(1f)
    case m => m
  }
}
```

Pretty straightforward stuff; the functions use the calculation abilities already present in the FlyingBehaviour, and then modify the results in specific ways.

Again, we should always be thinking about how we test our code, and here we have two Actors (FlyingBehaviour and DrinkingBehaviour) that will be the Pilot Actor's children, which means we have some behaviours that we can test in isolation and that also require mocking out in testing. To facilitate this, we'll provide some traits with factory methods, as usual.

```
trait DrinkingProvider {
  def newDrinkingBehaviour(drinker: ActorRef): Props =
```

```
    Props(DrinkingBehaviour(drinker))
}

trait FlyingProvider {
  def newFlyingBehaviour(plane: ActorRef,
                        heading: ActorRef,
                        altimeter: ActorRef): Props =
    Props(new FlyingBehaviour(plane, heading, altimeter))
}
```

We can now implement our Pilot's changes, which are vast enough to simply include the whole thing again.

```
class Pilot(plane: ActorRef,
            autopilot: ActorRef,
            heading: ActorRef,
            altimeter: ActorRef) extends Actor {
  this: DrinkingProvider with FlyingProvider =>
  import Pilots._
  import Pilot._
  import Plane._
  import context._
  import Altimeter._
  import ControlSurfaces._
  import DrinkingBehaviour._
  import FlyingBehaviour._
  import FSM._

  val copilotName = context.system.settings.config.getString(
    "zzz.akka.avionics.flightcrew.copilotName")
  override def preStart() {
    actorOf(newDrinkingBehaviour(self), "DrinkingBehaviour")
    actorOf(newFlyingBehaviour(plane, heading, altimeter), "FlyingBehaviour")
  }
  // We've pulled the bootstrapping code out into a separate receive
  // method. We'll only ever be in this state once, so there's no point
  // in having it around for long
  def bootstrap: Receive = {
    case ReadyToGo =>
```

```
val copilot = actorFor("../" + copilotName)
val flyer = actorFor("FlyingBehaviour")
flyer ! SubscribeTransitionCallBack(self)
flyer ! Fly(CourseTarget(20000, 250, System.currentTimeMillis + 30000))
become(sober(copilot, flyer))

}

// The 'sober' behaviour
def sober(copilot: ActorRef, flyer: ActorRef): Receive = {
    case FeelingSober =>
        // We're already sober
    case FeelingTipsy =>
        becomeTipsy(copilot, flyer)
    case FeelingLikeZaphod =>
        becomeZaphod(copilot, flyer)
}

}

// The 'tipsy' behaviour
def tipsy(copilot: ActorRef, flyer: ActorRef): Receive = {
    case FeelingSober =>
        becomeSober(copilot, flyer)
    case FeelingTipsy =>
        // We're already tipsy
    case FeelingLikeZaphod =>
        becomeZaphod(copilot, flyer)
}

}

// The 'zaphod' behaviour
def zaphod(copilot: ActorRef, flyer: ActorRef): Receive = {
    case FeelingSober =>
        becomeSober(copilot, flyer)
    case FeelingTipsy =>
        becomeTipsy(copilot, flyer)
    case FeelingLikeZaphod =>
        // We're already Zaphod
}

}

// The 'idle' state is merely the state where the Pilot does nothing at all
def idle: Receive = {
    case _ =>
```

```
}

// Updates the FlyingBehaviour with sober calculations and then
// becomes the sober behaviour
def becomeSober(copilot: ActorRef, flyer: ActorRef) = {
    flyer ! NewElevatorCalculator(calcElevator)
    flyer ! NewBankCalculator(calcAilerons)
    become(sober(copilot, flyer))
}

// Updates the FlyingBehaviour with tipsy calculations and then
// becomes the tipsy behaviour
def becomeTipsy(copilot: ActorRef, flyer: ActorRef) = {
    flyer ! NewElevatorCalculator(topsyCalcElevator)
    flyer ! NewBankCalculator(topsyCalcAilerons)
    become(topsy(copilot, flyer))
}

// Updates the FlyingBehaviour with zaphod calculations and then
// becomes the zaphod behaviour
def becomeZaphod(copilot: ActorRef, flyer: ActorRef) = {
    flyer ! NewElevatorCalculator(zaphodCalcElevator)
    flyer ! NewBankCalculator(zaphodCalcAilerons)
    become(zaphod(copilot, flyer))
}

// At any time, the FlyingBehaviour could go back to an Idle state,
// which means that our behavioural changes don't matter any more
override def unhandled(msg: Any): Unit = {
    msg match {
        case Transition(_, _, Idle) =>
            become(idle)
            // Ignore these two messages from the FSM rather than have them
            // go to the log
        case Transition(_, _, _) =>
        case CurrentState(_, _) =>
        case m => super.unhandled(m)
    }
}

// Initially we start in the bootstrap state
```

```
def receive = bootstrap
}
```

The behaviours have been decomposed out into several separate receive functions, and start in the `bootstrap` behaviour. You'll note how we've changed the way we create a reference to our children. We're still going to create them in the `preStart()` method, but instead of assigning them to vars, we'll take advantage of the fact that we're about to `become()` and pass the values in from the `bootstrap` behaviour to the sober behaviour as closed-over values. This is just another way to reference the data in your Actors, and is facilitated by the state changes we're implementing.

There's also the addition of the following:

```
def bootstrap: Receive = {
  case ReadyToGo =>
    ...
    flyer ! SubscribeTransitionCallBack(self)
    ...
}

// At any time, the FlyingBehaviour could go back to an Idle state,
// which means that our behavioural changes don't matter any more
override def unhandled(msg: Any): Unit = {
  msg match {
    case Transition(_, _, Idle) =>
      become(idle)
      // Ignore these two messages from the FSM rather than have them
      // go to the log
      case Transition(_, _, _) =>
      case CurrentState(_, _) =>
      case m => super.unhandled(m)
  }
}
```

The FSM allows external Actors to watch for state transitions in the same way that the `onTransition` method lets the FSM watch its own transitions. This allows the Pilot to see when the FlyingBehaviour moves from any state to the Idle state. The only time it goes to the Idle state is when it's no longer flying the Plane, and when this occurs, there's no reason for the Pilot to send

it anymore messages. The Pilot can also ignore any messages coming from the DrinkingBehaviour at the same time, so we let the Pilot become(`idle`).

Note that we're using a previously unseen method in the Actor's definition: `unhandled`. This is the same as the FSM's definition of `whenUnhandled` that we saw when implementing the FlyingBehaviour earlier. Since this state transition could occur in any of the previously defined states, we can put the handlers in this catchall. We could also have done this with the `orElse` function combinator as we did with the FlightAttendant earlier, but this is less cumbersome. It's also less explicit, since the function name is called `unhandled` as opposed to `flyingStateHandler`, or something like that, but this is still a nice mechanism.

And that's it. The Pilot can now react to the events received from the DrinkingBehaviour and translate those into new control calculations for the FlyingBehaviour FSM.

9.5 Some Challenges

There are so many things you can do with behavioural states, messages, and events that it's impossible to even imagine them. If you're looking for some interesting challenges that you'd like to implement against what we have so far, why not try:

- Implement a WeatherBehaviour Actor that randomly alters its state and beats on the Plane a bit.
 - For example, you could provide the Altimeter and HeadingIndicator with the ability to receive “offset” messages.
 - If the wind is blowing on the right side of the Plane, then you could set a -10 degree offset on the HeadingIndicator.
 - Varying the intensity of the wind will vary the offset.
 - The FlyingBehaviour should easily compensate for this small amount of offset naturally.
- Create a WayPoint Actor that distributes waypoints as time ticks by to anyone who wants to listen.
 - You can configure the waypoints in the Akka configuration system.

- The FlyingBehaviour Actor would certainly want to listen to this and you could implement a CourseChange message that it could use to alter the FlightData's target member.
- The Pilot and CoPilot can also listen to this. If the CoPilot watches, and it has access to the Heading and Altitude indicators, then it has all the data it needs to wrestle control away from the Pilot when he starts misbehaving.

9.6 Testing FSMs

Now that we have all this excellent stuff built, we should be able to test it. Akka has provided us with the FSM analog to the TestActorRef called TestFSMRef. The TestFSMRef gives us access to the internals of the FSM and allows us to poke and prod its state and data. For example, if we define the helper function to create FlyingBehaviours:

```
def fsm(plane: ActorRef = nilActor,
        heading: ActorRef = nilActor,
        altimeter: ActorRef = nilActor) = {
    TestFSMRef(new FlyingBehaviour(plane, heading, altimeter))
}
```

Then we can test the FlyingBehaviour with some simple test functions. For example, we can assert that the initially constructed state is correct with:

```
"FlyingBehaviour" should {
    "start in the Idle state and with Uninitialized data" in {
        val a = fsm()
        a.stateName must be (Idle)
        a.stateData must be (Uninitialized)
    }
}
```

And by being able to inspect the internal data, as well as the state, we can assert that the PreparingToFly behaviour works with the following:

```
"PreparingToFly state" should {
    "stay in PreparingToFly state when only a HeadingUpdate is received" in {
```

```
    val a = fsm()
    a ! Fly(target)
    a ! HeadingUpdate(20)
    a.stateName must be (PreparingToFly)
    val sd = a.stateData.asInstanceOf[FlightData]
    sd.status.altitude must be (-1)
    sd.status.heading must be (20)
}
"move to Flying state when all parts are received" in {
    val a = fsm()
    a ! Fly(target)
    a ! HeadingUpdate(20)
    a ! AltitudeUpdate(20)
    a ! Controls(testActor)
    a.stateName must be (Flying)
    val sd = a.stateData.asInstanceOf[FlightData]
    sd.controls must be (testActor)
    sd.status.altitude must be (20)
    sd.status.heading must be (20)
}
}
```

And we can ensure that the state transition from PreparingToFly to Flying creates the “Adjustment” timer with:

```
"transitioning to Flying state" should {
  "create the Adjustment timer" in {
    val a = fsm()
    a.setState(PreparingToFly)
    a.setState(Flying)
    a.timerActive_?("Adjustment") must be (true)
  }
}
```

Testing FSMs is made much simpler by including the TestFSMRef. We can inspect the internals just to ensure that the data and states are correct when we need them to be without having to subclass, or put in spies, or any other mechanism you might have to do otherwise.

Yeah, these guys know what they’re doing.

9.7 Testing the Pilot

We won't go too crazy here because you've met a lot of the TestKit up to this point, but there's an interesting couple of helper functions that you might want to be acquainted with. When we transition into the tipsy behaviour, for example, we send a couple of messages to the FlyingBehaviour in order to change the elevator calculation and the bank calculation.

It's rather brittle to test these in such a way that we assume an ordering. It would be better to ensure that we test so that we receive them all independent of their order. TestKit provides two mechanisms for this and we can use both in different ways.²

```
"Pilot.becomeZaphod" should {
    "send new zaphodCalcElevator and zaphodCalcAilerons to FlyingBehaviour" in {
        val (ref, _) = makePilot()
        ref ! FeelingLikeZaphod
        expectMsgAllOf(NewElevatorCalculator(Pilot.zaphodCalcElevator),
                      NewBankCalculator(Pilot.zaphodCalcAilerons))
    }
}
```

The `expectMsgAllOf()` will ensure that all of the given messages are received and matched. This is a tad limiting though, depending on your needs. If you want the equivalent of `expectMsgPF()`, which allows you to run a partial function across the match, then we can use `expectMsgAllClassOf()` with a subsequent match.

```
"Pilot.becomeTipsy" should {
    "send new tipsyCalcElevator and tipsyCalcAilerons to FlyingBehaviour" in {
        val (ref, _) = makePilot()
        ref ! FeelingTipsy
        expectMsgAllClassOf(classOf[NewElevatorCalculator],
                            classOf[NewBankCalculator]) foreach { m =>
            m match {
                case NewElevatorCalculator(f) =>
                    f must be (Pilot.topsyCalcElevator)
            }
        }
    }
}
```

²There are, in fact, many variations of these as there are many variations of the others, but they're variations that you can investigate on your own. The ScalaDoc is quite good.

```
case NewBankCalculator(f) =>
    f must be (Pilot.topsyCalcAilerons)
}
}
}
}
```

The `expectMsgAll*()` clan returns a sequence of the matched messages. You can then perform further assertions on these matches as we have above.

Just another tool to stick in your testing tool chest.

9.8 Chapter Summary

You should be feeling fairly awesome right about now. We've covered a *lot* of cool functionality in this chapter that will allow you to express your stateful systems with an elegance that isn't easily achieved through the mechanisms we've been given in the past. Massive chains of `if` statements and complex state data variables are not generally present in Akka's stateful Actor applications.

You can now wield:

- The state facilities of the `ActorContext` through `become()` and `unbecome()`
- The Finite State Machine, its state definitions, its transitions, timers, and handlers
- The FSM's testing facilities

On top of that you've also picked up some necessary skills aimed at appropriate behavioural decomposition by breaking behaviour out into separate Actors, as we did with the Pilot, DrinkingBehaviour, and FlyingBehaviour. The messages that travel between your decomposed functionality tie them back together in powerful and flexible ways.

This chapter has also reminded you about the power of immutability. The FlyingBehaviour FSM has been implemented as a stateful object, where all of the state-specific data is carried in a single, immutable object of its own. We can pass this through, filter it, and transform it as necessary.

We also covered a very important topic that faces many non-blocking, concurrent programmers: the multi-state algorithm with non-determinacy. We wanted to get the Plane ready to fly by collecting information from various sources, without having to block the application while we waited. This was done through a stateful transition as each piece was received in a completely non-blocking manner, without having to care about the order of the incoming data.

One thing that should become even more clear at this point is the power of the untyped Actor. All of the bits and pieces we have in the system can now send any type of messages to each other. One of the places this really shines is in testing. We can easily substitute the `testActor` for any of the components that our System Under Test (SUT) needs. In production, we can easily do the same thing.

You've also picked up more tools that help you implement Actor code, such as passing data to behavioural functions during the `become()` call and using the `unhandled()` message handler.

Chapter 10

Routing Messages

Remember that whole “strongly typed messages” and “untyped endpoints” stuff from a while back? Well, here’s another reason why that’s just awesome: *routing*. Akka’s routing feature lets you alter a message’s path from sender to receiver. Rather than horrifically shoving irrelevant message-handling behaviour into the sender and receiver, we can put that general handling logic into an intermediary.

10.1 Routers Are Not Actors

If Routers were implemented using Actors, with the full Dispatcher/Mailbox/Queue family of players, then you might expect that Routers would be bottlenecks to performance. Well, you don’t need to worry about that. The methods of the Router are invoked and routed entirely on the calling thread. This means that the Router’s implementation needs to be specially designed for concurrency, since those methods will not be running all safe and warm inside the Actor’s message receiver.

10.2 Akka’s Standard Routers

Akka provides a set of standard routers that implement routing patterns that frequently show up in those everyday problems. The nice thing about Routers is that, generally speaking, you can swap them for different implementations whenever you want. The loosely typed endpoints, and the beauty of the message handlers at the receiving end, make this a piece of cake.

Let’s look at what Akka provides for you out of the box.

RoundRobinRouter

The RoundRobinRouter sends messages to the Actors for which it fronts in a round-robin fashion. [Figure 10.1](#) depicts this particular router.

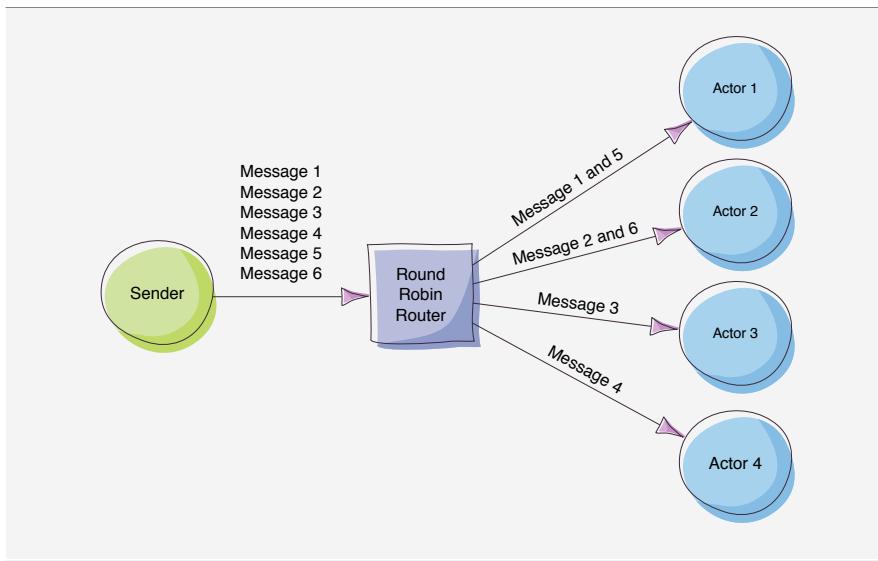


Figure 10.1 · Messages are routed to the composed Actors in a round-robin fashion.

Not much is surprising about the RoundRobinRouter. Here are some places in which you might want to use it:

- You want some concurrency beyond what a single instance of the working Actor would give you.
- All of the work that the Actors might be doing is fairly static. No incoming message requires more work than any other incoming message.
- Each instance of the Actor that lives under the Router is no different than any other.

Obviously, since message distribution will be deterministic around the ring of composed Actors, you have to be ready for the situation where a single Actor could be bogged down with work. This would happen when certain

messages take longer to process than others and bad luck would dictate that those longer-to-process messages always go to the same poor guy.

SmallestMailboxRouter

When it comes to load balancing, this is one of the niftiest ones. When a message comes into the Router, it decides to route the message to the composed Actor whose Mailbox is the smallest. There are further details to how it makes that choice, but for our purposes the obvious notion is perfectly reasonable. You should consult the documentation that ships with Akka if you're interested. Check [Figure 10.2](#) to get a picture of it.

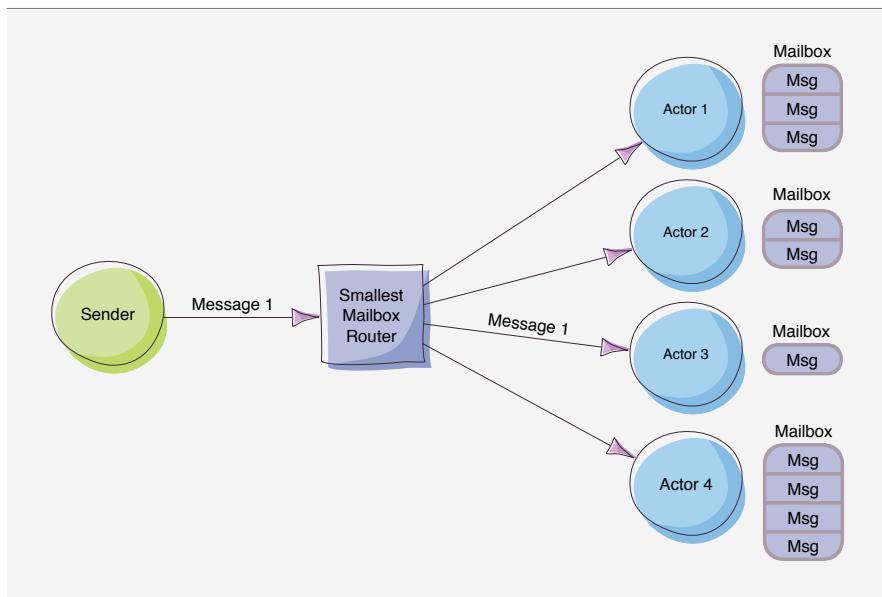


Figure 10.2 · In this case, the SmallestMailboxRouter will choose Actor 3 as the recipient of Message 1, since its Mailbox is clearly the smallest.

The SmallestMailboxRouter is a pretty good choice when it comes to balancing load among your composed Actors. The Mailbox size is generally a good indicator of how heavily loaded the Actor is, from a practical perspective. You should note that if the composed Actor is actually remotely deployed to another node, there's no way for the SmallestMailboxRouter to actually understand its Mailbox size. So, in these situations it might not be

the ideal choice of Router, but that clearly depends on the situation in which you find yourself.

Configuring a Router

The configurations are specified in the `akka.actor.deployment` block. Below is a simple configuration of a RoundRobinRouter, which would appear in your `application.conf` file:

```
akka {  
    actor {  
        deployment {  
            /DatabaseConnectionRouter {  
                router = "round-robin"  
                nr-of-instances = 20  
            }  
        }  
    }  
}
```

Clearly, this Router is intended to be used for Database connections. Assuming you have an Actor that communicates with a Database, you can now increase the parallel access to that Database using this Router. We would invoke it in code like this:

```
import akka.routing.FromConfig  
  
class DBConnection extends Actor { ... }  
  
val dbRouter = system.actorOf(Props[DBConnection].withRouter(FromConfig(),  
    "DatabaseConnectionRouter"), "DBRouter")
```

The `dbRouter` will be an instance of a RoundRobinRouter that will represent the database connection, and it will also be the parent of 20 instances of the `DBConnection` Actor. Assuming the `DBConnection` Actor and the Database can handle it, this will improve throughput. However, what's the sweet spot? Is it 20 Actors? Is it 10, or 40? It's hard to tell, and this is where the tuning comes in. The configuration aspect of Routers allows other people to help tune your application.

We'll be coming back to configuration again as we explore more about Routers. You choose whether or not to use configuration for your Routers;

you just need to be aware that if you choose to leave configuration out of your application, then it will be entirely up to you and your other programmers to tune your application and deliver a new build in order to effect those changes.

10.3 Routers and Children

Routers route to *routees*. You can create those routees dynamically by the Router, or you can assign them to it from an already created set. These two different methods of assigning routees have an impact on the relationship and supervision of those routees.

Letting the Router Create the Routees

There are some advantages to having the Router create the routees:

1. The Router handles the Supervision. You don't need to set up another (possibly parallel) hierarchy simply to give the routees a "home" and a supervisor.
2. It works well with configuration. If you want to allow the specification of `nr-of-instances` in the configuration file, then the Router can create those instances.

There are also some disadvantages:

1. You'll have a difficult time constructing anything but a single type of Actor. The Router uses the `Props(...)` object you supply as a factory for creating Actor instances. As such, it's quite difficult to have it create different types of instances using that sole factory method. Indeed, attempting to do otherwise is probably just a bad idea altogether; the trickery you might have to employ would just make your code difficult to understand.
2. You can't name them the way you might like. You don't have control over the second parameter to the `actorOf()` call, so you're stuck with whatever name the Router automatically generates.

If the disadvantages outweigh the advantages for you in a particular scenario, then you might want to opt for the other method of routee assignment, which allows you to specify the pre-created Actors.

Passing the Router Pre-Created Actors

The alternate assignment method has advantages that directly address the disadvantages of the previous method:

1. You have full control over creating the Actors, which allows you to specify anything from a different construction parameter value all the way up to entirely different Actor types.
2. Supervision is much more flexible. Your Actors need not be the children of any single parent, but each can have its own unique parent (or anything in between), allowing customized supervision strategies.
3. You can name the Actors whatever you like. If naming is an important consideration, then this method gives you that control.

And, of course, there are disadvantages as well:

1. You have to have parents for them. This is the yin to the advantage's yang. You have the power to assign whatever parentage you'd like, but you also have to assign that parentage. The same can be said for supervision.
2. It's not as flexible from a configuration perspective. If the Router can't create, then it has less power to do things on your behalf. This becomes more important when we recognize that Routers can dynamically resize their Actor pool, which we'll learn about soon.

Each method has its advantages and disadvantages and you'll need to pick the one that works best for your situation. Don't discount the power and flexibility of a configured Router, however. If you can, try to go with a configured Router, as it keeps the door open for possibilities you may not be able to anticipate.

The Router and Its Children's Life Cycles

When the Router creates the routees, they are its children, which means that the Router must manage their life cycles. By default, the Router will assign a `supervisorStrategy` that always escalates the decision to its parent. We've learned that this can have a fairly drastic effect, in so much as the escalation

will result in the strategy being applied at the level of the Router, instead of the single child. This means that a strategy of Stop, for example, will Stop the entire hierarchy, not just the Actor that threw the exception.

This may be just fine for your situation, but if it's not, you'll need to specify the strategy yourself, something like this:

```
val dbRouter = system.actorOf(Props.empty.withRouter(RoundRobinRouter(  
    nrOfInstances = 5,  
    supervisorStrategy = OneForOneStrategy {  
        // define your Decider here  
    }), "DBRouter")
```

If you'd rather use the standard default strategy, then you can assign that easily enough:

```
val dbRouter = system.actorOf(Props.empty.withRouter(RoundRobinRouter(  
    nrOfInstances = 5,  
    supervisorStrategy = SupervisorStrategy.defaultStrategy  
), "DBRouter"))
```

10.4 Routers on a Plane

It's time we put the Router to some use.¹ A BroadcastRouter would make the perfect component for allowing the passengers to receive important information, such as “Fasten Seat Belts,” don't you think? Let's build that into the Plane now, but first we'll need some passengers.

To do this, I created a bunch of passenger names using *The Random Name Generator*² and stuck them in the configuration file `src/main/resources/application.conf`. Feel free to use any names you'd like...you know, friends, relatives, loved ones, arch enemies, any personal nemesis you might have, whatever.

```
zzz.akka.avionics {  
    passengers = [  
        [ "Kelly Franqui", "01", "A" ],
```

¹It's times like this that I really wish the Akka team came up with a component called “Snake,” but in lieu of that, “Router” will have to do.

²<http://www.kleimo.com/random/name.cfm>

```
[ "Tyrone Dotts",      "02", "B" ],
[ "Malinda Class",    "03", "C" ],
[ "Kenya Jolicoeur",  "04", "A" ],
[ "Christian Piche",  "10", "B" ],
[ "Neva Delapena",    "11", "C" ],
[ "Alana Berrier",    "12", "A" ],
[ "Malinda Heister",  "13", "B" ],
[ "Carlene Heiney",   "14", "C" ],
[ "Erik Dannenberg",  "15", "A" ],
[ "Jamie Karlin",     "20", "B" ],
[ "Julianne Schroth", "21", "C" ],
[ "Elinor Boris",     "22", "A" ],
[ "Louisa Mikels",    "30", "B" ],
[ "Jessie Pillar",    "31", "C" ],
[ "Darcy Goudreau",   "32", "A" ],
[ "Harriett Isenhour", "33", "B" ],
[ "Odessa Maury",    "34", "C" ],
[ "Malinda Hiett",    "40", "A" ],
[ "Darcy Syed",       "41", "B" ],
[ "Julio Dismukes",   "42", "C" ],
[ "Jessie Altschuler", "43", "A" ],
[ "Tyrone Ericsson",  "44", "B" ],
[ "Mallory Dedrick",  "50", "C" ],
[ "Javier Broder",    "51", "A" ],
[ "Alejandra Fritzler", "52", "B" ],
[ "Rae Mcalleer",      "53", "C" ]
]
}
```

Given this, we can now build our Passenger Actor. As always, we'll start with our companion object that defines some public messages.

```
object Passenger {
  // These are notifications that tell the Passenger
  // to fasten or unfasten their seat belts
  case object FastenSeatbelts
  case object UnfastenSeatbelts

  // Regular expression to extract Name-Row-Seat tuple
```

```
val SeatAssignment = """([\w\s_]+)-(\d+)-([A-Z])""".r
}
```

To ease our testing, we'll add two traits that we can use to modify our dependencies during testing.

```
// The DrinkRequestProbability trait defines some
// thresholds that we can modify in tests to
// speed things up.
trait DrinkRequestProbability {
    // Limits the decision on whether the passenger
    // actually asks for a drink
    val askThreshold = 0.9f

    // The minimum time between drink requests
    val requestMin = 20.minutes

    // Some portion of this (0 to 100  // to requestMin
    val requestUpper = 30.minutes

    // Gives us a 'random' time within the previous
    // two bounds
    def randomishTime(): Duration = {
        requestMin + scala.util.Random.nextInt(
            requestUpper.toMillis.toInt).millis
    }
}

// The idea behind the PassengerProvider is old news at this point.
// We can use it in other classes to give us the ability to slide
// in different Actor types to ease testing.
trait PassengerProvider {
    def newPassenger(callButton: ActorRef): Actor =
        new Passenger(callButton) with DrinkRequestProbability
}
```

And now that we have the above, we can define the Passenger Actor.

```
class Passenger(callButton: ActorRef) extends Actor
    with ActorLogging {
    this: DrinkRequestProbability =>
    import Passenger._
```

```
import FlightAttendant.{GetDrink, Drink}
import scala.collection.JavaConverters._

// We'll be adding some randomness to our Passenger,
// and this shortcut will make things a little more
// readable.
val r = scala.util.Random

// It's about time that someone actually asked for a
// drink since our Flight Attendants have been coded
// to serve them up
case object CallForDrink

// The name of the Passenger can't have spaces in it,
// since that's not a valid character in the URI
// spec. We know the name will have underscores in
// place of spaces, and we'll convert those back
// here.
val SeatAssignment(myname, _, _) =
    self.path.name.replaceAllLiterally("_", " ")

// We'll be pulling some drink names from the
// configuration file as well
val drinks = context.system.settings.config.getStringList(
    "zzz.akka.avionics.drinks").asScala.toIndexedSeq

// A shortcut for the scheduler to make things look
// nicer later
val scheduler = context.system.scheduler

// We've just sat down, so it's time to get a drink
override def preStart() {
    self ! CallForDrink
}

// This method will decide whether or not we actually
// want to get a drink using some randomness to
// decide
def maybeSendDrinkRequest(): Unit = {
    if (r.nextFloat() > askThreshold) {
        val drinkname = drinks(r.nextInt(drinks.length))
        callButton ! GetDrink(drinkname)
    }
}
```

```
        }
        scheduler.scheduleOnce(randomishTime(), self, CallForDrink)
    }

    // Standard message handler
    def receive = {
        case CallForDrink =>
            maybeSendDrinkRequest()
        case Drink(drinkname) =>
            log.info("{} received a {} - Yum", myname, drinkname)
        case FastenSeatbelts =>
            log.info("{} fastening seatbelt", myname)
        case UnfastenSeatbelts =>
            log.info("{} UNfastening seatbelt", myname)
    }
}
```

Note that we've included another bit of configuration in this class; we've pulled in a collection of drinks. This configuration's definition is not shown for two simple reasons: 1) it would take up space in the book, and 2) you have more than enough skill to know exactly what it looks like anyway, so there's no point in showing it.

Also note that we're not really *doing* anything when we get the Drink, FastenSeatbelts, and UnfastenSeatbelts messages, other than logging something at info level. There are two reasons we're doing this as well: 1) it's easy to do this and there's no value in doing anything more difficult for illustration purposes and 2) we get to play with the Event Stream in a test.

We've also included a construction parameter in the Passenger: the callButton. We'll supply an ActorRef to this parameter that the Passenger can use to requests drinks.

Testing and the Event Stream

We haven't really covered too much of the Event Stream at this point, because we're waiting until we cover the EventBus, so it qualifies as "new." Interestingly, the Logger publishes to the Event Stream, which means that it's not entirely useless to us at a programmatic level. Normally when you "log" something (or, heaven forbid, println() something), it's useless to you at a programmatic level. Well, because the Logger is merely publishing

events to the Event Stream, and because the Event Stream is so awesome, we can use it in a test.

We'll test that the Passenger actually fastens his seat belt when the "Fasten Seat Belt" light comes on. We'll do this using two facilities that we haven't yet seen: a `TestProbe` and the `ActorSystem`'s Event Stream. Here's the test from the `Passenger` test spec:

```
trait TestDrinkRequestProbability extends DrinkRequestProbability {
    override val askThreshold = 0f
    override val requestMin = 0.milliseconds
    override val requestUpper = 2.milliseconds
}

class PassengersSpec extends TestKit(ActorSystem())
    with ImplicitSender {
    import akka.event.Logging.Info
    import akka.testkit.TestProbe

    var seatNumber = 9
    def newPassenger(): ActorRef = {
        seatNumber += 1
        system.actorOf(Props(new Passenger(testActor))
            with TestDrinkRequestProbability),
            s"Pat_Metheny-$seatNumber-B")
    }
    "Passengers" should {
        "fasten seatbelts when asked" in {
            val a = newPassenger()
            val p = TestProbe()
            system.eventStream.subscribe(p.ref, classOf[Info])
            a ! FastenSeatbelts
            p.expectMsgPF() {
                case Info(_, _, m) =>
                    m.toString must include ("fastening seatbelt")
            }
        }
    }
}
```

We've used a `TestProbe` instead of the `ImplicitSender` and `testActor`

that are present in our constructed tests. At times, you may need to isolate the messages that are coming into your test. For example, we've passed the `testActor` in for the `callButton` parameter in the `Passenger`. If the `Passenger` sends a `GetDrink` message to the `callButton`, then a bare `expectMsg()` call will see it, which makes our test difficult to verify. By using carefully placed `TestProbes`, we can isolate these messages from each other, making the tests easier to verify.

We use this `TestProbe` when we subscribe to the `ActorSystem`'s Event Stream, as follows:

```
system.eventStream.subscribe(p.ref, classOf[Info])
```

This says that we want the `TestProbe`'s `ActorRef` (`p.ref`) to be the handle to the subscribed Actor, and that we want it to receive events that match the class `akka.event.Logger.Info`. We can then use the `TestProbe`'s `expectMsgPF()` method to help us assess whether or not the `Passenger` did indeed fasten his seat belt.

```
p.expectMsgPF() {  
    case Info(_, _, m) =>  
        m.toString must include ("fastening seatbelt")  
}
```

Pretty nifty, no?

The Passenger Router

Now that we can create some `Passengers`, we can stick a `BroadcastRouter` in front of them. What creation strategy do you think we'll need for this? Well, based on the pros and cons we talked about earlier, and the fact that we've gone to so much trouble to *name* our passengers, then we'll probably need to go with the pre-creation route.

This will be more interesting than you might have originally thought. We need to create a `Supervisor` for our `Passenger`s, which will supervise them in whatever way we see fit, and then allow a `BroadcastRouter` to be created that will then route to them as `routees`.

But there's a catch: you'll need to wrestle the non-determinism of asynchronous programming to the ground. You see, the problem is that creating

the Supervisor and its children will be asynchronous. You can't give a collection of routees to a BroadcastRouter until you have a coherent collection of routees, and that can't happen until the asynchronous creation of those children is complete. Got it? OK, maybe not yet, but you will soon.

```
object PassengerSupervisor {  
    // Allows someone to request the BroadcastRouter  
    case object GetPassengerBroadcaster  
  
    // Returns the BroadcastRouter to the requestor  
    case class PassengerBroadcaster(broadcaster: ActorRef)  
  
    // Factory method for easy construction  
    def apply(callButton: ActorRef) = new PassengerSupervisor(callButton)  
        with PassengerProvider  
    }  
}
```

You can see from the companion object that the PassengerSupervisor will allow others to retrieve the BroadcastRouter. It will only return it once it can coherently construct it. Actually doing so and handling all of the concurrency is tricky, given the skills that we have thus far.

Figure 10.3 shows what we're going to construct. We'll have different supervision strategies for the PassengerSupervisor components and full construction will be integrated with the asynchronous nature of the Isolated-StopSupervisor's construction.

```
class PassengerSupervisor(callButton: ActorRef) extends Actor {  
    this: PassengerProvider =>  
    import PassengerSupervisor._  
  
    // We'll resume our immediate children instead of restarting them  
    // on an Exception  
    override val supervisorStrategy = OneForOneStrategy() {  
        case _: ActorKilledException => Escalate  
        case _: ActorInitializationException => Escalate  
        case _ => Resume  
    }  
  
    // Internal messages we use to communicate between this Actor  
    // and its subordinate IsolatedStopSupervisor  
    case class GetChildren(forSomeone: ActorRef)
```

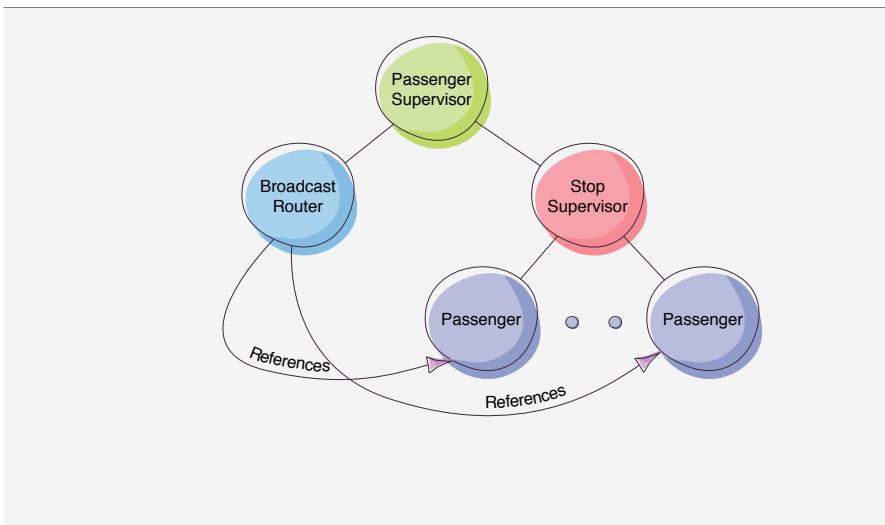


Figure 10.3 · The PassengerSupervisor's structure houses an IsolatedStopSupervisor to manage its Passengers and uses the default supervision strategy for its immediate children. The BroadcastRouter will be created after the IsolatedStopSupervisor is completely instantiated.

```
case class Children(children: Iterable[ActorRef], childrenFor: ActorRef)

// We use preStart() to create our IsolatedStopSupervisor
override def preStart() {
    context.actorOf(Props(new Actor {
        val config = context.system.settings.config
        override val supervisorStrategy = OneForOneStrategy() {
            case _: ActorKilledException => Escalate
            case _: ActorInitializationException => Escalate
            case _ => Stop
        }
        override def preStart() {
            import scala.collection.JavaConverters.-
            import com.typesafe.config.ConfigList
            // Get our passenger names from the configuration
            val passengers = config.getList("zzz.akka.avionics.passengers")
            // Iterate through them to create the passenger children
        }
    }))
}
```

```
passengers.asScala.foreach { nameWithSeat =>
    val id = nameWithSeat.asInstanceOf[ConfigList].unwrapped(
        ).asScala.mkString("-").replaceAllLiterally(" ", "_")
    // Convert spaces to underscores to comply with URI standard
    context.actorOf(Props(newPassenger(callButton)), id)
}
}

// Override the IsolatedStopSupervisor's receive
// method so that our parent can ask us for our
// created children
override def receive = {
    case GetChildren(forSomeone: ActorRef) =>
        sender ! Children(context.children, forSomeone)
    }
    "PassengersSupervisor")
}

... receive method to be defined ...
}
```

Much of this should be pretty familiar by now, but let's go through it quickly just to shore up the edges of our understanding:

- We specify a `supervisorStrategy` that resumes when immediate children fail.
- We create an anonymous internal child Actor to serve as the Supervisor of our Passengers. This Supervisor institutes a Stop strategy. We'll be happy to let our Passengers stay dead when they die.
- We'll let the Supervisor simply sit on the sidelines and do its job of supervising the children. We want to access those children in order to create a BroadcastRouter in front of them. As such, we've created a message handler that will return those children when requested.

The important bit that we have yet to see is the message handler inside the PassengerSupervisor itself. It's this handler that processes messages from the outside world, so it's the one in which we're most interested.

```
// TODO: This noRouter method could be made simpler by using a Future.  
// We'll have to refactor this later.  
  
def noRouter: Receive = {  
    case GetPassengerBroadcaster =>  
        context.actorFor("PassengersSupervisor") ! GetChildren(sender)  
    case Children(passengers, destinedFor) =>  
        val router = context.actorOf(Props().withRouter(  
            BroadcastRouter(passengers.toSeq)), "Passengers")  
        destinedFor ! PassengerBroadcaster(router)  
        context.become(withRouter(router))  
    }  
    def withRouter(router: ActorRef): Receive = {  
        case GetPassengerBroadcaster =>  
            sender ! PassengerBroadcaster(router)  
    }  
    def receive = noRouter
```

(Ignore the TODO for the moment; you'll get the full picture as to why it's there very soon.)

You can see that we've broken up our message handler into two separate states. We start out in the state where we have no BroadcastRouter (i.e., the noRouter state). In this state, we accept the GetPassengerBroadcaster message, which starts a dance between itself, the internal supervisor, and the outside world. [Figure 10.4](#) shows what we're going to implement.

We're doing this in a non-blocking, asynchronous manner. It's certainly possible to do this in a blocking manner as well, which is simpler, but far less instructive. The power of Akka programming is that it allows us to code safely and coherently without blocking.

When the outside world asks for the BroadcastRouter, the PassengerSupervisor doesn't yet have it. It also doesn't have direct access to the Stop Supervisor's children, so it can't just grab them and return them; it needs to ask for them. We also need to ensure that the original requester (the Outside World) is retained during this algorithm's execution.

All of the data we need is passed in the messages. Imagine how cumbersome it would be if we were to store the ActorRef for the Outside World as a var in the PassengerSupervisor. It's much easier to toss this information inside the messages.

Now, back to that TODO we left behind earlier. I said that we don't yet

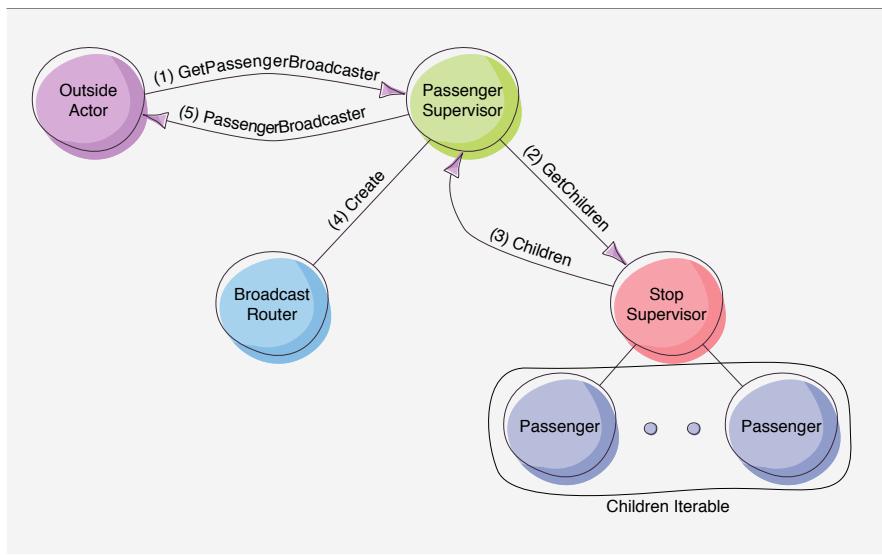


Figure 10.4 · The non-blocking, asynchronous algorithm we use to obtain the BroadcastRouter of the children of the PassengerSupervisor’s embedded Stop Supervisor.

possess the skill to remedy the TODO and that’s quite true, since the proper solution relies on the use of Futures. This would make it clear where the communication’s request/response nature goes.

Using the Passenger Router

We’ll illustrate how to use the BroadcastRouter using a test. Let’s examine the entire test file:

```
package zzz.akka.avionics

import akka.actor.{ActorSystem, Actor, ActorRef, Props}
import akka.testkit.{TestKit, ImplicitSender}
import scala.concurrent.util.duration._
import com.typesafe.config.ConfigFactory
import org.scalatest.{WordSpec, BeforeAndAfterAll}
import org.scalatest.matchers.MustMatchers

// A specialized configuration we'll inject into the
```

```
// ActorSystem so we have a known quantity we can test with
object PassengerSupervisorSpec {
    val config = ConfigFactory.parseString("""
        zzz.akka.avionics.passengers = [
            [ "Kelly Franqui", "23", "A" ],
            [ "Tyrone Dotts", "23", "B" ],
            [ "Malinda Class", "23", "C" ],
            [ "Kenya Jolicoeur", "24", "A" ],
            [ "Christian Piche", "24", "B" ]
        ]
    """)

    // We don't want to work with "real" passengers. This mock
    // passenger will be much easier to verify things with
    trait TestPassengerProvider extends PassengerProvider {
        override def newPassenger(callButton: ActorRef): Actor =
            new Actor {
                def receive = {
                    case m => callButton ! m
                }
            }
    }

    // The Test class injects the configuration into the
    // ActorSystem
    class PassengerSupervisorSpec
        extends TestKit(ActorSystem("PassengerSupervisorSpec",
            PassengerSupervisorSpec.config))
            with ImplicitSender
            with WordSpec
            with BeforeAndAfterAll
            with MustMatchers {
        import PassengerSupervisor._

        // Clean up the system when all the tests are done
        override def afterAll() {
            system.shutdown()
        }
    }
}
```

```
"PassengerSupervisor" should {
    "work" in {
        // Get our SUT
        val a = system.actorOf(Props(new PassengerSupervisor(testActor)
            with TestPassengerProvider))

        // Grab the BroadcastRouter
        a ! GetPassengerBroadcaster
        val broadcaster = expectMsgPF() {
            case PassengerBroadcaster(b) =>
                // Exercise the BroadcastRouter
                b ! "Hithere"
                // All 5 passengers should say "Hithere"
                expectMsg("Hithere")
                expectMsg("Hithere")
                expectMsg("Hithere")
                expectMsg("Hithere")
                expectMsg("Hithere")
                // And then nothing else!
                expectNoMsg(100.milliseconds)
                // Return the BroadcastRouter
                b
        }
        // Ensure that the cache works
        a ! GetPassengerBroadcaster
        expectMsg(PassengerBroadcaster(`broadcaster`))
    }
}
```

Technically speaking, this isn't much of a unit test since it tests a lot of stuff and has a fair number of moving parts, but it definitely gets the job done.

10.5 Magically Appearing Flight Attendants

The LeadFlightAttendant is really nothing more than a homegrown RandomRouter, so why not reimplement it as such? While we're at it, we'll simplify it and forgo the FlightAttendant names; just let the RandomRouter assign

them on its own. Further, let's create the Router in configuration rather than entirely in code. Let's start with the configuration:

```
akka.actor.deployment {  
    /Plane/LeadFlightAttendant {  
        router = "random"  
        resizer {  
            lower-bound = 4  
            upper-bound = 10  
        }  
    }  
}
```

The `resizer` section is new. Here's another neat aspect of Routers: the pool of Actors that it manages can grow as required! The `resizer` section above starts with only four FlightAttendants, but will grow that pool to have as many as ten, if required.

We can now modify the `Plane` to use this new Router configuration.

```
def startPeople() {  
    // Use the Router as defined in the configuration file  
    // under the name "LeadFlightAttendant"  
    val leadAttendant =  
        actorOf(Props(newFlightAttendant).withRouter(FromConfig()),  
               "LeadFlightAttendant")  
    val people =  
        actorOf(Props(new IsolatedStopSupervisor  
                      with OneForOneStrategyFactory {  
                def childStarter() {  
                    context.actorOf(Props(PassengerSupervisor(leadAttendant)),  
                                   "Passengers")  
                    // ... as before  
                }  
            }  
        })  
}
```

Here, we've wired the `PassengerSupervisor`'s `callButton` parameter up to the `LeadFlightAttendant`, which we've created from the configuration file

that declares it to be a RandomRouter, which will resize itself from a minimum of four FlightAttendants to a maximum of ten.

10.6 Sectioning off Flight Attendant Territory

What if the Flight Attendants had specific areas of the Plane in which they were supposed to work? In other words, the Flight Attendant in charge of rows 1-10 isn't supposed to handle rows 21-30.

To handle this type of routing, we can create a custom Router in code. The Router will statically allocate several FlightAttendants and then route based on the row number contained in the incoming sender's `sender.path.name` attribute.

```
package zzz.akka.avionics

import akka.actor.{Props, SupervisorStrategy}
import akka.routing.{RouterConfig, RouteeProvider, Route, Destination}
import akka.dispatch.Dispatchers

class SectionSpecificAttendantRouter extends RouterConfig {
    this: FlightAttendantProvider =>

    // The RouterConfig requires us to fill out these two
    // fields We know what the supervisorStrategy is but we're
    // only slightly aware of the Dispatcher, which we will be
    // meeting in detail later
    def routerDispatcher: String = Dispatchers.DefaultDispatcherId
    def supervisorStrategy: SupervisorStrategy =
        SupervisorStrategy.defaultStrategy

    // The createRoute method is what invokes the decision
    // making code. We instantiate the Actors we need and then
    // create the routing code
    def createRoute(routeeProps: Props,
                  routeeProvider: RouteeProvider): Route = {
        // Create 5 flight attendants
        val attendants = (1 to 5) map { n =>
            routeeProvider.context.actorOf(Props(newFlightAttendant),
                                         "Attendant-" + n)
        }
    }
}
```

```
// Register them with the provider - This is important.  
// If you forget to do this, nobody's really going to  
// tell you about it :)  
routeeProvider.registerRoutees(attendants)  
  
// Now the partial function that calculates the route.  
// We are going to route based on the name of the  
// incoming sender. Of course, you would cache this or  
// do something slicker.  
{  
    case (sender, message) =>  
        val Passenger.SeatAssignment(_, row, _) = sender.path.name  
        List(Destination(sender,  
            attendants(math.floor(row.toInt / 11).toInt)))  
    }  
}  
}
```

Above is the definition of our new `SectionSpecificAttendantRouter`. You should ignore all of the hard-coded numbers if they make you squeamish—neither of us would do such horrific things in the real world.

The goal of our new Router is to produce an instance of `Route`, which is defined as:

```
PartialFunction[(ActorRef, Any), Iterable[Destination]]
```

And the `Destination` is defined as:

```
case class Destination(sender: ActorRef, recipient: ActorRef)
```

You might note from the code, and from the definitions above, that the `createRoute()` method can actually return a host of sender/recipient pairs in the Iterable of Destinations, so while the world is essentially your oyster here, we keep it simple.

Testing the `SectionSpecificAttendantRouter`

It turns out that a custom Router is more difficult to test than you'd think. Based on what we've written in the `SectionSpecificAttendantRouter`, it would be nice if we could simply test the resulting `PartialFunction` from the `createRoute()`

method directly, but in order to do that we'd have to construct a RouteeProvider, which is non-trivial. So, we opt for testing the custom Router outside of the “unit” realm and go for a full integration test.

```
package zzz.akka.avionics

import akka.actor.{Props, ActorSystem, Actor, ActorRef}
import akka.testkit.{TestKit, ImplicitSender, ExtractRoute}
import akka.routing.RouterConfig
import org.scalatest.{WordSpec, BeforeAndAfterAll}
import org.scalatest.matchers.MustMatchers

// This will be the Routee, which will be put in place
// instead of the FlightAttendant
class TestRoutee extends Actor {
    def receive = {
        case m => sender ! m
    }
}

// The RouterRelay will tie the Router to the testActor so
// that we can see what's going on and verify it.
// Essentially, this will be our Passenger
class RouterRelay extends Actor {
    def receive = {
        case _ =>
    }
}

class SectionSpecificAttendantRouterSpec
    extends TestKit(ActorSystem("SectionSpecificAttendantRouterSpec"))
    with ImplicitSender
    with WordSpec
    with BeforeAndAfterAll
    with MustMatchers {

    override def afterAll() {
        system.shutdown()
    }

    // A simple method to create a new
    // SectionSpecificAttendantRouter with the overridden
```

```
// FlightAttendantProvider that instantiates a TestRoutee
def newRouter(): RouterConfig = new SectionSpecificAttendantRouter
    with FlightAttendantProvider {
    override def newFlightAttendant() = new TestRoutee
}

def relayWithRow(row: Int) =
    system.actorOf(Props[RouterRelay], s"Someone-$row-C")

val passengers = (1 to 25).map(relayWithRow)

"SectionSpecificAttendantRouter" should {
    "route consistently" in {
        val router = system.actorOf(Props[TestRoutee].withRouter(newRouter()))
        val route = ExtractRoute(router)
        val routeA = passengers.slice(0, 10).map { p => route(p, "Hi") }.flatten
        routeA.tail.forall { _.recipient == routeA.head.recipient } must be (true)
        val routeAB = passengers.slice(9, 11).map { p => route(p, "Hi") }.flatten
        routeAB.head must not be (routeAB.tail.head)
        val routeB = passengers.slice(10, 20).map { p => route(p, "Hi") }.flatten
        routeB.tail.forall { _.recipient == routeB.head.recipient } must be (true)
    }
}
}
```

You can see that we've set up a lot of plumbing to test this thing out, and just to make what we've done clear you can see the resulting layout in [Figure 10.5](#).

The idea behind the test is to strategically create a few passengers (mocked out with the Relay Actor) and have them send messages to the SectionSpecificAttendantRouter. The logic we wrote in the `createRoute()` method will then get invoked and the messages will route to the flight attendants (mocked out with the TestRoutee). The TestRoutee will then echo those messages back to their senders (i.e., the specific Relay in question) and the Relay will send the sender's ActorRef (i.e., the TestRoutee's ActorRef) to the `testActor`. Once the test function has collected the three ActorRefs from the Relays, it can verify that they are what they should be.

```
// This will be the Routee, which will be put in place
// instead of the FlightAttendant
```

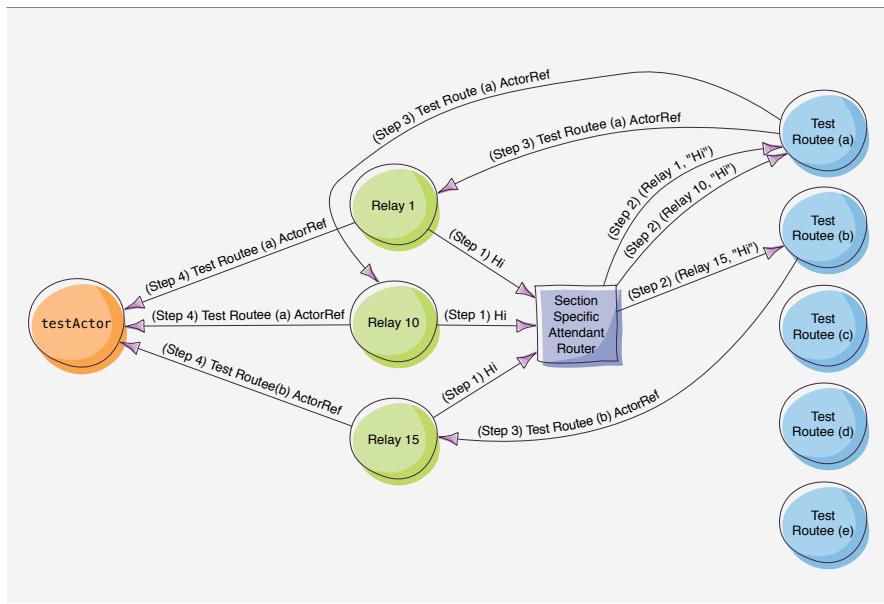


Figure 10.5 · In this layout for the `SectionSpecificAttendantRouter` test, the mocked pieces and messages are geared toward getting enough information into the `testActor` so we can verify what we expect to route where.

```
class TestRoute extends Actor {  
    def receive = {  
        case m => sender ! m  
    }  
}  
  
// The RouterRelay will tie the Router to the testActor so  
// that we can see what's going on and verify it.  
// Essentially, this will be our Passenger  
class RouterRelay extends Actor {  
    def receive = {  
        case _ =>  
    }  
}  
  
class SectionSpecificAttendantRouterSpec  
    extends TestKit(ActorSystem("SectionSpecificAttendantRouterSpec"))
```

```
with ImplicitSender
with WordSpec
with BeforeAndAfterAll
with MustMatchers {

override def afterAll() {
    system.shutdown()
}

// A simple method to create a new
// SectionSpecificAttendantRouter with the overridden
// FlightAttendantProvider that instantiates a TestRoutee
def newRouter(): RouterConfig = new SectionSpecificAttendantRouter
    with FlightAttendantProvider {

override def newFlightAttendant() = new TestRoutee
}

def relayWithRow(row: Int) =
    system.actorOf(Props[RouterRelay], s"Someone-$row-C")

val passengers = (1 to 25).map(relayWithRow)

"SectionSpecificAttendantRouter" should {
    "route consistently" in {
        val router = system.actorOf(Props[TestRoutee].withRouter(newRouter()))
        val route = ExtractRoute(router)
        val routeA = passengers.slice(0, 10).map { p => route(p, "Hi") }.flatten
        routeA.tail.forall { _.recipient == routeA.head.recipient } must be (true)
        val routeAB = passengers.slice(9, 11).map { p => route(p, "Hi") }.flatten
        routeAB.head must not be (routeAB.tail.head)
        val routeB = passengers.slice(10, 20).map { p => route(p, "Hi") }.flatten
        routeB.tail.forall { _.recipient == routeB.head.recipient } must be (true)
    }
}
}
```

So the messages from both the first and second Relay should have gone to the same TestRoutee, and the message from the third Relay should have gone to a different TestRoutee than the first and second message.

10.7 More You Can Do with Routers

Routers have many more possibilities, which you'll start to recognize as we explore more of what Akka has to offer. Here are some to consider:

- When we get into remote Actors, you can immediately see the potential of the ScatterGatherFirstCompletedRouter, as well as implementing some rudimentary load balancing across machines, or even some fault-tolerance.
- The business logic possibilities are pretty huge. Imagine an evolving protocol, where the messages contain the version of the protocol. As things evolve, you can put in a Router or even a family of cascading Routers that find the right version of the business logic to handle that protocol level. This helps eliminate the hideousness that often accompanies maintaining backward-compatibility in our code as protocols change.
- The Akka documentation's example of a custom Router involves the counting of Votes for the Republican or Democratic party, respectively. You can grow this solution while incorporating the same ideas behind the protocol versioning earlier, and simply segment your application in any manner you see fit. It's a clear win for isolating parts of your application logic from each other.
- You can also create a configurable Router, which we could have done with the SectionSpecificAttendantRouterSpec, but didn't. You follow the same kind of pattern, but you have to have a constructor that accepts the `com.typesafe.config.Config` object, which allows it to be dynamically configured. Consult the Akka documentation on Routing for more information.
- You can even code up your own configurable Resizers as well!

10.8 Chapter Summary

Remember a long time ago when I said that you'd be glad to leave the warmth of the type system behind in favour of the untyped Actor? Well, routing is a *huge* win for making that concession. Nice choice you made there.

You've gained more understanding about:

- Akka’s Routing concepts and methodology and how easy it is to use Akka’s pre-defined Routers, both from the configuration system and from code.
- How to structure your application so as to divorce you, the programmer, from the administrators (which may also be you) with respect to tuning your application’s performance.
- How to write your apps so that you can make some pretty drastic changes to it “in the field” without requiring a rebuild and reship. Empowering the customer to be the master of his or her own destiny is a *very* good and powerful thing.
- How to create your own Routers and you have some ideas as to why you might want to do that.
- The foundational aspects of Actor programming; you’ve seen more testing concepts and the power of substitutability when it comes to the untyped ActorRef. These concepts, and the acrobatics we can now perform when *structuring* our Actor applications, is absolutely vital to powerful Actor programming.
- The TestProbe and how Akka logs using the ActorSystem’s Event Stream.

You’re really starting to get a deep understanding of how the Akka programming paradigm works and you should have a lot of confidence right now. In fact, I would recommend that you take that high level of confidence and put it to good use; if you’re not married or otherwise attached, head out to your local singles bar and pick up. Everyone’s attracted to confidence at this insane level!

Chapter 11

Dispatchers and Mailboxes

We're getting ready to put Actors aside for the moment and move on to some of Akka's other features, but before we do that we need to learn a little bit about what keeps Akka ticking.

Way back in [Chapter 5](#), we discussed the Dispatcher and the Mailbox without going too deeply into what they are, what purpose they really serve, and what control you have over them. This chapter will help bring closure to our understanding of Actors while, at the same time bridge us over into the next topic of Futures. We'll also reference Routing, where we describe the “missing” Router.

11.1 Dispatchers

Up until now, we haven't really cared *how* our code was executed, so long as it worked. While you don't really have to understand how Akka does what it does, since the defaults work so well, it can certainly help you tune your application's execution to run ever so sweetly.

In case you hadn't figured it out yet, the job of actually executing the code you write, managing the threads, and all of that goodness is handled by the Dispatcher. The Dispatcher hides all (well, most) of the complexity from you, so that you can go about your business of writing code. You can tune the Dispatcher in a few different ways as well as supply different Dispatchers for different parts of your system. Let's review the Dispatchers that Akka provides.

The Dispatcher

The main Dispatcher, which is also the default Dispatcher, is simply called *Dispatcher*. It might become unclear as to when we're describing the concept Dispatcher, and the concrete default implementation Dispatcher. As such, I'll adopt a new name for the latter; we'll refer to the concrete default implementation as the *Event-Based Dispatcher*, and the concept will simply be *Dispatcher*.

There's a reason that the Event-Based Dispatcher is the default: it's *awesome*. It's generic, can work with any type of Actor and any type of Mailbox, and defaults to using the JSR166 Fork-Join Pool. This together means ultimate flexibility and blazing speed.¹ The Event-Based Dispatcher allows you a great level of freedom when writing your concurrent code. As long as you keep your message handlers short and sweet, the Event-Based Dispatcher will treat you very well.

PinnedDispatcher

There are times when you might want to deviate from the default. For example, you don't want to share your threads with 10,000,000 other Actors. Maybe you require an Actor to always be first in the queue for thread time, or you want to ensure that it will simply never starve for CPU attention. This is why the PinnedDispatcher was created; any Actor that you assign to the PinnedDispatcher will have a dedicated thread pool of size 1. That Actor will be the only Actor assigned to that thread pool and is therefore guaranteed thread time on that pool.

The PinnedDispatcher is good for solving the problems previously stated, but it makes an absolutely horrific default. If you have 10,000,000 Actors, you can't possibly use a PinnedDispatcher since you can't allocate 10,000,000 dedicated threads. Even if the JVM would let you do this before it tossed its cookies all over your motherboard, you'd spend more of your time context switching than doing anything else.

So, use these with care. The average program shouldn't need to house more than half a dozen Actors on a PinnedDispatcher. If you have a situation where you're doing more than that, either rethink your solution, rework your

¹We won't quote numbers in this book. Benchmarking is a black art, dripping with Voodoo and rubber chickens. It's *fast*, but if you want to really quantify it, grab some rubber chickens and go hog-wild.

problem, or if you really have a legitimate case, please toss me an email; I'd love to hear about it.

BalancingDispatcher

This one was “missing” when we talked about routing back in [Chapter 10](#). What came close was the SmallestMailboxRouter, which would find the Actor in its pool that had the smallest Mailbox and insert the message into that Mailbox. But it couldn’t “steal” work from one Actor and give it to another Actor that happens to be idle.

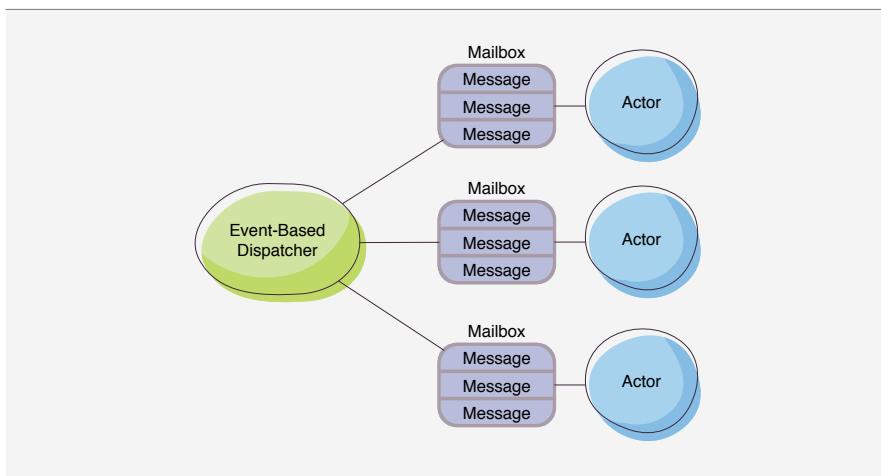


Figure 11.1 · The conceptual view of the Event-Based Dispatcher with respect to its Actors and the Mailboxes that contain their messages

The BalancingDispatcher fills this need. However, it can only do so because it is far more limiting than the previous two Dispatchers with respect to the Actors with which it can work. Whereas the Event-Based Dispatcher looks something like [Figure 11.1](#), the BalancingDispatcher looks more like [Figure 11.2](#). The fact that Akka decouples the Mailbox from the Actor and the Dispatcher means that no “stealing” needs to happen. Since only one Mailbox is shared between all of the Actors to which the BalancingDispatcher will dispatch, the BalancingDispatcher can pick any one it wishes without disturbing the others.

However, as you probably have already guessed, the Actors that the BalancingDispatcher feeds need to have the same Actor implementation. Truth-

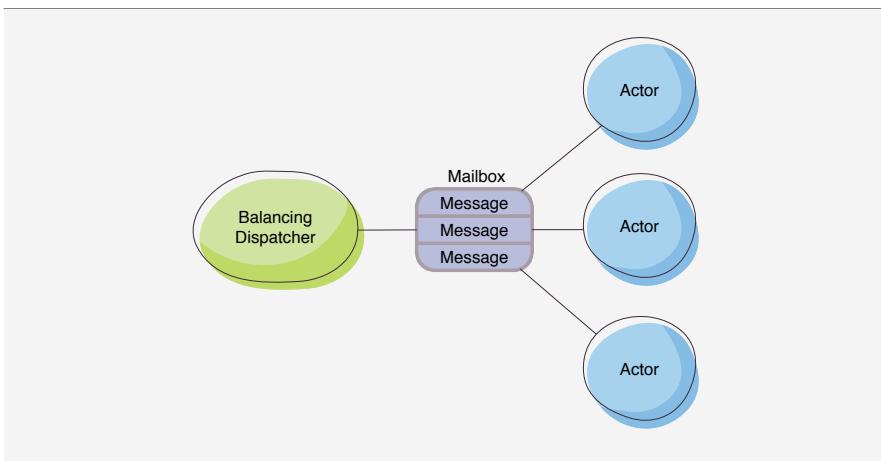


Figure 11.2 · The conceptual view of the BalancingDispatcher with respect to its Actors and the Mailbox that contains the messages for all of them

fully, nothing's stopping you from having different implementations for each Actor, but in practice you wouldn't do this. From your code's perspective, the BalancingDispatcher simply sends messages to one of your Actors at random. You can't guarantee which implementation will be chosen at any given time, so you'd simply make all of them identical to remove the complexity of the perceived randomness.

It shouldn't come as a surprise that a BalancingDispatcher's job isn't "intelligent" routing of messages; that is more suited to a Router.

CallingThreadDispatcher

The last Dispatcher that ships with Akka is almost something we shouldn't even mention... really. OK, I'll mention it, but for Ra's sake *don't use it*. The CallingThreadDispatcher was created to write deterministic tests and is, in fact, the dispatcher you use when you create an instance of the TestFSM-Ref, as we saw in [Chapter 9](#). This Dispatcher has no concurrency; it relies on the calling thread for execution. This means that your tests don't have to worry about concurrency, which is great, but if you used it in production code, you wouldn't have concurrency.

I've spoken with some who think they could be "clever" in using this Dispatcher in production code to serialize this, that, or the other thing for

some reason or another. *Don't*. Once you start taking a system that is designed to be concurrent across multiple threads, and then start synchronizing that across one thread, you're going to open yourself up to a world of pain and Akka isn't going to be there to help you. If you need to synchronize work across a number of algorithms, then Akka's got you covered. Use messages to sequence work, use Futures (to be seen soon), Dataflow (to be seen soon), etc... This is what Akka does! Don't try to get it to do what it already does in ways it's not intended to do it.

11.2 Dispatcher Tweaking

The Dispatcher has many of its parameters defined in the configuration system, which means you can tweak it. Actually *anyone* can tweak it in the running system, allowing people other than you to tune your application for performance. *That's nice*.

We won't go through all of the configuration options available to Dispatchers and their underlying thread choices—you can find that in the fantastic Akka reference documentation—but we will look at some of the key concepts. Let's make a fictitious Dispatcher in configuration to look at the possibilities.

```
zzz.akka.investigation {  
    # "a-dispatcher" is the name with which we refer to it  
    a-dispatcher {  
        # You could also use BalancingDispatcher, PinnedDispatcher or  
        # your own derivation of MessageDispatcherConfigurator.  
        type = "Dispatcher"  
  
        # You could also use thread-pool-executor  
        executor = "fork-join-executor"  
  
        # By increasing this value, you can maximize thread usage, at  
        # the cost of fairness.  
        throughput = 10  
  
        # Since we've chosen the executor to be a fork-join-executor, we  
        # need to configure it here  
        fork-join-executor {  
            # The minimum number of threads to have  
            parallelism-min = 2
```

```
# The scaling factor for calculating the number of threads to
# allocate based on the hardware capabilities. The formula
# used is "ceil(number of processors * parallelism-factor)"
parallelism-factor = 2.0

# The maximum number of threads to allocate
parallelism-max = 32

}

}

}
```

As stated, that's not everything that you can do with a Dispatcher configuration, but it hits some highlights. One of the most important properties in the Dispatcher configuration is throughput. When the Dispatcher grabs a thread and starts dispatching work on that thread, we can specify several messages that it can process before moving on. As the comment states above, you can maximize thread usage at the cost of "fairness."

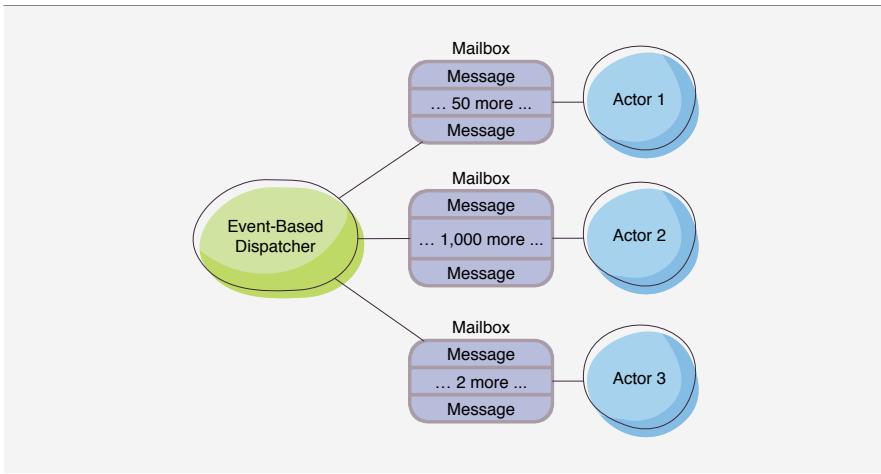


Figure 11.3 · By changing the throughput value in the Dispatcher configuration, we can alter how quickly these Mailboxes will drain.

In Figure 11.3, you can see that the sizes of the Mailboxes are different between Actors 1, 2, and 3. If we set the value of throughput to 1, then Actor 3's Mailbox will clearly drain before either Actor 1 or 2's Mailbox. However, if we set the value of throughput to 100, then Actor 1's Mailbox

may completely drain before Actor 3's, assuming that Actor 1's Mailbox gets thread time before Actor 3's.

That's the essence of the throughput property. It tells Akka to waste less time switching between Mailboxes and more time dispatching work from each Mailbox individually. However, this is less "fair" with respect to getting thread time per Mailbox. From the example in [Figure 11.3](#), we can see that Actor 3 is potentially treated unfairly, since its tiny Mailbox can't get thread time until we've completely drained Actor 1.

Setting a high value for throughput will increase the raw speed of message processing in some cases, but may also manifest certain latencies and create the feeling of "burstiness" from time to time, where work visibly gets done in chunks, with time gaps in between. If those aren't factors, then you may want to set throughput higher than you would otherwise.

Modifying the Default Dispatcher Configuration

The reference .conf that ships with the Akka Actor package holds the configuration for the default Dispatcher. You can always change the values of the default Dispatcher by overriding the configuration yourself. For example, if you wanted to change the value of throughput for the default Dispatcher, then you would put the following into your application.conf:

```
akka.actor.default-dispatcher.throughput = 20
```

You can do this for any parameter that you see for any configuration that you see. This doesn't merely apply to Dispatchers.

11.3 Mailboxes

Mailboxes are another one of those things we've been using for a long time, but haven't talked about much. They don't have a ton of complexity, but they do offer another component of which you can configure, modify, and even create your own implementation.

As we've seen, the Mailbox contains messages that have yet to process. The message that goes into the Mailbox is wrapped in an envelope that carries the sender along with it. The sender comes from when the message dispatches to your Actor.

In essence, the Mailbox is just a queue, or more accurately it *has* a queue, but the distinction here is of little value. Akka allows you to change the nature of the Mailbox in the Dispatcher's configuration. As with the Dispatcher, there are several pre-existing implementations from which you can choose.

UnboundedMailbox

This is the default. It's backed by a `java.util.concurrent.ConcurrentLinkedQueue`, does not block during the enqueue operation, and is not bounded by size. This is the default Mailbox and should serve you very well. The only time it should really be a problem is when it grows without bound and eats up all of your memory.

However, if you have a situation where the `UnboundedMailbox` fills up and causes an `OutOfMemory` Exception, then what you're probably seeing is a *symptom* of a problem, not the root cause and thus the fix lies elsewhere.

BoundedMailbox

The `BoundedMailbox` is an alternative to the `UnboundedMailbox` in that it can only contain a maximum number of messages. When the Mailbox is full, the next enqueue operation will block the calling thread for a specified period of time (defaults are in the configuration, which you can override).

This helps solve the `OutOfMemory` Exception problem, but it introduces another problem: *false errors*. Essentially, you can characterize the problem with the question, “What maximum size should I allow for the Mailbox?”

That's a tough question to answer most of the time. Perhaps situations exist where the Actor relies on a service that is slow for a few minutes. This may cause the Actor's Mailbox to fill up, and thus throw an error at someone eventually. But what happens when, two seconds after the error is thrown, the service speeds up again and drains the Mailbox very quickly? In most cases, that error is more of a pain than anything else.

You might want to have a `BoundedMailbox` when you have time limits elsewhere in the application; the idea being that the message is of no value if you can't enqueue it in the next 5 seconds. In that case, you'd want the error even though 500 milliseconds from now it could have worked, because it would still be too late if it had worked.

UnboundedPriorityMailbox and BoundedPriorityMailbox

The UnboundedPriorityMailbox and BoundedPriorityMailbox are “priority queue” versions of the previous two. When you create one of these, you specify how Akka should prioritize different messages and they will route properly to the Actor.

To create the logic that maps message types to priority levels, we use the PriorityGenerator, which gives a convenient method for this definition:

```
package zzz.akka.investigation
import akka.dispatch.PriorityGenerator
case class HighPriority(work: String)
case class LowPriority(work: String)
val = myPrioComparator PriorityGenerator {
    // Lower numbers mean higher priority
    case HighPriority(_) => 0
    case LowPriority(_) => 2
    // Default to "medium" priority
    case otherwise => 1
}
```

As you can see, the lower the number, the higher the priority. However, just specifying the PriorityGenerator isn’t enough. We need to create a class that holds the value created from the PriorityGenerator, so that we can specify it in the Dispatcher configuration.

```
package zzz.akka.investigation
import akka.actor.ActorSystem
import com.typesafe.config.Config
import akka.dispatch.UnboundedPriorityMailbox
class MyPriorityMailbox(settings: ActorSystem.Settings, config: Config)
    extends UnboundedPriorityMailbox(myPrioComparator)
```

And now we can specify the Mailbox in the Dispatcher configuration:

```
zzz.akka.investigation {
    my-priority-dispatcher {
```

```
executor = "fork-join-executor"
type = Dispatcher
mailbox-type = "zzz.akka.investigation.MyPriorityMailbox"
}
}
```

And that's all there is to it. When you assign an Actor's Dispatcher to `zzz.akka.investigation.my-priority-dispatcher`, it will create the `MyPriorityMailbox` under the covers and you're good to go.

Creating Your Own Mailbox

You can actually create your own Mailbox as well, which gives you an immense amount of control over how you want to handle your message queuing. We could come up with a toy example here, but there's more value in a real world application of this. Unfortunately, it's much too involved to go into detail here, but we can talk about the idea and show how it's hooked up.

You could say that the `BoundedMailbox` has a problem in that it's almost a blocking brick wall. At some point, you may find that enqueueing a message takes two nanoseconds, and then immediately following that, it takes 10 seconds before it times out and throws an Exception. What if you wanted to slow clients down instead of letting them run at full speed until the moment of death when you slam them in the face with an Exception?

The concept is embodied in the notion of a `PressureQueue`.² You can find the code for it at <https://github.com/derekwyatt/PressureQueue-Concept>. It's a bit rough, but the idea seems to work out OK.

The idea is that it takes longer to enqueue a message as the queue size increases. The size increases because the consumer of the queue's elements can't process them as quickly as the clients that are enqueueing the data. The consumer would be made happier if the clients would just slow down, so that it can do its work without having to force an Exception on them.

Let's say your database experiences a heavy load for a few minutes. If the queue continues to accept data at an astronomical rate, this could create a backlog of database activity that won't reach a state of stability for *hours*.

²I remember seeing a presentation that illustrated this concept a few years ago, but I can't remember what it was called or even who made it, so I can't give credit to the idea. After much searching, I've come up with nothing, so I can't give credit where it's due. Whoever you are, my apologies.

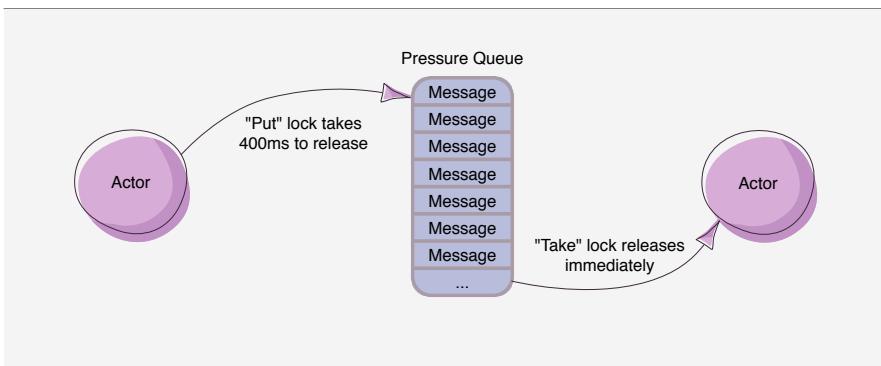


Figure 11.4 · A simple diagram of the PressureQueue: The Actor holds the “put” lock as long as the pressure algorithm demands. This length of time is a factor of the current queue size. The “take” lock releases immediately so that the queue’s consumer can empty it as quickly as possible.

(I’ve seen it happen). That’s a high price to pay for what is, essentially, only a few minutes of real pain. If the queue that accepts the work simply slows everyone down, and that slowdown is gracefully propagated back through the chain as far as it needs to be, then the period of high load will smooth out instead of creating an enormous spike of death.

Configuring the New Mailbox

Let’s say you have this PressureQueue. How do you stick it inside a Mailbox implementation so that Akka can instantiate it inside a Dispatcher? It’s really pretty simple actually, and we can show it in a short amount of code:

```
package zzz.akka.investigation

import akka.actor.{ActorContext, ActorSystem}
import akka.dispatch.{Envelope, MessageQueue, MailboxType,
    QueueBasedMessageQueue,
    UnboundedMessageQueueSemantics}

import com.typesafe.config.Config

// Our magical PressureQueue
import zzz.akka.investigation.PressureQueue

class PressureMailbox extends MailboxType {
```

```
// This constructor signature must exist, it will be
// called by Akka
def this(settings: ActorSystem.Settings, config: Config) = this()

// The create method is called to create the MessageQueue
final override def create(owner: Option[ActorContext]): MessageQueue =
  new QueueBasedMessageQueue with UnboundedMessageQueueSemantics {

    // Construct the PressureQueue such that the pressure
    // algorithm kicks in after the queue reaches size 10,
    // and the algorithm we use calculates the wait time
    // using the squared size of the queue resolved the
    // milliseconds
    final val queue = new PressureQueue[Envelope](10,
      PressureQueue.SQUARED_MILLISECONDS)
  }
}
```

Note that the element type that the PressureQueue holds is an Envelope. The Envelope contains your message and the sender, as described earlier.

And there you have it. We've created an entirely new type of Mailbox that behaves differently than any other Mailbox that Akka currently ships with and it really isn't all that hard. It's Unbounded and it has the blocking semantics that we've coded up. Used strategically, this could smooth out those load spikes we may get due to bottlenecks that exist in whatever components with which we have to work.

11.4 When to Choose a Dispatching Method

There's no hard-and-fast rule about which dispatching method you should choose for any particular problem, but there are some basic themes that might point you in the right direction:

Low Latency You should use the PinnedDispatcher when you have some requirements for low-latency in isolated parts of your app. For example, if you've put a maintenance probe into your app as an HTTP server, then you might want to pin that Actor to a dedicated PinnedDispatcher. This would ensure that you're not fighting to use a common event-based thread when performing some critical maintenance. However, you certainly can't put everything on a PinnedDispatcher, since

that would negate the whole idea of what Akka is helping you achieve. Keep the number *very* small here.

A “Bulk-Heading” Dispatcher You can use Dispatchers (generally, an Event-Based Dispatcher) for isolating different sections of your app from other sections. This is most common when you’re stuck using synchronous I/O in your app. Just as you would isolate the synchronous I/O to a separate thread pool from your CPU work, you can dedicate a Dispatcher for the same purpose. This will keep things responsive throughout your app, and it also gives you some external tuning of the I/O work vs. the CPU work via the configuration file.

Number Crunching When you have a bunch of pure CPU work to do, you want to keep your Actors as busy as possible, and this is where you want to employ the BalancingDispatcher.

Message Durability A feature of Akka’s Actors that we don’t discuss in this book is where the messages in an Actor’s queue are backed by a durable message store. If you have an Actor that uses this feature, then you will want to isolate it from Actors that don’t have a durable message store. The reason for this is the same reason for the I/O separation mentioned earlier.

11.5 Chapter Summary

Well, that was a long time coming. We’ve been working with the Dispatcher and the Mailbox since we thought about our first piece of Actor code, but never had a decent understanding of what they really were until now. Hopefully, all of the previous groundwork that we laid down made this chapter easier to consume.

There are some key takeaways here:

- The Dispatcher is the heart of the execution system in Akka. It’s where your threads are and its what decides when and how things execute.
- You can configure the Dispatcher or change it entirely. That throughput property is pretty important, so play with it and see what you get.

- The BalancingDispatcher helps you get that “work stealing” load balancer in your application if you need it. This will save you from having to figure out how to balance things on your own.
- The Mailbox’s role should now be pretty clear; the fact that it is decoupled from your Actor and the Dispatcher allows you to specify whatever you’d like.
- You can swap out all of the things we’ve seen for your own brilliant ideas. If you have a requirement that simply isn’t met by what Akka currently provides, it’s not at all difficult to slide your own implementations in place.

Above all, remember that the provided defaults are *extremely good*. You shouldn’t find yourself trying to modify things all that often and if you do, ask yourself why. The answer to that question had better be pretty darn good!

Remember that the path to many pink slips is paved with the intentions of programmers to increase performance. Don’t get too clever too quickly.

Chapter 12

Coding in the Future

We've come to the second great axis of Akka's concurrency paradigm, the *Future*. Futures have been around for a while now, but you've probably never seen them like this. These are Futures on steroids; tiger steroids, laced with the blood of Superman, dripping with the testosterone of Batman, and infused with the power of Green Lantern's Ring. If you're not ready to have your brain set on fire, close this book, curl up in the fetal position, and sing *Twinkle, Twinkle Little Star* softly to yourself until you're good to go.

The Future and the Actor share a certain number of similarities as well as differences, but they are also designed to work together very well. In this chapter, we'll see these similarities and differences as well as explore some patterns for tying the two components together.

12.1 What Is the Future?

One goal of the original Future design was geared toward bridging the gap between synchronous and asynchronous code. Legacy synchronous code that was forced to work with newer asynchronous APIs immediately had a problem: *How do you get a return value from a function call that won't return that value on the calling thread?* Coders needed a way to have their sequential code stay sequential in the face of an asynchronous function call, and the Future turned out to be a decent general mechanism for making that happen.

It's not much more complicated than what you see in [Figure 12.1](#). The sequential code wraps an asynchronous call inside of a Future (or the API call *returns* that Future), which allows the code to cross whatever threads it

wishes. It then returns the value to the caller's thread on which it is currently blocked while waiting for the result. The call that does the blocking is usually called `get()`.

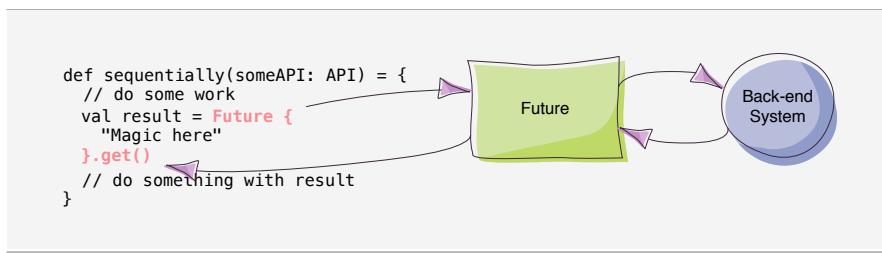


Figure 12.1 · A simple depiction of using a Future to synchronize an asynchronous API into pre-existing sequential code

Over the years, some enhancements have been made to create callbacks, and more doo-dads here and there, but essentially the concept has remained the same. The Future is a pocket of concurrency that executes independently of the calling thread and ultimately represents the “return value” from that code (which could obviously be `void` or `Unit`).

An Actor is a live object that can interact between itself and any number of other objects, functioning as long-lived message processors with potentially changing state. The Future, on the other hand, is intended as a one-shot, single-purpose entity that is only addressable by the chunk of code waiting for the Future’s Promise, and the other chunk of code that fulfills that Promise.

Akka Futures Are Composable and Functional

One of the latest concepts appearing in Futures¹ has been the ability to compose them within one another, which makes them much more functional. This is where we see a lot of the power behind Akka’s Futures. Essentially, they let you create *pipelines* of concurrent code that can weave into other code with ease.

¹Akka is not the first implementation to expose this type of feature set.

Futures Need an Execution Context

The Future doesn't just happen all by itself; it needs somewhere to actually execute, and that execution context is very much like the Dispatcher we met in [Chapter 11](#). In fact, if you happen to have a Dispatcher instance handy, you can use it to execute a Future.

[Listing 12.1](#) shows a bare-bones way to use a Future, outside of any help from existing Dispatchers, ActorSystems, or Actors, so we can calculate some Fibonacci numbers. It calculates the 100th Fibonacci number² inside of a Future, followed by a horrific thread-blocking *wait* and an assert.

The most important line of code for this example is...

```
val futureFib = Future { fibs.drop(99).head }(execContext)
```

...and do you know what's ugly about that? The ExecutionContext, that's what.

We don't want to pass that around all the time because that would be annoying. Fortunately, the API is a bit slicker than that, and will accept an *implicit* ExecutionContext, so we can change this to read:

```
val execService = Executors.newCachedThreadPool()
implicit val execContext =
    ExecutionContext.fromExecutorService(execService)
val futureFib = Future { fibs.drop(99).head }
```

The fact that the ExecutionContext is *implicit* is fairly awesome because it means that, from a practical perspective, we never really need to worry about it. Since Dispatchers are ExecutionContexts, we generally have one in scope all of the time, but you can always specify one explicitly if you need to, just like we have above.

12.2 Don't Wait for the Future

You've seen a lot of calls to `Await.result()` so far in this book, and you'll probably see more, but you shouldn't use them in anything but test code because they're *evil*. `Await.result()` does the same thing as the old `get()` call on a Future—it blocks the calling thread.

²0 is the first one, right?

```
package zzz.akka.investigation

import org.scalatest.{WordSpec, BeforeAndAfterAll}
import org.scalatest.matchers.MustMatchers

class MultiExecutionContextSpec extends WordSpec with MustMatchers {
    import scala.math.BigInt
    lazy val fibs: Stream[BigInt] = BigInt(0) #:: BigInt(1) #::
        fibs.zip(fibs.tail).map { n => n._1 + n._2 }
    "Future" should {
        "calculate fibonacci numbers" in {
            import scala.concurrent.{ExecutionContext, Future, Await}
            import scala.concurrent.util.duration._
            import java.util.concurrent.Executors
            // We need to create the ExecutionContext, which we can build from an
            // existing ExecutorService from plain ol' java.util.concurrent
            val execService = Executors.newCachedThreadPool()
            val execContext = ExecutionContext.fromExecutorService(execService)
            // We pass the ExecutionContext to the Future on which it will execute
            val futureFib = Future { fibs.drop(99).head }(execContext)
            // We then use the Await object's result() function to block the current
            // thread until the result is available or 1 second has passed
            val fib = Await.result(futureFib, 1.second)
            // Just make sure it's cool
            fib must be (BigInt("218922995834555169026"))
            // Shut down the ExecutionContext or this thread will never die
            execContext.shutdown()
        }
    }
}
```

Akka provides so many ways for you to compose your functionality inside other Futures, as well as having other great concurrency constructs (such as Actors), that you should never need to block a thread with anything but someone else's synchronous function call.

12.3 Promises and Futures

Technically speaking, the Future is only one half of a two-part relationship. The two parts make up a conduit of sorts, where the requester holds the Future and the servant holds the Promise; [Figure 12.2](#) shows the directionality of this relationship.

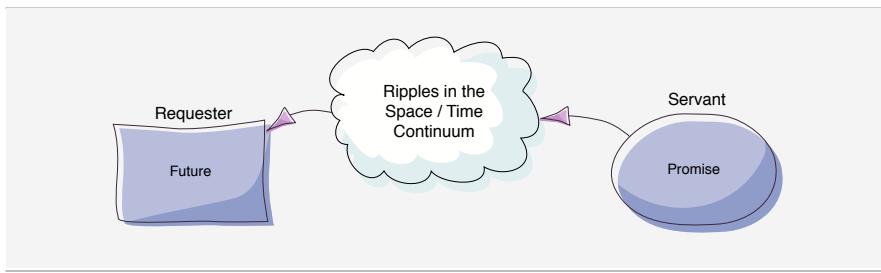


Figure 12.2 · Futures are fulfilled by a Promise, which is held by a servant that gives the Future to the requester. This creates the conduit through which the servant and requester communicate.

In terms of code, and ignoring the whole concurrency issue altogether, we can see how this relationship works:

```
import scala.concurrent.Promise  
  
// Create a Promise  
val promise = Promise[String]()  
  
// Get the associated Future from that Promise  
val future = promise.future  
  
// Successfully fulfill the Promise  
promise.success("I always keep my promises!")  
  
// Extract the value from the Future  
println(future.value)  
// Prints: Some(Right(I always keep my promises!))
```

This code sample isn't concerned with concurrency at all. We have a single thread that creates a Promise, from which it extracts the Future, giving it away to someone who wants it. At some point, the Promise is fulfilled and the usual magic can now occur in the Future. Since we're not concerned with concurrency here, and we *know* that the Promise has been fulfilled, it's pretty easy to extract the resulting value directly.

It's fairly rare for you to ever create Promises on your own, since these are generally created for you without you even knowing about it. Still, it's useful to understand how it works, and certainly important to understand if you ever need to create the Promise on your own.

Success and Failure

The resulting value in the Future is always an instance of Either. Indeed, we can see from the previous code snippet that the result of a successful fulfillment is a Right instance. This follows the convention of Either, where Right is success and Left is failure (when Either is used to model such things).

Therefore, it's not surprising that there are three main ways to complete a Promise:

- `success(result: T)`: As we've already seen, this completes the Promise successfully.
- `failure(exception: Throwable)`: The Throwable is the type used to fail a Promise, which maps quite well into the code we generally write.
- `complete(value: Either[Throwable, T])`: It shouldn't be a big shock to find out that the previous two mechanisms are merely sugar for this more general method.

There is one other way to complete a Promise and that's with another Future. This is one of the things that lets us chain these asynchronous bits of code together.

- `completeWith(other: Future[T])`: One of the ways to keep things functional.

However, this is not a function you should use directly, under normal circumstances. Akka uses it internally to combine Futures that are composed

with the more usual functional methods. We only discuss it here because you'll see it when you're looking at the API docs, and you should know that it's not something you're expected to use directly.

That's all we'll cover on Promises, since you deal with them so rarely in comparison to Futures. The Akka API documentation explains everything else you might need in order to create and manipulate Promises, so don't be afraid to look at the Promise documentation when you need it.

Subtle Sequentialism

There's something very important to understand when it comes using combinators to combine Futures with the for-comprehension sugar. The sequentialism is pretty obvious when you see the following:

```
val result = Future {  
    5  
} map { i => // has value 5  
    11  
}  
// 'i' is ignored, so the value of 'result' is Future(11)  
  
...or...  
  
val result = Future {  
    5  
} flatMap { i => // has value 5  
    Future {  
        11  
    }  
}  
// 'i' is ignored, so the value of 'result' is Future(11)
```

The code in the call to `flatMap` doesn't execute until the completion of the Future on which it is called. Even though the second Future completely ignores the return value of 5, it still runs *after* the 5 computes. Now let's rewrite the second version of the above code, using the much more pleasing for-comprehension:

```
for {
```

```
i <- Future(5)
j <- Future(11)
} yield j
```

Here, the beauty of the for-comprehension sugar can work against us. The for-comprehension hides all of that ugly nesting from us, making it look like these are on the same level, but nesting is indeed happening here. What might look concurrent, isn't.

To get them to run concurrently, we need to pre-define them so that the execution happens at the same level, but the binding of the results happens inside the nested flatMap.

```
val fiveFuture = Future(5)
val elevenFuture = Future(11)
val result = for {
    i <- fiveFuture
    j <- elevenFuture
} yield j
```

Now the `fiveFuture` and the `elevenFuture` run concurrently. The for-comprehension merely binds their return values together into a final Future (which is, of course, what it always does), but because they are pre-created, they can already be running. Effectively, we have this:

```
val result = fiveFuture.flatMap { i =>
    elevenFuture map { j => j }
}
```

instead of:

```
val result = fiveFuture.flatMap { i =>
    // Obviously won't be created until Future(5) is finished
    Future(11) map { j => j }
}
```

The last thing to remember is that this can show up in very subtle ways. For example, all we have to do is change the previous code to be defined as `defs` instead of `vals`.

```
def fiveFuture = Future(5)
def elevenFuture = Future(11)
val result = for {
    i <- fiveFuture
    j <- elevenFuture
} yield j
```

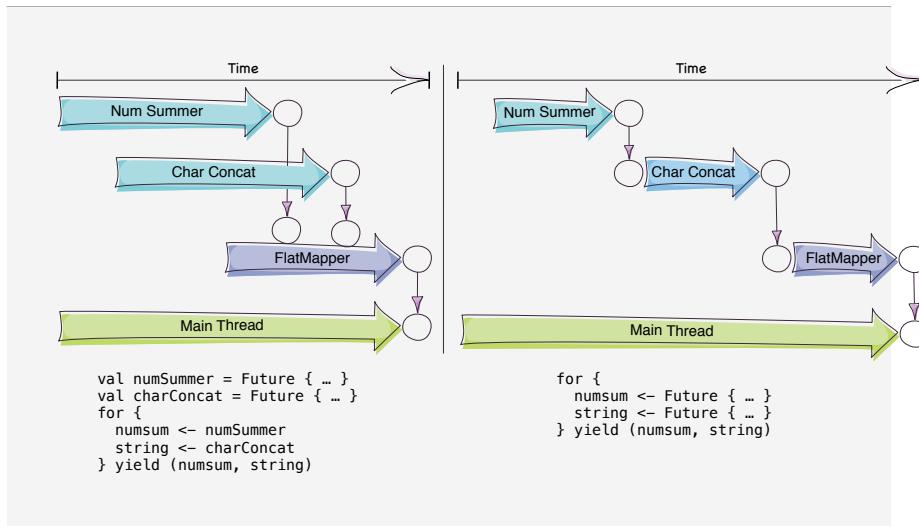


Figure 12.3 · There's a pretty clear difference between flatMapping pre-existing Futures and creating them on the fly.

And now we're back to sequential code. Again, *keep your eyes open* when you're writing this kind of code to make sure that you're getting concurrency where you need it. The difference should now be as clear as Figure 12.3.

Using filter

We left `filter` out of the usual `map/flatMap/filter` triad for a reason... it's *weird*. The most natural usage of `filter` is in collection-oriented programming. Something like this is pretty ubiquitous:

```
val evens = (1 to 42) filter { _ % 2 == 0 }
// evens == Seq(2, 4, 6, 8, 10, ...)
```

Or even with something like Options, it's pretty clear what should happen:

```
val number = Some(4)
// identity == Some(4)
val identity = number.filter { _ % 2 == 0 }
// none = None
val none = number.filter { _ % 3 == 0 }
```

Right? This makes sense because there's *something* left when these filter operations result in jack squat. If the predicate of the Option filter returns false, then we can return None. If the predicate returns false for every element of a collection, then we can return an empty collection.

But what about when the element contained is *code*? What is the result of the contents of a Future on which a filter predicate evaluates to false? What should the result be? With Option, it's an instance of Option. With Seq, it's an instance of Seq. By contract, the result of filter() must be a Future. But it can't be the identity, since the result of predicate has returned false; it has to be something else.

The only thing it can be is a *failure*. Failures are represented as exceptions that manifest when the Future is evaluated, so what we'll see is:

```
// Nothing bad happens when we assign the Future to 'oops'
val oops = Future { 5 } filter { _ % 2 == 0 }

// It's when the value of the Future is realized where we
// have the problem
Await.result(oops, 1.second)

java.util.NoSuchElementException:
  Future.filter predicate is not satisfied by: 5
```

Note that this is entirely different from filtering what the Future *contains*. In this case, we're filtering the *Future*, not its contents. For example, we don't use Future's filter here:

```
val no0ops = Future { 1 to 10 } map { seq =>
  seq filter { _ % 2 == 0 }
}
Await.result(no0ops, 1 second)
// evaluates to Vector(2, 4, 6, 8, 10)
```

In this case, we're filtering the Future's contents by mapping the Future to transform its result using the `filter` on the value the Future contains. That's a totally different box of frogs.

So, the `filter` method's purpose is to evaluate the Future's result, and either return that same Future or return an Exception. Got it? Good.

Using `transform`

The `transform` method offers another way to create Futures from Futures, but it allows you to transform the successful result as well as the `Throwable` at the same time. Assuming we have some cool function that retrieves some String data from a given URL, we can try to transform the result into an Int, and also provide a different Exception if things fail.

```
getRemoteString("http://somewhere/value.json").transform(  
    s => s.toInt,  
    t => new Exception("Couldn't get yer data: " + t))
```

The returned value from the `transform` call is a new Future that will result in either the transformed value or the transformed Exception, depending on what ends up happening.

Handling Future Failures

Futures are encapsulated pockets of concurrency, and that always makes detecting and processing failures interesting. The Future itself can't just throw an Exception; who would it throw it to? It's running on some independent thread somewhere, so the only person who can catch it is the guy who's running it, and he has no clue what the Future's real deal is. So throwing the Exception is out of the question.

We've already seen that the ultimate value of a Future (i.e., the actual `value` method on the Future) is an `Option[Either[Throwable, T]]`, where '`T`' is the type returned by the code that the Future will execute. This definition prepares for any success or failure.

Several strategies deal with failures, starting with the obvious one:

```
val oops = Future(5).filter { _ % 2 == 0 }  
val result = try {  
    Await.result(oops, 1.second)
```

```
    } catch {
      case e => 5
    }
  result must be (5)
```

That's the good ol' try/catch block, but it's not of much use to us in the real world because it requires the Future's *synchronous* evaluation using `Await.result()` and we don't do that in the real world.

But `Await.result()` has shown us something interesting. `Await.result()` is simply a blocking wrapper around the Future's `result()` method. The `result()` method returns an instance of type 'T', not an `Either[Throwable, T]`. We know that the Future contains an Exception; the Exception was created on another thread and stored as data in the Future for future extraction. The only thing that `result()` can do is throw the Exception it holds, since it clearly cannot return a value of type 'T'.

But using `result()` directly, isn't really something we tend to do, which means that the try/catch block is never really a good option. Instead, we can use the Future's functional nature to combine failure-handling code with the Future itself. There are two ways to do this.

fallbackTo

The `fallbackTo` combinator lets us bind a secondary Future to the main Future that will be substituted in the case of an Exception. It's pretty simple:

```
val oops = Future(5) filter { _ % 2 == 0 } fallbackTo Future(5)
val result = Await.result(oops, 1.second)
result must be (5)
```

However, there are times when you either want more granularity on your Exception handling, or you don't need a full-fledged Future as your handler.

recover

The `recover` method lets you supply a partial function that handles possible Exceptions and returns an *immediate* value instead of a *Future* value.

```
val oops = Future(5) filter { _ % 2 == 0 } recover {
  case e: NoSuchElementException => 5
```

```
}
```

```
val result = Await.result(oops, 1.second)
```

```
result must be (5)
```

Of course, if you fail to actually match and handle the Exception in the recover block, you'll still get an Exception when things finally evaluate:

```
val oops = Future(5) filter { _ % 2 == 0 } recover {
```

```
// This doesn't throw an ArithmeticException
```

```
case e: ArithmeticException => 5
```

```
}
```

```
// Test with an assertion
```

```
evaluating {
```

```
    val result = Await.result(oops, 1.second)
```

```
} must produce[NoSuchElementException]
```

recoverWith

RecoverWith is a different beast, since it allows us to recover the failure with a Future. It might not be immediately obvious how you can use it, but it actually allows you to do a bit of *speculative execution* when you might expect a failure.

For example, let's say we have a cache server out there that can serve up results to complicated questions really quickly, provided it's not overloaded with requests. There are times when the cache will time out, and the guy asking the question will be out of luck. You want to give him an answer regardless of the cache being available; you're going for the speed of the answer and nothing more. There's nothing to stop you from trying a "long" method and the cache method at the same time, and if the cache fails, then you can return the computed value instead.

```
class CacheTimedOutException(msg: String) extends Exception(msg)
```

```
val longCalculation = Future {
```

```
// It takes a long time to calculate 5... really
```

```
    5
```

```
}
```

```
val value = Future {
```

```
// Get the 5 from the cache, since it should be faster
// ...
// Except it's not
throw new CacheTimedOutException("That didn't work")
} recoverWith {
  case _: CacheTimedOutException =>
    longCalculation
}
Await.result(value, 1.second) must be (5)
```

The above code executes both the “long” calculation as well as the cache retrieval simultaneously. All the resources on the machine are available, so we don’t mind burning it up a bit, so long as we can get what we need. If the cache retrieval fails, we still win because we prepared for it ahead of time. The “long” calculation was kicked off before we even tried to get to the cache.

Normally, speculative execution is performed in anticipation of slow responses – fire off the same calculation on 20 machines and the first one back wins, much like the ScatterGatherFirstCompletedRouter. In this case, we’re speculatively executing something as a back-stop to failure. In the end, they’re the same concept applied to different situations.

Future.sequence

Do you remember *way* back to [Chapter 4](#) when we described the complexity involved with using Actors to multiply a whackload of matrices together? The core of the problem revolved around the non-determinacy of responses from Actors that were performing the multiplications. We had to remember who got what in order to maintain the right order of the responses. In a word, it’s *icky*.

Futures provide a *much* better solution to this problem. Multiplying the matrices together isn’t all that big of a deal—you just gotta do the math—the bookkeeping and maintenance of trying to parallelize the computation is the real pain, and Futures have a free solution to it.

Let’s set the stage for multiplying matrices with a mock Matrix class that multiplies things together and produces a resulting Some[Matrix] if things work and a None if they don’t.³

³Matrix multiplication needs the dimensions of the two matrices to line up such that if

First, our Matrix class:

```
case class Matrix(rows: Int, columns: Int) {  
    // "Multiply" the two matrices, ensuring that  
    // the dimensions line up  
    def mult(other: Matrix): Option[Matrix] = {  
        if (columns == other.rows) {  
            // multiply them...  
            Some(Matrix(rows, other.columns))  
        } else {  
            None  
        }  
    }  
}
```

Next, we define a function that will multiply a sequence of matrices together using `foldLeft`:

```
def matrixMult(matrices: Seq[Matrix]): Option[Matrix] = {  
    matrices.tail.foldLeft(Option(matrices.head)) { (acc, m) =>  
        acc flatMap { a => a mult m }  
    }  
}
```

Finally, we'll create a bunch of matrices that we'll multiply together:

```
// Generate some random indices  
val randoms = (1 to 20000) map { _ =>  
    scala.util.Random.nextInt(500)  
}  
  
// Turn them into rows and columns in a way that  
// ensures the dimensions all line up properly  
val matrices = randoms zip randoms.tail map {  
    case (rows, columns) => Matrix(rows, columns)  
}
```

the left side has dimensions $m \times n$, then the right side must have $n \times p$.

OK, the set up is finished. We have enough of a strawman in place that we can start multiplying them together in parallel. Next, we need to break up the `matrices` sequence into groups—groups of 500 should do. We'll then transform that sequence of groups into a sequence of `Futures` that are performing the multiplications.

```
val futures = matrices.grouped(500).map { ms =>
    Future(matrixMult(ms))
}.toSeq
```

The `futures` value now contains 40 `Futures` that are all multiplying their sequence of matrices. If the `ExecutionContext` they're running on happens to have 40 available threads, then they'll all go in parallel; if not, then they'll be executed in bits and pieces, but hopefully saturating your CPUs as much as possible.⁴

Once they're finished, we will have 40 resulting matrices (or, if there was a dimensional problem somewhere, we'll have one or more `None` values). We still need to multiply those 40 together into one final matrix. This is the part that was so damn tricky with the Actor-based solution. Questions arise:

1. How do we know when they're all done?
2. In what order did the responses return?
3. How do we ensure that we're multiplying the final 40 in the right order?

None of those questions have any meaning anymore, now that we're using `Futures` to do all of the bookkeeping for us. We just have to transform the sequence of `Futures` into a *single* `Future` that holds a sequence of *results*. When we have that single `Future`, we can easily transform the results using the `matrixMult()` function we used to multiply all of the intermediates.

```
val multResultFuture = Future.sequence(futures) map { r =>
    matrixMult(r.flatten)
}
```

⁴This is a very careful statement. There are a ton of factors that go into true CPU saturation, involving CPU caches, RAM hits, and all kinds of muck that isn't the focus of this book. Chances are you won't see your CPUs 100% busy due to the issues surrounding the details of your particular processor architecture.

The only difference here is that the intermediate matrices aren't of type `Matrix` but of type `Option[Matrix]`, so we need to flatten them before sending them to `matrixMult()`. However, that has nothing to do with the concurrency.

Now, `multResultFuture` holds a Future whose value is of type `Option[Matrix]`, which we can grab in our usual way:

```
val finished = Await.result(multiplyResults, 1.second)  
// We can't actually test the rows and columns since  
// they're random  
finished must not be ('empty)
```

And we're done! You understand, of course, that there was more code in that example to simply set up the problem and evaluate it than there was to actually perform the "complicated" parallelization, right? If we did that with Actors, the reverse would most certainly have been true!

If you need to scrape a bit of your brain off the floor and shove it back in through your ear due to the fact that your mind was just blown, I'll understand.

`Future.traverse`

The `sequence` method creates a Future with a sequence of values on which we can operate with a `map`. In some cases, you can see this as an "intermediate" sequence that you simply don't need. When you have a situation like this, you can use `traverse` to apply a mapping directly. We can illustrate by transforming a sequence of Future Ints into a sequence of Future Ints squared.

```
val futures = (1 to 20) map { i => Future(i) }  
// Grab the intermediate Future[Seq[Int]]  
val sequenced = Future.sequence(futures)  
// Transform to a Future[Seq[Int]] of squared values  
val seqSquared = sequenced map { seq =>  
    seq map { i => i * i }  
}  
// Just square them directly  
val trvSquared = Future.traverse(futures) { futurei =>
```

```
    futurei map { i => i * i }
}

val squaredFromSeq = Await.result(seqSquared, 1.second)
val squaredFromTrv = Await.result(trvSquared, 1.second)
squaredFromSeq must be (squaredFromTrv)
```

The `traverse` method merely eliminates the need for the intermediate `Future[Seq[T]]` when you don't need it. Use either one as you see fit.

Future.firstCompletedOf

Speaking of speculative execution: The `firstCompletedOf` method on the `Future` object provides the ability to pull out the first completed `Future` on a sequence of `Futures`.

If you want to run a bunch of tasks to grab the first winner's results, then this method makes perfect sense. Previously, we used speculative execution to calculate a value, presuming that grabbing it from the cache would fail. If the value actually was calculated *before* the cache timed out, then we still wouldn't have received it until the timeout occurred. With `firstCompletedOf`, we can perform the calculation and the cache retrieval, grabbing the first winner.

```
// Make them defs so that they aren't started
// until they go in the List
def longCalculation = Future {
  Thread.sleep(scala.util.Random.nextInt(60))
  "5 - From the calculation"
}

def cache = Future {
  Thread.sleep(scala.util.Random.nextInt(50))
  "5 - From the cache"
}

val futures = List(cache, longCalculation)
val result = Future.firstCompletedOf(futures) onSuccess {
  case result => println(result)
}

// Printout depends on the random sleeps
```

In the above code, the printout is either “5 - From the calculation” or “5 - From the cache,” depending on which one won.

You can also chain them together, which can be useful when you don’t have a list of Futures available to you but you know you want to add an optional optimization to a given Future, using either:

```
val result = cache either longCalculation
```

Things Are Still Running

Be aware that both of the above Futures will run. We just grab the first completed one, and the second one continues to completion. Depending on how you play with this functionality, you might see an interesting side effect. Let’s say we did this:

```
val futures = List(cache, longCalculation)
Await.result(Future.firstCompletedOf(futures) onSuccess {
    case result => println(result)
}, 1 second)

// Shut down the context, cuz we're done!
execContext.shutdown()
```

We’re not necessarily done by the time the execContext shuts down. We’ve waited long enough for the “fast” Future to finish, but the “slow” one is still going. There’s no problem with this, but it might look like there is. Because there’s something still running when the ExecutionContext shuts down, you might see an Exception on your terminal or in a log, for example.

```
Exception in thread "pool-291-thread-1"
java.util.concurrent.RejectedExecutionException
...
```

Or something along those lines. That’s the thread pool yelling at you, basically saying that some thread is doing something when it’s not supposed to, since there’s nothing on which it can do it.

These sorts of concurrency things pop up because, well, it’s concurrency. It’s up to you to either make sure all your ducks are lined up before you take that final shot, or if you know they’re not lined up, that you can ignore noise like this.

Future.fold

Folding is another one of those cool aspects of working with Akka's Futures. Much like `sequence` and `traverse`, `fold` manages the annoying aspects of keeping things in sequence as well as being invoked only when all of the Futures have completed.

We'll use `fold` to do something silly but illustrative. The following code will add up the ASCII values of several words, and just to make things quite clear, we'll include some side-effecting `println`s.

```
// Some of my favourite words
val words = Vector("Joker", "Batman", "Two Face", "Catwoman")

// Transform the words into Futures
val futures = words map { w =>
    Future {
        val sleepTime = scala.util.Random.nextInt(15)
        Thread.sleep(sleepTime)
        println(s"$w finished after $sleepTime milliseconds")
        w
    }
}

// Fold over them, adding up their ASCII values
val sum = Future.fold(futures)(0) { (acc, word) =>
    word.foldLeft(acc) { (a, c) => a + c.toInt }
}

// Assert
println("Waiting for result")
Await.result(sum, 1.second) must be (2641)

// The last time I ran it, it printed:
//
// Waiting for result
// Batman finished after 9 milliseconds
// Joker finished after 9 milliseconds
// Catwoman finished after 10 milliseconds
// Two Face finished after 10 milliseconds
```

Nifty, no?

Future.reduce

Reduce is the same variant of fold that we’re used to. The result value of the first Future in the sequence is used as the “zero” value to the supplied reducing function. The following illustrates the usage:

```
// Some of my favourite letters
val letters = Vector("B", "a", "t", "m", "a", "n")

// Transform the letters into Futures
val futures = letters map { l =>
    Future {
        val sleepTime = scala.util.Random.nextInt(15)
        Thread.sleep(sleepTime)
        println(s"$l finished after $sleepTime milliseconds")
        Thread.sleep(sleepTime)
        l
    }
}

// Reduce the letters down to a single word
val wordFuture = Future.reduce(futures) { (word, letter) =>
    word + letter
}

// Assert
println("Waiting for result")
Await.result(wordFuture, 1.second) must be ("Batman")

// The last time I ran it, it printed:
//
// Waiting for result
// n finished after 2 milliseconds
// t finished after 2 milliseconds
// a finished after 9 milliseconds
// m finished after 11 milliseconds
// B finished after 12 milliseconds
// a finished after 13 milliseconds
```

Again, it doesn’t matter when things finish, reduce works across the sequence, independent of the relative times in which the various Futures complete.

Future.find

If you want to locate a value in a set of Future results, you could conceivably filter those Futures, looking for anything that isn't a Left instance. However, you don't need to do that since Akka provides the `find` method.

The interesting difference about `find` from some of our previous methods is that it doesn't need the entire sequence of results in order to perform its duties. Let's illustrate this with another variation on our superhero code:

```
// He's just so awesome
val letters = Vector('B', 'a', 't', 'm', 'a', 'n')

// Transform the letters into Futures
val futures = letters map { l =>
    Future {
        Thread.sleep(scala.util.Random.nextInt(15))
        l
    }
}

// find anything less than or equal to 'm'
val foundFuture = Future.find(futures) { l => l <= 'm' }

val found = Await.result(foundFuture, 1.second)
println("Found " + found)

// The last five times I ran it, it printed:
//
//   Found Some(a)
//   Found Some(B)
//   Found Some(a)
//   Found Some(m)
//   Found Some(a)
```

The randomness in the Future completion times ensures that the found letter varies from run to run. Although the original sequence is always the same, the concurrency breaks it up into random completions that manifest in the results of the call to `find`.

Concurrency...it's just plain interesting stuff.

12.4 Side-Effecting

Up until now, we've only seen the Future's functional aspects. Akka has a mechanism for acting on results in a side-effecting manner as well, which can be useful depending on your needs.

Each callback takes a `PartialFunction` that has a different type, depending on the callback on which it is defined. Let's look at each one in turn.

onSuccess

The `onSuccess` method adds a callback to the Future that is executed when the Future completes and is successful (that is, the result is a `Right[T]` instead of a `Left[Throwable]`). Let's use the stuff we've seen to create an `onSuccess` that's a bit weird:

```
Future { 13 } filter {
    _ % 2 == 0
} fallbackTo Future {
    "That didn't work."
} onSuccess {
    case i: Int => println("Disco!")
    case m => println("Boogers! " + m)
}
// Prints: Boogers! That didn't work.
```

The Future will be successful no matter what because we have a `fallbackTo`, but (just for fun) we're using it to indicate to the callback that things didn't actually work. Due to the mixed types of `Int` and `String` from the combined Futures, the subsequent callback has type `PartialFunction[Any, Unit]`, which is just fine for us since we can pattern-match on the result quite easily.

onFailure

Of course, looking at failures with `onSuccess` is a bit strange. Normally, you would use `onSuccess` to do something when it's actually successful, and for failures, you'd use `onFailure`. For example, we can alter the previous code to do the obvious thing on failure with:

```
Future { 13 } filter {
```

```
_ % 2 == 0
} onFailure {
  case _ => println("Boogers! That didn't work.")
}
// Prints: Boogers! That didn't work.
```

onComplete

The previous two callback definitions are actually special cases of the onComplete callback, and if you find yourself actually caring about both success and failure, you may find it better to use the onComplete callback instead of the onFailure and onSuccess callbacks together.

```
Future { scala.util.Random.nextInt(100) } filter {
  _ % 2 == 0
} onComplete {
  case Right(num) => println("Disco! Got a " + num)
  case Left(_)   => println("Boogers! That didn't work.")
}
// Prints: Well... it depends on the value that comes
// out of Random.nextInt(100)
```

Side-Effects Order

You can actually add as many side-effects as you'd like. You can't chain them together since the results of these side-effecting callbacks is Unit, but if you have your Future available to you, you can do it without too much trouble.

```
val f = Future { 12 } filter {
  // Don't want to finish too soon
  Thread.sleep(50)
  _ % 2 == 0
}
f.onFailure {
  case _ => println("Boogers! That didn't work.")
```

```
    }
    f.onSuccess {
      case i: Int => println(i)
    }
    f.onSuccess {
      case i: Int => println(i + " is keen!")
    }
    f.onSuccess {
      case i: Int => println(i + " is WICKED keen!")
    }
    f.onComplete {
      case Left(t: Throwable) => println("Nuts! " + t)
      case Right(v: Int) => println("Yeah baby! " + v)
    }
}
```

Don't expect these to be called in any deterministic order though. Akka doesn't like wasting time, and if it wanted to guarantee that these went in some sort of order, then it would have to play a lot of games that slowed it down all the time...not worth it!

You should be avoiding side-effects anyway, since they're generally just a pain in the butt.⁵

12.5 Futures and Actors

You have a solid understanding of Actors, and you now have a pretty decent understanding of what Futures bring to the party. The keen thing is that they also work together really well.

sender ! response

When you're inside an Actor's message handler, and you're going to send a message to the sender of the current message, you use `!`. That works great if the sender is an Actor, but what if the sender is a Future?

Let's look and see how it's done:

⁵Actors notwithstanding. Yes, Actor programming is inherently side-effect based, but I expect you to hold this in your head at the same time as what you've just read and not have any cognitive dissonance, somehow.

```
// Specific imports to enable the features we're
// going to use
import akka.pattern.ask
import scala.concurrent.util.duration._
import akka.util.Timeout

// All 'asks' (a.k.a '?') need a timeout value
// after which the ask is considered failed.
implicit val askTimeout = Timeout(1.second)

// Create our Actor
val a = system.actorOf(Props[EchoActor])

// Make the Ask
val f = a ? "Echo this back"

// Verify the result
Await.result(f, 1.second) must be ("Echo this back")
```

Pretty simple stuff... you just simply make the ask, which we've seen before. The Actor definition that makes this work is:

```
class EchoActor extends Actor {
    def receive = {
        case m => sender ! m
    }
}
```

Yup. There's no difference between an Actor that sends messages to Actors and one that sends messages to Futures. The abstractions that Akka provides allow all Actors to be ignorant of both who and what the sender actually is. It can be the Dead Letter Office, a Future, a local Actor, a remote Actor, a router... *it just doesn't matter.*

The Details of '??'

Although not terribly complicated, there are some important concepts that surround 'ask' that you need to understand. Let's start by looking at how it gets imported.

- `import akka.pattern.ask` imports two methods named 'ask'.

- The first is an implicit conversion that converts an ActorRef to an AskableActorRef. The AskableActorRef contains the '?' method, giving us that nice DSL.
- The second is the actual 'ask' method we can use to communicate with the Actor itself. The '?' extension uses this method to perform the ask.

The usage method has the following signature:

```
def ask(ref: ActorRef, msg: Any)(implicit timeout: Timeout): Future[Any]
```

The ref and msg are pretty obvious; we're interested in the timeout and return type.

- timeout specifies the amount of time you're willing to wait for the ask to be fulfilled. This is *different* than the timeouts we've seen when performing an Await.result() call.
- The return type is Future[Any]. Up until this point, all of our Futures have been more strongly typed; we would have Future[Int] or Future[List[String]] or whatever. But in this case, we're talking to an Actor, and Actors exchange messages way up at the level of Any; thus, we can't expect anything lower down on the result type.

Chains of Timeouts

If you specify the timeout explicitly, instead of using the implicit, and you wait for the result, then you'll have something that looks like this:

```
val f = Await.result(a.ask("Echo this back")(Timeout(1 second)),  
                     1 second)
```

That's a mouthful, and at first glance those with glasses tend to take them off, give them a little clean, pop them back on, and have another look. Those without glasses immediately get a sense of *déjà vu*.

Both timeouts are necessary because we're performing two asynchronous, time-based functions. The first one is the argument to the 'ask' telling it to throw a TimeoutException if the Actor doesn't respond within 1 second. The

latter, of course, throws a `TimeoutException` if the Future itself doesn't *complete* within 1 second. The timers that are governed by these timeouts are running concurrently, so you don't have a total of 2 seconds. Therefore, if you change the definition to the following, chances are that you'll get a failure.

```
val f = Await.result(a.ask("Echo this back")(Timeout(1.second)),
1.microsecond)
```

The `Await.result()` will only wait a microsecond for the Future to complete and the Actor probably can't get back that fast.

Note

It's very important to understand that the *failure* here is not one such that the Future's value is a `Left(TimeoutException(...))`. This failure's manifestation will be a literal thrown Exception from the `Await.result()`.

You also have to think about what it looks like when you have a sequence of Futures created from asks, all with timeouts. To illustrate, let's create another echoing Actor that has a parameterized delay in it.

```
case class DelayedEcho(msg: String, millis: Long)
class DelayingActor extends Actor {
  def receive = {
    case DelayedEcho(msg, millis) =>
      blocking {
        Thread.sleep(millis)
        sender ! msg
      }
  }
}
```

Now we can write a test that deals with a series of asks against this `DelayingActor`, with different delays for each one.

```
// Create our delaying Actor
val a = system.actorOf(Props[DelayingActor])
// Create some asks, increasing the delays as we go
```

```
val futures = (1 to 10) map { i =>
    val delay = 10 + i * 2
    val str = s"Delayed for $delay milliseconds"
    // Put a timeout on them that's a little longer than the delays
    a.ask(DelayedEcho(str, delay))(Timeout((delay * 1.2).toInt)) andThen {
        case Left(e) => println(str + " - failed")
        case Right(m) => println(m + " - succeeded")
    }
}

// Wait for them all to be ready, whether or not they succeed or failed.
Await.ready(Future.sequence(futures), 1.second)

// Find out how many failed
val failed = futures filter { f =>
    f.value.isEmpty || f.value.get.isLeft
}

// Just print them out
println(failed.size + " failed")

// The last time I ran this, it printed:
//
// Delayed for 12 milliseconds - succeeded
// Delayed for 14 milliseconds - succeeded
// Delayed for 16 milliseconds - succeeded
// Delayed for 18 milliseconds - succeeded
// Delayed for 20 milliseconds - failed
// Delayed for 22 milliseconds - failed
// Delayed for 24 milliseconds - failed
// Delayed for 26 milliseconds - failed
// Delayed for 28 milliseconds - failed
// Delayed for 30 milliseconds - failed
// Delayed for 32 milliseconds - failed
// 6 failed
```

The machine I'm running this test on has only two cores,⁶ so the default settings almost ensure that there will eventually be some failed timeouts.

⁶It's a little 11-inch i5 dual core notebook, and I love it, so don't make fun of it.

Note that we used `Await.ready()` instead of `Await.result()`. We know for a fact that there will be some Exceptions in here, and if we try to go for *evaluation*, the Future will have no choice but to throw the Exception. It will also throw the first one it finds, which will make the rest of the code pretty tough to execute. `Await.ready()` ensures that the Futures are *ready* to be evaluated, but won't be evaluated until we try to extract the result, which of course we don't do. We just need to peek at it and see if it's either `None` (which it won't be, since `ready()` has already told us it's not, but we put it in there for good measure) or is a `Left` instance of `Either`.

Coercing from Any

Since Actor messages ensure that the Future type that an ask creates will be `Future[Any]`, you would expect that there's some support for coercing that type back down to something usable. You would be correct in that assumption.

But why not just use `asInstanceOf`? The reason is the same old reason we keep running up against: *concurrency*. When do you apply the `asInstanceOf`? You could apply it after you've received the result, but that's not terribly robust since it limits what you can do with the message. If it will travel through a bunch of intermediaries, you might now be depending on someone to cast it down who really doesn't know what's going on.

The right time do it is when it can be safely done, but you have to do it in a Future context since, in general, the thing you're casting down to doesn't yet exist. This is why we have the `mapTo` method on the Future. For example:

```
for {
    dbConn <- (systemActor ? GetDBConnection).mapTo[DBConn]
    result <- (dbActor ? Query(dbConn, query).mapTo[ResultSet]
} yield result.sortByKey
```

Get the idea? If we didn't coerce the result from the first ask, then we wouldn't have the right type for the second ask. However, you can't cast the result from the first ask down to a `DBConn` until the ask completes, which is why we use `mapTo`. It creates a new Future with the strongly typed result, or a `ClassCastException` if the cast fails.

Futures Do Request/Response

The ask syntax clearly implements a request/response paradigm. You could do the same thing with an anonymous Actor, such as this:

```
class MyActor extends Actor {  
    def receive = {  
        case GetSomeDataFrom(someActor) =>  
            // An anonymous Actor to provide a Request / Response  
            // context to communication with 'someActor'  
            context.actorOf(new Actor {  
                override def preStart() {  
                    someActor ! GiveMeStuff  
                }  
                def receive = {  
                    case SomeStuff(stuff) =>  
                        doSomethingWithStuff(stuff)  
                }  
            })  
    }  
}
```

The challenge with the request/response paradigm in asynchronous message systems is the act of assigning the appropriate *context*⁷ to the response, and that's a challenge that Futures meet head on and quite well.

But just to elaborate for a second, assume you have the following Actor:

```
class DiscoActor extends Actor {  
    def receive = {  
        case Go =>  
            (1 to 10) zip someListOfTenActors foreach {  
                case (num, actor) =>  
                    actor ! DoSomethingWithThisNumber(num)  
            }  
        case SomethingDone(result) =>  
            // Hmm... I wonder which request this response belongs to  
    }  
}
```

⁷Sorry about the overloaded word with the ActorContext, but the word *context* is just too important in concurrent programming.

There are three obvious ways to solve this problem, and probably a whole host of others that aren't so obvious.

1. Provide the ability to stash as much information as you want into the request message.
2. Keep some sort of mutable data structure lying around that can be indexed by some value, and let that store the intermediate state. The index can be a tag that you put on the message, or the ActorRef of the Actor to whom you're sending the request.
3. Create a closure.

OK, so **number 1** is just silly. How much data are you going to stash? What if it has to travel over a network? Are we going to trust people to put it in the response properly? Do they have to understand it? *Silly* is a very polite word for it.

Number 2 is better since it keeps things local to the guy doing the requests, but seriously? The suggestion is to use a 'tag' or the ActorRef of the Actor we're making the request of. If you said that the ActorRef is a good idea, then you failed the test; go back several chapters and read them again; *especially* the one on Routing.

Tip

Using the ActorRef as an index key is a horrible idea. The Actor you're sending to may never be the guy who's responding to you, so you'd have a dangler there. Bad idea.

Using a 'tag' instead is a much better idea and it has its merits in some situations,⁸ but it's not remotely ideal. It's cruft that is carried along with your message that Actors that see it still have to manage.

Number 3 is a fairly awesome solution most of the time. It keeps everything out of your messages, which makes them more flexible and reusable in many different contexts, and it keeps the intermediaries ignorant of the request/response nature of what you're doing. What's more is that a closure does not limit you to any amount of information that you want to close over.

That's why Futures handle the request/response problem—they allow you to create that closed over *context* with supreme ease.

⁸Performance is one. It's a lot cheaper to tag a message than it is to instantiate an Actor, put it on a thread, etc. But *do not* prematurely optimize.

Futures Are Not Your Actor

In the heat of coding, when your hair is on fire, you've pounded your fifth energy drink in the last 20 minutes and you've got Led Zeppelin turning your ears into a puddle of sublime oozing liquid, you might make a slight error; you accidentally walk into the Actor's fortress and start remodelling the first floor bathroom in a tacky leopard print.

It's easy to do, but once you see it, it's also easy to avoid. The problem comes from using a Future inside an Actor that modifies state directly. We saw this problem back when we looked properly using the Scheduler inside an Actor, and the problem here is exactly the same.

```
class MutableActor extends Actor {  
    var currentState = InitialState  
    def receive = {  
        case Go =>  
            (someOtherActor ? GetData) onSuccess {  
                case data =>  
                    // I'll just dislodge that Strawberry from the running  
                    // blender with my right index finger. It'll be fine.  
                    currentState = DataReceivedState(data)  
            }  
        case WebRequestReady(req) =>  
            currentState = WebReadyState(req)  
    }  
}
```

Yuck... you just lost your right index finger. Well done.⁹ You need to be careful of the behaviour you have inside your Future. This applies, of course, to every closure you create inside the fortress, whether it be inside a Future, another Actor, a Scheduler, or any other piece of code that leaves the warm, fuzzy innards of your Actor.

⁹Funny story. I know this girl who had to go to the emergency room with a blade from a food blender lodged in her index finger. The doctor was sympathetic, but also kinda laughed until he passed out.

Actor Message Piping

The ask pattern allows Actors to talk to Futures, and we've also seen the horrific way that you can mutate fortress-protected data inside an Actor. There's clearly something missing—how to talk to an Actor from a Future.

One way to do this is to use a side-effecting callback, since sending a message to an Actor is, effectively, a side-effecting operation. However, doing so just replicates the same code over and over again:

```
val future = someActor ? AskQuestion
future onSuccess {
  case m => anotherActor ! SuccessHappened(m)
} onFailure {
  case m => anotherActor ! FailureHappened(m)
}
```

The pipe pattern pulls this common code together into something that's reusable. So, we just have to:

```
// We need to import it in order for it to work
import akka.pattern.pipe

val future = someActor ? AskQuestion
future pipeTo anotherActor
```

It must be noted, however, that if the Future is in a failed state, then the message that gets sent to anotherActor is `akka.actor.Status.Failure(exception)`. So either the recipient Actor is ready to handle that message or it will get logged as something that's unhandled.

Piping is a fairly elegant way to bridge two Actors together while applying intermediate transforms to the message structure. For example:

```
import akka.pattern.pipe

// Get the value for the desired heading from
// actor1, which returns a String. Convert this
// value to an Int, pack it in a command message
// and pipe it to the Auto Pilot
(actor1 ? GetValue("Heading")).mapTo[String] map {
  headingString =>
```

```
        AlterHeading(headingString.toInt)
    } pipeTo autoPilotActor
```

12.6 Plane Futures

Remember the TODO we left behind in Chapter 10? It looked like this:

```
def noRouter: Receive = {
    case GetPassengerBroadcaster =>
        context.actorFor("PassengersSupervisor") ! GetChildren(sender)
    case Children(passengers, destinedFor) =>
        val router = context.actorOf(Props().withRouter(
            BroadcastRouter(passengers.toSeq)), "Passengers")
        destinedFor ! PassengerBroadcaster(router)
        context.become(withRouter(router))
}
```

This chunk of code is a classic request/response-style scenario and we can rewrite it using a Future.

```
implicit val askTimeout = Timeout(5.seconds)
def noRouter: Receive = {
    case GetPassengerBroadcaster =>
        val destinedFor = sender
        val actor = context.actorFor("PassengersSupervisor")
        (actor ? GetChildren).mapTo[Seq[ActorRef]] map { passengers =>
            (Props().withRouter(BroadcastRouter(passengers)), destinedFor)
        } pipeTo self
    case (props: Props, destinedFor: ActorRef) =>
        val router = context.actorOf(props, "Passengers")
        destinedFor ! PassengerBroadcaster(router)
        context.become(withRouter(router))
}
```

Does that look better? Well, it all depends on what you want out of it. The biggest key to this change is that the interaction's request/response nature is clear and codified. Before, if the Actor didn't receive the children response, nothing dealt with that problem. Now there is a timeout on

which we can play any of the tricks we now know, such as recover and recoverWith. We've also removed the silly notion of having to pass the equivalent of destinedFor to the underlying Actor. It merely has to return the children that it has, without having to walk around with this extra payload of the Actor to eventually respond to, which is something it doesn't really even want to know.

The simplifications we've made to the embedded supervisor's receive method make this more clear. We moved from this:

```
// Override the IsolatedStopSupervisor's receive
// method so that our parent can ask us for our
// created children
override def receive = {
    case GetChildren(forSomeone: ActorRef) =>
        sender ! Children(context.children, forSomeone)
}
```

to this:

```
// Override the IsolatedStopSupervisor's receive
// method so that our parent can ask us for our
// created children
override def receive = {
    case GetChildren =>
        sender ! context.children.toSeq
}
```

Instrument Status

Really cool planes have lots of dials and switches as well as flashing lights. If you wanted to put an “Everything is Hunky Dory” light on your plane, which lit up green when all of the instruments were operating correctly, you’d need to ask them all how they were feeling.

Let’s create a trait that we can mix into any instrument we create, which will report on the instrument’s status. As usual, we start with the companion object.

```
object StatusReporter {
```

```
// The message indicating that status should be reported
case object ReportStatus

// The different types of status that can be reported
sealed trait Status
case object StatusOK extends Status
case object StatusNotGreat extends Status
case object StatusBAD extends Status
}
```

And now the trait, which will be self-typed to an Actor. It will provide a receive handler to be composed into the ultimate receive handler and a required method to be implemented that reports status. Eventually, we'll be composing this inside whatever Actor we choose, that wishes to report its status.

```
trait StatusReporter { this: Actor =>
    import StatusReporter._

    // Abstract - implementers need to define this
    def currentStatus: Status

    // This must be combined with orElse into the
    // ultimate receive method
    def statusReceive: Receive = {
        case ReportStatus =>
            sender ! currentStatus
    }
}
```

Now let's inject it into the HeadingIndicator:

```
trait HeadingIndicator extends Actor
    with ActorLogging
    with StatusReporter { this: EventSource =>
    import StatusReporter._

    // The HeadingIndicator is always happy
    def currentStatus = StatusOK

    ...
    // Compose our receive method from the EventSource,
    // the StatusReporter and our own functionality
}
```

```
def receive = statusReceiveorElse
    eventSourceReceiveorElse
    headingIndicatorReceive
    ...
}
```

Assuming you have all of this done for the HeadingIndicator, and the Altimeter, as well as for other Instruments you'd create (such as a Fuel-Level Indicator, Airspeed Indicator, and a whole host of other things), then you can get an indication of the status of *all* Instruments with:

```
val instruments = Vector(actorFor("Altimeter"),
    actorFor("HeadingIndicator"),
    actorFor("AirSpeed"),
    actorFor("FuelSupply"))

def receive = {
    case GetAllInstrumentStatus =>
        Future.sequence(instruments map { i =>
            (i ? ReportStatus).mapTo[Status]
        }) map { results =>
            if (results.contains(StatusBAD)) StatusBAD
            else if (results.contains(StatusNotGreat)) StatusNotGreat
            else StatusOK
        } pipeTo sender
}
```

12.7 Chapter Summary

OK! We've just covered some pretty awesome stuff. Futures make up a very important axis of the Akka concurrency paradigm. If all you know about are Actors and Futures, you have enough information to do some pretty serious Akka programming. Of course, there's more to Akka and it continues to grow, but these two main concepts embody the entire Akka design. Let's summarize what we've covered:

- You should now understand that Actors are not a golden hammer. Sure, you can solve a ton of problems with them, but there are def-

initely a class of problems that cannot be solved with them nearly as easily as with Futures.

- You've learned to *compose* Futures, which is probably the single biggest win you have with them.
- You've learned to deal with failures in a very graceful way.
- You have a firm grasp on the nature of concurrency and timeouts.
- The request/response paradigm has been evaluated and while there's not necessarily a "right" answer to it under all circumstances, we now know that Futures provide a solid answer most of the time.
- The relationship between Actors and Futures is now also well understood. You're not mashing together Future functionality with Actor functionality because no mashing is required; they fit together like hand and glove.

Pat yourself on the back, grab a beer, eat that piece of cake you've been eyeing for the last few hours—you've *earned it*. You've assimilated some valuable information thus far and that should make you feel pretty great. Personally, I just slipped on my Green Lantern power ring in order to write this summary because, well, why not?

Chapter 13

Networking with IO

Whenever you need some non-blocking asynchronous network IO, Akka has your back. The IO package, which is already included the Actor module (so you don't need to go mucking about with SBT to set it up), provides two main mechanisms for achieving the kind of networking that will make your app stay reactive: Actor integration and Iteratees. We will briefly look at Iteratees once we've gone through the Actor integration, but it won't be an exhaustive investigation.

While we could build something cool for our Plane, such as landing gear or a really cool entertainment system, we have much more important things to worry about. How cool would it be if you could telnet into your Plane and ask it some questions? Yeah, way cooler than landing safely!

13.1 The Plane's Telnet Server

Telnet is one of those really cool apps that will survive until the end of the Internet, which will happen right around the same time you have to duck for flying pigs. We'll use this nifty little client by using it to attach to our Plane so that we can get some information about it.

For simplicity's sake, we'll build something that will help us grab the current heading and altitude from the Plane by typing in a couple of obvious commands. It'll look something like this:

```
-> telnet localhost 31733
Trying ::1...
Connected to localhost.
```

```
Escape character is '^]'.

Welcome to the Airplane!
-----
Valid commands are: 'heading' and 'altitude'

> heading
current heading is: 359.92 degrees

> altitude
current altitude is: 345.25 feet

> Hiya there Dave!
What?
```

Pretty awesome looking, no?¹ We'll implement this with a standard Actor; therefore, it will be as reactive as we're already used to and interact perfectly with everything else we have.

```
import akka.actor.{Actor, ActorRef, IO, IOManager, ActorLogging, Props}
import akka.util.ByteString
import scala.collection.mutable.Map

class TelnetServer(plane: ActorRef) extends Actor with ActorLogging {
    import TelnetServer._

    // The 'subservers' stores the map of Actors-to-clients that we need
    // in order to route future communications
    val subservers = Map.empty[IO.Handle, ActorRef]

    // Opens the server's socket and starts listening for incoming stuff
    val serverSocket = IOManager(context.system).listen("0.0.0.0", 31733)

    def receive = {
        // This message is sent by IO when our server officially starts
        case IO.Listening(server, address) =>
            log.info("Telnet Server listeninig on port {}", address)

        // When a client connects (e.g. telnet) we get this message
        case IO.NewClient(server) =>
            log.info("New incoming client connection on server")
            // You must accept the socket, which can pass to the sub server
    }
}
```

¹There's nothing more beautiful than a solid text-mode interface and anyone who claims otherwise deserves the evil eye.

```
// as well as used as a 'key' into our map to know where future
// communications come from
val socket = server.accept()
socket.write(ByteString(welcome))
subservers += (socket ->
    context.actorOf(Props(new SubServer(socket, plane))))
// Every time we get a message it comes in as a ByteString on
// this message
case IO.Read(socket, bytes) =>
    // Convert from ByteString to ascii (helper from companion)
    val cmd = ascii(bytes)
    // Send the message to the subserver, looked up by socket
    subservers(socket) ! NewMessage(cmd)
    // Client closed connection, kill the sub server
    case IO.Closed(socket, cause) =>
        context.stop(subservers(socket))
        subservers -= socket
    }
}
}
```

Simple, no? This class is the first of two; it's the main server that handles incoming client connections as well as future socket handling. Every single byte of incoming data comes through the TelnetServer, even those bytes that are from client connections that have already been made. Therefore, our TelnetServer must ensure that future incoming bytes are routed to the right Actor that's servicing that client, which [Figure 13.1](#) shows.

The IOManager will send our Actor (specified to the IOManager's listen method by the implicitly defined self) several messages. We're interested in the ones here that are IO.Listening, IO.Read, and IO.Closed. These three simple messages will give us everything we need in order to manage the incoming clients and make our server do what we want.²

The business logic belongs inside the Sub-Server, which we define as part of the TelnetServer's companion object (along with some helper stuff). Let's have a look:

```
object TelnetServer {
```

²I remember when writing a Telnet Server was *hard*, and something that made people think you were a god of some sort. Damn.

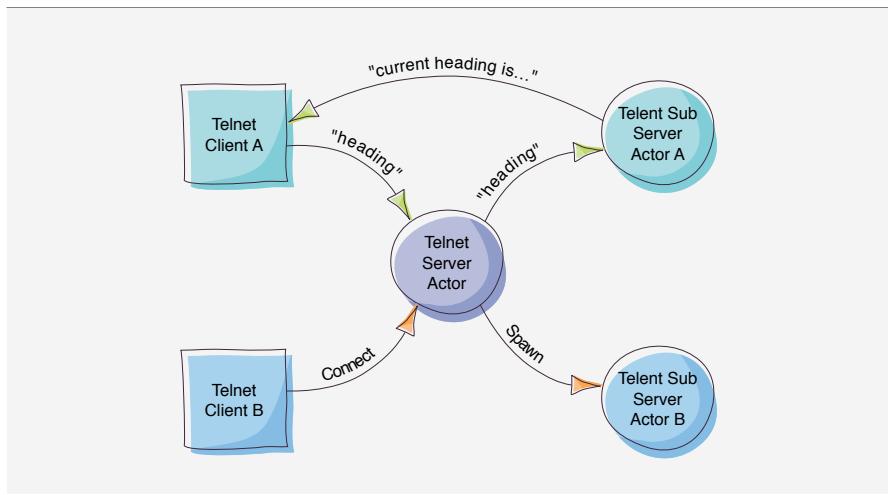


Figure 13.1 · The Telnet Server routes all incoming data to the appropriate Sub-Server Actor that has been instantiated to handle a specific client. It also instantiates that Sub-Server Actor when a new client connects, and clears it out when it disconnects.

```
// For the upcoming ask calls
implicit val askTimeout = Timeout(1.second)

// The welcome message was sent on connection
val welcome =
    """|Welcome to the Airplane!
  |-----
  |
  |Valid commands are: 'heading' and 'altitude'
  |
  |> """.stripMargin

// Simple method to convert from ByteString messages to
// the Strings we know we're going to get
def ascii(bytes: ByteString): String = {
    bytes.decodeString("UTF-8").trim
}

// To ease the SubServer's implementation we will send it Strings
// instead of ByteStrings that it would need to decode anyway
```

```
case class NewMessage(msg: String)

// The SubServer. We give it the socket that it can use for giving
// replies back to the telnet client and the plane to which it will
// ask questions to get status
class SubServer(socket: IO.SocketHandle,
                 plane: ActorRef) extends Actor {

    import HeadingIndicator._
    import Altimeter._

    // Helpers just to make it easier to format for the book :)
    def headStr(head: Float): ByteString =
        ByteString(s"current heading is: $head%3.2f degrees\n\n")
    def altStr(alt: Double): ByteString =
        ByteString(s"current altitude is: $alt%5.2f feet\n\n")
    def unknown(str: String): ByteString =
        ByteString(s"current $str is: unknown\n\n")

    // Ask the Plane for the CurrentHeading and send it to the client
    def handleHeading() = {
        (plane ? GetCurrentHeading).mapTo[CurrentHeading] onComplete {
            case Success(CurrentHeading(heading)) =>
                socket.write(headStr(heading))
            case Failure(_) =>
                socket.write(unknown("heading"))
        }
    }

    // Ask the Plane for the CurrentAltitude and send it to the client
    def handleAltitude() = {
        (plane ? GetCurrentAltitude).mapTo[CurrentAltitude] onComplete {
            case Success(CurrentAltitude(altitude)) =>
                socket.write(altStr(altitude))
            case Failure(_) =>
                socket.write(unknown("altitude"))
        }
    }

    // Receive NewMessages and deal with them
    def receive = {
        case NewMessage(msg) =>
```

```
msg match {
    case "heading" =>
        handleHeading()
    case "altitude" =>
        handleAltitude()
    case m =>
        socket.write(ByteString("What?\n\n"))
}
```

In terms of IO, there's really not much to it, but you can see that we've created a bit of plumbing in the Plane. We can now ask it for the CurrentAltitude and the CurrentHeading, which we tie inside of a Future and side-effect the result back to the socket. We don't need to examine these changes because they're so obvious to you now, but while a picture is probably overkill at this point, let's look at [Figure 13.2](#).

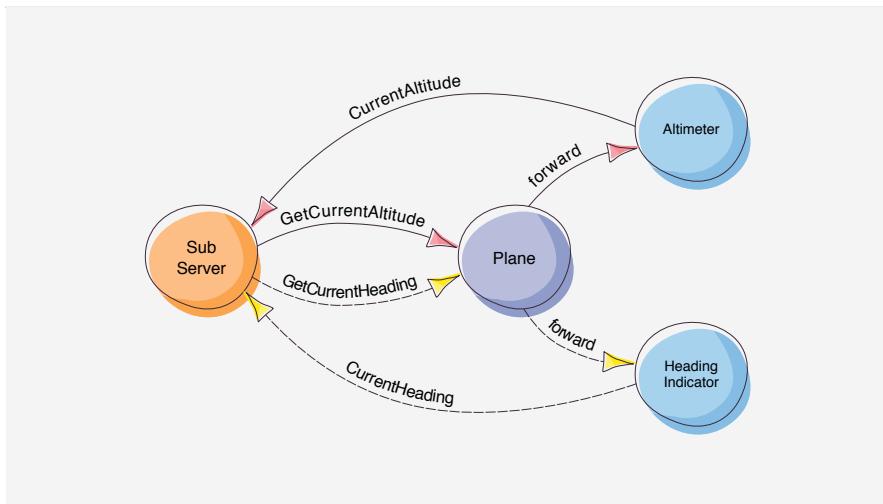


Figure 13.2 · The alterations that have been made to the Altimeter, HeadingIndicator, and Plane allow for the above message flow. The Plane acts as a relay, forwarding the requests that are then fulfilled to the original sender (the SubServer).

To get it running, we just have to hook the Plane up to it. The following implementation of the Avionics main method will do nicely.

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

object Avionics {
    val system = ActorSystem.create("PlaneSimulation")
    def main(args: Array[String]) {
        val plane = system.actorOf(Props[Plane](), "Plane")
        val server = system.actorOf(Props(new TelnetServer(plane)), "Telnet")
    }
}
```

Testing IO

Testing our TelnetServer is pretty darn simple. Since the TestKit and ImplicitSender turn our test class into what is effectively an Actor all by itself, we can use the IO system to test the IO system. Let's first create a simple Plane that we can inject into the TelnetServer to make it easy to test:

```
class PlaneForTelnet extends Actor {
    import HeadingIndicator._
    import Altimeter._

    def receive = {
        case GetCurrentAltitude =>
            sender ! CurrentAltitude(52500f)
        case GetCurrentHeading =>
            sender ! CurrentHeading(233.4f)
    }
}
```

That gives us a deterministic result that we can verify when we write to the socket. Now let's write the test that uses the socket and goes the full way around the network to verify the results.

```
val p = system.actorOf(Props[PlaneForTelnet])
val s = system.actorOf(Props(new TelnetServer(p)))
// The 'test' is now implicitly the Actor that the IOManager talks to
```

```
val socket = IOManager(system).connect("localhost", 31733)
// We expect the IOManager to send us IO.Connected
expectMsgType[IO.Connected]
// Skip the welcome message
expectMsgType[IO.Read]
// Verify the "heading" command
socket.write(ByteString("heading"))
expectMsgPF() {
    case IO.Read(_, bytes) =>
        TelnetServer.ascii(bytes) must include ("233.40 degrees")
}
// Verify the "altitude" command
socket.write(ByteString("altitude"))
expectMsgPF() {
    case IO.Read(_, bytes) =>
        TelnetServer.ascii(bytes) must include ("52500.00 feet")
}
// Close it up
socket.close()
```

That's it. We've tested the TelnetServer by using IO—definitely not a “unit” test, but quite awesome! It's so easy, why wouldn't you write this test?

IO Actor Integration

That's all we'll cover on the IO package's Actor integration. There's a bit more to it, but not a ton. The bottom line is that it just fits. Network programming with the IO package and Actors is almost like coding anything else with Actors; it's easy. If you want to make sure you have all the nuts and bolts nicely handled, head over to the Akka reference documentation and the ScalaDoc.

I will say that most, probably, wouldn't really write a Telnet-style of server for interrogating (or even manipulating) Actors inside a running application; they would go for an HTTP implementation instead. Now, while you could whip up your own HTTP server with Akka IO, you wouldn't. There are other fantastic HTTP servers that you can easily integrate with Akka and we talk about those later in the section discussing add-ons.

13.2 Iteratees

Iteratees have really come into their own lately—perhaps most prominently inside the Play framework.³ As stated earlier, we won’t go into a lot of detail about them because while they are quite useful, they represent a fairly “new” concept to many and would require an amount of explanation that is out of this book’s scope. The Iteratee is a concept that belongs to *functional programming* and there are several useful tutorials on it. If you’re going to use the Play framework, you should definitely consult the tutorial they have provided for their Iteratees.

If you’re looking for a functional solution to working with Akka’s IO module, then you should have a look at Iteratees. There’s absolutely no requirement to use them in order to have great success with IO, as we’ve seen. There’s also no need to even use the Akka implementation of Iteratees if you don’t want to; you should be able to use others out there with relative ease.

13.3 Chapter Summary

With respect to Actors, the IO module is pretty simple and eminently clear. Integrating IO into your Actor application should be a piece of cake, like one of those slices of cherry cheesecake you get for \$12 at some fancy restaurant where the cheese was hand-delivered by a Tibetan monk...on foot...through snow-covered mountains, hunted by furious goats.⁴

In fact, it might actually be considered an anti-pattern to not include some sort of network-accessible hookup for your app, considering how easy it is. With Akka, you have many choices available to you, including anything you can hook up through Camel⁵ or with the HTTP options you have, but if you just need something cheap and simple, IO is your very good friend.

³<http://www.playframework.org>

⁴If you’re lactose intolerant, just pretend you’re not for a moment. That’s a seriously good piece of cheesecake.

⁵<http://camel.apache.org>

Chapter 14

Going Multi-Node with Remote Actors

There are many different strategies for achieving a multi-node, interdependent application structure, involving everything from using the database as a communication mechanism to a “stateless” multi-node web tier. When we reach the scale of “big” applications, the devil is in the details; a relational database, for example, may be the cornerstone of a solution to a particular problem, and it may be violent screaming death for another problem’s solution. However, one aspect of large-scale programming continues to be a win in a large set of problem spaces: *messaging*. More than most development paradigms, messaging has the potential to tear into complexity and eat it for lunch. Inside the Akka toolkit, as you’ve seen, lies a ravenous beast from the likes of Greek mythology, and all it’s looking for is its next lunch.

Because you’ve been working with Akka, you’ve been coding in a really solid messaging system. Up until now, those messages have been sent intra-VM, but sending them out to a different VM, possibly on a different machine, is effectively the same thing. Of course, the potential for lost messages increases, but the mechanics of sending, receiving, and processing those messages doesn’t change.

14.1 Many Actors, Many Stages

We know that Actors have ActorRefs. They are part of an ActorSystem, and have a single parent and many children, supervise each other, and can watch any other Actor for death. How does all this fit into a world with multiple nodes?

Figure 14.1 provides some insight into how things work for everything

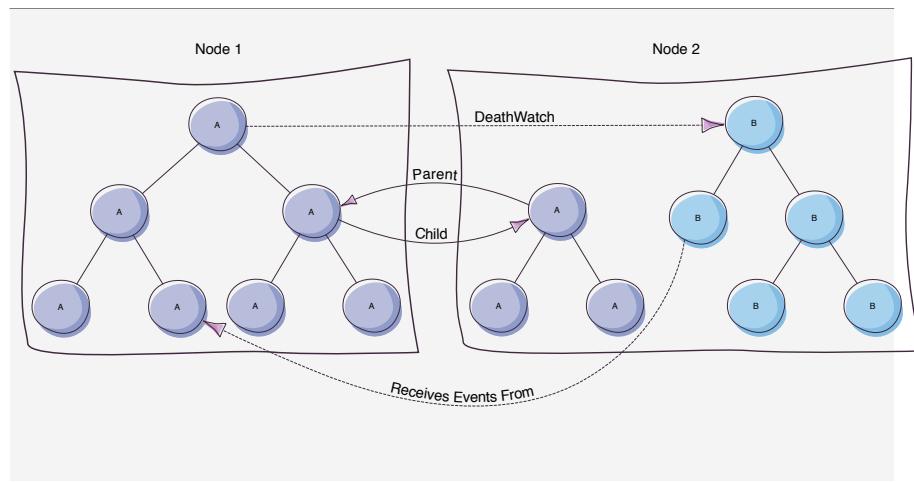


Figure 14.1 · Two nodes in an Akka application. An ActorSystem can span nodes and still maintain the parent/child relationships between them, including Supervision. All ActorRefs appear as ActorRefs, no matter where the Actors may reside, so DeathWatch and standard message passing still apply.

we've already learned about Actors. In many respects (but, of course, not all), the fact that multiple nodes are involved does not affect the Actor model. The ActorSystem isn't concerned about the fact that some of its Actors may be on separate nodes and DeathWatch works no matter where the Actors are running.

14.2 Simple Build Tool (SBT)

To use the Remote module, you'll need to make the requisite modifications to the SBT build file. Not a big deal, but it must be done.

```
libraryDependencies += Seq(  
    // as before  
    "com.typesafe.akka" % "akka-remote" % "2.1"  
)
```

14.3 Remote Airports

We'll start our investigation into remote Actors by introducing a new concept to our simulation: an Airport. It seems reasonable to have one of these on a remote system since airports and planes aren't generally co-located; at least none that I've personally seen.

Our Airport's function will be pretty simple; it will just send directional information to our FlyingBehaviour to help it fly the Plane to the Airport. In the process of creating our Airport, we'll see some cool configuration tricks as well as a simple way to bridge two Actors with incompatible message content.

The Beacon

An Airport will send out a beacon message at regular intervals, so planes can use it to calculate a course. This usually takes the form of an "*I'm at these coordinates*" message so that the plane can figure out in what direction it should head, but we'll simplify that. Our Airport will have a beacon that says, "*Change your heading to ...*". This is clearly silly, since it presumes the beacon knows where the plane is, but it'll do for our example.

We always need to think about testability of our code, which leads us to the following traits:

```
trait BeaconResolution {
    // Allows us to change the interval for testing purposes
    lazy val beaconInterval = 1.second
}

trait BeaconProvider {
    // factory method for creating a beacon
    def newBeacon(heading: Float) = Beacon(heading)
}
```

Let's also abstract away the idea we've been using for "publishing" of events into a common set of messages, which will help us bind a couple of Actors together more generally.

```
object GenericPublisher {
    // These messages are used by any 'event' publisher
```

```
case class RegisterListener(actor: ActorRef)
case class UnregisterListener(actor: ActorRef)
}
```

The Beacon will be quite simple; just something that continually sends a BeaconHeading message to anyone who wants it. It's just the same message over and over and over again.

```
object Beacon {
    // The BeaconHeading is the message that is sent from the Beacon to
    // its listener
    case class BeaconHeading(heading: Float)

    // Factory method to hide the 'ugliness' of the BeaconResolution from
    // everyone else
    def apply(heading: Float) = new Beacon(heading) with BeaconResolution
}

// The Beacon will continually emit the BeaconHeading message with the
// given heading constructor parameter
class Beacon(heading: Float) extends Actor { this: BeaconResolution =>
    import Beacon._
    import GenericPublisher._

    // Our usual timer message
    case object Tick

    // We use a specialized event bus for handling the pub/sub
    val bus = new EventBusForActors[BeaconHeading, Boolean]{
        _: BeaconHeading => true
    }

    // Our ticker to send out periodic beacon headings
    val ticker = context.system.scheduler.schedule(beaconInterval,
                                                beaconInterval, self, Tick)

    def receive = {
        // Subscribe for the Beacon
        case RegisterListener(actor) =>
            bus.subscribe(actor, true)
        // Unsubscribe for the Beacon
        case UnregisterListener(actor) =>
            bus.unsubscribe(actor)
    }
}
```

```
// Publish the BeaconHeading
case Tick =>
    bus.publish(BeaconHeading(heading))
}
}
```

Nearly everything here should be pretty familiar by now, so we don't need to discuss it. The one thing that's unfamiliar is the `EventBusForActors` val. This is a special class, which is not part of Akka, and we discuss it in [chapter 17](#).

The Airport

To create Airports, we'll need to specify some details; for example, the altitude at which incoming planes should stay and the heading that they should use in order to reach it. We'll also need messages and a bit of a helper to instantiate a specific airport.

```
trait AirportSpecifics {
    lazy val headingTo: Float = 0.0f
    lazy val altitude: Double = 0
}

object Airport {
    // Messages consumed by the Airport
    case class HeadTo(flyingBehaviour: ActorRef)
    case class Ignore(flyingBehaviour: ActorRef)

    // Factory method to instantiate the Toronto International Airport
    def toronto(): Actor = new Airport with BeaconProvider
        with AirportSpecifics {
            override lazy val headingTo: Float = 314.3f
            override lazy val altitude: Double = 26000
        }
}
```

If your plane wants to head to Toronto (and why wouldn't it?), then this airport will do you just fine. Let's define the class, so we can move some messages around.

```
class Airport extends Actor { this: AirportSpecifics with BeaconProvider =>
    import Airport._
    import Beacon._
    import FlyingBehaviour._
    import GenericPublisher._

    // Our beacon, which periodically sends out our airport's heading
    val beacon = context.actorOf(Props(new Beacon(headingTo)), "Beacon")

    def receive = {
        // FlyingBehaviour instances subscribe to this Airport in order to
        // be told where they should be flying
        case HeadTo(flyingBehaviour) =>
            val when = (1 hour fromNow).time.toMillis

            // But, we can't let them get BeaconHeading messages, since those
            // are not understood by FlyingBehaviour instances. We need to
            // transform those messages into appropriate 'Fly' messages
            context.actorOf(Props(new MessageTransformer(from = beacon,
                to = flyingBehaviour, {
                    case BeaconHeading(heading) =>
                        Fly(CourseTarget(altitude, heading, when))
                })))
        // Go simple. Realistically only one FlyingBehaviour instance is
        // going to be heading here, so we can just terminate all of the
        // children MessageTransformers (of which there's only one)
        case Ignore(_) =>
            context.children.foreach { context.stop }
    }
}
```

The new bit in here is the `MessageTransformer`. This is a very common pattern in message-oriented programming and the fact that these Actors are untyped enables it. We don't need to change the `Beacon` or the `FlyingBehaviour` in order to get them to work together. [Figure 14.2](#) pretty much says it all.

The code is really simple. We add on a specific registration piece in order to hide that aspect from the `Airport`, but it doesn't need to be so in the general case. You can pass the endpoints in through constructor arguments or the sender can be acquired through the `sender()` method, while the receiver

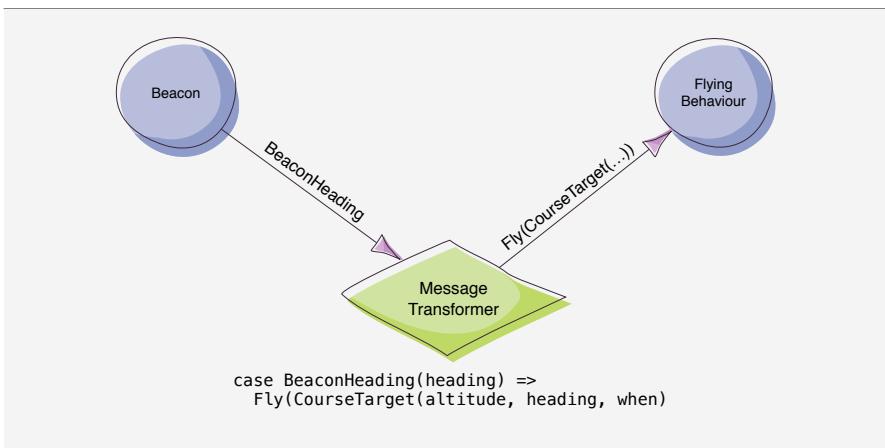


Figure 14.2 · A message transformer converts one message type to another, providing any data transform required during the transformation.

can be an argument in the message. It's all up to you.

```
class MessageTransformer(from: ActorRef, to: ActorRef,  
    transformer: PartialFunction[Any, Any]) extends Actor {  
  import GenericPublisher._  
  
  // Automatically register ourself for messages 'from'  
  override def preStart() {  
    from ! RegisterListener(self)  
  }  
  
  // Automatically deregister ourself for messages 'from'  
  override def postStop() {  
    from ! UnregisterListener(self)  
  }  
  
  // Take the incoming, transform, and send outgoing.  
  // Note how we keep the original sender as 'from', hiding the  
  // transformer from the ultimate receiver  
  def receive = {  
    case m => to forward transformer(m)  
  }  
}
```

14.4 Going Remote

We'll run the Airport on another instance of the JVM soon, but before we do, let's cover what's required in order for that to succeed.

1. The jar files for the application must be present on the Airport node. People often think that Akka will magically distribute code to remote nodes, but imagining it to be so does not make it so. At this point, you must have your code on the remote nodes.
2. The node must be configured. There are requirements that Akka imposes on the person managing the node with respect to configuration, and this means you're on the hook for that.

We'll take the code-deployment question and just assume it's cool. We all know how to install Java, Scala, drop in jar files in the right places, and get our apps to run, so let's mark that as golden.

Configuration is a bit more interesting. You can set or tweak a lot of parameters in the configuration to manipulate the remote system. We'll cover some of them, but not all. You can always refer to the Akka reference documentation for more information.

```
akka {  
    actor {  
        provider = "akka.remote.RemoteActorRefProvider"  
    }  
    remote {  
        transport = "akka.remote.netty.NettyRemoteTransport"  
        netty {  
            hostname = "127.0.0.1"  
            port = 2552  
        }  
    }  
}
```

The required setting is `akka.actor.provider = "akka.remote.RemoteActorRefProvider"`. It's essentially Akka magic that tells it that it can use a different implementation of its underlying guts when providing references to Actors. Everything else is optional. You can

manipulate many settings to mess with Netty (the underlying transport implementation), such as:

- Requiring a secure cookie between endpoints
- Deciding whether or not to reuse an incoming connection for outbound messages
- Changing the host name that the server sends in its messages
- Changing the port on which the server listens
- Determining how much memory to consume
- Deciding how to configure SSL

You can also configure aspects of the Akka-specific settings as well:

- Deciding whether or not to use “untrusted mode.” The messages that Akka uses to communicate with itself (known as “System Messages”) can be either ignored between nodes, or accepted. If they are ignored, then you’re in “untrusted mode” and if they are accepted you are in “trusted mode.”
- Logging of particular events. You can log received messages, sent messages, as well as remote life-cycle events.

For our purposes, we’re only concerned with the host name and the port of the configured instance.

Airport and Plane Configurations

We’ll have two configurations: one for the Airport(s) and one for the Plane. Once we have these configurations, we can run two Akka instances that communicate with one another.

To make testing easier, we’ll have two JVM instances on one machine rather than two JVM instances on two machines. Deploying to two different machines really doesn’t change what we’re doing, other than the configuration file changes that would occur (i.e., change the `akka.remote.netty.hostname` value).

We’ll also put these settings in the same configuration file, but scope them such that we can isolate the configurations from one another.

The Airport Configuration

The Airport configuration is pretty simple, since it looks almost identical to what we've already seen. It exists in the `application.conf` file as the following:

```
airport-remote {  
    akka {  
        actor {  
            provider = "akka.remote.RemoteActorRefProvider"  
        }  
        remote {  
            transport = "akka.remote.netty.NettyRemoteTransport"  
            netty {  
                hostname = "127.0.0.1"  
                port = 2552  
            }  
        }  
    }  
}
```

The Plane Configuration

The Plane is slightly different, since we'll have the Plane's JVM connect to the Airport's JVM. Thus, the Plane needs to understand the Airport and not the other way around. We'll configure the information about the remote Airport rather than hard-code that information into the Plane's source code.

```
plane-remote {  
    zzz {  
        akka {  
            avionics {  
                # The system, host and port of the Airport we want to use  
                # Alternatively you might want to have a list of these:  
                # e.g. airports = {  
                #                 [ "Airport1", "host1", 2552 ],  
                #                 [ "Airport2", "host2", 2552 ],  
                #                 [ "Airport3", "host3", 2552 ]  
                #             }  
            }  
        }  
    }  
}
```

```
    airport-system = "Airport"
    airport-host = "127.0.0.1"
    airport-port = 2552
  }
}
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"
    netty {
      hostname = "127.0.0.1"
      port = 2553
    }
  }
}
}
```

Note that the Plane's instance will be running on the local host, but on a different port than the Airport's instance (2553 vs. 2552 respectively).

14.5 Flying to the Airport

Not surprising, most of what's involved to make this work doesn't have much to do with Akka; it's just monkey work to run the two apps and make sure they're in good shape.

Currently, there isn't a ready-for-market solution that makes this work any easier, but it's close. Eventually, both a *multi-jvm* and *multi-node* SBT plugin will become available that makes running these sorts of tests incredibly easy.¹

¹I've tried it myself by compiling the required Akka source and publishing it locally on my own, and it works well. It's just not the sort of procedure you can reliably write about in a book.

The Execution Script

We're going to need something to help us run our code. With the help of SBT and a bit of shell scripting, we can make this pretty easily. This will only work on Mac OS X or Linux (or some other unix-like OS). On Windows, you'll have to do it the way you have to do everything else...with more difficulty².

```
#!/bin/bash

# Get the classpath with SBT's help
echo "Getting classpath..."
jars=$(sbt "show test:managed-classpath" 2>&1 | grep 'List' | \
    tr ',' '\n' | sed -e 's/.*(\([^\)]*\))*.*/\1/' | tr '\n' ':')

# You need to be in the root directory of the project for this
# If you're compiling against many versions of scala, this will take
# the "last one listed", whatever that is
echo "Expanding classpath..."
classes=$(ls -d target/scala*/classes | tail -1)
testclasses=$(ls -d target/scala*/test-classes | tail -1)
cp=$jars:$classes:$testclasses

# Create a run function to make things easier
function run {
    scala -cp $cp org.scalatest.tools.Runner -o -s "$@"
}

# Run the first N-1 in the background
c=1
while [[ $c -lt $# ]]
do
    eval echo Running $$c...
    eval run $$c &
    ((c=c+1))
done
# Run the last one as a blocking process
eval echo Running $$#...
eval run $$#
```

²Sorry, there's a reason I never use Windows ☺

If you don't know what that does, don't sweat it. The trickiest bit is the stuff that calculates cp (i.e., the class path), and if it looks like magic to you, then just accept the magic and move on.

Using this script, we'll execute two separate runtimes, which will talk to each other across your machine's loopback network.

AirportRemoteSpec

While we could just create a main function, instantiate our separate classes, and check the basic functionality, it's much nicer to use the facilities of ScalaTest and the TestKit. The AirportRemoteSpec isn't really much of a test class, but it will suffice to help us test the overall communication.

```
class AirportRemoteSpec
  extends TestKit(ActorSystem("Airport", RemoteConfig.config))
    with ImplicitSender
    with WordSpec
    with BeforeAndAfterAll
    with MustMatchers {

  // Breathes life into the test actor. If we don't do this then there won't
  // be any testActor for the other side to communicate with.
  val t = testActor

  override def afterAll() {
    system.shutdown()
  }

  "AirportRemote" should {
    "start up" in {
      if (RemoteConfig.runningRemote) {
        val toronto = system.actorOf(Props(Airport.toronto()), "toronto")
        // We're going to let the other side tell us when to shut down
        expectMsg("stopAirport")
      }
    }
  }
}
```

Pretty simple stuff here. We start up the system and just wait for the other side to tell us to shut down. The expectMsg() will time out after 3

seconds, by default, which is more than enough time to complete the test. The `expectMsg()` method allows you to increase that time if necessary.

Note how we're creating the `ActorSystem`. The `airport-remote` sub-config that we created earlier can be loaded dynamically as a configuration all on its own. This allows us to inject that configuration directly into the `ActorSystem` irrespective of any other sub-configs that might be present.

PlaneRemoteSpec

The setup for the `PlaneRemoteSpec` does much the same as the `AirportRemoteSpec`. The internals of the `PlaneRemoteSpec` are more interesting since it actually executes the test.

```
class PlaneRemoteSpec
  extends TestKit(ActorSystem("Plane", RemoteConfig.config))
  with ImplicitSender
  with WordSpec
  with BeforeAndAfterAll
  with MustMatchers {
  import Airport._
  import FlyingBehaviour._
  import RemoteConfig._

  override def afterAll() {
    system.shutdown()
  }

  // Get an ActorRef for the remote system's testActor
  def remoteTA(): ActorRef = system.actorFor(
    s"akka://$asys@$host:$port/system/testActor1")
  // Get an ActorRef for the remote system's toronto airport
  def toronto(): ActorRef = system.actorFor(
    s"akka://$asys@$host:$port/user/toronto")

  // Tells us whether or not the toronto airport has actually been found
  def actorForAirport: Boolean = toronto() != system.deadLetters

  "PlaneRemote" should {
    "get flying instructions from toronto" in {
      if (runningRemote) {
        // Wait for the toronto airport to come online
      }
    }
  }
}
```

```
awaitCond(actorForAirport, 3.seconds)
val to = toronto()
// Fly there
to ! HeadTo(testActor)
// We should be getting Fly directives
expectMsgPF() {
    case Fly(CourseTarget(altitude, heading, when)) =>
        altitude must be > (1000.0)
        heading must be (314.3f)
        when must be > (0L)
    }
    // We got it. Shut down.
    remoteTA() ! "stopAirport"
}
}
}
}
```

Note that we are pulling the ActorRef for the Toronto Airport and the remote testActor (we know it will be called `testActor1` and is a child of the `/system` guardian).

There's an invocation of TestKit's `awaitCond()` to ensure that we actually get the reference. We have to handle the race conditions imposed by starting up independent systems, so we'll use the `awaitCond()` to help us here. Akka's upcoming facilities give us "barriers" that solve this problem much better.

To run this, we use the script we created:

```
run_remote_tests.sh zzz.akka.avionics.AirportRemoteSpec \
                    zzz.akka.avionics.PlaneRemoteSpec
```

This test should succeed just fine, even though it does the following next-level (and pretty awesome) stuff:

- Launches the remote Airport and instantiates the Toronto instance.
- Retrieves that instance, using the remote address; however, the obtained reference looks just like any other ActorRef we've seen before.

- Registers a new ActorRef to the Toronto airport to receive events. The local ActorRef (i.e., the PlaneRemoteSpec's `testActor`) gets properly serialized and sent over the “wire” such that messages sent are routed back to the right spot—yes, that's *awesome*.
- Transforms the Beacon message into a Fly message, magically serializes that across the “wire,” drops it into the `testActor`'s Mailbox in a deserialized format, and processes it as normal.

Our code, aside from the need to look up the Toronto airport using a fully specified URI, is entirely ignorant of the complexity that is occurring, which is as it should be from any toolkit as advanced as Akka.

Take a minute and let that sink in. Now go tell a loved one that you may have just discovered the multi-node concurrency equivalent of the *Hammer of Thor*.

14.6 Programmatic Remote Deployment

One neat thing that you can do with the Akka remoting subsystem is to dynamically deploy an Actor to a remote node from inside another running Actor on the local node.

For Akka to perform the deployment, you need to specify a “scope” for that deployment. Much like before, we can have some test code, but this time it will deploy the Airport instead of connecting to it.

```
// Tells us when the Airport spec's testActor comes online
def actorForAirportTA: Boolean = remoteTA() != system.deadLetters

"PlaneRemoteDeployClientSpec" should {
    "deploy the Airport remotely" in {
        if (runningRemote) {
            // These are some new imports we haven't seen before
            import akka.actor.{Deploy, Address, AddressFromURIString}
            import akka.remote.RemoteScope

            // Wait for the Airport test system to come online
            awaitCond(actorForAirportTA, 3.seconds)

            // Another way to get an Address of a remote system
            val addr = Address("akka", asys, host, port.toInt)
        }
    }
}
```

```
// Deploy the Airport
val toronto = system.actorOf(Props(Airport.toronto()).withDeploy(
    Deploy(scope = RemoteScope(addr))), "toronto")

// Just verify that it looks like we expect
toronto.path.name must be ("toronto")
toronto.path.address must be (addr)

// Tell the Airport spec to validate the test
remoteTA() ! "goodToGo"
}

}

}
```

Notice how the Props usage states that it should have a specific mode of deployment. We use the Deploy class to specify how/where a particular Actor path should be created. Of particular interest here is how we're using the scope parameter to specify a new deployment scope for the Airport. The deployment scope directs Akka to deploy the Actor in a LocalScope or a RemoteScope. Generally speaking, there's no reason to use LocalScope and thus we're only interested in the RemoteScope here. The RemoteScope requires an Address in its construction, which defines the remote node on which the deployment is to take place.

The verifications we make in the test merely indicate what the particular aspects of the ActorRef's path become. These will contrast with the resulting path of the created Actor on the remote node. Let's look at that aspect of the test now.

```
// Breathes life into the test actor. If we don't do this then there won't
// be any testActor for the other side to communicate with.
val t = testActor

def actorForAirport: Boolean =
    system.actorFor("/user/toronto") != system.deadLetters

"AirportRemoteDeployServer" should {
    "let the Plane deploy an Airport over here" in {
        if (RemoteConfig.runningRemote) {
            expectMsg("goodToGo")

            // Wait for the toronto airport to come online
            awaitCond(actorForAirport, 3.seconds)
        }
    }
}
```

```
    val toronto = system.actorFor("/user/toronto")

    // Verify that it looks as we expect
    toronto.path.name must be ("toronto")
    toronto.path.address.system must be ("Airport")
    toronto.path.address.port must be (None)
    toronto.path.address.host must be (None)

}

}

}
```

Our “server” here simply spools up and waits to be told to verify the test. When the `testActor` receives the message “`goodToGo`”, it awaits the creation of the Toronto Airport and then validates the aspects of the created path.

Note how the path is entirely ignorant of the “remoteness” aspect of things. Akka is managing this for us.

It’s time to discuss the *Hammer of Thor* again. This Props class is something we’ve been using for quite a while, but its importance has never really been all that clear. The deployment aspect of it, and the fact that it’s encapsulating these construction-dependent details for us, makes it largely an awesome abstraction. The factory method (i.e., `Airport.toronto()`) gets transmitted across to the remote node. It’s not that we construct the object on the local node and then send it over; we’re sending the construction *command* instead. The Props class hides all of this stuff from us, allowing Akka to do the awesome work of constructing the Actor remotely and stitching its existence back across the wire so that we can communicate with it.

Relativity Effects

Performing a deployment like this is different than what we did previously, where we looked up a pre-existing Actor instance on a remote machine.

With the lookup strategy, our Plane system’s test simply had an `ActorRef` to an Actor running inside a remote node, but when we *deploy* from another system, the relationships are different. The relationship isn’t different than anything we’ve seen before, however; it’s a parent/child relationship.

To be more specific, let’s say we created the deployed Actor from within another Actor, like this:

```
context.actorOf(Props(Airport.toronto()).withDeploy(
```

```
Deploy(scope = RemoteScope(addr))), "toronto")
```

Then "toronto" becomes a child of that current Actor, subject to all of the usual aspects therein. It's now easy to see how you can couple Actor hierarchies together on remote nodes, if that's a strategy you need, or just as easily decouple them for whatever purposes you deem necessary. Flexibility is one of the design drivers for Akka, and it shows.

14.7 Configured Remote Deployment

We can also take the programmatic approach we just used via configuration. You should certainly consider this approach for particular aspects of your application that you want Administrators to have some control over.

The details of remote deployment are now well understood, so learning how to do it with configuration is a snap. In fact, in code, it looks even easier than it does without configuration; it actually doesn't look like we're doing anything at all!

We can create a configuration for our Actor that hooks on to the Actor's create path, and Akka will then intercept that creation attempt based on the specified creation path and forward that creation to the remote node.

```
akka.actor.deployment {  
    /toronto {  
        remote = "akka://Airport@airportHost:2552"  
    }  
}
```

Now in the code, we just need to create the Actor in a way that looks like we're creating it locally; we don't need to know what's going on since the Administrator will take on that burden.

```
// "Magically" gets created on the remote node!  
val toronto = context.actorOf(Props(Airport.toronto()), "toronto")
```

Hammer of Thor, dear reader, Hammer of Thor.

14.8 Routers Across Multiple Nodes

Remember that somewhat enigmatic ScatterGatherFirstCompletedRouter we saw earlier? Up until now, it probably seemed like a bit of an odd duck. Why spawn a bunch of the same jobs on the local machine, sucking all kinds of CPU, and then ignore most of that work? Well, yeah, that's pretty odd indeed. But, when you can magically deploy routees to remote nodes, things become a bit more impressive.

Just as Routers can be configured to do what they do, and remote deployment can be configured to do what it does, so can remote Router deployment. You understand Routing, and you understand Remoting, so now you just need to understand the configuration glue that sticks them together.

Assume you have an Actor that calculates Fibonacci numbers for you... *big ones*. You don't know how busy the machines in your network are, or how powerful they are, and you're an evil genius that has no problem sucking down CPU on tons of machines, so you define this:

```
akka.actor.deployment {  
    /evil/fibonacci {  
        router = "scatter-gather"  
        nr-of-instances = 20  
        target {  
            nodes = [  
                "akka://system@10.0.0.1:2552",  
                "akka://system@10.0.0.2:2552",  
                "akka://system@10.0.0.3:2552",  
                ...  
                "akka://system@10.0.0.20:2552"  
            ]  
        }  
    }  
}
```

Now when you send your request to calculate the 200,000,000th Fibonacci number to the /evil/fibonacci router, it will beat up 20 different machines and make them weep. The first one that finishes will return results to you, and the others will just continue chugging along until their hearts explode.

14.9 Serialization

Up to this point, serialization has been a no-brainer. We've been using data types that are all easily serialized by default, so there's literally no work for us to do. This is absolutely ideal, and if you can stay in this world, stay in it. Why write serializers and deserializers when you don't have to?

However, there are entities out there that cannot be serialized out of the box. They may be things that you create or things that someone else creates, but in either case if you want to send them to a remote Actor, you have to provide a serializer and deserializer.

When You Need Custom Serialization

There's a difference between implementing a class such that it is serializable and implementing your own serializer. If you merely want to make a certain class serializable, then you just need to implement the appropriate interface or trait, depending on which serialization method you need.

Another thing to note is that *case classes* are immediately serializable (assuming they aren't composed of something that isn't serializable). So if all you use are types that are serializable (e.g., String, Int, Boolean, ActorRef, etc.) and you compose these types inside *case classes*, then there's literally *nothing* you need to do.

However, if you really do have a need to implement a *serializer* of your own, you'll find that doing so isn't all that difficult. Let's assume you have this wonderful class that isn't a *case class* and you need to create a serializer for it.

```
class SerializeMe(val message: String) {  
    // When we don't use a case class, there's a lot we don't get for free,  
    // and this is one of them. At a minimum we need to implement equals()  
    // in order to test that serialization works.  
    override def equals(a: Any): Boolean = {  
        if (!a.isInstanceOf[SerializeMe]) false  
        else a.asInstanceOf[SerializeMe].message == message  
    }  
}
```

Creating the Serializer

We can create a serializer for `SerializeMe` by implementing what's required by Akka's `Serializer trait`.

```
class SerializeMeSerializer extends akka.serialization.Serializer {  
    // We don't need to inspect the type of this thing. We know what it is.  
    // If we wanted to be more general, Akka would ship the manifest to us  
    // and let us inspect what it is that we should be deserializing  
    def includeManifest: Boolean = false  
  
    // We need a unique identifier here. Make one up... intelligently  
    def identifier = 28591953  
  
    // Convert the object to a binary representation  
    def toBinary(obj: AnyRef): Array[Byte] = {  
        if (!obj.isInstanceOf[SerializeMe]) {  
            throw new java.io.NotSerializableException(  
                s"SerializeMeSerializer can't serialize ${obj.getClass.getName}%s")  
        }  
        val m = obj.asInstanceOf[SerializeMe]  
        m.message.getBytes("UTF-8")  
    }  
  
    // Convert the binary representation back to an object  
    def fromBinary(bytes: Array[Byte], clazz: Option[Class[_]]): AnyRef = {  
        new SerializeMe(new String(bytes, "UTF-8"))  
    }  
}
```

Wiring Up the Serializer

Next, we need to wire up the serializer to the class that it works with by modifying the configuration file. You would make these changes in the `application.conf` as usual.

```
akka.actor {  
    serializers {  
        special = "zzz.akka.investigation.SerializeMeSerializer"  
    }  
}
```

```
serialization-bindings {  
    "zzz.akka.investigation.SerializeMe" = special  
}  
}
```

Verifying the Serializer

And we can verify that it works by performing a manual serialization/deserialization operation:

```
import akka.actor.ActorSystem  
import akka.serialization._  
  
val system = ActorSystem("SerializationSpec")  
val serialization = SerializationExtension(system)  
  
"Serializer" should {  
    "serialize custom stuff" in {  
        val original = new SerializeMe("Hithere")  
        val serializer = serialization.findSerializerFor(original)  
        val bytes = serializer.toBinary(original)  
        val copied = serializer.fromBinary(bytes, manifest = None)  
        original must be(copied)  
    }  
}
```

Serialization Options

There are other options for serialization that make certain claims that you might find interesting—most notably, *speed*. The Akka reference topic on Serialization may be more current than this book, so you should consult the reference for the real answer. But at the time of writing, the alternatives are:

- *Protocol Buffers*: a serializer for serializing using the Google Protocol Buffer format.
- *Quickser*: a serialization library that claims to be “quick” (hence the name).
- *Kryo*: another serialization library that has some pretty good traction.

We won't look at or review them here; just be aware that you have options if you want.

Serialization Tips

Let's clarify a few things regarding serialization:

- The serializer we created isn't exactly ideal; you wouldn't want to declare a separate serializer in the configuration file for every class that you create. What you might do instead is implement serialization of the standard Java Serializer (i.e., implements `java.io.Serializable`) and not need to do anything else. You could, of course, do this for any of the other serialization options.
- Watch out for any accidental closures. If you're trying to get a default serializer to serialize something that you've defined as an inner class of something else, you may find that the "accidental" closure you've created fails to serialize.
- If you really do need to create your own serializer, you'd do so using a trait or an interface. The binding you would then create would declare that the interface is what is bound to the serializer. This is exactly how Akka handles Java Serialization, for example. The `akka.serialization.JavaSerializer` is bound to classes of type `java.io.Serializable`. In other words, we introduce a layer of indirection, and thus solve every known problem of the human race.

14.10 Remote System Events

An Aside into Language

Akka Remote Nodes operate in a "peer" sense; they're both clients and servers to each other. This is probably what you'd expect, and it makes perfect sense... until you start talking about it. The terminology we use to describe systems such as these hasn't quite evolved as well as the code has, and as such these discussions tend to become rather convoluted rather quickly.

The only clear way I've found to discuss these sorts of systems, in general, is to speak in terms of "client" and "server," but to bind those words to a particular node context. To ensure that your friends and co-workers don't

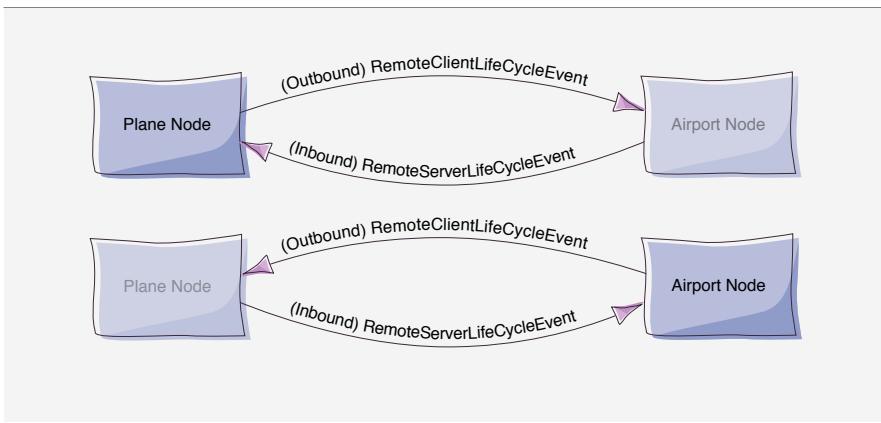


Figure 14.3 · The words `RemoteClientLifeCycleEvent` and `RemoteServerLifeCycleEvent` only have meaning when we bind them to a particular node context. A “client” event for the Plane Node is a corresponding “server” event for the Airport Node and vice versa.

sock you in the mouth during a heated Akka Remote discussion, you should be diligent about saying things like “Plane Server” and “Airport Client” instead of just “Server” and “Client.” The latter misstep will probably be something you eventually regret. The image of Figure 14.3 should be ever-present in your mind.

Remote Life-Cycle Events

Clearly Akka is doing some work for us behind the scenes that lies in the realm of the ultra-neato. If you need to snoop in on these events, for the purposes of reacting to them, you just need to look to the Event Stream.

To illustrate the events, how to use them and what to expect, we’ll implement the code for what you see in Figure 14.4. We’ll put this code into similar test specs that we’ve had before, for our Plane and our Airport, and they can be run using the same script we created earlier.

The Airport spec is wonderfully simple; we just need to wait for the Plane to tell it to shut down.

```
// Breathes life into the test actor. If we don't do this then there won't  
// be any testActor for the other side to communicate with.
```

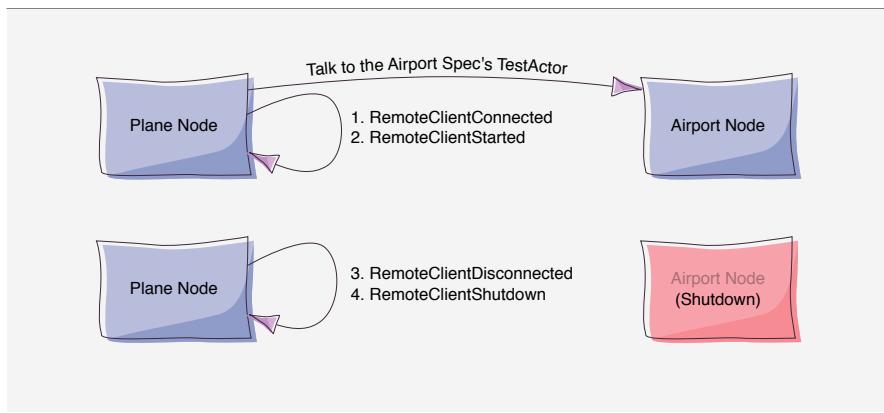


Figure 14.4 · Sending the message to the Airport Server connects and starts the Plane Client, while the shutdown of the Airport Server subsequently disconnects and shuts down the Plane Client.

```

val t = testActor
"AirportShutdownSpec" should {
  "shutdown when told to" in {
    if (RemoteConfig.runningRemote)
      expectMsg("stopAirport")
  }
}

```

We'll examine the Events in the Plane's spec. Note that these are simply Plane Client events that we're examining.

```

"PlaneShutdownSpec" should {
  "deploy the Airport remotely" in {
    if (RemoteConfig.runningRemote) {
      // Specifically, we're interested in the following events
      import akka.remote.{RemoteClientLifeCycleEvent,
        RemoteClientConnected,
        RemoteClientDisconnected,
        RemoteClientStarted,
        RemoteClientShutdown}
      import akka.actor.Address

```

```
// Subscribe to all events that pertain to the Plane Client
system.eventStream.subscribe(testActor, classOf[RemoteClientLifeCycleEvent])

// Wait for the Airport test system to come online
awaitCond(actorForAirportTA, 3.seconds)

// Tell the Airport to shut down
remoteTA() ! "stopAirport"

// Sending that message initiates all of the following events
expectMsgClass(classOf[RemoteClientConnected])
expectMsgClass(classOf[RemoteClientStarted])
expectMsgClass(classOf[RemoteClientDisconnected])
expectMsgClass(classOf[RemoteClientShutdown])

}

}

}
```

There are more events that you can look at, and you should consult the Akka reference documentation to see them. The mechanism by which you register for them and examine them doesn't change; the different events merely tell you different things.

14.11 On the Subject of Lost Messages

One thing that comes up a lot on the Akka Mailing list revolves around the notion of “guaranteed delivery.” Akka makes a very clear and concise statement on this subject: *Akka does not provide guaranteed delivery*. And, of course, this is often met with a *WTF?*³

A system that provides guaranteed delivery can be very expensive and complicated, and generally speaking you don't need it. There are a whole host of applications that work just fine without it and the Actor model, which provides deterministic behaviour in the case of failure, helps you handle these situations with a fair bit of grace.

With that said, we will cover a pattern (later in the book) that helps you handle these situations before they fail. For now, just hang tough. Guaranteed delivery isn't always the absolute-necessary-must-have-or-my-world-is-

³I'm pretty sure this stands for *Wallabies Taste Fruity*, but I'm not sure why people keep using it in this context.

going-to-be-sucked-into-a-black-hole thing that people often imagine it to be.

14.12 Clustering

At the time of this writing, the Akka team is designing and building a clustering feature. The details haven't been fully worked out yet and the feature itself is a bit of a moving target, so it's not settled enough to talk about in any great detail. But...

The clustering feature will fill in several needs in the "large computing" space. It will provide things such as load-balancing, redundancy, and no single-point-of-failure fault-tolerance. This is the kind of stuff that helps you handle the problems of trying to run your software for millions or hundreds of millions of concurrent users across oceans of time and space.

In this chapter, we didn't cover things like what to do when a node fails, for example. At the present time, you would have to code these solutions on your own. You can certainly do it, but the plan is to have the clustering feature solve these types of problems much better than you or I could solve them.

So, rest assured that the Akka team has your back if you're a large-scale shop. By the time you've read this book, and have become a proficient Akka developer, the clustering feature will be out or will be on the cusp.

14.13 Chapter Summary

That was quite the whirlwind tour of remoting in Akka. The bottom line is that the Akka team has done a magnificent job of keeping your application mostly ignorant of all of the complexity that revolves around the remoting system. Your skills have increased too; if this were an online multi-player game, then by now you'd be a Level 7 Akka Mage with the power to transmute squirrels into demons in charge of the undead.

Let's take a minute to sum up:

- ActorRefs are ActorRefs, no matter where the Actor is deployed. Your application doesn't need to understand where things are, and even sending an ActorRef in a message to an Actor on a remote machine works as expected.

- Message serialization is basically automatic. It's only when you do something special (like using a class that's not automatically serializable) that you need to do something "special" (like writing a serializer for it).
- When you want to deploy to remote nodes, you have the power to do so programmatically or via configuration. The configuration route allows you to put more power into the hands of your Administrators, and create less work for yourself.
- Routers and Remoting both start with "R", and they work very well together. Coincidence? Don't kid yourself.
- You've gained insight into the event model that surrounds the remoting module and can hook into the client and server life cycles of the nodes in question.
- You're more than ready for discussions around servers and clients in context.
- The subject of "guaranteed delivery" is firmly cemented in your neo-cortex⁴ and you're ready to write your applications with that in mind.

We've even talked about the yet-to-be-released clustering feature. Clustering isn't required for many things but it certainly is nice to have in your toolkit (well, it's a must have) when you want to go *really* big. The lack of clustering doesn't really stop you from going big at all, it just means you have a bit more work to do, and you have a lot of the skills to make this happen now, if necessary.

Before moving on to the next chapter, if you want to take some time transmuting a few squirrels into some undead demon masters, I'll totally understand. Just keep them away from the puppies.

⁴Actually, that's not where it went, but it sounds good.

Chapter 15

Sharing Data with Agents

Actors hide their data in a secure fortress, ensuring that access from the outside world is controlled and safe. When you want to get a value from an Actor, you send it a message and the result comes back at some point later. For reactive-based programming, this is great, but when all you want to do is modify and/or access some sort of data that is effectively *shared*, it can be a little less than optimal. Agents are designed for this sort of situation.

Agents invert a concept that you normally would write on your own. For example, if you want an Actor to represent some sort of counter, you might write this:

```
object CounterActor {
    case class AlterBy(value: Int)
    case object GetValue
}

class CounterActor extends Actor {
    import CounterActor._

    var counter = 0

    def receive = {
        case AlterBy(value) =>
            counter += value
        case GetValue =>
            sender ! counter
    }
}

implicit val askTimeout = Timeout(1.second)
```

```
val counter = system.actorOf(Props[CounterActor])
counter ! AlterBy(1)
Await.result(counter ? GetValue, 1.second) must be (1)
```

We code the logic for knowing what to do to the internal state into the Actor itself; the outside world sends a message to the Actor that invokes the logic within it. Agents invert this by being more general. An Agent accepts logic that operates on the internal data and modifies the value using that logic. From a practical perspective, the following is identical to the previous example's functionality:

```
implicit val awaitTimeout = Timeout(1.second)
val counter = Agent(0)
counter send { _ + 1 }
counter.await must be (1)
```

At this point in your understanding, it's not hard to guess what the Agent is doing. Agents are helpful wrappers around Actors. While they invert the paradigm that we would normally write (i.e., moving the logic outside the Actor instead of inside), they also provide a non-message-based interface to them, which can give us more convenient access to their functionality. But the heart of the Agent is, most definitely, an Actor.

15.1 SBT

Agents are a separate module in Akka, which means that we need to include them in the SBT build in order to use them.

```
libraryDependencies += Seq(
  // as before
  "com.typesafe.akka" % "akka-agent" % "2.1"
)
```

15.2 Agents as Counters

One of the most common uses for Agents is to implement *counters*. A counter in this context denotes something that is used to collect statistics

about a running application. You can have your Actors publish stats to an Agent, or set of Agents, that you can then interrogate as often as you see fit. You can also log them regularly on a timer, if you wish.

We can add a whole host of counters to our Plane, such as the number of drinks served, the number of course changes, how many times the auto pilot was engaged, the mean time between control changes, and even the number of times people used the bathroom.

In the spirit of continuing to learn by example, let's build the bathrooms on the Plane and add a couple of counters to it.

The Most Important Part of the Plane

I don't know what you think the most important part of the plane is, but I used to fly from Canada to Switzerland regularly and, while I'm sure there are all kinds of useful doo-dads and whatcha-hoozits that "flying people" will tell you are really important, when you're flying for 10 hours non-stop with two meals and a lot of drinks, the lack of a solid lavatory would be *noticed*.

So, let's build some Lavatories using the skills we have attained thus far. Since only one person can use it at once, and it can be put into a state of Vacant or Occupied, we'll model this with a state machine.

```
// GenderAndTime will be what's stored in two Agents, one for Male and
// one for Female passengers
sealed abstract class Gender
case object Male extends Gender
case object Female extends Gender

case class GenderAndTime(gender: Gender, peakDuration: Duration, count: Int)

object Bathroom {
    // The States for our FSM
    sealed trait State
    case object Vacant extends State
    case object Occupied extends State

    // The Data' for our FSM
    sealed trait Data
    case class InUse(by: ActorRef, atTimeMillis: Long,
                    queue: Queue[ActorRef]) extends Data
    case object NotInUse extends Data
}
```

```
// Messages to and from the FSM
case object IWannaUseTheBathroom
case object YouCanUseTheBathroomNow
case class Finished(gender: Gender)

// Helper function to update one of the counters
private def updateCounter(male: Agent[GenderAndTime],
                           female: Agent[GenderAndTime],
                           gender: Gender, dur: Duration) {

    gender match {
        case Male => male send { c =>
            GenderAndTime(Male, dur.max(c.peakDuration), c.count + 1)
        }
        case Female => female send { c =>
            GenderAndTime(Female, dur.max(c.peakDuration), c.count + 1)
        }
    }
}

class Bathroom(femaleCounter: Agent[GenderAndTime],
               maleCounter: Agent[GenderAndTime]) extends Actor
    with FSM[Bathroom.State, Bathroom.Data] {

    import Bathroom._

    startWith(Vacant, NotInUse)

    when(Vacant) {
        case Event(IWannaUseTheBathroom, _) =>
            sender ! YouCanUseTheBathroomNow
            goto(Occupied) using InUse(by = sender,
                                         atTimeMillis = System.currentTimeMillis,
                                         queue = Queue())
    }

    when(Occupied) {
        // Can't use the bathroom now... queue up
        case Event(IWannaUseTheBathroom, data: InUse) =>
            stay using data.copy(queue = data.queue.enqueue(sender))
        // Note that we guard the case by the identity of the sender
        case Event(Finished(gender), data: InUse) if sender == data.by =>
```

```
// Update the appropriate counter
updateCounter(maleCounter, femaleCounter, gender,
              Duration(System.currentTimeMillis - data.atTimeMillis,
                       TimeUnit.MILLISECONDS))

// Move on to the next state
if (data.queue.isEmpty)
    goto(Vacant) using NotInUse
else {
    val (next, q) = data.queue.dequeue
    next ! YouCanUseTheBathroomNow
    stay using InUse(next, System.currentTimeMillis, q)
}
}

initialize
}
```

Our bathrooms in all their glory are now ready for use. We can see that the counters get updated properly in the bathrooms themselves, but we still have to supply those counters. In general, the best place to do this is at the application level, which in our case is the Plane. The modifications to it are fairly straightforward and our goal is to produce something like Figure 15.1.

```
class Plane extends Actor with ActorLogging {

    ...

    val maleBathroomCounter = Agent(GenderAndTime(Male, 0.seconds, 0))
    val femaleBathroomCounter = Agent(GenderAndTime(Female, 0.seconds, 0))

    // Start up the bathrooms behind a 4-instance router
    def startUtilities() {
        context.actorOf(Props(new Bathroom(femaleBathroomCounter,
                                           maleBathroomCounter)).withRouter(
            RoundRobinRouter(nrOfInstances = 4,
                            supervisorStrategy = OneForOneStrategy() {
                case _ => Resume
            })), "Bathrooms")
    }

    def startPeople() {
```

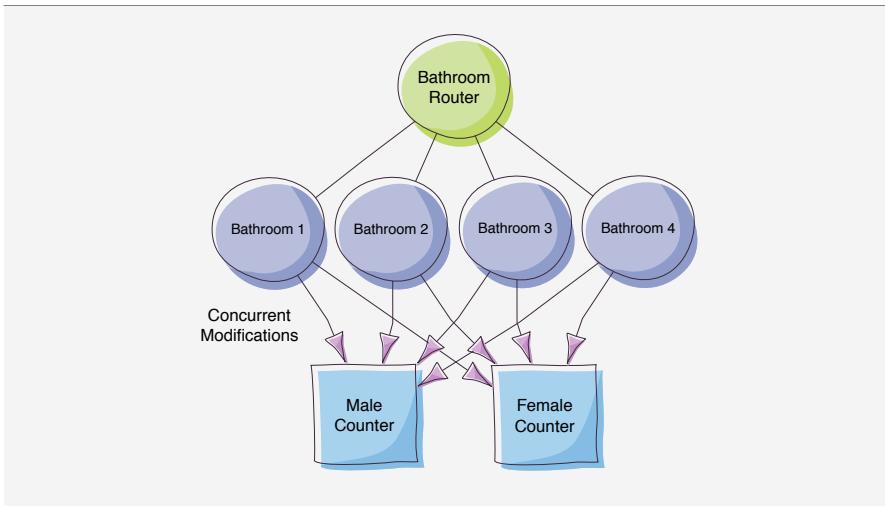


Figure 15.1 · To create more than one bathroom, we'll use a Round Robin Router and give each bathroom access to the counter Agents (which will be housed inside the Plane).

```
...
val bathrooms = actorForControls("Bathrooms")
...
val people = actorOf(Props(new IsolatedStopSupervisor with OneForOneStrategyFactory {
    def childStarter() {
        context.actorOf(Props(PassengerSupervisor(leadAttendant, bathrooms)),
                      "Passengers")
    ...
}
override def preStart() {
    // Get our children going. Order is important here.
    startControls()
    startUtilities()
    startPeople()
    ...
}
override def postStop() {
    // Await the values. If the plane is shutting down then, most likely,
```

```
// the whole system is going down. If we get a Future result to the values
// then we may not get them until the system has disappeared. Bad news
val male = maleBathroomCounter.await
val female = femaleBathroomCounter.await

// Explicitly close them so they can be reaped
maleBathroomCounter.close()
femaleBathroomCounter.close()

log.info(s"${male.count} men used the bathroom")
log.info(s"${female.count} women used the bathroom")
log.info(s"peak bathroom usage time for men was ${male.peakDuration}")
log.info(s"peak bathroom usage time for women was ${female.peakDuration}")

}
```

These are the interesting parts of the modifications we need to make; we won't belabour the point by going through all the changes required to get the bathrooms router instance sent down to the passengers since it's essentially monkey work and you know how to do all that.

What's important to understand here is that the counters themselves are held at the Plane level. The Plane is the granddaddy of the application life cycle and thus it's a good spot to store things like this. It's also a convenient spot to report the Plane's shutdown time. When we're reporting, there are more interesting things to note:

- We block on the results from the Agents. Normally, in Akka programming, we never block since blocking a thread is pretty evil; however, in this case, blocking is important.
- We want to log our statistics at the time the Plane shuts down.
- If we just get the current value instead of awaiting the final value, then our statistics wouldn't be as accurate as we'd like.
- If we use a Future to get the value, then by the time the Future executes, the system may be entirely shut down, which would make the Future fail.
- We explicitly close the Agents. This allows them to be reaped by the garbage collector. We know that until we stop an Actor the Actor is alive and can't be collected—this is the exact same thing.

We haven't covered some aspects of the Agent's API yet, though we've hinted at them above. Now that you have a pretty solid understanding of the Agent's design aspects, we'll cover the API in some more detail.

15.3 Working with Agents

Agents are a little more involved than we've seen thus far. Note that they're not *complicated*, just more *involved*. This is actually a pretty common thing that I tend to say about Akka—it's not complicated; in fact, it's really quite simple. If you could call anything complicated, it would be the concurrency that Akka helps you implement. In this case, the Agent interface has more methods than we've seen thus far that ease the real-life aspects of concurrency, which follows the theme of Akka as a whole.

There's Always a Value, But...

An Agent has some aspects that match Actor and Future, but they aren't identical to them. For example, if you try to get the value from a Future when there's no value yet available, you'll get an Exception. An Agent differs because it always has a value. That value may change, but at any given point in time it always has a value. This is obvious from the fact that constructing an Agent requires an initial value:

```
val secretAgent = Agent(007)
```

So, no matter what happens concurrently with anything in your app anywhere, anytime, you can always get a value from this Agent, even it's only 007. But of course, this is about concurrency, which means that you need to be realistic about what "current" really means. Before we start dissecting the API, remember that concurrency makes determining the following assertion impossible:

```
val secretAgent = Agent(007)
secretAgent send { _ + 1 }
// secretAgent.get() must be ... what?
```

At the time when we evaluate `get()`, the `secretAgent` could be any one of five values. This is because `get()` is being evaluated on the main thread while the other operations are being evaluated on a separate thread. This happens through the magic of Scala’s Software Transactional Memory (STM) implementation, and doesn’t concern us. It’s nifty and awesome, and the fact that we can ignore it right now is probably its greatest aspect.

15.4 The API

Let’s look deeper at the API, so you can see how to manipulate the awesomeness in your own apps.

Setting and Getting Values

We’ve already seen this, so there’s not a ton to say. We initialize an Agent at construction time, so there’s never a moment where the value can’t be obtained.

```
val secretAgent = Agent(007)
```

If we want to get the value out of this Agent, we have two options; we can either use the `get()` method or the `apply()` method:

```
val sevenByGet = secretAgent.get()
val sevenByApply = secretAgent()
sevenByGet must be (sevenByApply)
```

It’s up to you which you want to use. Personally, I prefer `get()` just because it makes it obvious that the object you’re calling it on isn’t just your ordinary object. It’s not like getting an integer out of an integer box—you’re crossing boundaries of space and time into Scala’s STM module to grab this particular integer.

There’s another way to get a value out of an Agent, which we’ve already seen, by awaiting the result. Normally, of course, we try to avoid awaiting on anything in a concurrent application, but there are times when you need to do it. This helps us solve the non-determinacy we saw before.

```
// Timeout is needed for await below
implicit val agentTimeout = Timeout(1.second)
```

```
val secretAgent = Agent(007)
secretAgent send { _ + 1 }
secretAgent.await must be (11)
```

When we call `await`, we're queuing a message in the Agent's Mailbox. It will ensure that we get the value that exists some time after the last `{ _ + 1 }` call. We're not actually *guaranteed* the value will be 11, since someone else could conceivably make one or more calls to `send` in between one of the ones we made (assuming they had access to it, of course), but we do know that the value will be as it existed sometime *after* our last call to `send`.

Instead of blocking with `await`, we can use the corollary method, `future`. This will give us a Future to the same result that `await` would deliver, except that it's now non-blocking and functional. We know all about Futures now, so it's easy enough to guess what you can do with them.

```
// Timeout is needed for await below
implicit val agentTimeout = Timeout(1.second)
val secretAgent = Agent(007)
secretAgent send { _ + 1 }
secretAgent send { _ + 1 }
Await.result(secretAgent.future, 1.second) must be (9)
```

Modifying Agent Values

It's the modification of Agents that really makes them shine. The inversion of control, coupled with the asynchronous execution and the now-standard messaging semantics of the Actor model, give us a solid and flexible scheme for modification.

`send`

We've already seen `send`, but what isn't absolutely clear is that the function that `send` puts into the Agent's Mailbox executes on the thread pool on which the Agent's Actor is running. In other words, it looks a bit like this:

```
class NotARealAgent[T](init: T) extends Actor {  
    var t: T = init  
    def receive = {  
        case ApplyThisFunction(f) =>  
            t = f(t)  
    }  
}
```

Clearly, that's not a real Agent, but from an execution semantic point of view, it's basically the same as what send does.

sendOff

Agents can also execute your code off in another thread. This prevents it from executing on the same reactive thread pool on which the Agent normally executes (generally the default Dispatcher's thread pool for the entire Actor System). The `sendOff` method allows this to happen and you use it when you're sending a function to the Agent that will be "long running."

Normally in an Actor, we break logic up that might be long running, or give it to a Future to complete, or give it to another Actor, or.... In an Agent's case, we don't have these possibilities. The Agent will run our code synchronously, and modify the variable based on the result. It can't run different operations in parallel because that would ruin the modifications' sequential nature. And, since the Agent has no idea what's going on because you're the one specifying the logic, you also have to specify the length of that operation using either `send` or `sendOff`.

```
// Timeout is needed for await below  
implicit val agentTimeout = Timeout(1.second)  
val secretAgent = Agent(007)  
secretAgent sendOff { i => Thread.sleep(200); 5 }  
secretAgent send { 10 }  
secretAgent.await must be (10)
```

In the above code, the call to `sendOff` will delegate that work to another thread, but will suspend the normal Agent operations so that the `send` won't get processed until the `sendOff` completes. This keeps things ordered, but ensures that the standard thread pool won't get swamped with long-running operations and, thus, starve everyone else.

alter and alterOff

send and sendOff have return values of Unit. In contrast, alter and alterOff have return values of Future[T], which allow you to operate on the result of your function's application to the Agent's internal value. This might not be much use when you simply provide a *value* to Agent, but it's useful when you've given it a real computation to perform.

Note

The url <http://www.assembla.com/spaces/akka/tickets/2344> states that the Agent needs some love. This means, for one, that the curried timeout parameter needs to be implicit. You need to fix this section once that happens.

```
// Timeout is needed for await below
implicit val alterTimeout = Timeout(1.second)
val secretAgent = Agent(007)
val f1 = secretAgent.alter({ i => i + 1 })(alterTimeout)
val f2 = secretAgent.alter({ i => i + 1 })(alterTimeout)
Await.result(f2, 1.second) must be (9)
Await.result(f1, 1.second) must be (8)
```

alterOff works analogously to sendOff with no real surprises. If you have a long computation for an Agent and you want to get a Future to the result, use alterOff.

update

The last method you have for modifying an Agent's value is to use update. This is fundamentally the same as using send with a value instead of a function; in fact, the current implementation of update in the Akka source does exactly that.

```
implicit val agentTimeout = Timeout(1.second)
val secretAgent = Agent(007)
secretAgent update 12
secretAgent.await must be (12)
```

Note that, of course, update is meant to take only values of type T, not single parameter functions of type T => T.

Functional Agents

Agents are also functional, in that they implement `map`, `flatMap`, and `foreach`. However, the semantics of these functional operations differ from what we've seen thus far. Agents are, by definition, side-effect beings; we continually change what they represent. While the values they represent are immutable (or at least, should be), their representations constantly change. Functional programming works when side-effects aren't present and so we can't have the same semantics with the functional side of Agents as we do with the Actor-based system we've seen thus far. This doesn't make them more or less useful in the functional paradigm; it simply is a different paradigm with which we can work in order to get the job done.

For example, we can use `map` directly, transforming one Agent into another Agent, leaving the first untouched and compare both directly.

```
// Type annotations added for affect
val secretAgent: Agent[Int] = Agent(007)
val secretAgent2: Agent[Int] = secretAgent map { _ + 1 }
secretAgent2.await must be (secretAgent.await + 1)
```

Or we can use for-comprehensions to abstract the `map` and `flatMap` calls away while we operate on the Agents' contents:

```
val result = for {
    first <- Agent(7)
    second <- Agent(8)
    intermediate = first + 1
} yield first + second + intermediate
result.await must be (23)
```

It's pretty standard stuff, but you have to be realistic about things with respect to concurrency. For example, `map` will work just fine, but you're not guaranteed what value you'll map over (I mean, how could you?). The value that you'll map over is whatever value is obtained from calling `get()` at that nanosecond. If you have something more deterministic (e.g., the value after you've done a `send`), then the Agent's functional aspects won't work out in that situation.

A Note about Suspend and Resume

Agent has two methods on it called suspend and resume. “Clever”¹ programmers might want to use these functions to do “interesting” things. Don’t.

Akka uses suspend and resume internally to assist in the operation of sendOff and alterOff. While either of these is executing, the Agent’s standard operation must be suspended in order to ensure that modifications remain sequential. You’re not intended to use them.

15.5 Transactional Agents

As far as I can tell right now... these don’t work.

15.6 Chapter Summary

Agents are another tool in the Akka toolshed that ease the creation, manipulation, and management of shared data. They’re really great for implementing statistics counters, since any entity can update them at any time, safely and deterministically. Access to the data they represent is fast when you don’t care whether modifications are pending, and can be done more deterministically using a message-based approach of retrieval.

You can use Agents to represent other types of data as well, but counters are by far the most obvious and easy to reason about. In complex systems, data without state can become quite complex and difficult to reason about. For example, were you to implement the idea of a bank account with an Agent, it would be very difficult to represent when the account becomes overdrawn. In these situations, you might want to model that piece of data with a Finite State Machine instead.

When it comes to Agents, it might be best to keep things simple. It’s a lot easier to reason about small pieces of data that require no “context” in order to be easily understood, and are relevant at any moment in time. Concurrency, in general, just makes this problematic, and the other Akka facilities that we’ve seen thus far, which allow us to mix behavioural state with values as they progress through time, can make these more complex problems much simpler to solve.

¹Yup, I’m being sarcastic.

Chapter 16

Granular Concurrency with Dataflow

Up to this point, we've been working with a concurrency that has been *coarse*; that is, a function must execute entirely before control returns to the thread on which it is currently executing. By taking advantage of Scala's Continuations feature, along with the power of Futures, Akka has developed a feature implementation known as *Dataflow*, along the same lines as the feature of the same name implemented¹ in the Oz programming language.²

Dataflow concurrency lets you write a function so that it *appears* as a sequential method, but is actually broken up into discrete functions that enable the “sequential” function to execute concurrently with other parts of its own implementation. For example, here's a mind-bending piece of code that you can implement with Dataflow (written in pseudo code):

```
function() {  
    // Assign value1 with value2, which doesn't yet exist  
    value1 = value2 + 10  
    // Give value2 a value  
    value2 = 7  
    return value1  
}  
  
result = function()
```

¹<http://www.mozart-oz.org/documentation/tutorial/node8.html#chapter-concurrency>

²You can learn more about Oz and the Mozart Programming System at <http://www.mozart-oz.org/>

In that chunk of code, `value2` gets used before it gets assigned, which doesn't make any sense in a sequential function definition. However, what if we rewrote it to look more like this:

```
// Declare value2 as a Promise
value2 = Promise[Int]()

anonymousFunction() {
    // Assign value1 with value2, which doesn't yet exist
    // We're going to pretend that 'get()' is a blocking call
    value1 = value2.future.get() + 10
    return value1
}

anonymousProcedure() {
    // Fulfill the Promise of value2
    value2 complete 7
}

// Run the anonymousProcedure asynchronously
Future { anonymousProcedure() }

// Run the function and get the result
result = anonymousFunction()
```

Now it looks much more plausible, right? Given that the `anonymousProcedure()` runs concurrently with the `anonymousFunction()` and a Future helps bridge their separate dimensions, this code now makes sense. Of course, Akka won't do something as silly as create a blocking call like the `value2.future.get()` we have above, but the goal of the illustration was to introduce Futures and asynchrony to you.

16.1 Caveats

Before we go hog wild and re-envision everything we've seen so far as solvable via Dataflow, we have to understand that Dataflow is meant to work with *pure functions*. A pure function has no side effects, which means:

1. It is defined solely as a function that operates on its input parameters and nothing else.

2. It will always return the exact same result given the exact same input parameters, regardless of which planet or in which spatial dimension it operates.
3. You can't use things like `scala.util.Random`.
4. You can't read from or write to disk.
5. You can't make calls out to the network or send messages to Actors that are equally non-pure (which, in general, they're not).

This does limit the number of things you can solve with Dataflow, but that limitation is quite small. For example, if you need data from disk, simply obtain it before you enter Dataflow. Once you have the data, you can pass it to pure functions inside Dataflow and get the asynchronous operations that you're looking for.

16.2 With That Said...

The aspect of *purity* is core to an aspect of Dataflow in which you have the power to accept or reject as you see fit. Concurrency is complex and has the potential to cause pernicious bugs, but Dataflow, with the acceptance of and adherence to the caveats, has the power to put a lot more determinism into your concurrent code. If a deadlock is going to happen, for example, it will happen *all the time*; it is no longer a Heisenbug.

However, if you find the caveats too limiting, then you can choose to reject them and code with whatever side effects you want. In doing so, you trade off that determinism and re-introduce the potential to create data races or other types of concurrency problems. Software development is always about making intelligent trade-offs. If the trade-off is an intelligent one, then go for it.

16.3 Getting Dataflow into the Build

At the present time, getting Dataflow into the build isn't as easy as we've become accustomed to thus far. We need to use the more sophisticated type of SBT configuration, which is implemented via `.scala` files instead of `.sbt` files. The following example will pull in Dataflow, along with defining the right bits that get everything working together.

```
import sbt._  
import Keys._  
  
object MyBuild extends Build {  
    lazy val dataflow = Project(  
        "dataflow",  
        file("."),  
        settings = Defaults.defaultSettings ++ Seq(  
            autoCompilerPlugins := true,  
            version := "0.1",  
            scalaVersion := "2.10",  
            libraryDependencies +=  
                "com.typesafe.akka" % "akka-dataflow" % "2.1" cross CrossVersion.full,  
            libraryDependencies <+= scalaVersion { v =>  
                compilerPlugin("org.scala-lang.plugins" % "continuations" % "2.10")  
            },  
            scalacOptions += "-P:continuations:enable"  
        )  
    )  
}
```

16.4 Dataflow Values

The difference between a *value* and a *variable* is that values can't vary, but variables can – cute, huh? As we learned back in Section 2.3, immutability is a lot easier to reason about, with respect to concurrency, than mutability is. Dataflow depends on and revolves around the idea of values in order to achieve its goals.

A Dataflow value can be written to once but read from an infinite number of times. The key to a Dataflow value is that we implement it as a Future and can therefore compose it with anything that wants to use it. It also means that there's nothing new here for us; a Dataflow value is just an Akka Promise. We're about to see how they work inside flow.

16.5 Flow

If Dataflow were just Futures, then Futures would also be Dataflow, but the difference between them is flow. Much like a for-comprehension, flow lets us compose Futures together in a way that's convenient and easy to read.

```
import akka.dataflow._
import scala.concurrent.Await
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.util.Duration
import scala.concurrent.util.duration._
import scala.math.BigDecimal

// Calculate pi to 'n' places
def calculatePiTo(places: Int): Future[BigDecimal] = ???

// calculate the first 'n' Fibonacci numbers
def fibonaccis(n: Int): Future[Seq[BigDecimal]] = ???

// The flow block will return a Future
val perfect = flow {
    // 'pie' is now a Dataflow value, which is pi to
    // 3,000,000 decimal places returned in a Future
    val pie = calculatePiTo(3000000)

    // So is 'fibs', which is the first 31,402nd
    // Fibonacci numbers returned in a Future
    val fibs = fibonaccis(31402)

    // The 'perfect' area
    val lastFibs = fibs().last
    pie() * lastFibs * lastFibs
}
println(Await.result(perfect, 1.second))
```

Both pie and fibs will calculate asynchronously and then be used concurrently inside the flow block. In addition, the value of lastFibs and perfect are not blocking operations; they will be dependent on the results of the Future Dataflow values, pie and fibs. In other words, the flow block will return immediately and it will return a Future.

Akka handles all of the work involved in splitting up your code into composable bits and reorganizing it so that it can execute in a non-blocking manner.

Accessing Values in Flow

You probably noticed `pie()` and `fibs()`, and you also might have noticed that they look a bit weird. We know that `pie` and `fibs` are `Futures`, but we've never seen them used in this way before. That is because a `Future` doesn't have an `apply()` method on it; i.e., you wouldn't be able to call `pie()` if it were just a plain ol' `Future`.

Akka enhances the `Future` object into something that defines the `apply()` method, which holds the hooks that the `Continuations` feature needs in order to recompose your code into the appropriate structure. It also provides you with the type you're interested in, which makes your code more natural than it would be if you were just using `Futures` directly.

```
val fibs = fibonaccis(31402)  
// Won't work - fibs is a Future and doesn't have a "last" method  
val lastFibs = fibs.last // <- ERROR  
  
// Works just fine. The apply method resolves the Future into the  
// Seq[BigDecimal] in which we're interested  
val lastFibs = fibs().last
```

Sure the first representation looks more natural (i.e., it doesn't have the `()`), but it doesn't work; Dataflow needs a hook that will help it recompose your code, and that hook is the `apply` method.

Creating Values in Flow

There are times when you might want to create a Dataflow value directly, rather than working with the result of a `Future`, which you obtained from some other source. When we want to create a value in a `flow` block, we need to create the `Future`'s *Promise*, with a `Promise` instance. Have a look back to [Section 12.3](#) to refresh your memory on the relationship between `Promises` and `Futures`, if you're a little fuzzy on that.

To properly create a Dataflow value, we just need to create a `Promise` and then assign to it once we have a concrete value we can stick in it.

```
def calculatePiTo(places: Int): Future[BigDecimal] = ???  
  
val perfect = flow {  
    // Create our Promises  
    val pie = Promise[BigDecimal]()  
    val fibs = Promise[Seq[BigDecimal]]()  
  
    // Assign to the first with the results from the Future  
    pie << calculatePiTo(3000000)  
  
    // Assign this one directly with the 31402nd Fibonacci number  
    fibs << BigDecimal("11787086955526126 ... 133457816720073973026")  
  
    // Use them just like we always have  
    val lastFibs = fibs().last  
    pie() * lastFibs * lastFibs  
}  

```

Both Promise and Future have had the apply method added on to them so that we can use either one in the same way in our code – just another way the Akka team has made our lives easier.

Note that we can assign a future value to a Promise or we can assign an immediate value to that Promise, depending on whatever happens to be most convenient at the time. This is yet another makes-life-easier addition to the API.

In either case, all we have to do is *shift*³ the value into the Promise in order to complete it.

Assigning More Than Once

You can't assign more than once. So don't. The compiler will happily let you do the following:

```
pie << calculatePiTo(3000000)  
if (someInterestingConditionHappened)  
    pie << BigDecimal(3.1415926535)
```

³This is not standard Akka terminology; I'm appropriating the word for Dataflow because I think it really fits nicely. Not only is << historically known as a “shift operator,” but it also makes a nice picture in my head. In asynchronous code, the idea of immediate assignment is often inaccurate, and the notion of *shifting* an eventual value into an object after we've already moved on seems to speak to the issue much better than “assignment” does.

However, this will fail at runtime. At some point, that promise will already be completed, and Akka will throw an error that states precisely that. It'll look something like this:

```
java.lang.IllegalStateException: problem in scala.concurrent internal callback
... stack trace stuff ...
Caused by: java.lang.IllegalStateException: Promise already completed.
... more stack trace stuff ...
```

Dataflow guarantees that you'll get this error anytime you hit the `if` block. This is fantastic, since we're looking to reduce the effects of Heisenbugs, but it is a runtime effect. Just be aware that the compiler will happily let you shift into Dataflow value more than once, but you're not going to get away with it at runtime. If you think about and treat Dataflow values as standard Scala vals, then you'll have no problem understanding how to use them.

Multiple Flows

Just to be clear, there's nothing to stop you from declaring multiple flow blocks, inside one another or external to one another, all of which might be working with multiple Promises or Futures defined throughout your application. We can play with our previous code to illustrate:

```
// Calculate pi to 'n' places
def calculatePiTo(places: Int): Future[BigDecimal] = ???

// calculate the first 'n' Fibonacci numbers
def fibonaccis(n: Int): Future[Seq[BigDecimal]] = ???

// Get some Future values
val pie = calculatePiTo(500)
val fib = flow { fibonaccis(31402)().last }

// Get the perfect area
val perfect = flow {
    pie() * fib() * fib()
}

// Get an imperfect area
val imperfect = flow {
```

```
    BigDecimal(3.14) * fib() * fib()
}

// Calculate the ratio of the two
val ratio = flow { perfect() / imperfect() }

// Print out that ratio
println(Await.result(ratio, 1.second))
```

See? We have a lot of flows here, mixed in with some basic Futures and it all hangs together quite nicely. You can see how `perfect` and `imperfect` both work in parallel, depending on the eventual value of `fib`. The fact that `fib` is defined outside of flows allows different flows to access it, as well as ensures that it will execute in parallel. This is no different than how we avoid “Subtle Sequentialism,” which we covered in [Section 12.3](#).

16.6 Another Way to Get Instrument Status

Back in [Section 12.6](#), we used Futures to collect the status of our instruments. There was nothing wrong with what we did and there’s no reason to change it. But rather than aggregate status, what if we wanted to simply display it on the screen? Using Dataflow, we can accomplish that goal in a manner that might feel more natural to you.

```
flow {
    val altStatus = actorFor("Altimeter") ? ReportStatus
    val headStatus = actorFor("HeadingIndicator") ? ReportStatus
    val airStatus = actorFor("AirSpeed") ? ReportStatus
    val fuelStatus = actorFor("FuelSupply") ? ReportStatus
    val status = s"""|Altimeter      : ${altStatus()}
                    |HeadingIndicator : ${headStatus()}
                    |AirSpeed        : ${airStatus()}
                    |FuelSupply      : ${fuelStatus()}"""
    status.stripMargin
} onComplete println
```

Again, there’s nothing stopping you from doing this with standard Future combinators or using a for-comprehension; it’s totally up to you.

One thing we must note about the above code, however, is that it’s *not pure*. Because we interact with Actors, there are aspects of real life that can

come to bite us on the butt; we can get timeouts or network failures, for example. The lesson here is that just because the flow works the first time doesn't mean it will work every time. We sacrificed the caveat discussed earlier regarding pure functions for Dataflow's ease of use. If we had done this with a for-comprehension, then it wouldn't have been any different; real life could still bite us in the butts.

16.7 When to Use Dataflow

You can apply Dataflow in a bunch of situations and its lack of obtrusiveness makes its application quite nice in many cases. Whenever you might use a for-comprehension, you may choose to use Dataflow instead. It's really up to you and how you want your code to look.

Remember that Dataflow is really just Futures under the hood, so anything you can do with Dataflow you can do with Futures, plain and simple. The Scala compiler and the Akka toolkit are helping you keep the plumbing mechanisms of Futures out of your way, which is fantastic, but this doesn't lend any capabilities to your code that weren't already available to you.

You also have to remember that not all of the Future's facilities are part of Dataflow. There's nothing magical in Dataflow that will change the way that you might use `Future.sequence` or `Future.either`, for example. Dataflow helps you compose your Futures by hiding the syntax associated with `flatMap` (see [Section 12.3](#)) and other combinators, such as `completeWith`, and side-effecting code, such as `onFailure`.

16.8 Chapter Summary

That's Dataflow! There's really not a lot to it, and in fact the implementation inside Akka to realize it is quite small. Dataflow provides us with a syntactic variation on the work we've done with Futures up to this point, but provides us with a less *coarse* implementation that looks much more like imperative code than we've seen to date. It can be a very powerful mechanism that makes your code easier to reason about and clearer to others, as well as your future self.

Chapter 17

Patterns for Akka Programming

When a toolkit presents you with as many possibilities as Akka, it can be very helpful to have a set of “go to” patterns that you can employ to solve your design issues. In this chapter, we’ll summarize some of the things we’ve learned and distill them down into a set of patterns you can use.

This won’t be a rigorous definition as you might find in *Design Patterns: Elements of Reusable Object Oriented Software* by Erich Gamma, et al. (Addison Wesley, 1994). We’ll be much less formal here; you should take these examples, improve upon them, and tailor them to whatever your particular needs might be. This section aims to help you simplify the way you approach your Akka application development, so that you can focus on your application logic.

17.1 Behavioural Composition

We know that you can break up your Actor’s behaviour into multiple methods that return `receive` partial functions. We also know that you can combine these together using Scala’s `orElse` function combinator, which can make things interesting.

As the number of behavioural functions increase, the number of permutations increases as well. Keeping everything straight can be a real problem, and constructing the new `receive` method at each point in the code can also be quite problematic. To handle this problem, we create a new derivation of the Actor that allows us to compose the `receive` partial function without having to know about every component that goes into its construction. It eliminates the ability to implement `receive` and replaces it with a new

concept; we add partial functions to a map on construction instead, and let the composer create the `receive` method for us.

```
trait ReceiveCompositingActor extends Actor {
    import scala.collection.mutable.Map

    // This map will hold the bits and pieces of our ultimate 'receive'
    lazy val receivePartials = Map.empty[Int, Receive]

    // A couple of constants you can use if you'd like. They indicate that
    // the start is a low number and the end is a high number
    val StartOfReceiveChain = 0
    val EndOfReceiveChain = 10000

    // Convenience wrapper around the 'become' operation
    def becomeNew(key: Int, behaviour: Receive) {
        receivePartials += (key -> behaviour)
        context.become(composeReceive)
    }

    // Composes the ultimate 'receive' partial function by combining the
    // partials in sorted order using 'orElse'
    def composeReceive: Receive = {
        receivePartials.toSeq.sortBy {
            case (key, _) => key
        }.map {
            case (_, value) => value
        }.reduceLeft { (a, b) => a orElse b }
    }

    // Immediately becomes the composed entity
    override def preStart() {
        super.preStart()
        context.become(composeReceive)
    }

    // Pointless now. We're going to become something new immediately
    final def receive: Receive = { case _ => }
}
```

To illustrate its use, let's create a few traits that self-type to the `ReceiveCompositingActor` and thus mix in various bits of behaviour.

```
trait HelloHandler { this: ReceiveCompositingActor =>
    val HelloReceiver = 10
    def helloHandler: Receive = {
        case "Hello" => sender ! "Hithere"
    }
    // Hello belongs in slot '10' in the chain of receivers
    receivePartials += (HelloReceiver -> helloHandler)
}

trait MiddleHandler { this: ReceiveCompositingActor =>
    val SmallTalkReceiver1 = 15
    val SmallTalkReceiver2 = 16
    def smallTalkHandler1: Receive = {
        case "So How's the Weather?" => sender ! "Rainy, and lousy..."
    }
    def smallTalkHandler2: Receive = {
        case "How about the Kids? Good?" => sender ! "Sure"
    }
    // The small talk belongs in the "middle" of the chain
    receivePartials += (SmallTalkReceiver1 -> smallTalkHandler1)
    receivePartials += (SmallTalkReceiver2 -> smallTalkHandler2)
}

trait GoodbyeHandler { this: ReceiveCompositingActor =>
    val GoodbyeReceiver = 20
    def goodbyeHandler: Receive = {
        case "Goodbye" => sender ! "So Long"
    }
    // Goodbye belongs in slot '20' in the chain of receivers
    receivePartials += (GoodbyeReceiver -> goodbyeHandler)
}

class ComposedActor extends ReceiveCompositingActor
    with HelloHandler
    with GoodbyeHandler
    with MiddleHandler {
    val MoodReceiver = 50
    def alternateSmallTalk1: Receive = {
        case "So How's the Weather?" => sender ! "Sunny! Amazing!"
    }
}
```

```
}

def alternateSmallTalk2: Receive = {
    case "How about the Kids? Good?" => sender ! "Funny! Smart! Awesome!"
}

def moodHandler: Receive = {
    // Change the small talk from grumpy to happy
    case "Happy" =>
        becomeNew(SmallTalkReceiver1, alternateSmallTalk1)
        becomeNew(SmallTalkReceiver2, alternateSmallTalk2)

    // Change it back to grumpy
    case "Grumpy" =>
        becomeNew(SmallTalkReceiver1, smallTalkHandler1)
        becomeNew(SmallTalkReceiver2, smallTalkHandler2)
}

// We'll put the messages that change the mood at the end of the chain
receivePartials += (MoodReceiver -> moodHandler)
}
```

You can use more sophisticated mechanisms for choosing your keys, if you like, but this is a pretty simple approach. You could imagine using types that solidify the knowledge at compile time instead of runtime. You could also be more sophisticated than that and create a class that orders itself irrespective of its “key.” This would separate the ordering from the key and thus be a little safer. If you need to, go for it.

The bottom line is that now we can change the behaviour of a *piece* of the functionality without having to specify unrelated pieces of that functionality. When we make the call to `becomeNew(key, value)`, we don’t need to worry about all of the other unrelated pieces, and that’s the goal of the entire `ReceiveCompositingActor`.

17.2 Isolated and Parallel Testing

The ScalaTest framework is quite flexible and gives you a fair amount of leeway into how you structure your tests. When you implement the simple test with Akka, there are some drawbacks depending on what you’re doing. Let’s recap the simple test:

```
import akka.actor.ActorSystem
```

```
import akka.testkit.{TestKit, ImplicitSender}
import org.scalatest.{WordSpec, BeforeAndAfterAll}
import org.scalatest.matchers.MustMatchers

class SimpleSpec extends TestKit(ActorSystem("SimpleSpec"))
    with ImplicitSender
    with WordSpec
    with BeforeAndAfterAll
    with MustMatchers {

    override def afterAll() = system.shutdown()

    "Simple" should {
        "do something" in {
            // And it works
            "Akka" must be ("Akka")
        }
    }
}
```

Nine times out of ten this works just fine. We have our ActorSystem, our TestKit, our ImplicitSender, and everything works great. However, it fails to work well under certain situations:

- We have an Actor name that conflicts between tests (i.e., you get an InvalidActorNameException due to the fact that another one already exists).
- Things become a bit slow and you'd like to run things in parallel.
- You'd like to have each test hold its own `testActor`, but you'd rather not use a bunch of different test probes.
- You're modeling some other interesting behaviour that gets messed up between tests because it's inside the same ActorSystem.

In these cases, you're better off isolating the ActorSystems from each other. This involves creating Fixtures that contain the ActorSystem, the TestKit, and the ImplicitSender rather than the Suite. We start by defining the “Base” test spec from which we can derive a couple of specializations:

```
// Helps us generate unique names for ActorSystems
object TestSystemCounter {
    val sysId = new AtomicInteger()
}

trait BaseSpec extends fixture.WordSpec with MustMatchers {
    import TestSystemCounter._

    type Fixture <: AkkaFixture
    val specType: String

    // ScalaTest needs to know what our Fixture parameter type is
    type FixtureParam = Fixture

    // Our basic Fixture. You would derive from this if you want to
    // specialize it with more fields, methods, etc...
    class AkkaFixture extends TestKit(
        ActorSystem(s"$specType-${sysId.incrementAndGet()}"))
        with ImplicitSender

    // Abstract. Derivations must implement this
    def createAkkaFixture(): Fixture

    // This is how our tests get run
    override def withFixture(test: OneArgTest) {
        val sys = createAkkaFixture()
        try {
            test(sys)
        } finally {
            sys.system.shutdown()
        }
    }
}
```

Given that, we can now derive a specialization that runs tests in isolation but still sequentially, and another that runs them in isolation but in parallel.

```
// Runs each test sequentially but provides fixture isolation
trait SequentialAkkaSpecWithIsolatedFixture extends BaseSpec {
    val specType = "Seq"
}

// Runs each individual test in parallel
```

```
trait ParallelAkkaSpec extends BaseSpec with ParallelTestExecution {  
    val specType = "Par"  
}
```

These new derivations can be used quite simply. We'll use the ParallelAkkaSpec as an example, since it's the most fun.

```
class TestInParallelSpec extends ParallelAkkaSpec {  
    type Fixture = AkkaFixture  
    def createAkkaFixture(): Fixture = new AkkaFixture  
  
    "TestInParallel" should {  
        "work 1" in { f => import f._  
            val a = system.actorOf(Props[Echo], "Echo")  
            a ! "Ping"  
            expectMsg("Ping")  
        }  
        "work 2" in { f => import f._  
            val a = system.actorOf(Props[Echo], "Echo")  
            a ! "Ping"  
            expectMsg("Ping")  
        }  
    }  
}
```

Note how the tests take a Fixture parameter that is bound to f. By importing f._, we get the convenience of using the TestKit facilities just as though we included TestKit at the Suite level.

If we didn't use the Fixture approach for these tests, the second one would fail with an Exception, due to the fact that the Actor named "Echo" already exists. So not only do these tests run in parallel, they also run correctly, which they wouldn't if the TestKit was included at the Suite level.

17.3 Strategies for Implementing Request/Response

As we've already learned, the problem with the request/response idiom is establishing a *context* in which to understand the eventual response. For example, let's say we have an Actor that makes the plane go up and down. To any control amount that comes in, we want to tack on some sort of random

“deflection” amount, which an external Actor calculates. Here’s the bad way to do it:

```
class BadIdea extends Actor {  
    val deflection = context.actorOf(Props[DeflectionCalculator])  
    def receive = {  
        case GoUp(amount) =>  
            deflection ! GetDeflectionAmount  
        case GoDown(amount) =>  
            deflection ! GetDeflectionAmount  
        case DeflectionAmount(amount) =>  
            // Hmm... was I going up or down?  
    }  
}
```

When we get the response from the deflection calculator, we don’t know whether we wanted to go up or down (i.e., we’ve lost our *context*). We can solve this in a few different ways.

The Future

Using a Future lets us close over the data we might need and also provides a fairly natural contextual reference for our operation. We can take the BadIdea above and convert it to a FutureIdea like this:

```
class FutureIdea(controlSurfaces: ActorRef) extends Actor {  
    val deflection = context.actorOf(Props[DeflectionCalculator])  
    def receive = {  
        case GoUp(amount) =>  
            // Ask for the deflection amount, transform it to a StickBack message  
            // and pipe that to the Control Surfaces  
            (deflection ? GetDeflectionAmount).mapTo[DeflectionAmount].map { amt =>  
                val DeflectionAmount(deflection) = amt  
                StickBack(amount + deflection)  
            } pipeTo controlSurfaces  
        case GoDown(amount) =>  
            // Ask for the deflection amount, transform it to a StickForward message  
            // and pipe that to the Control Surfaces  
            (deflection ? GetDeflectionAmount).mapTo[DeflectionAmount].map { amt =>
```

```
    val DeflectionAmount(deflection) = amt
    StickForward(amount + deflection)
  } pipeTo controlSurfaces
}
}
```

A really nice thing about the Future is that we can put a timeout on the request. If the responder fails to respond in a timely manner, the Future will throw an exception that we can deal with however we like. And it will throw that exception in a context-sensitive manner, so we can know that request "B" failed due to a timeout, but request "D" succeeded just fine.

The Actor

Sometimes a Future won't work out for you all that well. There may be many messages you want to handle, many states you want to transition, or just have complex logic that you want to employ. Even when they may be possible inside of a Future, an Actor may be a clearer representation of what you want to do. You can either cook up an Actor to handle the problem for you, or you can do it anonymously. We'll show the anonymous Actor method here:

```
class ActorIdea(controlSurfaces: ActorRef) extends Actor {
  val deflection = context.actorOf(Props[DeflectionCalculator])
  def receive = {
    case GoUp(amount) =>
      // Spin up an anonymous Actor to send the StickBack message to the
      // Control Surfaces based on the deflection amount
      context.actorOf(Props(new Actor {
        override def preStart() = deflection ! GetDeflectionAmount
        def receive = {
          case DeflectionAmount(deflection) =>
            controlSurfaces ! StickBack(amount + deflection)
            // Remember to stop yourself!
            context.stop(self)
        }
      }))
    case GoDown(amount) =>
```

```
// Spin up an anonymous Actor to send the StickForward message to the
// Control Surfaces based on the deflection amount
context.actorOf(Props(new Actor {
    override def preStart() = deflection ! GetDeflectionAmount
    def receive = {
        case DeflectionAmount(deflection) =>
            controlSurfaces ! StickForward(amount + deflection)
            // Remember to stop yourself!
            context.stop(self)
    }
}))
```

This is much like the Future example, but it's not so easy to put the timeout on the responses. You'd have to use the ReceiveTimeout message to indicate that something failed to be received. If you have more than one thing happening here, though, that can get tricky as the receive timeout resets when it gets *any* message at all.

Internal Actor Data

We can use mutable data inside the Actor to help the operational context survive between message processing.

```
class VarIdea(controlSurfaces: ActorRef) extends Actor {
    val deflection = context.actorOf(Props[DeflectionCalculator])
    var lastRequestWas = ""
    var lastAmount = 0f

    def receive = {
        case GoUp(amount) =>
            lastRequestWas = "GoUp"
            lastAmount = amount
            deflection ! GetDeflectionAmount
        case GoDown(amount) =>
            lastRequestWas = "GoDown"
            lastAmount = amount
            deflection ! GetDeflectionAmount
    }
}
```

```
    case DeflectionAmount(deflection) if lastRequestWas == "GoUp" =>
      controlSurfaces ! StickBack(deflection + lastAmount)
    case DeflectionAmount(deflection) if lastRequestWas == "GoDown" =>
      controlSurfaces ! StickForward(deflection + lastAmount)
  }
}
```

This is pretty hideous, in the general sense, but in certain specific cases it may serve you well. There are several pitfalls to it:

1. It relies on the fact that you get requests and responses in a particular order; i.e., GoUp, DeflectionAmount, GoDown, DeflectionAmount, GoDown, DeflectionAmount, etc. If you get GoDown, GoDown, GoUp, DeflectionAmount, then it won't work all that well.
2. It doesn't survive a restart very well, due to the fact that it resets the variable data to their initialized states.
3. It's not very resilient to change. If your Actor becomes more complex, this varying state of the data may become entirely unwieldy. The state of the request is not localized to the request, but is now global to the Actor. It really can only be doing one thing, as opposed to processing several different things between requests and their paired responses.
4. You have to deal with timeouts. If the responder doesn't respond by the time you'd like, you need to figure out how to deal with that.

Message Data

You can also use the message to store this data, which gives you the low overhead of machine usage (don't have to spin up a Future or Actor), at the price of muddying up the protocol a bit.

```
class MsgIdea(controlSurfaces: ActorRef) extends Actor {
  val deflection = context.actorOf(Props[DeflectionCalculator2])

  def receive = {
    case GoUp(amount) =>
      deflection ! GetDeflectionAmount("GoUp", amount)
    case GoDown(amount) =>
```

```
    deflection ! GetDeflectionAmount("GoDown", amount)
    case DeflectionAmount(op, amount, deflection) if op == "GoUp" =>
      controlSurfaces ! StickBack(deflection + amount)
    case DeflectionAmount(op, amount, deflection) if op == "GoDown" =>
      controlSurfaces ! StickForward(deflection + amount)
  }
}
```

The price paid here is that the `DeflectionCalculator2` must package up the context data in its responses. There are pitfalls here as well:

1. It opens the door to runtime errors should the data's repackaging get screwed up somehow.
2. It increases the coupling between requester and responder.
3. It decreases code flexibility as you wish to add or remove fields.
 - This can be mitigated by providing a single case class member that holds all of the fields, so that the responder remains somewhat ignorant of the issue.
4. It increases payloads that have to go across the network should you want to send these messages to remote nodes.
5. You have to deal with timeouts. If the responder doesn't respond by the time you'd like, you need to figure out how to deal with that.

Internal/Message Data Hybrid

You can hit a middle ground between the internal data and the message data approaches that solves some of the issues. You put a minimal context into the message, which we call a *tag*, and then use that tag as an index into some internal data to the Actor.

```
class TagIdea(controlSurfaces: ActorRef) extends Actor {
  import scala.collection.mutable.Map
  val deflection = context.actorOf(Props[DeflectionCalculator3])
  // The map of tags to context
  val tagMap = Map.empty[Int, Tuple2[String, Float]]
```

```
// Our 'tags' will be integers
var tagNum = 1

def receive = {
    case GoUp(amount) =>
        // Add the req / rsp context to the map
        tagMap += (tagNum -> ("GoUp", amount))
        deflection ! GetDeflectionAmount(tagNum)
        tagNum += 1
    case GoDown(amount) =>
        // Add the req / rsp context to the map
        tagMap += (tagNum -> ("GoDown", amount))
        deflection ! GetDeflectionAmount(tagNum)
        tagNum += 1
    case DeflectionAmount(tag, deflection) =>
        // Get the req / rsp context from the map
        val (op, amount) = tagMap(tag)
        val amt = amount + deflection
        // Remove the context from the map
        tagMap -= tag
        if (op == "GoUp") controlSurfaces ! StickBack(amt)
        else controlSurfaces ! StickForward(amt)
    }
}
```

We can now have multiple things going on at once, which is great, but of course there are still pitfalls here:

1. The message protocol is still a bit messy. What if they send back the wrong tag? Yeah, that could be serious death.
2. We still have response timeouts to worry about. You might have to have something that stores the request timestamp in the map so that you can sweep through it occasionally to see if anything has timed out.
3. You'd have to decide how you behave if you restart.
4. We still have that pesky timeout problem.

Roundup

OK? There are several different ways to handle the request/response issue and there are really two major things that seem to sway the decisions on how you might want to do it:

1. Ease of implementation. Future wins here... big time. You can easily close over what you'd like, naturally create the response context you need, and everything is hunky dory.
2. Speed. Spinning up a Future to do the work for you may be costly so you might need to avoid it. That said, *don't prematurely optimize*. If you're trying to do millions of request/responses per second, that may or may not be costly. If a profiler tells you that it is, and your users are experiencing a latency that you can blame on the Future request/response, then you might consider changing it for those rare cases where you need to. At that point, you need to pick one of the other methods that works in your situation.

17.4 Mechanisms for Handling Non-Deterministic Bootstrapping

There are times when an Actor's initialization takes time. You can argue that an Actor should be initialized on construction, and as such cannot be in a state of limbo. This is a fine goal to achieve but it's not always possible or, if it is, the solution can be worse than the original problem. For example, if you Actor restarts, it may need to read fresh information from the database; however, if it was constructed with the information from the database, then it won't receive fresh data. To do that, you'd have to add a layer of indirection to the hierarchy, and let the parent read the new information and then reconstruct the child, but really, what's the difference? It's just a more complex way of doing the same thing.

We saw a flavour of this when we created the FlyingBehaviour FSM; it had to acquire the controls as well as seed a heading indication and an altitude indication before it could move on to actually flying the plane.

Things get more interesting when the Actor is visible to the outside world and is expected to handle requests. If the Actor isn't initialized to the point where it actually *can* handle those requests, then what do you do? Unfor-

tunately, the answer to that question is rather dependent on your problem domain, but we'll cover some solutions here.

However, I must make a key point clear: the bootstrapping algorithm takes place at the same time when other entities are making requests. That's just the nature of concurrency. During this time, it is very important to ensure that the incoming requests have a deterministic outcome. If the bootstrapping fails, these messages shouldn't just go to the Dead Letter Office, unless the client is happy to have that happen (and generally it we would expect that clients wouldn't be happy having their requests be silently ignored). It's this deterministic client interaction that is the focus of this pattern.

Denial

Denial is the simplest mechanism to deal with this problem. Your server is unable to handle the request right now, so you give an error back to the client and tell it to try again later. For example, let's say that the Actor in question implements a web service, but during its initialization, it must suck up some data from an external data store. Until its initialization is complete, it will send clients a 500 HTTP error.

```
class WaitForInit extends Actor {  
    def uninitialized: Receive = {  
        case DBResults(results) =>  
            context.become(initialized(results))  
        case HTTPRequest(req) =>  
            req.INTERNAL_ERROR("System not ready. Try again soon.")  
    }  
  
    def initialized(data: Map[String, String]): Receive = {  
        case HTTPRequest(req) =>  
            // dispatch request and fulfill  
    }  
  
    def receive = uninitialized  
}
```

We simply start in the uninitialized state, which sends errors to all HTTP requests. Once the database information is in, it can become the initialized state and handle requests properly.

Stashing

Akka ships a trait that you can mix into your Actors and lets you stash messages away so that you can process them later. We can use this functionality to save messages while the Actor is being bootstrapped.

```
class StashingActor extends Actor with Stash {  
    def uninitialized: Receive = {  
        case DBResults(results) =>  
            // Unstash everything since the behaviour we're about to 'become'  
            // will be able to handle whatever was stashed away  
            unstashAll()  
            context.become(initialized(results))  
        case HTTPRequest(_) =>  
            // Can't handle it now. Stash it away.  
            stash()  
    }  
  
    def initialized(data: Map[String, String]): Receive = {  
        case HTTPRequest(req) =>  
            // dispatch request and fulfill  
            req.OK("Here you go")  
    }  
  
    def receive = uninitialized  
}
```

When we switch to the `initialized` behaviour, the messages that we stashed away will be processed.

Now, with that said, there are timeouts to consider. You probably can't just keep messages stashed forever—people will eventually want a result, and you can't just suck up an infinite amount of memory either. These are things you need to consider when you're stashing messages.

Caveats

You can't use the `Stash` trait without a particular type of Mailbox. This means you need to create a configuration for a dispatcher that specifies the `UnboundedDequeBasedMailbox` for its `mailbox-type`, like this:

```
zzz.akka.investigation.stash-dispatcher {
```

```
    mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
}
```

Then you need to construct the Actor using the appropriate method on the Props class:

```
val a = system.actorOf(Props[StashingActor].withDispatcher(
  "zzz.akka.investigation.stash-dispatcher"))
```

You need to use this specific type of Mailbox because the unstashed messages are prepended to the Mailbox, which is an expensive operation if you've only got a Queue to work with. A Deque provides very fast prepend functionality.

Override of preRestart()

The Stash trait overrides preRestart(), which makes it sensitive to where you mix it in; it must be mixed in before anything else that overrides preRestart(). For example:

```
class SomeMix { this: Actor =>
  override def preRestart() {
    ...
  }

  // You can do this:
  class WithStashed extends Actor with Stash with SomeMix

  // But you CANNOT do this:
  class WithStashed extends Actor with SomeMix with Stash
```

Regarding Restarts

The stashed messages do not survive an Actor restart. Generally speaking, this should be perfectly fine for most cases where you might want to stash, but you need to be aware that a restart of your Actor is going to empty your stash. You may need to put some try/catch blocks in your code to ensure that the Actor doesn't restart when you don't want it to.

17.5 The Circuit Breaker

You might remember the days of the Twitter “Fail Whale.” Every once in a while you’d see this picture pop up when you went to Twitter of an enormous smiling whale, hopelessly being lifted by a bunch of tiny birds, indicating that they were under high load and simply couldn’t service your request.¹

The Fail Whale showed up in order to protect the server from load spikes as a last-resort measure. Twitter simply can’t do what you want because the load on the system is too high, and if they were to try to help you out, it would only make things worse. So they tell you to get lost for a while until things calm down. This type of behaviour is nicely implemented using what is known as a Circuit Breaker.

We won’t go into great detail about this here, since Akka already ships this pattern and the Akka reference documentation does a perfectly fine job of explaining it to you.

Put simply, the Circuit Breaker works by watching timeouts and failures. You wrap calls to certain operations inside an instantiated Circuit Breaker and drop in a handler that is called when the Circuit Opens (an open circuit is bad, a closed circuit is good). This gives you a chance to change your Actor’s behaviour to one that fails immediately and sends the user a Fail Whale right away, which eliminates the load on your machine (much like we did when denying access during initialization in the previous section).

17.6 Breaking Up a Long-Running Algorithm into Multiple Steps

You know that whole thing about “state” being held inside the messages instead of inside the Actor? Well, this pattern is classic for really underscoring that idea. You employ it when you have to do a stack of work, and you’re not finished until you’ve done it all.

For example, let’s say you want to get a bunch of information about a particular user’s installed applications. You first need to pull up his application list from some external data store, and then you need to iterate through each one, grabbing various bits of data. Essentially, this algorithm iterates $N + 1$ times—once for the initial application list, and then once for each found application. But you need to do it *asynchronously*, and that’s the big

¹This was back in the day *before* Twitter converted their backend to Scala, of course.

fly in the ointment. Most of the time, you'd quite happily tie up a thread doing it all synchronously, but in a large application, this will be *death*, so you need to do it asynchronously instead.

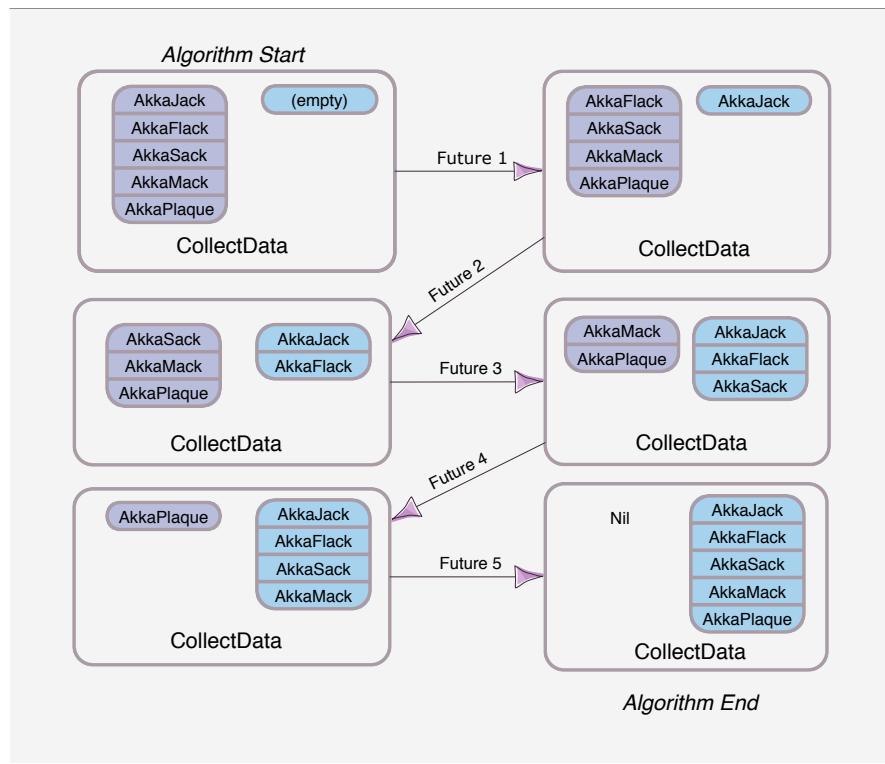


Figure 17.1 · The multi-stage asynchronous algorithm is essentially the propagation of evolving copies of a single message type. The message has two member lists; the left list contains work to be done, which decreases over time, while the results list on the right increases over time.

The idea here is to break the problem up into multiple stages and to process those stages using messages, where the messages contain all of the data, including the data that indicates the work to do, as well as the data containing the completed work. We shrink the list of work to do as the list of results grows, as Figure 17.1 depicts. Now let's look at the code:

```
object MultiStageAlgorithm {  
    // Successful result
```

```
case class UserAppData(username: String,
                      appData: List[Map[String, String]])  
// Failed result  
case class CollectionFailed(error: String)  
}  
  
  
class MultiStageAlgorithm(username: String,  
                         dataStore: ActorRef,  
                         returnTo: ActorRef) extends Actor {  
    import MultiStageAlgorithm._  
  
    implicit val askTimeout = Timeout(5.seconds)  
    // This is the internal message we use to continue  
    // collecting the app information from the data store.  
    // The appList contains the list of apps to collect  
    // information for, and the appDetails contains the list  
    // of collected information  
    case class CollectData(appList: List[String],  
                          appDetails: List[Map[String, String]])  
  
    override def preStart() {  
        // Start by asking for the list of applications  
        dataStore ! GetAppList(username)  
    }  
  
    def receive = {  
        // The data store has returned the list of applications  
        case AppList(appList) =>  
            // Start our work with a full list of work to be done  
            // and no results  
            self ! CollectData(appList, List.empty)  
        // When the work to be done is Nil, we're finished.  
        // Send the results and die.  
        case CollectData(Nil, appData) =>  
            returnTo ! UserAppData(username, appData)  
            context.stop(self)  
        // When there's more to be done, we pull the head off of  
        // the app list and retrieve its information. Once the  
        // results are in, we can remove the head from the list
```

```
// of work to be done, prepend the results to the
// results list and pipe the transformed message back to
// 'self'.
case CollectData(leftToCheck, appData) =>
  dataStore ? GetAppData(leftToCheck.head) map { msg =>
    msg match {
      case AppData(propertyMap) =>
        CollectData(leftToCheck.tail, propertyMap :: appData)
    }
  } recover {
    case e => CollectionFailed(e.toString)
  } pipeTo self
// If things failed then we need to state as such and die
case m: CollectionFailed =>
  returnTo ! m
  context.stop(self)
}
}
```

There are many different ways to implement this pattern. You could, for example, use a Finite State Machine and have the data travel through the states instead of using messages. Alternatively you could use `become()` to carry the state as well or you could also do this with a series of Futures.

One other thing to note about this pattern is that it's asynchronous-sequential, not asynchronous-parallel. You apply this pattern when you need to do things sequentially, but you spend the bulk of the time doing the work largely outside of your code (e.g., it's IO bound).

17.7 Going Parallel

When you want to fire off a bunch of parallel work, this is generally best done using Futures. We can rewrite the example we used to collect the application data using Actors and Messages from before, but this time we will use Futures and run the algorithm in parallel.

```
implicit val askTimeout = Timeout(5.seconds)

// Instantiate our "data store"
val ds = system.actorOf(Props[AppDataStore])
```

```
// Get a future to the initial list of applications
val results = (ds ? GetAppList("me")).mapTo[AppList].flatMap { applist =>
    // Extract the list
    val AppList(list) = applist
    // Create a Future on a list of (future) results from obtaining all of
    // the particular application information in parallel
    Future.sequence(list.map { appname =>
        (ds ? GetAppData(appname)).mapTo[AppData].map { data =>
            val AppData(propertyMap) = data
            propertyMap
        }
    })
}
// Now do whatever you'd like with the (Future) results
```

Before we leave this pattern, I should note that the non-Akka side of Scala can do something along these lines. It's not quite as asynchronous as the Akka version, but it still may work for you in certain situations. Scala's parallel collections let you create simple blocks of *fork-join* code, like this:

```
val applist = Await.result((ds ? GetAppList("me")).mapTo[AppList], 5.seconds)
// Extract the list
val AppList(list) = applist
val results = list.par.map { appname =>
    // We're converting the list of app strings into a list of app properties
    // so we need to resolve the Future right now.
    val data = Await.result((ds ? GetAppData(appname)).mapTo[AppData],
                           1.second)
    val AppData(propertyMap) = data
    propertyMap
}
```

Here, we use Akka's Future merely to synchronize work to the current thread (i.e., the ask syntax), so that we can then run the rest in parallel using a parallel sequence from Scala's collections library.

The main difference with this structure is that synchronizing to the main thread happens at different points. Before we can turn the list of applications into a parallel sequence (i.e., `list.par`), we must have that list. We could

have done that inside of the Future, but it's a little clearer to take that out of the equation.

The key difference is on lines 7 and 8. Since the parallel sequence is helping us implement fork-join, we need to eventually perform the join. So, rather than have everything happen “in the future” as in the previous example, eventually our work happens in “the present.” The advantage here is that it happens in parallel, but it does tie up our current thread while we wait for everything to join up.

Scala’s parallel collections are extremely powerful and can provide a syntactically terse method of parallelization when all you need is a simple fork-join pattern, so don’t forget that it’s available to you.

17.8 An Actor EventBus

Akka’s EventBus concept is an excellent one; however, you don’t always require the generality of it. Often, you just want to create a bus where the subscribers are Actors. This pattern helps us achieve a reusable Actor EventBus.

```
// This is a phantom type implementation of a concept that supplies default
// values to type parameters - something lacking in Scala. Solution is not
// my own. This is provided by Aaron Novstrup from the Stack Overflow post
// http://stackoverflow.com/a/6629984/230401
sealed class DefaultsTo[A, B]
trait LowPriorityDefaultsTo {
    implicit def overrideDefault[A, B] = new DefaultsTo[A, B]
}
object DefaultsTo extends LowPriorityDefaultsTo {
    implicit def default[B] = new DefaultsTo[B, B]
}

import akka.event.{ActorEventBus, LookupClassification}
object EventBusForActors {
    val classify: Any => Class[_] = { event => event.getClass }
}
class EventBusForActors[EventType, ClassifierType](
```

```
classifier: EventType => ClassifierType = EventBusForActors.classify)
(implicit e: EventType DefaultsTo Any,
 c: ClassifierType DefaultsTo Class[_]) extends ActorEventBus
                                         with LookupClassification {

    // Declares that this bus can publish events of any type
    type Event = EventType

    // We're going to classify our events by the class type
    type Classifier = ClassifierType

    // These next three methods are abstract in the LookupClassification that
    // we've mixed in. The LookupClassification fills in the methods that the
    // EventBus requires so that it can expose more specific requirements on
    // us that present less of a burden than what it's managing for us.
    protected def classify(event: Event): Classifier = classifier(event)
    protected def mapSize(): Int = 32
    protected def publish(event: Event, subscriber: Subscriber): Unit =
        subscriber ! event
}
```

The `DefaultsTo` helper lets us define the `EventBusForActors` in a way that just makes it easier to instantiate with some defaults, since you often want an `Event` type of `Any` and a `Classifier` of `Class[_]`. We implement the methods that are required by the `LookupClassification`, declare the types we need (the `Subscriber` type is supplied by the `ActorEventBus` mixin), and we're off to the races.

17.9 Message Transformation

The `MessageTransformer` we saw before is an incredibly powerful tool in your patterns arsenal. A transformer's general implementation is:

```
class MessageTransformer(from: ActorRef, to: ActorRef,
                        transformer: PartialFunction[Any, Any]) extends Actor {

    // Take the incoming, transform, and send outgoing. Note how we
    // keep the original sender as 'from', hiding the transformer
    // from the ultimate receiver.
    //
    // This may not work for your situation. If you need this transformer
```

```
// to transform responses back in the other direction, then you'd want
// to change the 'forward' to a 'tell'
def receive = {
    case m => to forward transformer(m)
}
}
```

17.10 Retry Behaviour

Akka does not guarantee message delivery, as we know. The reasoning for this is quite sound, but that doesn't really matter to most people that are contemplating using remote Actors. When you send a message to a remote Actor, you want to know that it processed. Before we dive into this, we need to cover off some expectations.

The Dirt on Retry

Networks are unreliable; messages do get lost, both requests and responses. These are rare, but they do happen. At the moment, you need to focus on the fact that they are rare, not that they happen. While it may be quite necessary that you account for these problems, remember that you'll be writing a fair amount of code and running an algorithm 100% of the time to handle 0.00000001% of the cases. You need to assess whether eliminating the problem is better than gracefully handling the failure.

Timeouts will come into play here, big time. It's not the networks that tend to be the problem, your application's generally the bigger problem, or it's what your application works with that's the problem (e.g., a database server that's under high load).

Idempotency

Idempotency is the biggest factor in this exercise. A message retry involves potentially duplicating that message, and thus duplicating the processing on the backend. If your backend cannot handle a second (or third, or fourth) invocation of the logic surrounding that message, you cannot implement retry behaviour.

We can drive the idempotency problem home with a deadly example:

```
class NuclearWeaponSilo extends Actor {  
    import NuclearWeaponSilo._  
  
    def live: Receive = {  
        case Launch =>  
            sender ! LAUNCHED_!!!!  
        case ToggleLiveliness =>  
            throw new Exception("Ooops")  
    }  
    def stubbed: Receive = {  
        case Launch =>  
            sender ! NoLaunch  
        case ToggleLiveliness =>  
            context.become(live)  
            sender ! Toggled  
    }  
    def receive = stubbed  
}
```

Now let's drive the problem home with a quick test. We'll toggle it from 'stubbed' to 'live' and back again, putting a toggle retry in place when we time out on the toggle response.

```
// Go to 'live' state  
a ! ToggleLiveliness  
expectMsg(Toggled)  
// Go back to 'stubbed' state  
a ! ToggleLiveliness  
expectNoMsg(100.milliseconds)  
// Hmm... didn't get a response... retry the toggle  
a ! ToggleLiveliness  
// Awesome, it worked!  
expectMsg(Toggled)  
// Run a test fire  
a ! Launch  
expectMsgPF() {  
    case LAUNCHED_!!!! =>  
        println("WHAT?!  Nooooo!!!!")  
}
```

Idempotency is a tricky thing to handle, and potentially deadly if you screw it up. You'll need to ensure that you have a really solid handle on it with respect to your particular domain. We'll present a simple strategy here and you can tailor it to fit your needs if possible.

Using a High Watermark

This is a simple strategy for handling retries; we use a high watermark. When a client makes a request of a server, it specifies a numerical value that indicates which message, in an overall sequence, this message represents. The server will compare that to what it thinks is the appropriate next message number and act accordingly.

This pattern implementation is quite complex and limited. This is due to the solution's generality; thus, it is possibly best as an illustration rather than a true implementation.

Caveats

To implement this solution, we'll simplify the problem. The relationship between our client and server will be limited to the following:

- It will only be request/response from client to server. The client cannot make a one-way communication to the server; to ensure the watermarks match up, the server *must* respond to all messages and the client *must* receive and process those responses.
- This is a one-to-one relationship. The server does not accept messages from other clients and the client does not accept responses from other servers.
 - Both client and server can accept non-watermarked messages from other Actors, but if they are watermarked, things will go badly.

Common Abstractions

We implement this pattern in two parts: the server and the client. We do it in such a way as to attempt to abstract the idempotency protection away from the implementations. As such, both aspects of the client and server share some common properties, which we will investigate now.

```
object HighWaterMark {  
    // The HWM data type will help us coerce some of the ugliness to the side  
    // with the use of implicits.  
    object HWM {  
        import language.implicitConversions  
        def apply(i: Int) = new HWM(i)  
        implicit def int2HWM(i: Int): HWM = HWM(i)  
        implicit def HWM2Int(hwm: HWM): Int = hwm.hwm  
    }  
    class HWM(val hwm: Int) {  
        def +(i: Int): HWM = HWM(hwm + i)  
        override def toString() = hwm.toString()  
    }  
    // Used by the client to mark a message  
    case class WaterMarkedMessage(num: Int, msg: Any)  
    // Responses from the server to the client that indicate the state  
    // of particular message processing issues  
    case class MessageAlreadyProcessed(msgNum: Int, hwm: Int, msg: Any)  
    case class MessageHasBeenMissed(msgNum: Int, hwm: Int, msg: Any)  
}
```

The `MessageAlreadyProcessed` message indicates the given `msgNum` has already been processed, and that the server's high watermark is `hwm`. The `MessageHasBeenMissed` indicates that the server has missed a message somewhere. The message that the client sent is in `msgNum` and the server's high watermark (i.e., the next message number it's expecting) is in `hwm`.

The Server Side

The server side implements a higher message abstraction on top of the `Any` that we normally use. The only message type that's reasonable for input to the server is a `WaterMarkedMessage` type. The internal values inside the `WaterMarkedMessage` are extracted and processed independently from the wrapped message. If the watermark information is fine, then the base server code sends the message down to the derivation; otherwise the client side is notified of any interesting issues.

```
object HighWaterMarkServer {
```

```
import HighWaterMark.HWM

class HWMHolder(var hwm: HWM)
class ClientWrapper(client: ActorRef) {
    def !(message: Any)(implicit sender: ActorRef,
                         hwmHolder: HWMHolder): Unit = {
        client ! message
        hwmHolder.hwm += 1
    }
}
}

trait HighWaterMarkServer extends Actor {
    import HighWaterMark._

    import HighWaterMarkServer._

    // The current high water mark
    implicit val watermark = new HWMHolder(HWM(0))

    // The 'client' is what the server implementation can use to
    // send responses to the Client. This will keep the
    // watermark in check
    var client = new ClientWrapper(context.system.deadLetters)

    // Derivations must implement this in order to process messages that
    // are deemed as valid (i.e. not already processed or missed)
    def messageProcessor: Receive

    // The main business end of the server. Processes the incoming messages
    // checking them for validity and passing them on to the implementation
    final def receive = {
        case WaterMarkedMessage(num, msg) =>
            if (num < watermark.hwm)
                sender ! MessageAlreadyProcessed(num, watermark.hwm, msg)
            else if (num > watermark.hwm)
                sender ! MessageHasBeenMissed(num, watermark.hwm, msg)
            else {
                client = new ClientWrapper(sender)
                messageProcessor(msg)
            }
        case msg =>
            client = new ClientWrapper(sender)
```

```
    messageProcessor(msg)
  }
}
```

Implementations must use `client` in order to send responses to the client. Using the `client` ensures that the watermark is updated properly, and can be used from outside of the current running context (i.e., you can send back to the client from a `Future`).

The Client Side

The client side is a bit more complicated. We need to increment the watermark when the server responds to the client, and we also need to wrap client messages in a `WaterMarkedMessage` for the server to parse. The problem is complicated enough that we use a Finite State Machine to solve the problem, since it provides a clear implementation that isn't available in other methods.

```
object HighWaterMarkClientFSM {
  sealed trait State
  case object WaitingForRequest extends State
  case object WaitingForResponse extends State

  sealed trait Data
  case object Init extends Data
  case class NoPendingRequest(hwm: Int) extends Data
  case class PendingRequest(hwm: Int, msg: Any, retries: Int) extends Data

  // Messages that can be sent back to the true client
  case class FailureToSend(msg: Any)
  case class OneRequestAtATime(pendingMsg: Any, yourMsg: Any)
}

class HighWaterMarkClientFSM(client: ActorRef,
                           server: ActorRef,
                           retryInterval: Duration = 5.seconds,
                           retryLimit: Int = 5)
  extends Actor with FSM[HighWaterMarkClientFSM.State,
                           HighWaterMarkClientFSM.Data] {
  import HighWaterMarkClientFSM._

  import HighWaterMark._
```

```
case object RetrySend

startWith(WaitingForRequest, NoPendingRequest(0))

when(WaitingForRequest) {
    // This can happen if retries are piling up in the
    // server. When it finally does respond, we move to
    // this state but it's going to send a number of
    // MessageAlreadyProcessed messages due to the retries
    case Event(m: MessageAlreadyProcessed, _) =>
        stay
        // The client makes a request
    case Event(request, NoPendingRequest(hwm)) if sender == client =>
        server ! WaterMarkedMessage(hwm, request)
        goto(WaitingForResponse) using PendingRequest(hwm, request, 0)
}

onTransition {
    // Create the retry timer
    case WaitingForRequest -> WaitingForResponse =>
        setTimer("retry", RetrySend, retryInterval, repeat = true)
    // Clear the retry timer
    case WaitingForResponse -> WaitingForRequest =>
        cancelTimer("retry")
}

when(WaitingForResponse) {
    // The number of retries has been exhausted
    // We terminate here.
    case Event(RetrySend, PendingRequest(_, msg, `retryLimit`)) =>
        client ! FailureToSend(msg)
        stop()
    // We can retry again
    case Event(RetrySend, PendingRequest(hwm, msg, retries)) =>
        server ! WaterMarkedMessage(hwm, msg)
        stay using PendingRequest(hwm, msg, retries + 1)
    // The request has already been fulfilled
    case Event(m @ MessageAlreadyProcessed(num, hwm, _), _) =>
        client ! m
        goto(WaitingForRequest) using NoPendingRequest(hwm)
```

```
// A message has been missed. We consider this death.  
case Event(m @ MessageHasBeenMissed(num, hwm, _), _) =>  
    client ! m  
    stop()  
  
// The response has come in from the server, so we can change states  
// and send it back to the client  
case Event(response, PendingRequest(hwm, _, _)) if sender == server =>  
    client ! response  
    goto(WaitingForRequest) using NoPendingRequest(hwm + 1)  
  
// The client has tried to make another request while we have one  
// outstanding. This isn't allowed.  
case Event(request, PendingRequest(_, msg, _)) if sender == client =>  
    client ! OneRequestAtATime(msg, request)  
    stay  
}  
}  
}
```

Now that we have the FSM, we can use it with the client. The client will embed the FSM and provide access to it in the implementation as though it were the server.

```
object HighWaterMarkClient {  
    import HighWaterMark.{HWM, WaterMarkedMessage}  
  
    // This is a bit of jiggery-pogery that we use to wrap the calls to '!'  
    // The client derivation can still use '!' to contact the server but we'll  
    // intercept it and wrap it in a WaterMarkedMessage  
    class ServerWrapper(server: ActorRef) {  
        def !(message: Any)(implicit sender: ActorRef, hwm: HWM): Unit = {  
            server ! WaterMarkedMessage(hwm, message)  
        }  
    }  
      
    abstract class NewHighWaterMarkClient(serverActor: ActorRef,  
                                           retryInterval: Duration = 5.seconds,  
                                           retryLimit: Int = 5)  
        extends Actor with ActorLogging {  
        import HighWaterMarkClientFSM._  
        import HighWaterMark._
```

```
val server =  
    context.actorOf(Props(  
        new HighWaterMarkClientFSM(self, serverActor,  
            retryInterval, retryLimit)))  
  
def messageProcessor: Receive  
def handleAlreadyProcessed(num: Int, hwm: Int, msg: Any): Unit =  
    log.info("Dumping response to message already processed ({} , {} , {})",  
        num, hwm, msg)  
  
def handleMissedMessage(num: Int, hwm: Int, msg: Any): Unit = {  
    log.info("Dumping response to message skipped ({} , {} , {})",  
        num, hwm, msg)  
    context.stop(self)  
}  
  
final def receive = {  
    case MessageAlreadyProcessed(num, hwm, msg) =>  
        handleAlreadyProcessed(num, hwm, msg)  
    case MessageHasBeenMissed(num, hwm, msg) =>  
        handleMissedMessage(num, hwm, msg)  
    case m =>  
        messageProcessor(m)  
}  
}  
  
abstract class HighWaterMarkClient(serverActor: ActorRef)  
    extends Actor with ActorLogging {  
    import HighWaterMark._  
    import HighWaterMarkClient._  
    import language.implicitConversions  
  
    // Made implicit so that the ServerWrapper has easy access to it  
    implicit var watermark = HWM(0)  
  
    // We create 'server' as a specific instance to make it natural  
    // for the client  
    val server = new ServerWrapper(serverActor)  
  
    // Required to be implemented by the derivation so it can process  
    // messages as normal  
    def messageProcessor: Receive
```

```
// overridable callback that does something when a message is
// reported as already processed by the server
def handleAlreadyProcessed(num: Int, hwm: Int, msg: Any): Unit = {
    log.info("Dumping response to message already processed ({} , {} , {})",
        num, hwm, msg)
    watermark = hwm
}

// overridable callback that does something when the server says
// that "some" message (or messages) has been missed
def handleMissedMessage(num: Int, hwm: Int, msg: Any): Unit = {
    log.info("Dumping response to message skipped ({} , {} , {})",
        num, hwm, msg)
    // Things are probably completely screwed here...
    // The only reasonable default is to toss our cookies
    context.stop(self)
}

// The watermark's main business end. Distributes messages from the server
// to the callbacks as well as increments the watermark when the server
// responds. It also allows for other Actors to send messages without
// mucking with the watermark.
final def receive = {
    case MessageAlreadyProcessed(num, hwm, msg) =>
        handleAlreadyProcessed(num, hwm, msg)
    case MessageHasBeenMissed(num, hwm, msg) =>
        handleMissedMessage(num, hwm, msg)
    case m if sender == serverActor =>
        messageProcessor(m)
        watermark += 1
    case m =>
        messageProcessor(m)
}
```

Usage

With that massive amount of work, we can now implement simple clients and servers. An example of a simple Ping-Pong server might be:

```
class Server extends HighWaterMarkServer {  
    def messageProcessor = {  
        case "Ping" =>  
            client ! "Pong"  
    }  
}
```

Note the use of `client` instead of `sender` for sending responses to the client. This is critical to ensure that we manage the watermark properly. The corresponding client is also pretty simple, but it has to manage the possible failure to send to the server.

```
class Client(svrActor: ActorRef) extends HighWaterMarkClient(svrActor) {  
    import HighWaterMarkClientFSM.FailureToSend  
  
    // We're going to send 3 pings, and we'll keep track of them here  
    var pingSends = 0  
  
    // Send the first ping  
    override def preStart() = server ! "Ping"  
  
    def sendPing(): Unit = {  
        pingSends += 1  
        if (pingSends < 3) server ! "Ping"  
    }  
  
    def messageProcessor = {  
        case "Pong" =>  
            sendPing()  
        // This can happen, and if it does, we've lost our "server" so  
        // we're going to kill ourselves as well  
        case FailureToSend(msg) =>  
            println("Ping send failed. Gotta die.")  
            context.stop(self)  
    }  
  
    override def handleAlreadyProcessed(num: Int,  
                                         hwm: Int,  
                                         msg: Any): Unit = {  
        // OK, our Ping was already handled but we lost the result somewhere  
        // No problem, we didn't need the result, we'll just Ping again if  
        // we need to  
    }  
}
```

```
    sendPing()  
}  
}
```

We're using `server` to talk to the server instead of `svrActor`, which is as critical as using `client` in the server to talk to the client. If the client's internal FSM happened to do a retry, then the server would possibly respond stating that the message was already handled.

In our simple Ping Pong case, the business logic dictates that we just optionally send another Ping. Real business logic might dictate that we retrieve some value from the server instead. For example, the lost response was from a request that was meant to create some sort of backend data, which would have been returned in that response, then we still need to get that data. The server should expose a mechanism for retrieving the data that was in that lost response.

Conclusion

This is hard stuff. There are messaging platforms that can give you delivery semantics at any of the levels you'd like; *At Least Once*, *At Most Once*, or *Once and Only Once*². However, Akka only guarantees *At Most Once*, and in request/response, there are two opportunities for this to be zero. There are solid reasons for this that include notions of complexity and throughput, but the core concept of embracing failure in the design is also key.

We've presented here a limited but still quite complex *At Least Once* system, but with idempotency protection for the server. The server doesn't need to be concerned about processing messages with side effects more than once, which might otherwise be terribly dangerous.

Now that we've covered the pattern, you should take a sober look at the problem to which you think you need to apply it. The alternative approach is:

- Put a timeout on the request/response pair (e.g., with a Future) and propagate that failure as far back to the original caller as you can. Embrace that failure at the most knowledgeable level of the business logic and let it take whatever action it feels is necessary (which may be to retry through Actor Supervision).

²An AMQP (Advanced Message Queuing Protocol) implementation, such as RabbitMQ, for example.

- Handle the idempotency problem.
 - Make the Server operations idempotent. If this is possible, then you're good to go.
 - Handle the idempotency at the client side. For example, instead of making the request to create a resource, wrap it in an if; for example, “If the server says the resource isn't yet created, tell the server to create it.”

17.11 Shutting Down When All Actors Complete

Developers seem to have a desire to have a bunch of Actors do some work and then shut the system down when they complete. Akka doesn't have any direct support for this because it's not easy to say when an Actor is finished, in a general sense. Still, the desire is common, so let's look at an approach for dealing with it.

To give a bit of concrete understanding, assume we have an Actor that adds numbers together, from which we can retrieve the result. We'll send a whole bunch of Add commands to a group of these Actors and then grab the results. All of this will be happening asynchronously, so by the time we reach the point of shutting down the system, the processing will not have completed.

We want to block the mainline thread before calling `shutdown()` on the system, so that we can allow the Adder Actors to drain their queues and respond with values before the app terminates.

```
// Create ten Adder actors
val actors = (1 to 10).map { _ =>
    sys.actorOf(Props[Adder])
}
actors.foreach { a =>
    // Add a couple of thousand numbers
    (1 to 2000).foreach { i => a ! Add(i) }
}
// print out the results of the additions
actors.foreach { a =>
    (a ? Get) onSuccess {
        case result => println(result)
    }
}
```

```
    }
}

// Shut them down gracefully
val stopped = actors.map { a => gracefulStop(a, 5.seconds) }
// Wait for the graceful stop to complete
Await.result(Future.sequence(stopped), 5.seconds)
// Shut down the system
sys.shutdown()
```

By sending all of the Actors a `gracefulStop()` command, we place a message in their Mailboxes that exists behind the request to get the value. By waiting on the result from the `gracefulStop()`, we block the main thread until both get messages have been received and the Actors have subsequently stopped. We can then deterministically shut down the ActorSystem.

17.12 Chapter Summary

You've learned an absolute ton about Akka to this point. The patterns you've just read through provide you with a set of intellectual tools you can turn to when you're presented with a particular problem. They're not solutions; they're meant to frame a particular situation into an implementable pattern that you can then mold into the final shape you need. Perhaps the Message-Transformer almost works, except that you need it to be a constant man-in-the-middle rather than a forwarder. You now have the skills to make that happen. Add that new pattern to your arsenal and spread the love.

It's a good idea to understand these patterns at a high-level of competency, not necessarily because they're so valuable as patterns but more because some of them try to provide reusable Akka components where it's reasonable. Understanding these helps you to understand how to build reusable components in your own code, which increases your own library of tools that you can employ to build applications faster and more reliably.

Patterns are good things to help understanding and to create a vocabulary in your own teams and working environments. But as with any pattern-based design approach (usually, of course, with a much more rigorous pattern definition), don't get bogged down in them. Occasionally, adhering to previously used patterns can blind you to the much simpler solution that might be staring you right in the face.

Chapter 18

Antipatterns for Akka Programming

Akka is a flexible and powerful toolkit that tames the razor-toothed beasts of both concurrency and parallelism. As comforting as that may be, it doesn't make dangling your favourite personal appendage into the gaping maw of one of those beasts a really great idea. Akka gives you enough rope to hang yourself with, just like anything else worth using in the world of software development.

In this chapter, I'll shed a little light on some of the traps you really don't want to step into. Some of them are painted neon pink and are easy to see, while others are covered in leaves just waiting for you to make that wrong move. Fortunately, there aren't a ton of these, so we should be able to blast past them without a lot of thought.

18.1 Mutability in Messages

You can't do this. Akka states clearly that you must use immutable messages between Actors and they aren't kidding. I even got slammed by this one recently; in fact, it was while writing this book. I tried to be "clever"¹ when implementing the Retry pattern. Maintaining that pesky watermark was pretty tricky, and in a previous version I tried making it a mutable object in the client side. However, I failed to realize that it was being passed as part of the message, which made my watermarks go all wonky because I didn't make a defensive copy of it.

So, did I make a defensive copy? *No*. I realized the pure evil that comes up to bite you in rear when you least expect it merely because you have (even

¹Yup. It's the root of a ton of evils.

inadvertently) used something mutable in a message.

What exactly are we talking about here? Well, if you see things like this in your code, be scared:

```
// var... var is bad
class SomeMessage(var some: Int, var msg: String)

// Fine and dandy outside of a message
class HighWaterMarkHolder(var hwm: Int)

// Which makes its use here a terrible idea
case class WaterMarkMessage(hwm: HighWaterMarkHolder)
```

It was that last one that bit me in my special place. At the point of usage, it doesn't look bad—we right case classes all the time that look like that—but the fact that the object being composed inside the case class is ultimately mutable causes us pain.

The good news is that this is absolutely the *first* time I've ever been bitten by it, and it was because I was being foolishly clever.

18.2 Loosely Typing Your Messages

This one's pretty simple. Don't use *values* for messages. Sure, we've seen a lot of stuff like this in the book:

```
def receive = {
  case "Hello" =>
    sender ! "Hi"
}
```

But that doesn't mean it's a good idea. Basically, don't use numbers or Strings for messages. Use the type system; it's there to catch you when you fat-finger or miss your morning coffee, or when you refactor. If you use values, the type system laughs at you when get that bug report back from the Fortune 500 company stating that they lost 200 million dollars because you pattern matched on 5 instead of 6.

18.3 Closing over Actor Data

This one is a little less obvious and can creep up on you at the wrong moment. It shows up mostly when you spawn a Future or child Actor from within an Actor. If you close over some mutable state or a non-referentially transparent method in your Future or child Actor, then you've committed a great evil.

As a reminder, let's look at an easy way to commit great evil inside an Actor:

```
class SomeActor extends Actor {  
    def receive = {  
        case Request =>  
            (Server ? DoSomething) map { _ =>  
                sender ! RequestComplete  
            }  
    }  
}
```

See it? The Future, which is created from the ?, has closed over `sender`. This is a bad idea because `sender` is a method, not a `val`. You haven't closed over the `ActorRef` that `sender` returns, which is what you wanted to do.

To handle this particular case, Akka has added the `pipeTo` pattern that we discussed earlier. You would alter the above to the following:

```
class SomeActor extends Actor {  
    def receive = {  
        case Request =>  
            (Server ? DoSomething) map { _ => RequestComplete } pipeTo sender  
    }  
}
```

Now `sender` is outside the Future's scope and back in the Actor's scope, which is immediately evaluated to the `ActorRef` that it returns.

That's the nasty version because it *looks* like `sender` is a `val`. It's quite possible for you to create the same problems in your own code, so watch out for it.

The less nasty and more obvious version shows up as private mutable data in your Actors, which is a direct consequence of data that you create in your Actor.

```
class SomeActor extends Actor {
    var counter = 0
    def receive = {
        case Request =>
            (Server ? DoSomething) map { _ =>
                // Nope. Don't do it.
                counter += 1
                RequestComplete
            } pipeTo sender
    }
}
```

The counter is being accessed outside of the Actor's private fortress and is thus being accessed across multiple threads. There are memory barriers involved in this, concurrent access violations, and the like. Don't do it.

If there's some side-effect work you need to perform as part of your spawned concurrency, you need to join that behaviour back to your Actor. For example, we might do the following:

```
class SomeActor extends Actor {
    var counter = 0
    def receive = {
        case Request =>
            val requester = sender
            (Server ? DoSomething) map { _ =>
                (requester, RequestComplete)
            } pipeTo self
        case (relayTo: ActorRef, message) =>
            counter += 1
            relayTo ! message
    }
}
```

There are lots of ways to do it, but the key here is that we've moved the alteration of the Actor's mutable data to where it belongs—inside the Actor's `receive` method. We've closed over a frozen instance of the `sender`, frozen as `requester`, and piped the intermediate result back to `self` in order to operate on it further inside the Actor's private `receive` method.

Another thing you can't close over is the ActorContext. The ActorContext carries some of the Actor's most intimate details. It should live as long as the Actor lives and no longer, and it should never be accessed from outside of the Actor itself.

18.4 Violating the Single-Responsibility Principle

Akka programming doesn't let you violate this principle—it's generally a good principle across the spectrum of software development and that makes it a good rule of thumb. The principle exists to ensure that your classes or functions are easily understood, easily reused, and are loosely coupled. In Akka, the definition includes another dimension: *your Actors are more naturally resilient*.

The Supervisor of a given Actor has a simple Decider that it employs in order to help your application heal itself. But the Decider isn't exactly all that psychic; if you have very complex logic in your Actor and it can fail for any one of a dozen reasons, the Decider will have a tough time knowing what to do.

The more complex your Actor becomes, the more difficult it is to understand what you should do when it throws an Exception. To illustrate, we will take a simple and obvious example.

```
class ComplexActor(initial: Double) extends Actor {  
    import ComplexActor._  
  
    var interestDivisor = initial  
    def receive = {  
        case Divide(dividend, divisor) =>  
            sender ! Quotient(dividend / divisor)  
        case CalculateInterest(amount) =>  
            sender ! Interest(amount / interestDivisor)  
        case AlterInterest(by) =>  
            interestDivisor += by  
    }  
}
```

What is this Actor's Supervisor strategy? Clearly, there can be a Divide by Zero Exception, but it comes in two possible cases: Either the safe Di-

vide message or the not-so-safe CalculateIntrest message. What should the Decider do?

Well, in the case of a Divide error, we could Restart or Resume. Computationally, it's safer to Resume of course, since there's no value in Restarting. But, in the case of CalculateIntrest, there's a worse problem: the state of the Actor is *messed up*. If the business is happy to put the `interestDivisor` back to its initial state (and let's pretend that it is), then we must Restart this Actor since every calculation from here on out will throw a Divide by Zero Exception. So if you choose to Restart, then the Actor will Restart even when the Exception comes from the Divide message, which is a bad thing as well.

The Supervisor is in *no position* to decide what to do. This Actor does more than it should and needs to be broken up into two separate pieces of functionality. The beauty of doing that is that the problems completely go away and are now properly managed by the default strategy in both cases. Restarts are no big deal for the Divide case, and they are now appropriately applied to the CalculateIntrest case.

Keeping things simple makes the problems recursively solvable, which makes the hierarchies simple, which is a good thing. The only other major place that this shows up is when you have to apply the same Decider to multiple children. Just as we have the same Exception meaning two different things in the Actor above, we can have the same Exception meaning two different things for two different Actors that are both supervised by the same Supervisor. Don't do this. Keep it simple.

18.5 Inappropriate Relationships

Actors present the developer with a bunch of “live stuff” which, for the uninitiated, presents them with relationship problems². When you have a Master-Slave relationship between Actors, sometimes the knee-jerk reaction is to have Masters know about Slaves. It is perfectly reasonable if the Master creates the Slaves as children. If you're in a multi-node, non-parent-child relationship, it's backwards.

If you have to reconfigure the Master node in order to ensure that it can talk to newly running Slaves, then you have it backwards. It's very hard to keep the Master “knowledgeable” about all of the Slaves, where they are and

²My wife still isn't happy about all these Actors I'm seeing on the side

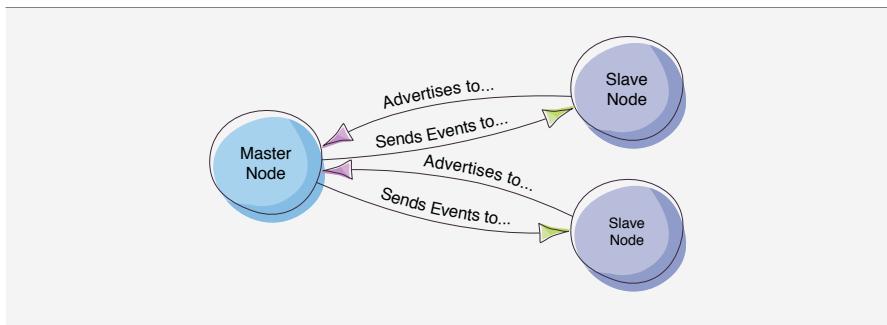


Figure 18.1 · Masters send events to Slaves, but it only does this after the Slaves have advertised their existence to the Master. Don't do it the other way around.

who they are. On the other hand, it's very easy to tell Slaves where the one Master is. This is more resilient to failures and restarts, and scales much better (as new Slaves are added dynamically).

This sort of thing shows up once in a while when you're pulling your relationships together in Actor-based programs. Make sure that they point in the right direction to keep things flexible.

18.6 Too Much actorFor()

We looked at this in some detail when we were learning about Supervision, and it's really quite an important point. While `actorFor()` is a very useful utility, it creates a coupling between Actors that can be very brittle over time.

From the Actor's point of view, there are really only two major reasons for using `actorFor()`:

1. Looking up your dependencies
2. Looking up your descendants

As a general rule of thumb, looking up your descendants is pretty resilient. In general, you're in charge of creating those descendants so the path to them and their names is reasonably stable. In fact, using `actorFor()` to look up descendants is usually a heck of a lot better than anything else you might want to do. Remember our Plane? It needed to start its “children”

as children of supervisors (i.e., grandchildren). This means that the Plane itself is not making direct `context.actorOf()` calls on them, which means it doesn't get the ActorRefs directly, which means it has to get them some other way. The easiest way to do it is to use `actorFor()`.

```
def actorForControls(name: String) = actorFor("Controls/" + name)

...
val controls = actorForControls("ControlSurfaces")
val autopilot = actorForControls("AutoPilot")
val altimeter = actorForControls("Altimeter")
val heading = actorForControls("HeadingIndicator")
val bathrooms = actorForControls("Bathrooms")
val leadAttendant = actorOf(Props(newFlightAttendant).withRouter(
    FromConfig()), "LeadFlightAttendant")
val people = actorOf(Props(new IsolatedStopSupervisor
    with OneForOneStrategyFactory {
    def childStarter() {
        context.actorOf(Props(PassengerSupervisor(leadAttendant, bathrooms)),
            "Passengers")
        context.actorOf(Props(newCoPilot(plane, autopilot, altimeter)),
            copilotName)
        context.actorOf(Props(newPilot(plane, autopilot,
            heading, altimeter)), pilotName)
    }
}), "People")
Await.result(people ? WaitForStart, 1.second)
```

However, looking up your dependencies is more problematic. Dependencies are usually outside your list of descendants and hard-coding knowledge of their location outside your own subtree will create brittle coupling between you and them.

- It makes them more difficult to test. This alone is a pretty big indicator that something's wrong. If you have to set up a dozen mock Actors in a specific tree outside your Actor under test, then it'll be brutal to test the Actor.

- It limits the ability for another developer to re-jig the supervisor of those dependencies, since their paths will change and the lookups will change.
- `actorFor()` returns the Dead Letter Office when it can't find what it's looking for. So, unless you're diligent about checking what you have, you could happily be sending messages right down to `/dev/null`.

Inject dependencies as you normally would in any OO-style program. It will make for more flexible systems in the end.

18.7 Not Enough Config

Every time you have a parameter that you use in an Akka program, and it isn't a reflected configuration value, you need to ask "why?". Developers (yeah, you) can never fully predict or understand how their systems will be used in the real world, and users have a very keen insight into how they want to use your software that you may never have envisioned.

The bottom line is that doing the following is easy, the moment you truly realize that the parameter is one you want to keep in your code:

```
# application.conf
com {
    mycompany {
        rocks {
            cutoff = 7.0
        }
    }
}

class ResonanceActor extends Actor {
    val config = context.system.settings.config
    val cutoff = config.getDouble("com.mycompany.rocks.cutoff")
    ...
}
```

You know what's not easy? Doing it after you release the product and the customer has a huge problem that would be fixed by letting them tune that parameter.

If you have magic values in your code, then they should be in your configuration.

18.8 Needless Future Plumbing

When you spend a lot of time with one Akka feature, you can start to forget about the other features. At one point, I wrote a piece of code inside of an Actor that looked like this:

```
case GetSomeInformation =>
    (otherActor ? GetSomeInformation) pipeTo sender
```

There's nothing technically wrong with that; the Future is there to bridge between the original sender and the reply from the otherActor. It's suboptimal though, because it's functionally equivalent to:

```
case m @ GetSomeInformation =>
    otherActor forward m
```

The moral of the story is a bit more general than the title would suggest; keep your eyes on the whole system. Akka is a pretty involved toolkit with lots of options and provides you with several ways to solve a problem. Try to remember what you have available to you and pick the best one.

18.9 Chapter Summary

The Akka toolkit certainly makes writing resilient, asynchronous programs *a lot* easier than anything that has come before it, but yes...there are some potential pitfalls. These are things that you'll see in your early stages, but they'll disappear *very* quickly. Just be aware that there are some anti-patterns out there, which have tripped people up in the past. Some of them aren't *death*, such as violating the single-responsibility principle, but if they stick around too long and get buried too deep in your application, they can really start to sting.

You have your patterns, your anti-patterns, and a lot of understanding on how to code in the new world. You're almost at the plateau, and it's time to look at what else Akka provides for you to develop your apps with speed, and a hell of a lot of style.

Chapter 19

Growing Your App with Add-On Modules

Akka is a pretty big, pretty involved toolkit, but it's also very modular. As a result of the modularity, it's continually growing, which is a good thing for you because it means that you can grow with it.

Akka's foundation is so flexible and so reusable that you can implement a *ton* of stuff in the paradigm it enables. Not only that, Akka allows you to extend beyond its basic core right inside of the ActorSystem itself. The toolkit isn't a framework, like we've already seen, but it gives you the power to create any framework you might need to make implementing your applications easier and faster.

In this chapter, I won't go into great detail about how to use or implement these extra aspects of Akka; I'll just give an overview of them so you know they're there, and why you might use them. By the time we're done, you should have a good sense of what's available so you don't do something silly like write it yourself.

19.1 Extensions

Extensions come in the form of add-ons to the ActorSystem. Akka itself has several extensions that have been instantiated for you already. Some of the others we have yet to see, but they are also included in the Akka distribution. For example:

Serialization Serialization is implemented as an Extension in order to put some globally required methods and information right into the Ac-

torSystem for you. You would use it when you want to serialize something manually:

```
val system = ActorSystem()
val original = "woohoo"

// Here we extract the extension from the system we just instantiated
val serialization = SerializationExtension(system)

// Now we can use it to obtain the serializer for the appropriate type
val serializer = serialization.findSerializerFor(original)

// Serialize it
val bytes = serializer.toBinary(original)

// Deserialize it
val back = serializer.fromBinary(bytes, manifest = None)
```

TestKit The TestKit stores some of its core settings in an Extension. It makes for a pretty nice place to keep this stuff when it's not relevant to any particular instantiated instance of something (e.g., the instantiated TestKit).

ZeroMQ To incorporate the ZeroMQ library into Akka, you need several support methods, along with configuration and a special guardian Actor. An Extension is the perfect place to house these bits and pieces.

Transactors Actors themselves can participate in transactions, which require a bit of configuration, which is stored in an Extension.

If you ever get to the point where you think you need to create a new “module” for Akka, you might be in a position to create an Extension. If there are global configurations, helper functions, external players you need to hook into, counters you need to set, barriers you want to activate on, or whatever else you might need, the Extension might just be the right avenue on which to grow your app.

19.2 Working with Software Transactional Memory

Scala ships with the Software Transactional Memory (STM), but Akka has support for working with it.

1. The Agents that we've already seen support interacting with the STM by being able to add themselves to a transaction's successful commit. When the transaction successfully commits, the Agent can execute the appropriate send to update itself.
2. Actors can participate in transactions through the Transactor module.

Agent-send-on-successful-commit is fairly straightforward, so we won't go into much detail about it, but a "transactor" merits a bit more explanation.

Transactors

You can use the STM without any help from Akka. You can use it privately inside a given Actor just to ensure that a heap of modifications either successfully completes or does not, atomically. You can also expose a shared transactional data structure across many different Actors and let STM ensure consistency between the shared accesses.

Transactors come into play when you want to coordinate *operations* amongst a group of interrelated Actors. For example, imagine you have Actors that represent a group of friends. These friends are playing a game that requires that they choose a destination in which to travel, and they then vote on a proposal. The votes can be cast in a transactional nature, completing successfully upon full agreement or rolling back to the proposal stage.

The Transactor provides a DSL-like API and a set of Messages and objects that you can use to work with the STM. It can take a little while to wrap your head around, but it is certainly possible. You can consult the Akka documentation for more information.

19.3 ZeroMQ

A while back, ZeroMQ¹ came on the scene with some nifty claims regarding speed. And speed being what it is to developers—you know, like catnip is to a cat that's already high on some designer pharmaceutical drug—it has become pretty well known. Akka provides hooks into ZeroMQ that allow you to use it as an intermachine transport layer.

¹ZeroMQ at <http://www.zeromq.org>

If you’re a devotee to the 0MQ or you need to integrate with it, or you just want to test out the catnip for yourself, Akka has you covered. You can use the ZeroMQ module to get what you need.

19.4 Microkernel

We haven’t covered how you would actually package, distribute, and run an Akka application. There are a couple of reasons for this:

- You gotta leave some stuff out of the book (that’s a rule).
- There are many options available to you.

Because Akka is a toolkit, you can run it inside your application server just fine. You can also roll your own application with a `main` function, launch it with scripts, distribute it in a `.jar` or `.tar` file, or you can use a web framework that’s already designed to work with Akka (more on this later).

The Microkernel helps when you’re looking to build a “pure” Akka application from scratch—you have no web container, or Java app server of any kind, or something that pre-exists to which you’ll add some Akka goodness.

The Microkernel handles the problem of building, packaging, distributing, and executing these standalone apps for you. If you find yourself in this situation, look at the reference documentation for more information on the Microkernel, use it, and then collect your profits.

19.5 Camel

Ah, Camel.² Camel is one of the most impressive integration systems to grace the Enterprise and the Cloud... well, ever. If you are a Java Enterprise developer or Java Cloud developer and don’t know what Camel is, you need to do two things: 1) Go stand in the corner for 20 minutes and feel bad about yourself and 2) go read about Camel right now, or promise to do it no more than 8 milliseconds after you finished reading this book.

Camel provides an integration mechanism between systems that speak any number of protocols and APIs, such as HTTP, SOAP, TCP, FTP, JMS, the Filesystem, or pretty much anything else you might encounter. It’s the

²<http://camel.apache.org>

greased plumbing that binds your apps together, especially those you didn't get right in the first place (you know, the ones that are just plain wrong).

Akka integrates extremely well with Camel. Since Camel essentially provides message-based plumbing and Akka is a message-based pile of awesomeness, you get a wonderful integration between Camel and the internals of the application that you've wisely written in Akka.

Just to give you a quick taste, we'll write a file consumer. This Actor will use Camel's file-system producer that will listen to files in a given directory (creating that directory if need be) and convert those file system events into messages for an Actor.

```
class FileConsumer extends Consumer {  
    def endpointUri = "file:/tmp/actor"  
  
    case class PrintThis(m: String)  
    def receive = {  
        case msg: CamelMessage =>  
            self ! PrintThis(msg.bodyAs[String])  
        case PrintThis(msg) =>  
            println(s"""  
| Hey I got a message!  
| I got it from Camel but this handler doesn't  
| really know that. The message is:  
|  
| ${msg}""".stripMargin)  
    }  
}
```

Now to use it is pretty simply. We can just echo some text into a file in the appropriate directory and they are sent to our Actor.

```
% echo "Hi there Mr. Camel, Dood!" > /tmp/actor/file
```

And this produces the right thing, of course:

```
Hey I got a message!  
I got it from Camel but this handler doesn't  
really know that. The message is:
```

```
Hi there Mr. Camel, Dood!
```

We can keep echoing into that file all day long and it will keep sending those messages to our FileConsumer. Normally, you just have to stick one of these guys in front of another Actor that's expecting the PrintThis message, but for simplicity we stuck the Consumer Actor and the "business" Actor into the same thing.

The beauty of the untyped Actor means that you can swap things out without having to worry about upstream guys. I once converted the messaging conduit in my application from a homegrown WebSocket connection to one that spoke JMS to an upstream system in 60 lines of code in less than 2 hours of work. It's that powerful and that easy.

19.6 Durable Mailboxes

Durable Mailboxes let you specify a Mailbox for the Actor that can survive machine-death. The Mailbox itself is backed by some sort of durable storage; the default Akka implementation being a journaled file on the file system. Advanced Message Queuing Protocol (AMQP) does have an implementation available, but it isn't distributed with Akka.

Beyond the basic file system durability, Akka provides a mechanism by which you can write your own durability solution. It's not terribly difficult to implement, assuming you have a solid piece of kit that can reliably store your messages for you (e.g., a database server, a memory-backed distributed key-value storage system, etc.).

There are systems that require some sort of durability, history, or constant consistency, and in these cases, the durability module might be exactly what the doctor ordered. In real-time systems, durability may not be the solution you need if all you're looking to do is record events for later mining or post-event processing.

19.7 Clustering

Clustering is the big feature of Akka 2.x ($x \geq 1$) and it's the feature that will bring Akka into the center stage for those *really* huge applications that want to have load-balancing and fault-tolerance across dozens to thousands of nodes.

Unfortunately, it's not complete at the current time, which makes it diffi-

cult to write about.³ As with most things, the authoritative source for information on Clustering will be the Akka reference documentation and website.

The Clustering feature's main thrust is to provide an even more general remote experience, to help load balance Actors, migrate them between nodes, detect failure and institute automatic recovery, and many more things while doing it without having a single point of failure. It's not clear at this time how this will all be realized, but the architecture's bones look like they're well placed to make it happen.

19.8 HTTP

As far as the general public is concerned and, let's face it, for most developers, the Internet is synonymous with the Web.⁴ And the Web is HTTP. Fortunately, there are some pretty awesome HTTP solutions for Akka, the chief one being Play.⁵

The Play framework is a fully loaded, fully asynchronous powerhouse web framework that helps you build truly awesome and highly interactive websites using either Java or Scala with Akka. It is actually built on top of Akka and uses it to power the asynchronous, non-blocking engine.

The only complaint that some people have had is that Play has been a framework for those who want the full-course dinner and are looking to build huge websites with it with page templates and rendering engines and all kinds of cool stuff. For those whose applications only need connectivity and a bit of HTTP routing, Play is just too much to swallow. For those people, Akka recommends Play Mini, which provides a small footprint with few dependencies that makes it very easy to write small HTTP-enabled applications.

Play and Play Mini have gained a pretty happy following because they solve a set of core problems that developers of large-scale websites (or, at least, HTTP enabled applications) need. Play has managed to implement the reactive, asynchronous model in a very clear and manageable way so your applications are easy to understand, and that's a good thing.

³...and even if it were, it wouldn't fit in this book anyway.

⁴I apologize to all the Gopher devotees in the audience.

⁵<http://www.playframework.org>

19.9 Monitoring

Typesafe owns Akka as well as Scala and is the company responsible for bringing you all of this goodness. A business has to make money somehow and this is one of the ways they plan to do it: Large applications tend to need some sort of monitoring tool as well as support from the key players in their implementation; Typesafe provides both of these at a cost, which helps fuel their business.

The Typesafe Console is a web-enabled application written with the Typesafe Stack (Scala, Akka, and Play) that gives you really detailed information about your running application, across your nodes, your Actors, your JVMs, and many other bits of information you need to know. There's also a programmatic interface to it that you can access from JMX or REST.

You can check it out at <http://typesafe.com/products/console>.

19.10 Chapter Summary

Akka is a modular system, which works in your favour because you can interact with it and create your own modules with relative ease. The modules that Akka has already provided, which we now have the briefest insight into, allow you to expand your applications in various ways, sometimes non-intrusively. The ability to add durability by tossing in a new type of Mailbox is a great example of this—the Actor(s) that gain durability are completely ignorant of that fact, and what could be better than ignorance?

You don't really know how to use these modules or the sweet spots where they might be valuable, or have any real experience with them right now, but that isn't the point. You do know that they exist, that they solve particular problems, and that they're available to you to assist in your application development. You probably don't need to invent these particular wheels because they're already done.

Chapter 20

Using Akka from Java

I'm not going to lie to you, the absolute shining API for Akka is the Scala API. While the Akka team has done a great job with the Java API, Scala is a more expressive language and the team has been amazing at exploiting that expressiveness. If you can use the Scala API, you'd be doing yourself a great amount of good by using it exclusively. However, you'll find that the Java API is entirely usable and pleasant.

There are differences between the two, of course. It's not reasonable to cover *all* of the differences, since the Akka reference documentation and the API do a fine job of handling the details, but for those who need to understand how things work in Java, this chapter is for you. I'll assume that you know Java well enough to recognize the more verbose typing, the different import styles, and the other Java-isms we haven't seen thus far.

20.1 Immutability

We've covered immutability a ton, but it's more important in Java because the convention is opposite to that of Scala. For example, the case class in Scala is immutable by default, and considering that's our bread-and-butter message class we generally get immutability "for free." In Java, the opposite is true; you have to specify immutability all over the place.

So, don't forget those `finals`! Akka can't check them for you so you're on your own here. The `final` keyword should be ubiquitous in your code.

20.2 Differences Overall

The differences you'll encounter fall into two main buckets: Java's lack of support for *implicits* and functions as first class members of the language. Java also lacks pattern matching, which makes things a bit more cumbersome, but it does so only in the Actor `receive()` method.

This simply means you have “more stuff” in your code than you would with Scala. But if you’re a Java programmer, you’re already used to more stuff, in general anyway, so this shouldn’t be a huge problem.

20.3 Glue Classes

Where Scala is an FP/OO-hybrid language, Java is only OO, which means that we need to augment a bit of the FP stuff with classes. As such, there are a few classes in the Java API that you need to understand.

20.4 akka.japi.Procedure

The `Procedure` class embodies the notion of void function on one parameter. It allows Java to express Scala’s equivalent of `(T) => Unit`. Essentially, it looks like:

```
interface Procedure<T> {  
    public void apply(T param);  
}
```

If you want to define a `Procedure`, you can simply implement an anonymous `Procedure` interface, like this:

```
Procedure<Integer> p = new Procedure<Integer>() {  
    public void apply(Integer i) {  
        System.out.println(i);  
    }  
};  
p.apply(9);  
// Prints '9'
```

akka.japi.Function

Beyond the Procedure, we generally need functions—those that return values—and this is why we have the Function interface.

```
interface Function<T, R> {  
    public R apply(T param);  
}
```

Now we have an interface for a function on one parameter. We can define a simple function that will convert an integer to its string equivalent as:

```
Function<Integer, String> f = new Function<Integer, String>() {  
    public String apply(Integer i) {  
        return i.toString();  
    }  
};  
String s = f.apply(9);  
assertEquals("9", s);
```

akka.japi.Function2

Much like Function we have c{Function2, which gives us a function on two parameters. If we want to define a function that takes two integers and returns a string representing the sum, we could write:

```
Function2<Integer, Integer, String> f =  
    new Function2<Integer, Integer, String>() {  
        public String apply(Integer i, Integer j) {  
            return new Integer(i + j).toString();  
        }  
};  
String s = f.apply(9, 10);  
assertEquals("19", s);
```

akka.japi.Option

A lot of Scala code takes advantage of the concept of a value that may or may not be defined, which is embodied in the Option. In Scala, the Option has several fine operations defined on it; however, these aren't necessarily required for Java, so there's a slightly stripped down version in this implementation.

Assuming you get an Option from Akka, there are some simple ways you can use it:

```
// When the Option has something
Option<String> opt = new Option.Some<String>("Something");
assertTrue(opt.isDefined());
assertFalse(opt.isEmpty());
assertEquals("Something", opt.get());
assertTrue(opt.iterator().hasNext());
assertEquals("Something", opt.iterator().next());

// When the Option has nothing
Option<String> none = Option.none();
// Or equivalently: Option.option(null);
assertFalse(none.isDefined());
assertTrue(none.isEmpty());
try {
    none.get();
    fail();
} catch(Exception e) {
    // yup... it threw
}
assertFalse(none.iterator().hasNext());
```

akka.dispatch.Mapper

The Mapper class provides us with a bit more than the Function interface does and is generally used in situations where we're *mapping* from one object type to another (of possibly the same type). Generally speaking, you simply have to provide an instance to the appropriate function (e.g., map) and let Akka do the rest.

```
Mapper<Integer, String> f = new Mapper<Integer, String>() {
    public String apply(Integer i) {
        return i.toString();
    }
};
assertEquals("9", f.apply(9));
```

20.5 Messages

We've basically standardized on the case class when we're implementing messages due to all of the magnificent benefits we get from them. In Java, we need to do more work. At a minimum, a message's Java form should look something like this:

```
public class JavaMessage {
    // FINAL - Make it FINAL - gotta be FINAL.
    public final String msg;
    // Value-style constructor
    public JavaMessage(String msg) {
        this.msg = msg;
    }
    // If you don't do this it's going to make things a real pain
    public boolean equals(Object that) {
        if (that instanceof JavaMessage)
            return ((JavaMessage)that).msg.equals(this.msg);
        else
            return false;
    }
    // Ditto here
    public int hashCode() {
        return this.msg.hashCode();
    }
    // Makes diagnostics and the like a lot nicer
    public String toString() {
        return "JavaMessage(" + this.msg + ")";
    }
}
```

```
}
```

Because of this extra verbosity, it might be tempting to eliminate the usage of a message “class” altogether and just go for the primitive type (a String, in this case). Please avoid that temptation. Using specific classes to embody your protocol will pay off in type safety and extensibility later. If the verbosity annoys you, take a good look at your editor to see if it can help out in any way. (For example, I use a Vim plugin that helps me avoid all of this boilerplate by generating it for me.)

20.6 The Untyped Actor

Creating a Java version of the untyped Actor is pretty simple. It gets a bit more complex as you want to go beyond the basics, but it’s still not a big deal.

```
import akka.actor.UntypedActor;

public class JavaActor extends UntypedActor {
    // The main 'receive' method for the Actor
    public void onReceive(Object message) throws Exception {
        // We don't have pattern matching so we just use 'if' blocks,
        // and a whole lot of 'instanceof' invocations
        if (message instanceof JavaMessage) {
            final JavaMessage m = (JavaMessage)message;
            if (m.msg.equals("Hi there"))
                // If we don't specifically send ourself as the second parameter
                // to 'tell' then the Dead Letter Office will get put in its place
                getSender().tell(new JavaMessage("Hi back"), getSelf());
            else if (m.msg.equals("Bye"))
                getSender().tell(new JavaMessage("So long"), getSelf());
        } else {
            // We've got to call this by hand otherwise any unhandled message
            // will be silently discarded
            unhandled(message);
        }
    }
}
```

Creation

You can create this Actor using the usual mechanism of Props passed to an ActorSystem, which will create the Actor using Java Reflection.

```
ActorSystem system = ActorSystem.create("JavaActorSystem");
final Props props = new Props(JavaActor.class);
final ActorRef actor = system.actorOf(props);
```

If we want to create an Actor that doesn't use the default constructor, then we have to supply a factory object that will create it for us. For example, if we have a class with no default constructor, then we have to specify a factory that passes in the required arguments (in this case, a single string value):

```
import akka.actor.UntypedActorFactory;
final Props props = new Props(new UntypedActorFactory() {
    public UntypedActor create() {
        return new JavaActorNoDefault("Start Value");
    }
});
final ActorRef actor = system.actorOf(props);
```

Of course, you should never return a reference to a static, or otherwise pre-existing Actor that might be referenced elsewhere, or have gone through a standard Actor life cycle. This would just be the worst idea ever; don't do it.

Members and Methods

The members of the UntypedActor are basically all there, but some look slightly different than they do in the Scala version:

getSelf() Equivalent to Scala's self value

getSender() Equivalent to Scala's sender method

onReceive() Equivalent to Scala's receive method

supervisorStrategy() Equivalent to Scala's receive method

unhandled() Equivalent to Scala's unhandled method

getContext() Equivalent to Scala's `context` value. And within the context, we have the following:

- actorOf()** Create children as usual.
- become()** Change running states.
- getChildren()** Get the Iterable to the children.
- parent()** The guy who owns you.
- watch() / unwatch()** Throw a DeathWatch onto an ActorRef.
- system()** Access the Actor System.

Life-cycle methods These do what you would expect:

- `preStart()`
- `preRestart()`
- `postRestart()`
- `postStop()`

These all do the equivalent of what you'd expect from the Scala API. The main difference is that the `become()` functionality is limited to what some rudimentary Java definitions can provide. No real effort has been made to implement the power of the Scala `PartialFunction` in Java—that would just be silly. As a result, you're limited to what you can do with basic Java. For example:

```
// Create what will be an initial onReceive implementation
Procedure<Object> receiveA = new Procedure<Object>() {
    @Override
    public void apply(Object message) {
        if (message instanceof String)
            getSender().tell("You gave me a String.", getSelf());
        else {
            getSender().tell("You gave me something else.", getSelf());
            getContext().become(receiveB);
        }
    }
};
```

```
// Create a second flavour of the 'onReceive' procedure
Procedure<Object> receiveB = new Procedure<Object>() {
    @Override
    public void apply(Object message) {
        if (message instanceof Integer)
            getSender().tell("You gave me an Integer.", getSelf());
        else {
            getSender().tell("You gave me something else.", getSelf());
            getContext().become(receiveA);
        }
    }
};

// It needs to be implemented but we're going to swap it out
// before it can get used, so we'll throw an exception just
// for fun.
public void onReceive(Object message) throws Exception {
    throw new Exception("This shouldn't have happened");
}

// Override the 'preStart' to instantiate the 'receiveA' as the
// initial 'onReceive' implementation.
@Override
public void preStart() {
    getContext().become(receiveA);
}
```

TestKit

The Java API comes with a TestKit variant that we've made so much extensive use of in the Scala implementation. The implementation is pretty slick and has a lot of the cool stuff we're used to.

The basic pattern you use to hook it up is the following (using JUnit4 as our example test framework). We start with a bunch of imports, part for our test framework and part for Akka.

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.AfterClass;
```

```
import org.junit.BeforeClass;  
import akka.actor.ActorSystem;  
import akka.actor.Props;  
import akka.actor.ActorRef;  
import akka.testkit.JavaTestKit;
```

Now we can implement our JUnit 4 test that looks a lot like the Scala version. The JavaTestKit is a pretty good representation of what we would have in Scala.

```
public class JavaActorSpec {  
    // We'll store it as static and assign it later  
    static ActorSystem system;  
  
    @BeforeClass  
    public static void setup() {  
        // Create the ActorSystem  
        system = ActorSystem.create();  
    }  
  
    @AfterClass  
    public static void teardown() {  
        // Shutdown the ActorSystem  
        system.shutdown();  
    }  
  
    @Test  
    public void testMethod() {  
        // We write our code inside a new JavaTestKit instance  
        new JavaTestKit(system) {{  
            // Create your actor  
            final Props props = new Props(JavaActor.class);  
            final ActorRef actor = system.actorOf(props);  
            // Send it a message - getRef() is the our test actor  
            actor.tell(new JavaMessage("Hi there"), getRef());  
            // Uses our JavaMessage.equals() and error messages will  
            // use JavaMessage.toString()  
            expectMsgEquals(duration("1 second"), new JavaMessage("Hi back"));  
        }};  
    }  
}
```

```
}
```

That's the basic idea. When you need to know more, the API reference can help you out, but you'll find that it's very close to what we've already seen; you're just going to do it from Java instead of Scala, which will just make it a bit clunkier.

20.7 Futures

There's nothing really surprising about the Futures implementation for Java beyond what we already expect, but we'll go over a bit of it here to give you an idea of how it works.

Imports

You need to do a fair bit of importing to get things to work out. Generally, you can find what you need in `akka.dispatch.*`, `scala.concurrent.*` and `java.util.concurrent.*`. If you can't see what you need immediately, it's probably in one of these key places. In the examples that follow, we're pulling in our imports as follows:

```
// Basic Akka Imports
import akka.dispatch.ExecutionContexts;
import akka.dispatch.Futures;
import akka.dispatch.Mapper;
import akka.dispatch.Recover;
import akka.dispatch.OnSuccess;
import akka.dispatch.OnFailure;
import akka.dispatch.OnComplete;
import akka.util.Timeout;
import static akka.dispatch.Futures.future;

// Things we need from Scala
import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
import scala.concurrent.Await;
import scala.concurrent.util.Duration;

// Help we get from Java itself
```

```
import java.util.ArrayList;
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
```

Testing Futures

We'll be driving some aspects of the Java Futures through some JUnit tests, which will be set up in the following framework:

```
public class JavaFutureSpec {

    // The Future needs somewhere to execute, which exists as the
    // ExecutionContext (we can use a Dispatcher as well, but we're going
    // to do this a little lower down).
    static ExecutorService es;
    static ExecutionContext ec;

    // We're going to need a timeout later, so we'll use it from here
    static Timeout timeout = new Timeout(Duration.parse("2 seconds"));

    // Sets up the ExecutionContext
    @BeforeClass
    public static void setup() {
        es = Executors.newFixedThreadPool(2);
        ec = ExecutionContexts.fromExecutorService(es);
    }

    // Shuts down our thread pool
    @AfterClass
    public static void teardown() {
        es.shutdown();
    }

    // ...
}
```

Around the API

Now we'll look at how we can manipulate Futures through Java to achieve the same things we've seen in the past with the Scala API.

A Vanilla Future

Creating a simple Future requires that we instantiate a Callable that can execute the business logic inside the Future. We pass in the ExecutionContext on which the Future will execute:

```
// Use the 'future' factory method to construct the Future
Future<String> f = future(new Callable<String>() {
    public String call() {
        return "Fibonacci";
    }
}, ec);
assertEquals("Fibonacci", (String)Await.result(f, timeout.duration()));
```

Using Map

We can use a Future's functional aspects as well, which are great for building up delayed executions without having to block any threads. Doing so requires that we change from using a Callable to a Mapper:

```
// Create a Future that computes a String
Future<Integer> f = future(new Callable<String>() {
    public String call() {
        return "Fibonacci";
    }
    // And then map that string into an Integer
}, ec).map(new Mapper<String, Integer>() {
    public Integer apply(String s) {
        return s.length();
    }
}, ec);
assertEquals(new Integer(9), Await.result(f, timeout.duration()));
```

Using flatMap

As with map, we can flatMap over a Future as well. The indirection that we need merely creates a Mapper that returns a Future with a result instead of the result directly:

```
// Ultimately we're going to get back a Future<Integer>
Future<Integer> f = future(new Callable<String>() {
    // But we start with a String
    public String call() {
        return "Fibonacci";
    }
    // flatMap it to a Future<Integer> (i.e. not an Integer as before)
}, ec).flatMap(new Mapper<String, Future<Integer>>() {
    public Future<Integer> apply(final String s) {
        // Construct a new Future to an eventual Integer
        return future(new Callable<Integer>() {
            public Integer call() {
                return s.length();
            }
        }, ec);
    }
}, ec);
assertEquals(new Integer(9), Await.result(f, timeout.duration()));
```

Sequencing

We can also sequence a list of Futures with Java as we would with Scala. Let's sum up a bunch of Integers to see how you do this:

```
// Create an ArrayList of 200 Future Integers
ArrayList<Future<Integer>> v = new ArrayList<>();
for (int i = 0; i < 200; i++) {
    final int num = i;
    v.add(future(new Callable<Integer>() {
        public Integer call() throws Exception {
            return new Integer(num);
        }
    }, ec));
}
// Compose that list of Future Integers into a Future of a
// List of Integers
Future<Iterable<Integer>> futureListOfInts = Futures.sequence(v, ec);
```

```
// Map over the results and sum them up
Future<Long> f = futureListOfInts.map(
    new Mapper<Iterable<Integer>, Long>() {
        public Long apply(Iterable<Integer> ints) {
            long sum = 0;
            for (Integer i : ints) sum += i;
            return sum;
        }
    }, ec);
assertEquals(new Long(19900), Await.result(f, timeout.duration()));
```

Callbacks

Callbacks probably have the most obvious need for implementation with classes, so the following should be fairly self-explanatory:

```
Future<Integer> f = future(new Callable<Integer>() {
    public Integer call() {
        return new Integer(42);
    }
}, ec);

// Add the onSuccess callback
f.onSuccess(new OnSuccess<Integer>() {
    public void onSuccess(Integer result) {
        System.out.println("Awesome! I got a number: " + result);
    }
}, ec);

// Add the onFailure callback
f.onFailure(new OnFailure() {
    public void onFailure(Throwable failure) {
        System.out.println("Aw, poo! It didn't work: " + failure);
    }
}, ec);

// Add the onComplete callback
f.onComplete(new OnComplete<Integer>() {
    public void onComplete(Throwable failure, Integer result) {
        if (failure != null)
```

```
        System.out.println("Aw, poo! It didn't work: " + failure);
    } else
        System.out.println("Awesome! I got a number: " + result);
    }
}, ec);
```

Failures

We can create a failure with a simple Exception, as usual:

```
Future<String> f = future(new Callable<String>() {
    // We need to declare that the main Callable method can throw
    public String call() throws Exception {
        throw new Exception("Boo!");
    }
}, ec);
Await.result(f, timeout.duration());
```

Recovering

Recovering from failures is essentially the same in Java Futures as it is in Scala with the now common alteration that the function be represented as an instance of a class.

```
Future<Integer> f = future(new Callable<Integer>() {
    public Integer call() throws Exception {
        throw new Exception("You ain't gettin' no Integer from me.");
    }
}, ec).recover(new Recover<Integer>() {
    public Integer recover(Throwable t) throws Throwable {
        return new Integer(0);
    }
}, ec);
assertEquals(new Integer(0), Await.result(f, timeout.duration()));
```

Using `recoverWith()` is equivalent, but for the extra Future level of indirection, as with `flatMap()`. In other words, you recover with a `Recover<Future<Integer>>` instead of just a `Recover<Integer>`.

20.8 Manipulating Agents

Agents are another point of difference that's worth describing—not because it's a huge problem, it's just different. The bottom line here is due to the fact that Java has no support for functions as first class members of the language, which means that in order to alter the value inside an Agent, you must compose the function inside of a class.

20.9 Finite State Machines

Roll your own! Unfortunately, the FSM that ships with Akka relies heavily on Scala. So heavily, in fact, that porting it to Java just isn't possible. Check the Akka reference documentation for some pointers on how to make something that's equivalent for Java.¹

20.10 Dataflow

Dataflow uses a specific feature of Scala called *Continuations* that has no analogue in Java. Sorry, folks, but if Dataflow is the killer feature for you, then you'll have to move to Scala. But if you do that, then you'll probably just get Dataflow as well as every other amazing thing you've ever wanted in life, so what the heck are you waiting for?!

20.11 Chapter Summary

Akka is *very* accessible from Java,² so there's no reason why you shouldn't adopt the Akka paradigm to help you accomplish your concurrent needs whether you're in Scala or Java. Of course, Akka was written in Scala and it doesn't confine itself unnecessarily to being 100% compatible with Java (since that would just punish us Scala guys). This means you don't get to use all of the features that you might get in Scala, but that doesn't mean you're having to go without, either. The Java API is very rich, and should provide you with everything you need.

¹Or just switch to Scala, already!

²In fact, I'm using it in my day job and it works great.

Chapter 21

Now that You’re an Akka Coder

Congratulations! You’ve achieved a great deal in digesting these pages. You haven’t just learned the core parts of the greatest concurrency toolkit to grace the JVM to date, you’ve opened up your mind to one of the most important programming paradigms of the decade. Non-blocking, fully asynchronous application design and development—which helps you create applications that are resilient in the face of failure and scales both vertically to the hardware available on your machine and horizontally across a farm of machines—is the must-have development paradigm in our multi-core world.

This book can’t possibly hope to make you an expert Akka programmer, and certainly not an expert in the field of this new paradigm, but you’ve climbed over that all-too-difficult hump of a learning curve that puts you in a position to learn much faster. So where do you go from here?

21.1 Akka.io

The Akka website (<http://akka.io>) is clearly the definitive source for information on the toolkit. The reference documentation is some of the best on the web and you should head there right after you chew on this for a while. The reference documentation helps you with the “what” and a decent amount of the “how,” but it’s a lot easier to understand and to put into the right context when you have a solid grasp of the “why,” which you now possess.

If you’re interested in some of the modules that we didn’t cover, head over there and eat them up. The Scaladoc API is also a very useful resource that you should be able to sail through now, using it for the simple reference

tool that it's intended to be.

21.2 The Mailing List

If you look at the Akka website, you'll see a link for the Mailing List. There are a ton of helpful people on there and now is the right time to sign up; you've reached the point where you have a solid foundation, you know where the reference documentation is, and you can even talk the talk. It's at this point where you can start asking some good questions that people will enjoy answering. I know I really like it when I see a question on the mailing list that's actually fun to answer—a design question that presents an interesting use case or one that lets you expand the user's knowledge to a higher level of Akka knowledge.

21.3 Have *Fun!*

There's a lot of joy in writing software that we can be *proud of*. I don't know about you, but when I can write a system that scales well and handles a ton of concurrent incoming events with grace and speed, while at the same time being resilient and *responsive*, that makes me feel like a proud papa!

If you're diligent about applying the paradigm, and grow your applications within it, and even stretch it from time to time, you'll find that your applications are in that first-class tier of reliability and usability. And if looking like the god of concurrent programming doesn't make you feel awesome and if you don't find the process *fun*, then you've probably made the wrong career choice. But if you've read this book, then you're not that person.

Akka is powerful, expressive, and fun. Use it and be awesome.

Cheers!

About the Author

Derek Wyatt is a Software Architect and Developer specializing in large-scale, real-time applications for the World Wide Web. He's been working with Akka since the early days of 1.0 and has used it to implement a number of applications both large and small. After spending many years writing large concurrent systems in C++ using traditional concurrency mechanisms, Derek now embraces the sophisticated, and beautiful simplicity of the paradigm presented in the Akka Toolkit. He also harbours a love of the Vim text editor and the Unix command line that borders on the unhealthy.