

Akka

IN ACTION

Raymond Roestenburg
Rob Bakker
Rob Williams



MEAP



MANNING



**MEAP Edition
Manning Early Access Program
Akka in Action
Version 13**

Copyright 2014 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

brief contents

- 1. Introducing Akka*
- 2. Up and Running*
- 3. Test Driven Development with Actors*
- 4. Fault tolerance*
- 5. Futures*
- 6. Your first distributed Akka App*
- 7. Configuration, Logging and Deployment*
- 8. System structure*
- 9. Routing*
- 10. Message channels*
- 11. Finite State Machines and Agents*
- 12. Working with Transactions*
- 13. Integration*
- 14. Clustering*
- 15. Akka persistence*
- 16. Performance Analysis and Tuning*

Introducing Akka

In this chapter

- An overview of Akka
- Actors and Actor Systems
- Applicability of Akka

In this first chapter, you will be introduced to the various aspects of Akka, what it makes possible, and how it differs from existing solutions. Focus will be on how to take these capabilities and craft powerful concurrent and distributed applications. Akka is at once revolutionary: breaking from the container tradition, and yet doing so by employing ideas that have been around for some time: Actor Systems (if you've never heard of Actors, fear not, we will discuss them at length in the course of the book). You will learn how Akka simplifies the implementation of asynchronous and concurrent tasks, and how it also offers an exciting new solution to distributed fault tolerance. Finally, we will discuss how these concepts, combined with the Akka runtime, deliver on the promise of apps that are more reliable and scalable and yet easier to write (and less prone to rewriting).

Too often, developers wait until defects or scaling issues present themselves to consider these issues. Akka makes it easy to build them in from the first line of code in the project, or add them to an existing project. These are not corner case, late-stage requirements: every app has some part that handles request and processing loads, or runs jobs that have to manage a lot of work. The dominant approach to these problems has been to graft largely home-grown solutions onto already written code. Even if you are adding to an existing codebase, the new code

will be concurrent, scalable and fault tolerant from its inception. This book will convince you that Akka's ability to confer these capabilities is so lightweight and elegant that putting them in the later bin makes no sense.

1.1 What is Akka

While Akka is written in Scala, it is usable from both Scala and Java. Its primary goal is to make the achievement of performance, reliability, and scalability simpler. This is one of the most exciting aspects of Actors: they allow programmers to just focus on how to most efficiently implement solutions, not on having to make their code also manage scaling issues like batching or resource management. Most solutions, as they grow, are pulled simultaneously on these two axes: having to expand capacity while still evolving the application's capabilities. Actors let the programmer just focus on getting the work done; the system provides means outside the code for scaling it when the demand curve grows (or shifts).

For systems that demand real-time, or near real-time, the dominant model of having components synchronously depend upon services that are often unavailable is a showstopper. Facile notions of forking a process or spawning a thread to handle each request are not tenable long term. While containers deploy thread pools, once the execute thread calls the corresponding application code, there is nothing keeping that task from running for as long as it likes, including potentially waiting on many other system components. With the rise of the cloud, truly distributed computing is quickly becoming a reality. While this means more opportunities to rapidly assemble apps that leverage other services, it also means dependence upon those services make simplistic serial programming impossible in the face of performance demands.

Let's consider the items on our wish list:

- Handle many requests in parallel
- Concurrent interaction with services and clients
- Responsive asynchronous interaction
- An event-driven programming model

With Akka, we can accomplish these things today, as the examples throughout the book will illustrate. The example in this chapter will provide a cursory look at how Akka provides for concurrency and fault tolerance. Later, we will tackle the rest of these items, with a host of examples.

The Akka team refers to their creation as a toolkit rather than a framework. Frameworks tend to be a mechanism for providing a discrete element of a stack (e.g. the ui, or the web services layer). Akka provides a set of tools to render any part of the stack, and to provide the interconnects between them. It does this by a specific architectural configuration that lets the seeker of a solution interface simply call methods, rather than worry about enqueueing messages, which is done seamlessly by the Akka runtime.

Akka is made up of modules that are distributed as JAR files; you use them just like any other library. Great care has been taken to minimize the dependencies that are needed in every module. The akka-actor module has no dependencies other than the standard Scala library and is in fact distributed with Scala since version 2.10. Modules are provided to ease the use of Akka for different parts of the stack: remoting, clustering, transactions and dataflow concurrency (as we will see later). There are also modules focused on integration with other systems like the camel and zeromq module. Figure 1.1 shows the main elements in the Akka stack, and how client code is shielded from most of the details.

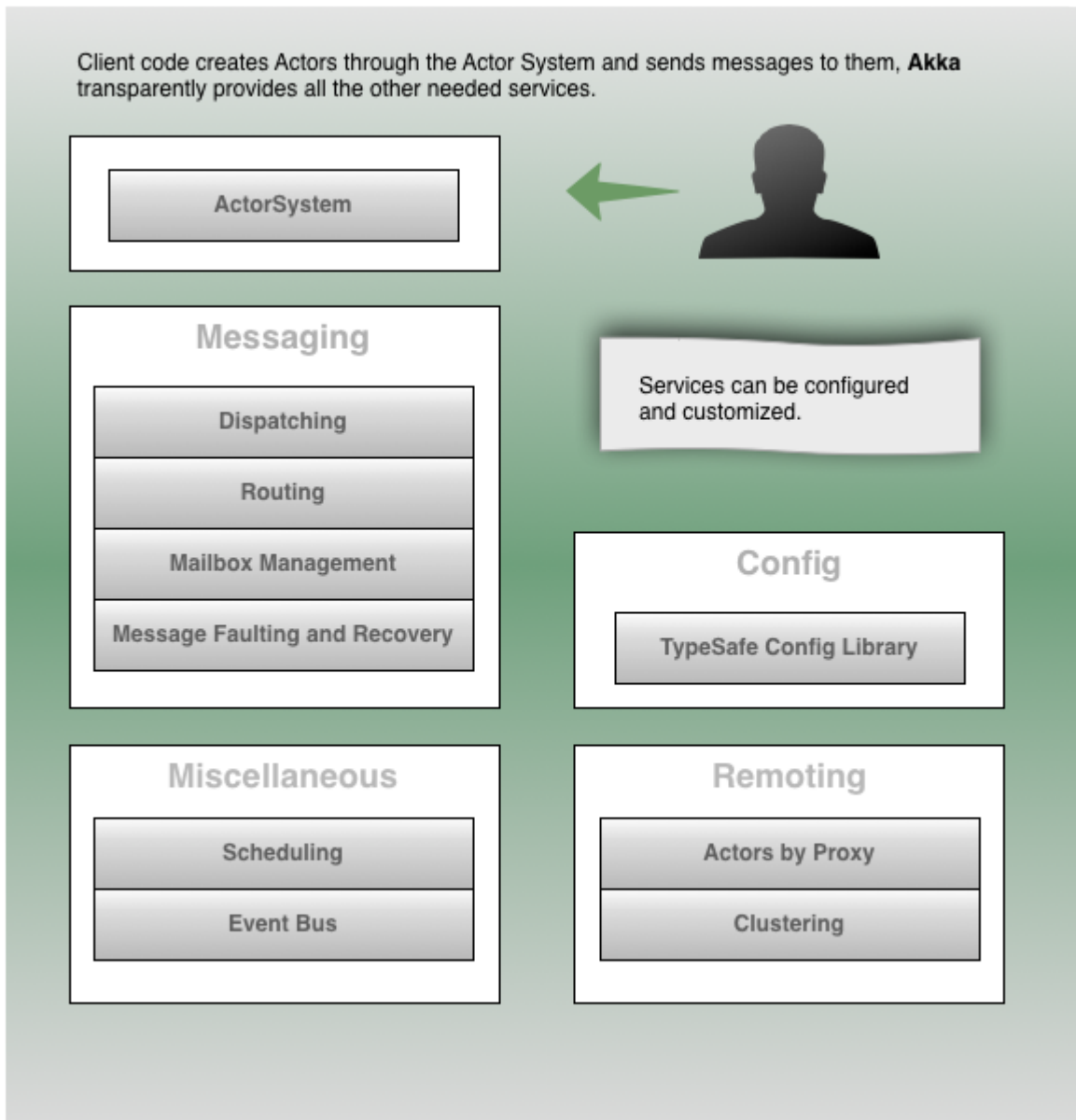


Figure 1.1 The Akka Stack

Akka also provides a runtime. The core of the runtime is the basic actor services and a flexible means of configuring the other modules that you would like to use. There is a microkernel called Play-mini available which you can use to deploy your applications. You will be shocked when you see how much you can do, and how well it can scale, from so little code, with a runtime that has such a tiny footprint.

So what is a typical Akka application made up of? Well, the answer is, Actors. Akka is based on the Actor programming model, which we will discuss in great detail in this book. The history of Actors goes back forty years. It emerged as a way to get scores of machines/CPU's to work together to solve large problems. But

it was also inspired by some of the difficulties encountered in early object languages, growing into the notion of a 'process calculus' that has been embraced by functional languages like Erlang and Fantom. But again, Akka deploys both the language side of the Actor model, and the runtime. In the next section, we will start our exploration of Akka by taking a look at how the Actor programming model makes possible a simpler means of concurrency.

As per the mission of this press, we believe strongly that when you see the Akka versions of some of the common solutions to these problems, you will be convinced of both their superiority, and the much greater ease and elegance afforded the developer in accomplishing them. In the next section, we will start to see the specific elements of Akka that make all this possible.

1.1.1 Simpler Concurrency

In this section, we are going to go over an example application that needs to handle concurrent requests. For an application to be truly concurrent, the processing of the requests must also execute simultaneously, with the executors collaborating to complete the task at hand. Some experience with writing concurrent code using threads in the JVM, and some of the hard problems that come with that territory, is assumed. So here's to hoping you find it as hard as we do and would love a simpler model than threads and locks for concurrency, and as an added bonus, a lot less code to write. First, we'll consider concurrency at the conceptual level, then look at the two ways (shared mutable state/message passing (Akka)) to solve the problem of selling tickets.

THE EXAMPLE PROBLEM: SELLING TICKETS

In this example, customers buy tickets to Events from TicketingAgents. We will look at this example in more depth later in the book, for now, we use it to illustrate the two approaches to concurrency: the traditional one in which the developer is responsible for using threads to distribute the load, and the message-based one, where messages are sent and processing involves simply working through a queue of them. One other difference between the models: the original one employs shared mutable state, the message one, immutability.

What is immutability? As a quick refresher, if something is immutable, it means it is given its state at construction and cannot be changed after. The last 20 years of programming have seen increasing emphasis on the importance of immutability (in the C++ and Java communities). This is a key aspect of the message-oriented model; by having collaborators only interact with immutable

representations, we forego the primary source of difficulties: multiple parties transforming the same objects. This example illustrates that sometimes this means a bit more work in the initial implementation, but weighed against the decrease in defects and the scaling benefits, it's a tiny price to pay (and the result is a cleaner implementation that makes what the code is actually doing much clearer, hence more readable and maintainable). Finally, it also makes the code easier to verify because unit tests rarely account for what happens when state changes are the work of multiple parties. Figure 1.2 shows the most basic flow of the ticketing selling application: what happens when a Customer presents who wishes to purchase a ticket for a given Event.

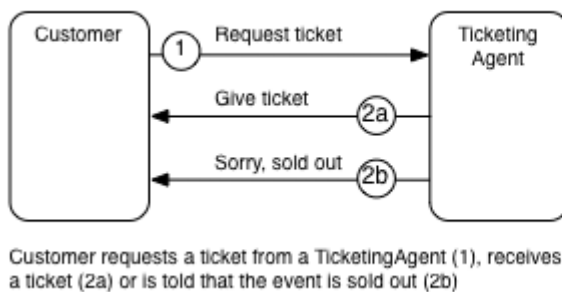


Figure 1.2 Buying Tickets

If you think about the problem of selling tickets to Customers, it seems like a good candidate for concurrency, because the demand side of the model could potentially be overwhelming: what would happen if a million people tried to purchase at the same time? In a thread-based model, the most common simple implementation: forking or spawning a thread for each request, would lead to immediate catastrophic failure. Even a thread pool is going to run the risk of failure, because the requests are probably coming over lines that have timeouts, so if the execute threads are busy for too long, the client will fail before even being serviced. Furthermore, the presumption in the first model that most programmers operate under: that another thread will mean a linear improvement in capacity, is often wrong because those threads are going to contend with each other for the shared resources they are jointly transforming. The combination of messages, and no shared mutable state, will relieve these ills.

Before looking at the two approaches, let's consider the domain model briefly. Clearly, we would have a Venue, which has Seats and schedules Events. For each Seat we will print a Ticket that can be sold to an Customer by a TicketingAgent. You can probably anticipate where the problems are going to come from: each

request for a ticket is going to require us to find one, see if the Customer wants it, and if they do, change that Ticket's state from Available to Sold and then print it out for the Customer. So if our plan for scaling this is to just have more TicketingAgents, clearly giving them all orderly access to the pool of available tickets (under all conditions) is going to be the critical factor in a successful outcome. (What happens if we fail? In the next section, we will discuss the other issue that haunts such pursuits: fault tolerance, discussion of concurrency will prompt many thoughts about faulting strategies.) Figure 1.3 shows how we will have to accommodate these constraints, by having multiple TicketingAgents to deal with the multitude of Customers that may demand service at any given time.

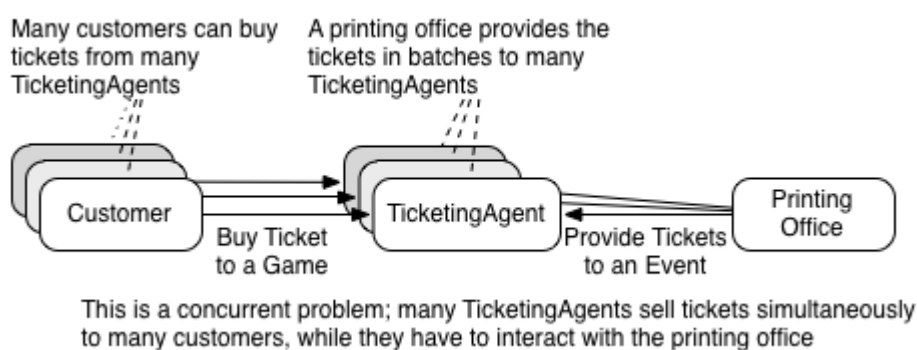


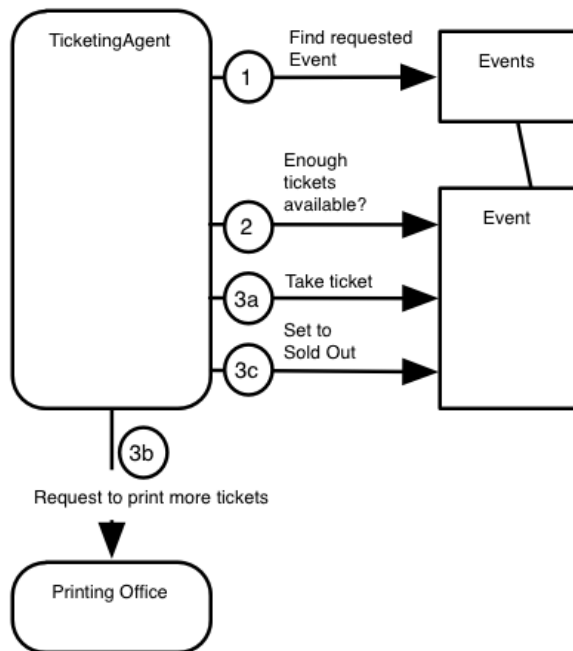
Figure 1.3 Relationship between Customers, TicketingAgents and the printing office

We have only one Printing Office, but that does not mean we will fail catastrophically if it encounters a problem (we will discuss in the next section). The important point here is that we managed to make it possible to have any number of TicketingAgents fielding requests from Customers without blocking on access to the pool of tickets.

OPTION 1: SHARING STATE APPROACH

As the diagrams show, TicketingAgents must compete with each other for access to the list of available Tickets. Some of that wait time is going to result in a Customer waiting while the Tickets of another Customer are being printed, which makes no sense. This is clearly a case of implementation shortcomings poisoning domain model sanctity.

There might be a way to optimize this; figure 1.4 shows a modified approach.



The TicketingAgent takes a ticket (3a) and requests more tickets to be printed when the tickets are running low (3b). If the game is out of print, and there are no more tickets, the kiosk sets the game to sold out (3c).

Figure 1.4 Buying sequence

You can see, we didn't have to go to great lengths to keep the Agents from blocking each other's access to Tickets, and the result is also a bit clearer than what we would probably end up with if someone came in later and rewrote the selling sequence to make it concurrent.

There are three threading pitfalls that can result in catastrophic runtime failure:

1. thread starvation - when the execute threads are either all busy, or have all become deadlocked, so there's literally no one to service any new requests, this effectively shuts down the server (though it probably still looks to an admin like it's up and running).
2. race conditions - when shared data is being changed by multiple threads while a given method is executing, it can cause the internal logic to be subverted and produce unexpected results
3. deadlock - if two threads start an operation that requires locking of more than one resource, and they don't acquire those locks in the same order, a deadlock can ensue: thread 1 locks resource A, thread 2 resource B, then as 2 attempts to lock A and 1 lock B, we have a deadlock.

Even if you manage to avoid these problems, the process of making sure the lock based code is optimized ends up being painstaking. Locking is often done too much or too little; when too many locks are used, nothing really happens at the

same time. Difficult bugs can occur when too few locks are used. Getting to the right balance is hard.

The TicketingAgents have to wait for each other at times, which means that the customers have to wait longer for their tickets. The TicketingAgents also have to wait when the printing office is busy adding tickets. Some of these wait times just feel unnecessary. Why would a customer at one TicketingAgent have to wait for a customer at another TicketingAgent? This is clearly a case of implementation shortcomings poisoning domain model sanctity.

OPTION 2: MESSAGE PASSING APPROACH

Our strategy for making things simpler (with Akka) consists of avoiding the root of all the ills we just discussed: shared state. Here are the three changes we need to make:

1. The Events and the Tickets will only be passed as immutable messages between the TicketingAgent and the printing office.
2. Requests of the agents and printing office will be enqueued *asynchronously*, without the thread waiting for method completion or even a receipt.
3. The agents and the printing office, rather than holding references to each other, contain *addresses* where they can send messages for each other. Those messages will be temporarily stored in a *mailbox*, to be processed later, one at a time, in the order they arrived. (Don't worry, how this 'toolkit' makes this possible, that will be explained later.)

Of course, a new requirement springs from this reconfiguration of collaborators: TicketingAgents will need to have a way to get more Tickets when they have sold the ones they have. While that is an additional piece, once we have done it by simply passing immutable messages, we will not have to worry about whether our application can withstand drastic (and rapid) increases in demand. Figure 1.5 shows the TicketingAgent sequence.

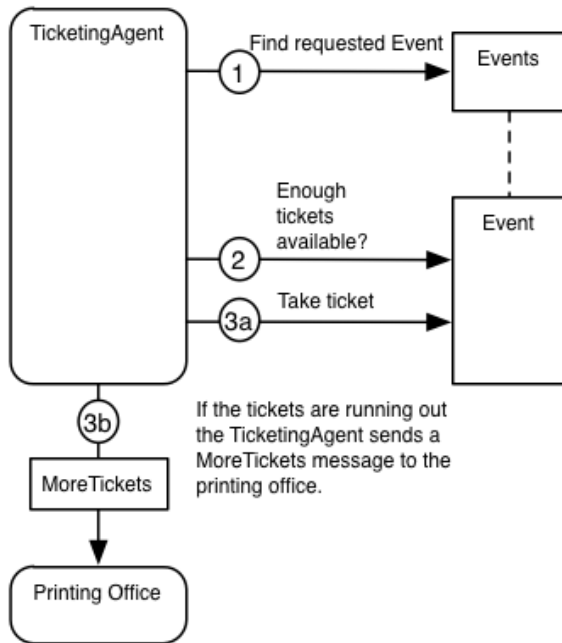


Figure 1.5 TicketingAgent sequence

So how do the TicketingAgent get tickets? Well, the printing office will send Event messages to the agents. Every Event message will contain a batch of Tickets and some stats for the Event. There are quite a few ways to send tickets to all the TicketingAgents, in this case we have chosen to let them relay the messages to each other, which makes the system more peer-based, and given some simple rules on when to pass, is a means of scaling we will go into in more detail later. The TicketingAgents know each others' addresses and send the event message along to each other as shown in figure 1.6 . (We will see chains many times in the Actor world, ranging from this simple means of propagating working state, up to a distributed, actor-based version of the Chain of Responsibility Design Pattern.)

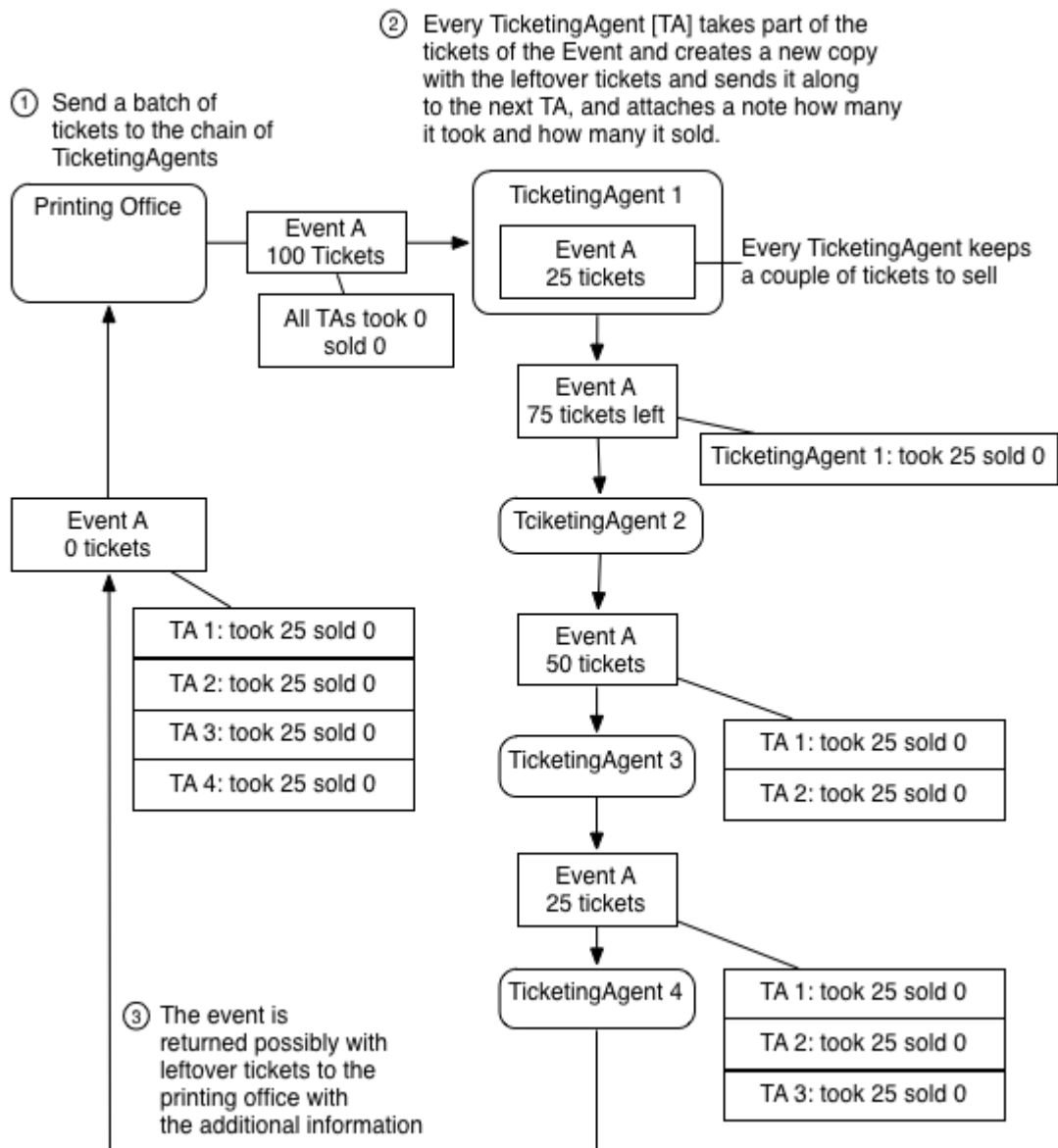


Figure 1.6 Distributing tickets

The printing office will send an Event message with the next batch of tickets to the chain of TicketingAgents when needed. The Event message can indicate that the Event is sold out or out of print. If a TicketingAgent in the chain has leftover tickets, we've chosen to relay the tickets to other Agents rather than just return them to the available pool. Another option would be to expire the tickets; each TicketingAgent is given the ticket, but once a certain amount of time has passed, what they have is no longer valid and can no longer be sold.

The message passing approach has a couple of benefits. Most importantly, there are no more locks to manage! Any number of TicketingAgents can sell tickets in parallel without locking, since they have their own tickets to sell. A

TicketingAgent does not have to wait for a response from the printing office when it sends a 'More Tickets' message, whenever the printing office is ready it will send a response. A buffering scheme could also be implemented, where the message for new tickets is sent once the on hand level has dropped below a certain number (with the idea that they will arrive before the cache has been emptied).

Even if the printing office were to crash, all agents can continue to sell tickets until they run out. If a TicketingAgent sends a message to a crashed printing office, the TicketingAgent does not wait, or crash. Since the agent does not know the printing office directly, a message to the printing office address could be handled by some other object, if the message system were to swap out the crashed printing office for a new one, and give the new printing office the old address (more on this later).

This is a typical implementation in Akka. The TicketingAgent and the printing office in the *message* passing example can be implemented with Akka Actors. Actors do not share state, can only communicate through immutable messages and do not talk to each other directly but through actor references, similar to the addresses we talked about. This approach satisfies the three things we wanted to change. So why is this simpler than the shared mutable state approach?

- We don't need to manage locks. We don't have to think about how to protect the shared data. Inside an actor we're safe.
- We are more protected from deadlocks caused by out of order access by multiple threads, that cause the system to wait forever, or other problems like race conditions and thread starvation. Use of Akka precludes most of these problems, relieving us of the burden.
- Performance tuning a shared mutable state solution is hard work and error prone and verification through tests is nearly impossible.

SIDEBAR The Actor Model is not new

The Actor Model is not new at all and has actually been around for quite a while, the idea was introduced in 1973 by Carl Hewitt, Peter Bishop, and Richard Steiger. The Erlang language and its OTP middleware libraries, developed by Ericsson around 1986, supports the Actor Model and has been used to build massively scalable systems with requirements for high availability. An example of the success of Erlang is the AXD301 switch product which achieves a reliability 99.9999999%, also known as nine nines reliability. The Actor Model implementation in Akka differs in a couple of details from the Erlang implementation, but has definitely been heavily influenced by it, and shares a lot of its concepts.

So are there no concurrency primitives like locks used *at all* in Akka? Well, of course there are, it's just that *you* don't have to deal with them *directly*. Everything still eventually runs on threads and low level concurrency primitives. Akka uses the `java.util.concurrent` library to coordinate message processing and takes great care to minimize the number of locks used to an absolute bare minimum. It uses lock free and wait free algorithms where possible, for example compare-and-swap (CAS) techniques, which are beyond the scope of this book. And because nothing can be shared between actors, the shared locks that you would normally have between objects are not present at all. But as we will see later, we can still get into some trouble if we accidentally share state.

We also saw that in the message passing approach we needed to find a way to redistribute the tickets. We had to model the application differently, which is what you would probably expect, no such thing as a free lunch. But the advantage is that we have traded unknown, perhaps vast amounts of work in scaling and preventing disasters, for a few amendments to the interactions of the core collaborators, in this case, simply distributing the load (as tickets to be sold). Later in the book, we will see that having load accommodations in the domain layer will be beneficial when we want to scale, as we have already provided a means of spreading the work around to make use of additional resources.

There are other benefits that stem from the message passing approach that Akka uses, which we will discuss in the next sections. We have touched on them briefly already:

1. Even in this first, simple example, the message passing approach is clearly more fault tolerant, averting catastrophic failure if one component (no matter how key) fails.
2. The shared mutable state is always in one place in the example (in one JVM if it is kept entirely in memory). If you need to scale beyond this constraint, you will have to (re)distribute the data somehow. Since the message passing style uses addresses, looking ahead, you can see that if local and remote addresses were interchangeable, scaling out would be possible without code changes of any kind.

So again, we have paid a small price in terms of more explicit cooperation, but reaped a clear long-term benefit.

1.1.2 Fault Tolerance

We just touched on how quickly and easily we were granted a significant measure of fault tolerance in our example; let's take a moment to go into that a bit more here. (Fault tolerance is going to be discussed in more detail in chapter 3.) The Ticketing Example mentioned that the message passing approach allows the system to keep functioning even when part of it is waiting forever and not functioning at all. One reason for this is isolation, the actors do not talk to each other directly. An actor can never block or wait forever because it sent another actor a message. Because the message is delivered to a mailbox, the sender is immediately free to go do other work. Maybe the message will never reach its intended actor, or maybe it will never get processed, but the sending actor at least will never fail trying to send a message to an address. The same cannot be said for objects and calling methods. Once you call a method, you are all-in. Figure 1.7 shows the difference in this regard between actors and objects.

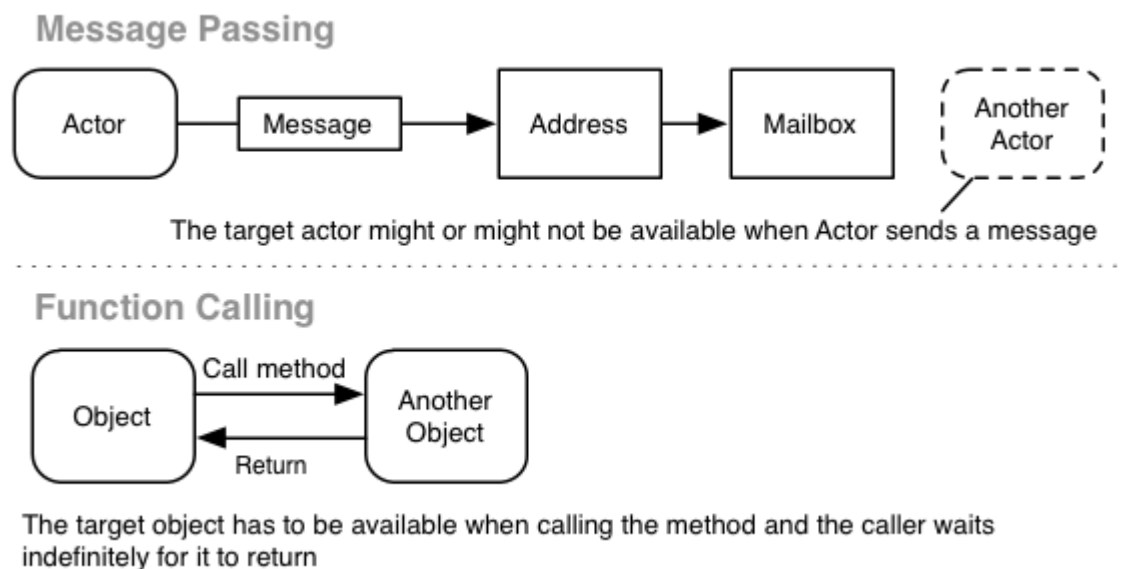


Figure 1.7 Message Passing vs. Function Calling

This isolation also offers another opportunity. What if an actor crashes, and is replaced by another actor, reachable at the same address? Any message sent to the address will just reach the new actor, as long as the crashed actor is replaced properly. The fault is contained in the misbehaving actor and any consequent message could be handled by a new actor. (For this to work, the erroneous actor should not be allowed to continue to process messages at an exception. It should just crash and burn, never to be seen again.) Where frameworks like Spring

leverage that at the container/bean level, Akka brings substitutability and replaceability to runtime (this is the Actor concept of 'let it crash'): we anticipate that for whatever reasons, there will be cases of individual handlers failing to carry out their tasks, and we are prepared to reroute that work to avoid a catastrophic failure.

Also of note here: this is bidirectional. Even if the failure occurs up the chain, underlings can carry on until a new supervisor arrives. There is no broken chain, and most importantly, it is no longer the duty of the developer to find all the ways that chain might break catastrophically and provide specialized handling for each case (exceptions are our only tool in the shared mutable state model, and as we'll shortly see, they are bound to the current VM, and propagation up and down the callstack). A failed Actor (in this case, the PrintingOffice), can be replaced without even halting the other collaborators, as shown in figure 1.8 .

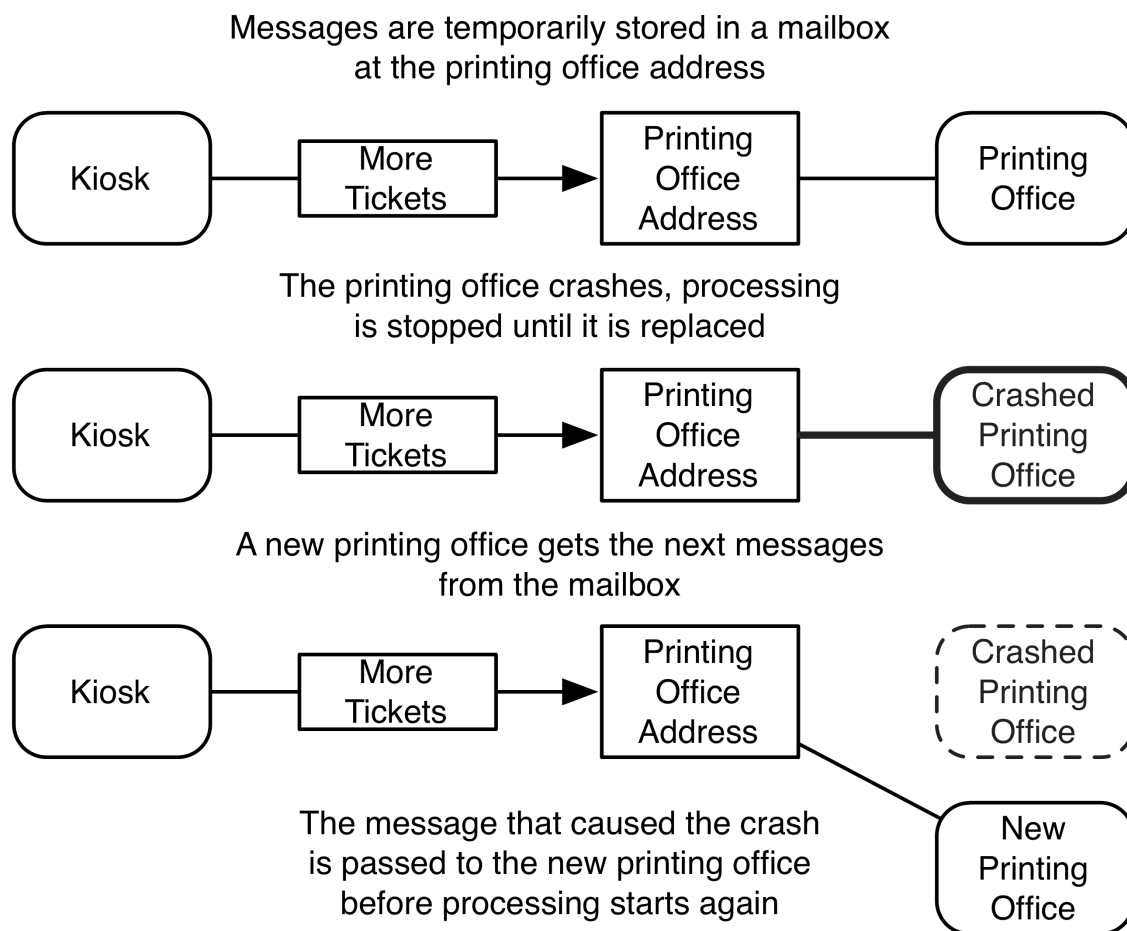


Figure 1.8 Replacing a crashed printing office

As we mentioned before, messages are always temporarily stored in a mailbox.

The message in the mailbox that was processed by the crashed printing office can be passed to the new printing office (when it's been determined that it was not handled). The agents don't notice any problem and the system just continues to sell tickets. This scenario is one example of a fault tolerance strategy that Akka provides, which is called the Restart strategy. Other strategies that can be used are Resume, Stop and Escalate. These strategies are explained in more detail in chapter 4. Akka provides a way to select strategies for specific exceptions that can occur in Actors. Since Akka controls how all messages between Actors are processed and knows all the addresses of the Actors, it can stop processing messages for an Actor that throws an exception, check which strategy should be used for the specific exception and take the required action. (This is kind of a Cool Hand Luke Universe: the only failures, are failures of communication.)

Fault tolerant does not mean that every possible fault is caught and recovered from completely. A fault tolerant system is a system that can at least contain and isolate faults in specific parts of the system, averting a full system crash. The goal is to keep the system running, as was achieved by restarting the printing office. Different faults require different corrective strategies. Some faults are solved by restarting a part of the system, other faults might not be solvable at the point of detection and may need to be handled at a higher level, as part of a larger subsystem. We'll see how Akka provides for such cases with its notion of Supervision later in the book.

As you would probably expect, replacing malfunctioning objects in a shared mutable state approach is almost impossible to do, unless you are prepared to build a framework of your own to support it. And this is not limited to malfunctioning objects, what if you just wanted to replace the behavior of a particular object? (As we will see later, Akka also provides functionality to hot swap the behavior of an Actor.) Since you do not have control over how methods are called, nor possess the ability to suspend a method and redirect it to another new object, the flexibility that is offered by the message passing approach is unmatched.

Without going into a lot of detail, let's look briefly at exception handling. Exceptions in standard, non-concurrent code are thrown up the call hierarchy. An object needs to handle the exception, or rethrow it. Whenever an exception occurs you need to stop the regular business that you are doing and fallback to error handling, to immediately continue where you left off after the error has been dealt with. Since this is quite hard, most developers prefer to just throw an error all the way up the stack, leaving it up for some type of framework to handle, aborting the

process at the first sign of an error. In Java, the situation is made murkier by the fact that there are two types of exceptions (checked and runtime), and confusion about the applicability of each has made it more likely that the default impulse is just going to be to punt and hope someone else handles it.

And that's just talking about non-concurrent exception handling. Exceptions are almost impossible to share between threads out of the box, unless you are prepared to build a lot of infrastructure to handle this. Exceptions cannot automatically be passed outside of the thread group that the thread is part of, which means you would have to find some other way to communicate exceptions amongst threads in different thread groups. In most cases if a thread encounters some kind of exception the choice is made to ignore the error and continue, or stop execution completely, which is the simplest solution. You might find some evidence in a log that a thread crashed or stopped, but communicating this back to other parts of the system is not easy. Restarting the thread and providing the state it needs to function is very hard to do manually. Things become an order of magnitude harder when these threads are distributed across several machines. Akka provides one model for handling errors, for both actors on one machine, as scaled out across many, as we will see later in chapter 3. Hopefully, you can see in the midst of these contrasted approaches to core problems, that Akka is not just a solution to concurrency issues; the fault tolerance advantages can, in many cases, be just as great a lure.

1.1.3 Scale Up and Out

In this section we are going to briefly look at how a message passing style provides some benefits for scaling up and out. In our Ticketing example, scaling up would mean getting more TicketingAgents running on our one server, scaling out would be bringing up TicketingAgents on a number of machines.

SCALE OUT

Without explicitly stating it, so far we have looked at the agents and printing office example on one machine, and one JVM. So how do we scale out to many machines?

The message passing style uses addresses to communicate, so all we need to change is how the addresses are linked to actors. If the toolkit takes care of the fact that an address can be local or remote, we can scale the solution just by configuring how the addresses are resolved. Figure 1.9 shows how the Akka toolkit solves this for us.

The toolkit provides the proxies to the remote actors
and has remote client and server components
to send messages across the network

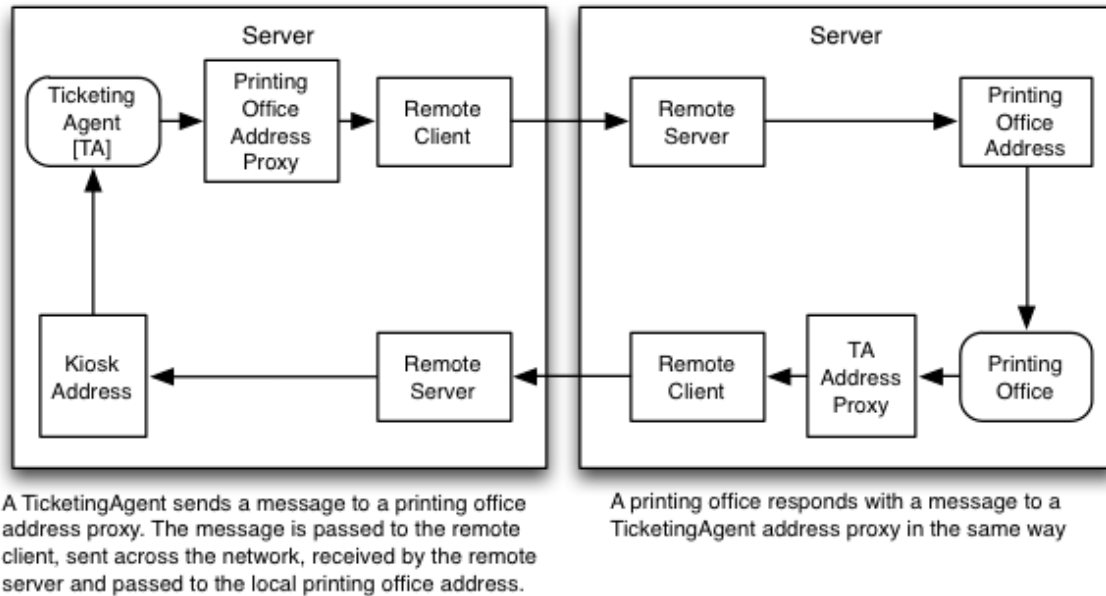


Figure 1.9 Transparent Remote Actors (through Proxies)

Akka provides what it calls *Remote Actors* (which we will discuss in chapter 5) that enable the transparency we seek. Akka passes messages for a remote actor on to a remote machine where the actor resides, and passes the result back across the network. The TicketingAgent does not know that it just talked to a remote printing office. The solution can stay almost the same as the in-memory example. The only thing that has to change is how the reference to remote actors is looked up, which can be achieved solely through configuration, as we will see later. The code stays exactly the same. Which means that we can often transition from scaling up to scaling out without having to change a single line of code.

The flexibility of resolving an address is heavily used in Akka, as we will show throughout this book. Remote actors, clustering and even the test toolkit use this flexibility.

The shared mutable state example uses locks which only work within one JVM. So how easy is it to scale out and make this example work across many machines? There are no features in the JVM that we can directly use to achieve this. The most common solution is to push all of the state out of the JVM and put it all in a database.

If we follow the database path, the code has to be changed completely. The biggest threat is that under the demands of having to exercise more control, the

data store will become preeminent, and the middle tier will collapse: the classes will just be DTOs or stateless fronts that have no real behavior. The really dark side of this is that then when more sophisticated behavior is required, the database will be asked to handle that as well, which is a disaster. Also, the transparency of our system, how easily you can trace the flow, turns instead into patches of behavior intertwined with elaborate control mechanisms. In some architectures, message queues and web services will appear to prevent this, but the result will be a model that has all the same advantages of Akka, except rendered by pulling them up into the application layer where the code will serve multiple masters: the business requirements and the communication mechanisms. Akka saves us from these painful eventualities by building on messages instead of shared mutable state, and giving us transparent remoting to allow us to scale both up and out from one codebase.

SCALE UP

So what if we want to just increase the performance on one machine and scale up? Imagine that we upgrade a machine's number of CPU cores. In the shared mutable state example we could increase the number of threads that the agents run on. But as we have seen, locks result in contention, which will mean the number of threads doing work at any one time is often less than the total number, as some will have to wait on each other to finish. Sharing as little as possible means locking as little as possible, which is the goal of the message passing approach.

Using the message passing approach, the agents and printing office can run on fewer threads and still outperform the locking version, as long as the toolkit optimizes the processing and dispatching of messages. Every thread has a stack to store runtime data. The size of the stack differs per operating system, for instance on the linux x64 platform it is normally 256kB. The stack size is one of the factors that limits the number of threads that run at the same time on a server. Around 4096 threads can fit in 1GB of memory on the linux x64 platform.

Actors run on an abstraction which is called a dispatcher. The dispatcher takes care of which threading model is used and processes the mailboxes. Similar to the way thread pools work, but with the all-important messaging scheme on top; the pool just handles task scheduling, Akka's dispatcher/mailbox combo handles that and messaging. Best part is we can map a dispatch strategy through the configuration layer, meaning we can change it without changing code. In later chapters we will see that Akka's ease of configuration makes performance tuning a breeze.

Actors are lightweight because they run on top of dispatchers, the actors are not necessarily directly proportional to the number of threads. Akka Actors take a lot less space than threads, around 2.7 million actors can fit in 1GB of memory. A big difference compared to 4096 threads, which means that you can create different types of actors more freely than you would when using threads directly. There are different types of dispatchers to choose from which can be tuned to specific needs. It's possible to share a dispatcher between actors, or use different ones for different actors. Being able to configure and tune the dispatchers and the mailboxes that are used throughout the application gives a lot of flexibility when performance tuning.

Now that we've learned the benefits of a message passing approach to concurrency, fault tolerance, and scaling, it's time to look at some specifics: the concrete components that Akka uses and how these work together to provide the message passing toolkit.

1.2 About Akka Actors and ActorSystems

Akka has a set of components for every concept that has been discussed so far; the address of an actor which provides a level of indirection, the mailbox that is used to temporarily keep messages and of course the actor itself. Our goal here is to look briefly at the supporting components that are needed and how these connect together, discuss how Akka abstracts the low-level threading machinery and how Actors run on top of this abstraction. Actors are part of an Actor System, which has the responsibility to both provide the supporting components and make it possible for actors to find each other.

We've already talked quite a bit about what an actor is on a conceptual level in the previous sections. We identified that we had to make the following changes to get to a message passing style:

1. No mutable shared data structure.
2. Immutable message passing.
3. Asynchronous message sending.

It's important to remember that when you are using a toolkit or a framework, the things that you need to know are clearly delineated from the things the toolkit is doing for you. As you can see in figure 1.10 , an actor is an object that you can send messages to: it's not your responsibility to worry about the delivery of those messages, or the receipt of them. Your code can just focus on the fact that an actor has behavior, it does something specific when certain messages are received. And

it collaborates with others, by participating in a protocol that represents the language of the messages (like a DSL in messages rather than method signatures).

Let's look at how Akka implements actors and which components compare to the concepts we've talked about so far: actors, addresses and mailboxes. Akka is implemented in Scala and provides both a Java and Scala API which can be used to build actors, throughout this book we will be showing examples in Scala, using the Scala API. In the Scala API, Akka provides an Actor trait that you can extend from to build your own actor. A TicketingAgent could be implemented as an Akka Actor, which extends from the Actor trait and keeps the state of the games and the tickets it is going to sell in an internal list. What we have called an address so far is called an ActorRef in Akka, short for actor reference. The ActorRef to the TicketingAgent would be used by the Printing Office as the address to send messages to the TicketingAgent. An ActorRef comes in many shapes in Akka, although most of these are hidden from view. As a user of the API you just work with the ActorRef class.

We will walk through a simple example, shown in figure 1.10 that illustrates what our code will have to do, and what the ActorSystem will do for us. As you can see, we get to just ask for a ref to the TicketingAgent actor, then once we have it, we simply send it a buy request (steps 1 and 5). The ActorSystem shields us from all the details about which actor is going to process our requests, where the mailbox is, whether it was delivered, etc.

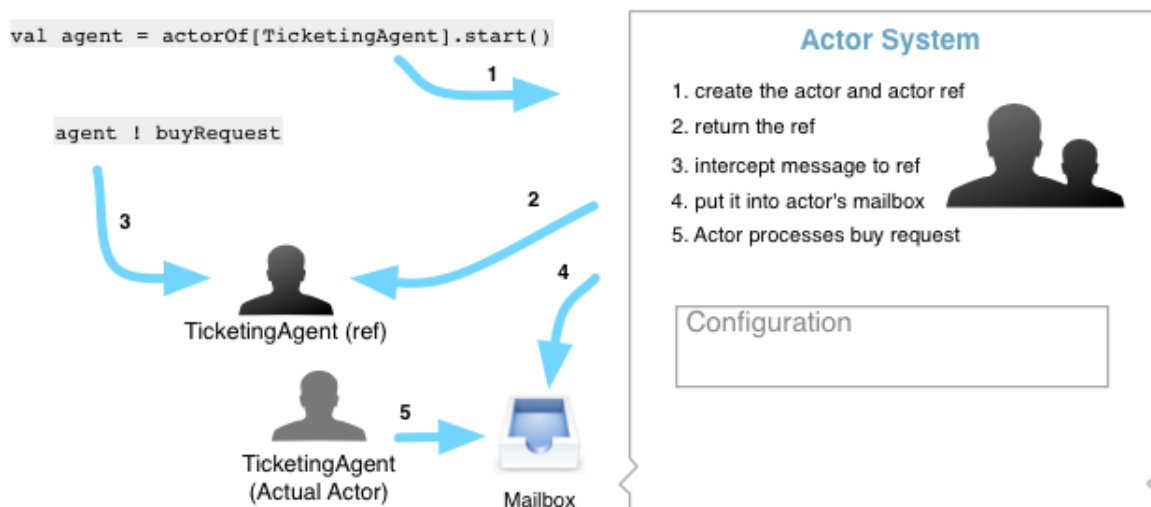


Figure 1.10 Actor Request Cycle in Action

Of course, all the rules we discussed before apply: we are not blocking and waiting for our request to be processed, we are not embedding any shared state in

the request, and the actual processing of the purchase request doesn't lock anything, or access any shared state so we have done nothing to prevent requests from being handled concurrently.

So how do you get an actor reference to an actor in the hierarchy? This is where ActorPaths come in. You could compare the hierarchy of actors to a URL path structure. Every actor has a name. This name needs to be unique per level in the hierarchy, two sibling actors cannot have the same name (if you do not provide a name Akka generates one for you, but it is a good idea to name all your actors). All actor references can be located directly by an actor path, absolute or relative, and it has to follow the URI generic syntax. An actor path is built just like a URI, starting with a scheme followed by a scheme-specific part, as shown in figure 1.11 :

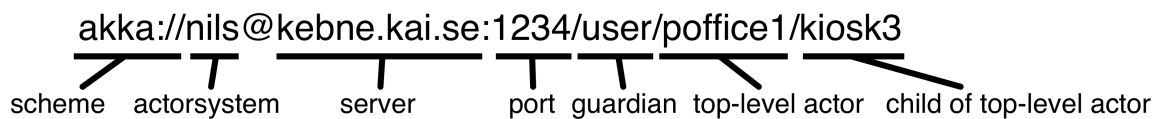


Figure 1.11 ActorPath URI syntax

In the above example, the printing office actor 'poffice1' can refer to 'TicketingAgent2' by simply using the path 'TicketingAgent2'. TicketingAgent2 can refer to its sibling by using '../TicketingAgent3'. The guardian actor is always called 'user'.

The notion of parents extends beyond simply organizational tidiness: one of the most important concepts in Akka, Supervision, is a function of the hierarchy: every actor is automatically the supervisor of its children. This means that when a child actor crashes, the parent gets to decide which strategy should be taken to correct the fault. This also makes it possible to escalate an issue up the hierarchy, where concerns that are of a more global nature, can be handled by Supervisors whose focus is on such matters.

In summary, Akka prevents an actor from being created directly by a constructor because that would circumvent the actor hierarchy; creating an actor directly by using a constructor would give direct access to the methods of the actor, which breaks the rules for a message passing style of concurrency. The actor system creates actor references, provides a generic way to locate actor references, provides the root actor to create a hierarchy of actors and connects the runtime components to the actors.

NOTE**Core Actor Operations**

Another way to look at an actor is to describe the operations that it supports. An Akka actor has *four core operations* :

1. **CREATE:** An actor can create another actor. In Akka, actors are part of an actor system, which defines the root of the actor hierarchy and creates top-level actors. Every actor can create child actors. The topology of the actors is dynamic, it depends on which actors create other actors and which addresses are used to communicate with them.
2. **SEND:** An actor can send a message to another actor. Messages are sent asynchronously, using an address to send to an Actor associated with a given Mailbox.
3. **BECOME:** The behavior of an actor can be dynamically changed. Messages are received one at a time and an actor can designate that it wants to handle next messages in a different way, basically swapping its behavior, which we will look at in later chapters.
4. **SUPERVISE:** An actor supervises and monitors its children in the actor hierarchy and manages the failures that happen. As we will see in chapter 3, this provides a clean separation between message processing and error handling.

1.3 Summary

We had two large conceptual realms to cover here: the theory and practice of Actor Systems and Akka's implementation. We went into just enough detail on the implementation side so that you are ready to use Akka (and we will shift immediately in the next chapter into getting you up and running), but we laid out some of what Akka is doing behind the scenes so you know why things are done as they are. Subsequent chapters will contain more details about what is under the covers, and how you can configure and customize it for more complex scenarios, but for now, we already know enough to see the tremendous power of Akka. In the next chapter, you will see that the cost for this power is low: you can not only get up and running very quickly, but the complexity of managing messaging and concurrency is largely shouldered by the toolkit.

The things we saw here, that we will carry forward, included:

- Message passing enables an easier road to real concurrency
- With that concurrent approach, we will be able to scale up and out
- We can scale both the request and the processing elements of our application
- Messages also unlock greater fault tolerance capabilities
- Supervision provides a means of modeling for both concurrency and fault tolerance
- Akka infuses our code with these powers in a lightweight, unobtrusive manner

Hopefully you have been convinced that this approach is better than the old one of writing the code as though there are no concurrency requirements, then retrofitting it with locks as mutable state issues arise.

In the next chapter we will get up and running with an Akka instance, building and running our first working example.

Up and Running

In this chapter

- Fetch a project template
- Build a minimal Akka App for the cloud
- Deploy to Heroku

Our goal here is to show you how quickly you can make an Akka app that not only does something nontrivial, but is built to do it to scale, even in its easiest, early incarnations. We will clone a project from github.com that contains our example, then we'll walk through the essentials that you will need to know to get started building Akka apps. First we will look at the dependencies that you need for a minimal app, using TypeSafe's Simple Build Tool (SBT) to create a single jar file that can be used to run the app. Throughout the book, we're going to build a ticket selling app and this chapter will show the first iteration. In this first iteration we will build a very minimal set of REST services. We'll keep it as simple as possible to focus on essential Akka features. Finally we will show you how easy it is to deploy this App to the cloud and get working on popular cloud provider, Heroku. What will be most remarkable is how quickly we will get to this point!

One of the most exciting things about Akka is how easy it is to get up and running, and how flexible it is given its small footprint runtime, as you'll soon see. We will ignore some of the infrastructure details (like the REST implementation), chapter 8 will go into more detail on how to use Spray, but you will leave this chapter with enough information to build serious REST interfaces of all types, and we'll see in the next chapter, how we can combine this with TDD (Test-Driven Development).

2.1 Clone, Build and Test Interface

To make things a bit easier we've published the source code for the App on github.com. The first thing you have to do is clone the repo to a directory of your choice:

```
git clone https://github.com/RayRoestenburg/akka-in-action.git
```

This will create a directory named *akka-in-action* that contains a directory *chapter2* which contains the example project for this chapter. We're expecting that you are already familiar with git and github amongst other tools. Please checkout the *Appendix* for how to install the tools that you will need to follow along. We will be using sbt, git, the heroku toolbelt and httpie in this chapter. The appendix also has details on how to setup popular IDE's like IntelliJ IDEA and Eclipse in case you would like to set that up.

Let's look at the structure of the project. SBT follows a project structure similar to Maven with which you are probably familiar. The major difference is that SBT allows for the use of Scala in the build files, and it has an interpreter. This makes it quite a bit more powerful. For more information on SBT, see the Manning title *SBT in Action*. Inside the *chapter2* directory all the code for the server can be found in `src/main/scala`, configuration files and other resources in `src/main/resources`, tests and test resources respectively in `src/test/scala` and `src/test/resources`. The project should build right out of the box. Run the following command inside the *chapter2* directory and keep fingers crossed:

```
sbt assembly
```

This should show sbt booting up getting all needed dependencies, running all the tests and finally building one fat jar into `target/scala-2.10/goticks-server.jar`. You could run the server by simply running:

```
java -jar target/scala-2.10/goticks-server.jar
```

If you got curious and ran this you should see something similar to the following output

```
Slf4jEventHandler started
akka://com-goticks-Main/user/io-bridge started
akka://com-goticks-Main/user/http-server started on /0.0.0.0:5000
```

Now that we have verified that the project builds correctly it's time to talk about what it does. In the next section we will start with the build file and then look at the resources and of course the actual code for the services.

2.1.1 Build with SBT

Let's first look at the build file. We're using the simple SBT DSL for build files in this chapter because it gives us all we need right now. It's pretty compact because the only thing we are employing that is not part of the native Akka stack is Spray. Of course, as we go forward in the book, we will be back to add more dependencies to this file, perhaps configure the build to support different target environments (e.g. Develop, Test, Product), but you can see that for your future projects you will be able to get going quite quickly, and without the aid of a template, or by cutting and pasting large build files from other projects. If you have not worked with the SBT DSL before it is important to note that you need to put an empty line between lines in the file (this is the price we pay for not telling Scala where each expression ends). The build file is located directly under the chapter2 directory in a file called build.sbt. The build file starts with two imports:

```
import AssemblyKeys._
import com.typesafe.startscript.StartScriptPlugin
```

These imports are needed for two SBT plugins. The first makes it possible to make the "one jar" file we saw before, goticks-server.jar. This jar contains the source code for the goticks App and all dependencies it needs. The second import is required for deploying to Heroku which we will look at in a later section. Next the build file specifies some project details; the name, version and organization of our app and the version of scala to use:

```
name := "goticks"
```

```

version := "0.1-SNAPSHOT"

organization := "com.goticks"

scalaVersion := "2.10.0"

```

The following lines specify which maven repositories to use for downloading the dependencies:

```

resolvers ++=
Seq("repo" at "http://repo.typesafe.com/typesafe/releases/",
  "Spray Repository" at "http://repo.spray.io",
  "Spray Nightlies" at "http://nightlies.spray.io/")

```

The `resolvers` is just a list of maven repositories that will be resolved by SBT. As you can see, we have added the Typesafe and Spray repositories (the build tool will download all the libraries from them). Next are the dependencies for the project:

```

libraryDependencies ++= {
  val akkaVersion      = "2.1.2"      ❶
  val sprayVersion     = "1.1-20130123"
  Seq(
    "com.typesafe.akka" %% "akka-actor"      % akkaVersion,      ❷
    "io.spray"          % "spray-can"        % sprayVersion,
    "io.spray"          % "spray-routing"    % sprayVersion,
    "io.spray"          %% "spray-json"      % "1.2.3",
    "com.typesafe.akka" %% "akka-slf4j"     % akkaVersion,
    "ch.qos.logback"   % "logback-classic" % "1.0.10",
    "com.typesafe.akka" %% "akka-testkit"   % akkaVersion % "test",
    "org.scalatest"    %% "scalatest"      % "1.9.1"           % "test"
  )
}

```

- ❶ The version of Akka we are using
- ❷ The akka actor module dependency

The `libraryDependencies` is just a list of dependencies that SBT will get from the specified maven repositories. Every dependency points to a maven artifact in the format `organization % module % version` (the `%%` is for automatically using

the right scala version of the library). The most important dependency here is the *akka-actor* module. The build file concludes with some settings for the plugins which we will look at in a later section. Now that we have our build file setup we can compile the code, run the tests and build the jar file. Run the below command in the chapter2 directory:

```
sbt clean compile test
```

The above command cleans out the target directory, compiles all the code and runs all the tests in the project. If any dependencies still need to be downloaded SBT will do that automatically. Now that we have the build file in place, lets take a closer look at what we are trying to achieve with this example in the next section.

2.1.2 Fast forward to the *GoTicks.com* REST server

Our ticket selling service which will allow customers to buy tickets to all sorts of events, concerts, sports games and the like. Let's say we're part of a startup called *GoTicks.com* and in this very first iteration we have been assigned to build the backend REST server for the first version of the service (it's part of a minimum viable product). Right now we want customers to get a numbered ticket to a show. Once all the tickets are sold for an event the server should respond with a 404 (Not Found) HTTP status code. The first thing we'll want to implement in the REST interface will have to be the addition of a new event (since all other services will require the presence of an event in the system). A new event only contains the name of the event, say "RHCP" for the Red Hot Chili Peppers and the total number of tickets we can sell for the given venue.

The requirements for the REST interface are shown below:

Table 2.1 REST API

Description	HTTP Method	URL	Body	Response example
Create an event with a number of tickets.	PUT	/events	{ event: "RHCP", nrOfTickets : 250}	HTTP 200 OK
Get an overview of all events and the number of tickets available.	GET	/events		[{ event : "RHCP", nrOfTickets : 249}, { event : "Radiohead", nrOfTickets : 130},]
Buy a ticket	GET	/ticket/:eventName		{ event: "RHCP", nr: 1 } or HTTP 404

Lets build the app and run it inside SBT. Go to the chapter2 directory and execute the following command:

```
sbt run
```

This automatically runs the goticks app inside SBT. You should see something along the lines of this output:

```
akka://com-goticks-Main/user/http-server started on /0.0.0.0:5000
```

If you get an error make sure that you are not already running the server in another console, or that some other process is already using port 5000. Lets see if everything works by using `httpie` (a human readable HTTP command line tool that makes it very simple to send HTTP requests). It has support for JSON and handles the required housekeeping in headers, among other things. See the Appendix for more information on how to get it setup.) First lets see if we can create an event with a number of tickets:

```
http PUT localhost:5000/events event=RHCP nrOfTickets:=10
```

The above command creates a JSON request entity with an event and a nrOfTickets; { event: "RHCP", nrOfTickets : 10}. It creates a PUT request to our running server and prints out the response. You should see a response that looks like this:

```
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: text/plain
Date: Tue, 16 Apr 2013 11:22:48 GMT
Server: GoTicks.com REST API
OK
```

The event is now created. Lets create another one:

```
http PUT localhost:5000/events event=DjMadlib nrOfTickets:=15
```

Now lets try out the GET request:

```
http GET localhost:5000/events
```

Which should give the below response:

```
HTTP/1.1 200 OK
Content-Length: 92
Content-Type: application/json; charset=UTF-8
Date: Tue, 16 Apr 2013 12:39:10 GMT
Server: GoTicks.com REST API
[
  {
    "event": "DjMadlib",
    "nrOfTickets": 15
  },
  {
    "event": "RHCP",
    "nrOfTickets": 10
  }
]
```

Notice, we are seeing both events, and of course, all the tickets are still available. Now lets see if we can buy a ticket for the RHCP event:

```
http GET localhost:5000/ticket/RHCP
```

The app issues a ticket and sends it to us, here is our Ticket (again, as JSON):

```
HTTP/1.1 200 OK
Content-Length: 32
Content-Type: application/json; charset=UTF-8
Date: Tue, 16 Apr 2013 12:40:00 GMT
Server: GoTicks.com REST API
{
  "event": "RHCP",
  "nr": 1
}
```

If we GET on /events again you should see the following response:

```
HTTP/1.1 200 OK
Content-Length: 91
Content-Type: application/json; charset=UTF-8
Date: Tue, 16 Apr 2013 12:41:42 GMT
Server: GoTicks.com REST API
[
  {
    "event": "DjMadlib",
    "nrOfTickets": 15
  },
  {
    "event": "RHCP",
    "nrOfTickets": 9
  }
]
```

As expected there are now only 9 tickets left. You should get a 404 after repeating the /ticket/RHCP request 11 times:

```
HTTP/1.1 404 Not Found
Content-Length: 83
Content-Type: text/plain
Date: Tue, 16 Apr 2013 12:42:57 GMT
```

```
Server: GoTicks.com REST API
The requested resource could not be found
but may be available again in the future.
```

That concludes all the API calls in the REST interface. Clearly, at this point, the application supports the basic Event CRUD cycle from creation of the actual Event, through the sale of all the tickets until they are sold out. This is not comprehensive of course, for instance, we are not accounting for Events that will not sell out, but whose tickets will need to become unavailable once the actual Event has started. Now let's look at the details of how we're going to get to this result in the next section.

2.2 Explore the Actors in the App

In this section we're going to look at how the App is built. You can try and build the Actors yourself or just follow along from the source code on github.com. As you now know, actors can perform four operations; create, send/receive, become and supervise. In this example we'll only touch on the first two operations. First, we'll take a look at the overall structure: how operations will be carried out by the various collaborators (actors) to provide the core functionality: creating events, issuing tickets, finishing events.

2.2.1 Structure of the App

The App consists of three actor classes in total. The first thing we have to do is create an actor system that will contain all the actors. After that the actors can create each other. The below figure shows the sequence:

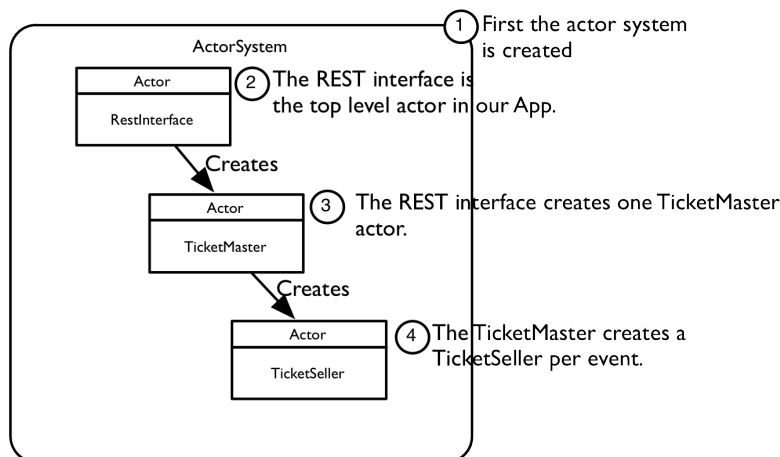


Figure 2.1 Actor Creation Sequence Triggered by REST Request

The REST Interface Actor will handle the HTTP requests. It is

basically an adapter for HTTP: it takes care of converting from and to JSON and provides the required HTTP response. Even in this simplest example, we can see how the fulfillment of a request spawns a number of collaborators, each with specific responsibilities. The TicketSeller eventually keeps track of the tickets for one particular event and sells the tickets. The figure shows how a request for creating an Event flows through the actor system (this was the first service we showed above):

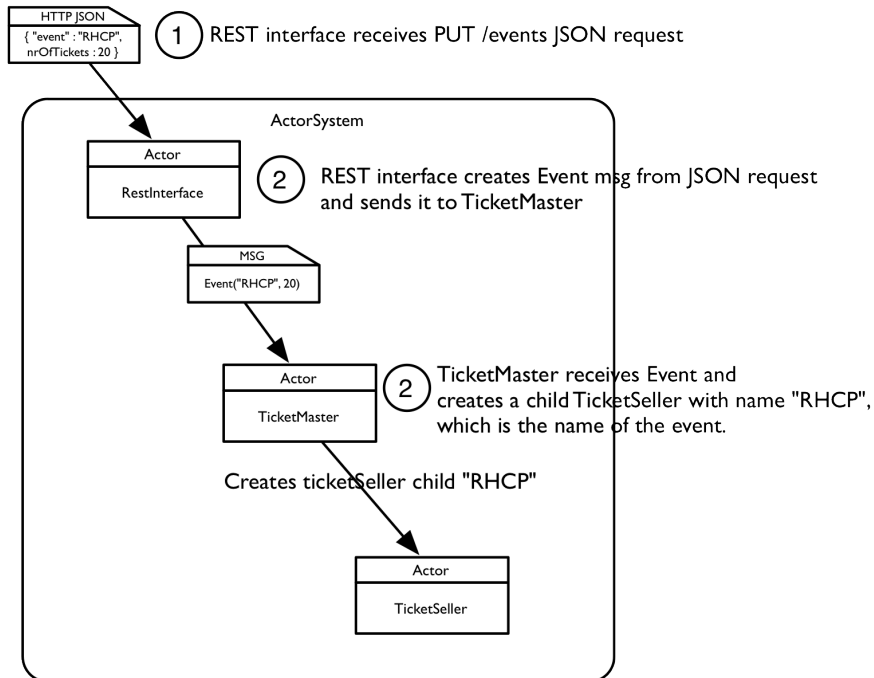


Figure 2.2 Create an Event from the received JSON request

The second service we discussed above was the ability for a Customer to purchase a ticket (now that we have an Event). The figure shows what should happen when such a ticket purchase request is received (as JSON):

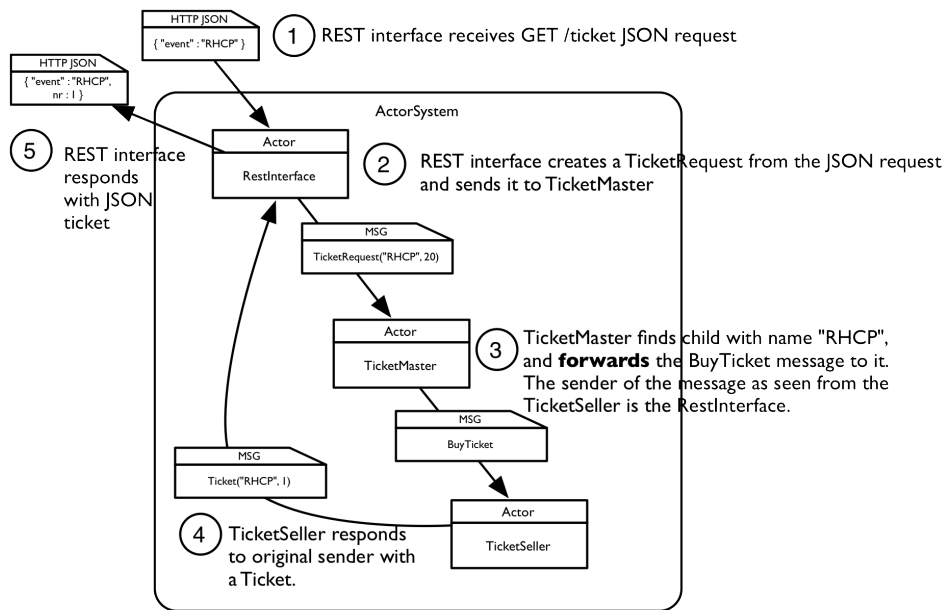


Figure 2.3 Buy a Ticket

This is the most important flow in this tiny application: our business is essentially about connecting Customers to available Tickets. And, as we discussed, this is also where we will most likely come under pressure later, when we are actually using this in the real world.

Let's step back and start looking at the code as a whole, first up the Main class, which starts everything up. The Main object is a simple Scala App that you can run just like any other Scala App. It's similar to a Java class with a main method. Below is the listing of the Main class:

Listing 2.1 Application Bootstrap: Main Class Starting Spray (and its ActorSystem)

```

package com.goticks
import spray.can.server.SprayCanHttpServerApp
import akka.actor.Props
import com.typesafe.config.ConfigFactory

object Main extends App with SprayCanHttpServerApp {
  val config = ConfigFactory.load()
  val host = config.getString("http.host")
  val port = config.getInt("http.port")
  val api = system.actorOf(
    Props(new RestInterface()),
    "httpInterface"
  )
  newHttpServer(api) ! Bind(interface = host, port = port)
}

```

- ① A Spray-can App
- ② Load a configuration file
- ③ Get some HTTP configuration values
- ④ Create the top level REST Interface actor
- ⑤ Start a Spray HTTP server

Main extends App with the SprayCanHttpServerApp trait. Simply by adding the trait, we can then just pull out our config parameters for the server settings and inject them into the call to create the RestInterface actor, which will be given the responsibility to handle requests that come to this instance. You really don't need to know the details of how Spray is making this possible (we will cover them in detail later). The last line in the Main class actually starts the server and binds it to a host and port. The App uses the following messages internally between the actors: REST Interface Message Classes

```

case class Event(event:String, nrOfTickets:Int)
case object GetEvents
case class Events(events>List[Event])
case object EventCreated

```

```

case class TicketRequest(event:String)      5
case object SoldOut                          6
case class Tickets(tickets>List[Ticket])   7
case object BuyTicket                        8
case class Ticket(event:String, nr:Int)     9

```

- ① Message to create an event
- ② Message for requesting the state of all events
- ③ Response message that contains current status of all events
- ④ Signal event to indicate an event was created
- ⑤ Request for a ticket for a particular event
- ⑥ Signal event that the event is sold out
- ⑦ New tickets for an Event, created by BoxOffice
- ⑧ Message to buy a ticket from the TicketSeller
- ⑨ The numbered ticket to an event

As is typical of REST apps, we have an interface that revolves around the lifecycles of the core entities: Events and Tickets. Remember, Akka is going to get these parts to go together with immutable messages, so the Actors have to be designed to get all the information they need, and produce all that is needed if they enlist any collaborators. This lends itself well to REST. In the next sections we're going to look at the Actors in more detail. We'll start from the TicketSeller and work our way up.

2.2.2 The Actor that handles the sale: TicketSeller

The TicketSeller is created by the BoxOffice and just simply keeps a list of tickets. Every time a ticket is requested it takes the next one from the head of the list. The below listing shows the code for the TicketSeller:

Listing 2.3 TicketSeller Implementation

```

package com.goticks

import akka.actor.{PoisonPill, Actor}

class TicketSeller extends Actor {
  import TicketProtocol._

  var tickets = Vector[Ticket]() ❶

  def receive = {

    case GetEvents => sender ! tickets.size ❷

    case Tickets(newTickets) =>
      tickets = tickets ++ newTickets ❸

    case BuyTicket =>
      if (tickets.isEmpty) { ❹
        sender ! SoldOut
        self ! PoisonPill
      }

      tickets.headOption.foreach { ticket =>
        tickets = tickets.tail
        sender ! ticket
      }
  }
}

```

- ❶ The list of tickets
- ❷ Return the size of the list when GetEvents is received
- ❸ add the new tickets to the existing list of tickets when Tickets message is received
- ❹ Report SoldOut and kill self if there are no more tickets. Otherwise get the head ticket and leave the rest in the tickets list.

The TicketSeller takes a so called `PoisonPill` after it has reported it is sold out. This nicely cleans up the actor (deletes it) when it has no more work to do. In the next section we're going to look at the `BoxOffice` actor.

2.2.3 BoxOffice

The `BoxOffice` needs to create `TicketSeller` children for every event and delegates the selling to them. The below listing shows how the `BoxOffice` responds to an `Event` message:

Listing 2.4 BoxOffice Creates TicketSellers

```

case Event(name, nrOfTickets) =>

  if(context.child(name).isEmpty) {
    val ticketSeller = context.actorOf(Props[TicketSeller], name)

    val tickets = Tickets((1 to nrOfTickets).map{
      nr=> Ticket(name, nr)}.toList)
    ticketSeller ! tickets
  }

  sender ! EventCreated

```

1 If TicketSellers have not been created already

The `BoxOffice` creates `TicketSellers` for each event. Notice that it uses its `context` instead of the actor system to create the actor; Actors created with the context of another Actor are its children and subject to the parent Actor's supervision (much more about that in subsequent chapters). It builds up a list of numbered tickets for the event and sends these tickets to the `TicketSeller`. It also responds to the sender of the `Event` message that the Event has been created (the sender here is always the `RestInterface` actor). The below listing shows how the `BoxOffice` responds to the `TicketRequest`:

```

case TicketRequest(name) =>

  context.child(name) match {
    case Some(ticketSeller) => ticketSeller.forward(BuyTicket)
    case None                => sender ! SoldOut
  }

```

The `BoxOffice` tries to find a child `TicketSeller` for the event and if it finds it it forwards a `BuyTicket` message to the `ticketSeller`. If there is no child for the event it sends a `SoldOut` message back to the sender (The `RestInterface`). The next message is a bit more involved and will get you extra credit if you get it the first time. We're going to ask all `TicketSellers` for the number of tickets they have left and combine all the results into a list of events. This gets interesting because ask is an asynchronous operation and at the same time we don't want to wait and block the `BoxOffice` from handling other requests.

The below code uses a concept called `Futures` which will be explained further

in a chapter 7 so if you feel like skipping it now that's fine. If you're up for a challenge though lets look at the code below!

Listing 2.5 Using Futures

```
import akka.pattern.ask

val capturedSender = sender

def askEvent(ticketSeller:ActorRef): Future[Event] = { ❶

    val futureInt = ticketSeller.ask(GetEvents).mapTo[Int] ❷

    futureInt.map { nrOfTickets => ❸
        Event(ticketSeller.actorRef.path.name, nrOfTickets)
    }
}

val futures = context.children.map { ticketSeller => ❹
    askEvent(ticketSeller)
}

Future.sequence(futures).map { events =>
    capturedSender ! Events(events.toList)
}
```

- ❶ A local method definition for asking GetEvents to a TicketSeller.
- ❷ Ask for the number of tickets that are left without waiting. The futureInt will at some point get the value
- ❸ Transform the future value from an Int to an Event.
- ❹ Ask all the children how many tickets they have left for the event.

Right now we will skim over this example and just look at the concepts. What is happening here is that an ask method returns immediately with a *Future* to the response. A Future is a value that is going to be available at some point in the future (hence the name). Instead of waiting for the response value (the number of tickets left) we get a future reference (which you could read as 'use for future reference'). We never read the value directly but instead we define what should happen once the value becomes available. We can even combine a list of Future values into one list of values and describe what should happen with this list once all of the asynchronous operations complete.

The code finally sends an Events message back to the sender once all responses have been handled.

Don't worry if this is not immediately clear, we have a whole chapter devoted to this subject. We're just trying to get you curious about this awesome feature, check out chapter 7 on Futures if you can't wait to find out how these non-blocking asynchronous operations work. That concludes the salient details of the BoxOffice. We have one actor left in the App which will be handled in the next section; the REST interface.

2.2.4 REST Interface

The REST interface uses the Spray routing DSL which will be covered in detail in chapter 9. Services interfaces, as they grow, need more sophisticated routing of requests. Since we are really just creating an Event and then selling the Tickets to it, our routing requirements are few at this point. Let's look at the details of doing simple request routing in the below listings. First the REST interface needs to handle an Event request:

Listing 2.6 Creation of the BoxOffice Actor

```
val BoxOffice = context.actorOf(Props[BoxOffice])
```

The REST Interface creates a BoxOffice child actor when it is constructed. It also creates a temporary Responder actor which stays around for the lifetime of the HTTP request:

Listing 2.7 Responder being created

```
def createResponder(requestContext: RequestContext) = {
  context.actorOf(Props(new Responder(requestContext, BoxOffice)))
}
```

This responder sends messages to the BoxOffice and handles the responses that are sent back from the TicketSeller and BoxOffice actors. The below code shows a snippet of the DSL that is used to handle HTTP requests:

Listing 2.8 Spray Route Definition (DSL)

```

path("ticket") {
  get {
    entity(as[TicketRequest]) { ticketRequest => requestContext =>
      val responder = createResponder(requestContext)
      BoxOffice.ask(ticketRequest).pipeTo(responder)
    }
  }
}

```

The request entity is unmarshalled into a `TicketRequest` object, freeing us from having to write code to specifically adapt messages into objects. The `BoxOffice` is asked for the `ticketRequest`. The response is piped to the responder actor using the *pipe* pattern. The Responder completes the HTTP request when it receives the `Ticket` or `SoldOut` response from the `TicketSeller`:

Listing 2.9 Responder Handles Request Completion: the Response

```

class Responder(requestContext:RequestContext,
                BoxOffice:ActorRef)
  extends Actor with ActorLogging {
  import TicketProtocol._
  import spray.httpx.SprayJsonSupport._

  def receive = {

    case ticket:Ticket =>
      requestContext.complete(StatusCodes.OK, ticket)
      self ! PoisonPill

    case EventCreated =>
      requestContext.complete(StatusCodes.OK)
      self ! PoisonPill

    case SoldOut =>
      requestContext.complete(StatusCodes.NotFound)
      self ! PoisonPill

    case Events(events) =>
      requestContext.complete(StatusCodes.OK, events)
      self ! PoisonPill

  }
}

```

As you can see the responder kills itself after it has completed the request. It's only around for as long as the HTTP request is being processed. The messages are automatically converted back to JSON. You can find the details of how this is done in the chapter on Spray. That concludes all the actors in the first iteration of the GoTicks.com application. If you followed along or even tried to rebuild this yourself, congratulations! You've just seen how to build your first fully asynchronous akka actor App with a fully functional REST interface. While the app itself is rather trivial, we have already made it so that the processing is fully concurrent, and the actual selling of the Tickets is both scalable (because it's already concurrent) and fault tolerant (we will see much more on that). This example also showed how we can do asynchronous processing within the synchronous request/response paradigm of the HTTP world. We hope that you've found that it takes only a few lines of code to build this app. Compare that to a more traditional toolkit or framework and I'm sure that you are pleasantly surprised to see how little code was needed. For a little cherry on the top, we're going to show you what we need to do to deploy this minimal App to the cloud. We'll get this app running on Heroku.com in the next section.

2.3 Into the Cloud

Heroku.com is a popular cloud provider which has support for Scala applications, and free instances that you can play with. In this section we're going to show you how easy it is to get the GoTicks.com App up and running on Heroku. We're expecting that you have already installed the Heroku toolbelt. If not please refer to the Appendix for how to install it. You will also need to sign up for an account on heroku.com. Visit their site the signup speaks for itself. In the next section we're first going to create an app on heroku.com. After that we will deploy it and run it.

2.3.1 Create the App on Heroku

First login to your heroku account:

```
heroku login
```

Next create a new heroku app that will host our GoTicks.com App. Execute the following command in the chapter2 directory:

```
heroku create
```

You should see something like the following response:

```
Creating damp-bayou-9575... done, stack is cedar
http://damp-bayou-9575.herokuapp.com/
| git@heroku.com:damp-bayou-9575.git
```

We need to add a couple of things to our project so Heroku understands how to build our code. First the `project/build.properties` file:

```
sbt.version=0.12.2
```

This file needs to specify which version of SBT you are using. Let's also look at the `project/plugins.sbt` file:

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.8.7")

addSbtPlugin(
  "com.typesafe.startscript" % "xsbt-start-script-plugin" % "0.5.3"
)
```

This file registers all the plugins we have been using so far. The `startscript` plugin creates a script for running our App on Heroku. We also need a so-called *Procfile* right under the `chapter2` directory which tells heroku that our App should be run on a *Web Dyno*, which is one of the types of processes heroku runs on its virtual Dyno Manifold. The ProcFile is shown below:

```
web: target/start com.goticks.Main
```

It specifies that Heroku should run the script that the `start-script-plugin` has built with the `Main` App class as argument. Let's first test to see if everything runs locally. Execute the following command:

```
sbt clean compile stage
```

This is the same command that Heroku will run on the cloud later on. It builds all the code and creates the target/start script. You can now run the ProcFile locally by using Heroku's *foreman* tool:

```
foreman start
```

This should show something like the following on the command line:

```
01:15:05 web.1 | started with pid 90796
01:15:07 web.1 | Slf4jEventHandler started
01:15:10 web.1 | akka://com-goticks-Main/user/io-bridge started
01:15:11 web.1 | akka://com-goticks-Main/user/http-server started
```

This is all that's required to prepare an application for deployment on Heroku. It lets us go through the whole cycle locally so that we are working at maximum speed while getting our first deploy done. Once we actually deploy to Heroku, you'll see that all subsequent pushes to the cloud instances are accomplished directly through git by simply pushing the desired version of the source to our remote instance. The deployment of the app to the Heroku cloud instance is described in the next section.

2.3.2 Deploy and Run on Heroku

We've just verified that we could locally run the App with foreman. We created a new app on heroku with `heroku create`. This command also added a *git remote* with the name `heroku` to the git configuration. All we have to do now is make sure all changes are committed locally to the git repository. After that push the code to heroku with the following command:

```
git push heroku master
```

The above assumes that you committed any changes to your master branch. Heroku hooks into the git push process and identifies the code as a Scala App. It

downloads all dependencies on the cloud, compiles the code and starts the application. Finally you should see something like the below output:

```

----> Scala app detected
-----> Installing OpenJDK 1.6...
.... // resolving downloads, downloading dependencies
....
-----> Compiled slug size is 43.1MB
-----> Launching... done, v1
http://damp-bayou-9575.herokuapp.com deployed to Heroku

To git@heroku.com:damp-bayou-9575.git
* [new branch]      master -> master

```

The above shows the console on creation of the App: note that Heroku figured out that our app is a Scala app, so it installed the OpenJDK, then compiled and launched the source in the instance. The App is now deployed and started on Heroku. You can now use `httpie` again to test the App on heroku:

```

http PUT damp-bayou-9575.herokuapp.com/events \
event="RHCP" nrOfTickets:=10
http GET damp-bayou-9575.herokuapp.com/ticket/RHCP

```

The above commands should result in similar response as before. Congratulations, you just deployed your first Akka App to Heroku! With that we conclude this first iteration of the GoTicks.com app. Of course, now that the app is deployed on Heroku, you can call it from anywhere.

2.4 Summary

In this chapter you have seen how little is necessary to build a fully functional REST service out of Actors. All interactions were asynchronous. The service performed as expected when we tested it with the `httpie` commandline tool.

We even deployed our app (via Heroku.com) into the Cloud! We're hoping you got excited about what a quick out of the box experience Akka offers. Of course the GoTicks.com App is not ready for production yet. There is no persistent storage for tickets. We've deployed to Heroku, but web dynos can be replaced at any time so purely storing the tickets in memory will not work in real life. The App is scaled up but has not scaled out yet to multiple nodes.

But we promise to fix that in later chapters where we will gradually get closer to a real world system. In the next chapter, we're going to look at how to test actor systems.

Test Driven Development with Actors



In this chapter

- Testing Actors
- Some Rules to Follow
- Speed of Development

It's amusing to think back to when TDD first appeared on the scene, the primary objection was that tests take too long, and thus hold up development. Though you rarely hear that today, there is a vast difference in the testing load both between different stacks, and through different phases (e.g. unit vs. integration tests). Everyone has a rapid, fluid experience on the unit side, when testing is confined to a single component. Tests that involve collaborators is where ease and speed generally evaporate rapidly. Actors provide an interesting solution to this problem for the following reasons:

- Actors are a more direct match for tests because they embody behavior (and almost all TDD has at least some **BDD** (Behavior-Driven Development) in it)
- Too often regular unit tests test only the interface, or have to test separately interface and functionality
- Actors are built on messaging, which has huge advantages for testing as you can easily simulate behaviors by just sending messages

Before we start testing (and coding), we will take several of the concepts from

the previous chapter and show their expression in code, introducing the Actor API for creating actors then sending and receiving messages. Important details about how the actors are actually run and some rules you have to follow to prevent problems will be covered. After that, we will move on to the implementation of some common scenarios, taking a test-driven approach to writing the actors, immediately verifying that the code does what we would expect. At each step along the way, we will focus first on the goal that we are going to try to achieve with the code (one of the main points of TDD). Next, we'll write a test specification for the Actor, which will start the development of the code (TDD/Test First style). Then, of course, we will write enough code to make the test pass, and repeat. Rules that need to be followed to prevent accidentally sharing state will be discovered as we go, as well as some of the details of how actors work in Akka that have an impact on test development.

3.1 Testing Actors

First, we need to learn how to test sending and receiving messages, first in fire and forget style (one-way) followed by request response style (two-way) interaction. We are going to use the *ScalaTest* unit testing framework that is also used to test Akka itself. ScalaTest is a xUnit style testing framework, if you are not familiar with it and would like to know more about it, please visit <http://www.scalatest.org/> for more information. The ScalaTest framework is designed for readability, so it should be easy to read and follow the test without much introduction. Upon first exposure, testing Actors is more difficult than testing normal objects for a couple of reasons:

- **Timing** - Sending messages is asynchronous, so it is difficult to know when to assert expected values in the unit test.
- **Asynchronicity** - Actors are meant to be run in parallel on several threads. Multi-threaded tests are more difficult than single-threaded tests and require concurrency primitives like locks, latches and barriers to synchronize results from various actors. Exactly the kind of thing we wanted to get a bit further away from. Incorrect usage of just one barrier can block a unit test which in turn halts the execution of a full test suite.
- **Statelessness** - An actor hides its internal state and does not allow access to this state. Access should only be possible through the ActorRef. Calling a method on an actor and checking its state is prevented on purpose, which is something you would like to be able to do when unit testing.
- **Collaboration/Integraton** - If you want to do an integration test of a couple of actors, you would need to eavesdrop in between the actors to assert that the messages have the expected values. It's not immediately clear how this can be done.

Luckily Akka provides the *akka-testkit* module. This module contains a number of testing tools that makes testing actors a lot easier. The testkit module makes a couple of different types of tests possible:

- **Single threaded unit testing** - An actor instance is normally not accessible directly. The testkit provides a `TestActorRef` which allows access to the underlying actor instance. This makes it possible to just test the actor

instance directly by calling the methods that you have defined, or even call the receive function in a single threaded environment, just as you are used to when testing normal objects.

- **Multi-threaded unit testing** - The `testkit` module provides the `TestKit` and `TestProbe` classes, which make it possible to receive replies from actors, inspect messages and set timing bounds for particular messages to arrive. The `TestKit` has methods to assert expected messages. Actors are run using a normal dispatcher in a multi-threaded environment.
- **Multiple JVM testing** - Akka also provides tools for testing multiple JVMs, which comes in handy when you want to test remote actor systems. Multi-JVM testing will be discussed on chapter 5.

The `TestKit` has the `TestActorRef` extending the `LocalActorRef` class and sets the dispatcher to a `CallingThreadDispatcher` that is built for testing only. (It invokes the actors on the calling thread instead of on a separate thread.) This provides one of the key junction points for advancing the above-listed solutions.

Depending on your preference, you might use one of the styles more often. The option that is of course closest to actually running your code in production is the multi-threaded style testing with the `TestKit` class. We will focus more on the multi-threaded approach to testing, since this can show problems with the code that will not be apparent in a single-threaded environment. (You probably will not be surprised that we also prefer a classical unit testing approach over mocking).

Next, we'll cover what needs to be done to get ready to start writing tests (setup). Then, we'll go into testing with messages, and will see some of our first working examples.

3.1.1 Preparing to Test

Before we start we will have to do a little preparation so that we don't repeat ourselves unnecessarily. The `HelloWorld` example showed that once an actor system is created, it is started and continues to run until it is stopped. Let's build a small trait which we can use for all the tests that makes sure that the system under test is automatically stopped when the unit test ends.

Listing 3.1 Stop the system after all tests are done

```
import org.scalatest.{ Suite, BeforeAndAfterAll }
import akka.testkit.TestKit

trait StopSystemAfterAll extends BeforeAndAfterAll {
  ①
  this: TestKit with Suite => ②
  override protected def afterAll() {
    super.afterAll()
    system.shutdown() ③
  }
}
```

- ① Extend from the BeforeAndAfterAll ScalaTest trait.
- ② This trait can only be used if it is mixed in with a test that uses the TestKit.
- ③ Shutdown the system after all tests have executed.

We will mixin this trait when we write our tests, so that the system is automatically shutdown after all tests are executed. The TestKit exposes a `system` value, which can be accessed in the test to create actors and everything else you would like to do with the system.

In the next sections we are going to use the testkit module to test some common scenarios when working with actors, both in a single threaded and in a multi-threaded environment. There are only a few different ways for the actors to interact with each other. We will explore the different options that are available and test the specific interaction with the testkit module.

3.2 One-way messages

Remember, we have left the land of invoke a function and wait on the response, so the fact that our examples are all just sending one way messages with `tell` is deliberate. Given this fire and forget style, we don't know when the message arrives at the actor, or if it even arrives, so how do we test this? What we would like to do is send a message to an actor, and after sending the message check that the actor has done the work it should have done. An actor that responds to messages should do something with the message and take some kind of action, like send a message to another actor, or store some internal state, or interact with another object, or with I/O for instance in some kind of way. If the actor its behavior is completely invisible from the outside, we can only check if it handled the message without any errors, and we could try to look into the state of the actor with the `TestActorRef`. There are a three variations that we will look at:

- **SilentActor** - An actor's behavior is not directly observable from the outside, it might be an intermediate step that the actor takes to create some internal state. We want to test that the actor at least handled the message and did not throw any exception. We want to be sure that the actor has finished. We want to test the internal state change.
- **SendingActor** - An actor sends a message to another actor (or possibly many actors) after it is done processing the received message. We will treat the actor as a black box and inspect the message that is sent out in response to the message it received.
- **SideEffectingActor** - An actor receives a message and interacts with a normal object in some kind of way. After we send a message to the actor, we would like to assert if the object was effected.

We will write a test for each type of actor in the above list that will illustrate the means of verifying results in tests you write.

3.2.1 SilentActor Examples

Let's start with the `SilentActor`. Since it's our first test lets go briefly through the use of `ScalaTest`:

Listing 3.2 First test for the silent actor type

```

class SilentActor01Test extends TestKit(ActorSystem("testsystem")) ❶
  with WordSpec ❷
  with MustMatchers ❸
  with StopSystemAfterAll { ❹
    "A Silent Actor" must { ❺
      "change state when it receives a message, single threaded" in { ❻
        //Write the test, first fail
        fail("not implemented yet")
      }
      "change state when it receives a message, multi-threaded" in {
        //Write the test, first fail
        fail("not implemented yet")
      }
    }
  }
}

```

- ❶ extend from TestKit and provide an actor system for testing.
- ❷ WordSpec provides an easy to read DSL for testing in the BDD style
- ❸ MustMatchers provides easy to read assertions
- ❹ Make sure the system is stopped after all tests
- ❺ Write tests as textual specifications
- ❻ Every 'in' describes a specific test

The above code is the basic skeleton that we need to start running a test for the silent actor. We're using the `WordSpec` style of testing, since it makes it possible to write the test as a number of textual specifications, which will also be shown when the test is run. In the above code, we have created a specification for the silent actor type with a test that should as it says "change internal state when it receives a message." Right now it always fails since it is not implemented yet, as is expected in *Red-Green-Refactor* style, where you first make sure the test fails (Red), then implement the code to make it pass (Green), after which you might refactor the code to make it nicer. First we will test the silent actor in a single-threaded environment. We've included the `TestKit` already since we are also going to test if everything works well in a multi-threaded environment a bit later, which is not necessary if you only use the `TestActorRef`. Below we have defined an Actor that does nothing, and will always fail the tests:

Listing 3.3 First failing implementation of the silent actor type

```
class SilentActor extends Actor {
  def receive = {
    case msg => () 1
  }
}
```

- 1** swallows any message, does not keep any internal state

Now let's first write the test to send the silent actor a message and check that it changes its internal state. The `SilentActor` actor will have to be written for this test to pass, as well as an object called `SilentActorProtocol`. This object contains all the messages that `SilentActor` supports, which is a nice way of grouping messages that are related to each other as we will see later.

Listing 3.4 Single-threaded test internal state

```
"change internal state when it receives a message, single" in {
  import SilentActorProtocol._ 1
  val silentActor = TestActorRef[SilentActor] 2
  silentActor ! SilentMessage("whisper")
  silentActor.underlyingActor.state must (contain("whisper")) 3
}
```

- 1** Import the messages
- 2** Create a `TestActorRef` for single-threaded testing
- 3** Get the underlying actor and assert the state

This is the simplest version of the typical TDD scenario: trigger something and check for a state change. Now let's write the `SilentActor` actor:

Listing 3.5 SilentActor implementation

```

object SilentActorProtocol { 1
  case class SilentMessage(data: String) 2
  case class GetState(receiver: ActorRef)
}

class SilentActor extends Actor {
  import SilentActorProtocol._
  var internalState = Vector[String]()

  def receive = {
    case SilentMessage(data) =>
      internalState = internalState :+ data 3
  }

  def state = internalState
}

```

- 1** A 'protocol' which keeps related messages together
- 2** The message type that the SilentActor can process
- 3** the state is kept in a Vector, every message is added to this Vector.
- 4** the state method returns the built up Vector

Since the returned list is a immutable, the test can't change the list and cause problems when asserting the expected result. It's completely safe to set/update the internalState var, since the Actor is protected from multi-threaded access. In general it is good practice to prefer vars in combination with immutable data structures instead of vals in combination with mutable data structures.

Now lets look at the multi-threaded version of this test, as you will see we will have to change the code for the actor a bit as well. Just like in the single-threaded version where we added a state method to make it possible to test the actor, we will have to add some code to make the multi-threaded version testable.

Listing 3.6 Multi-threaded test of internal state

```
"change internal state when it receives a message, multi" in {
  import SilentActorProtocol._ 1
  val silentActor = system.actorOf(Props[SilentActor], "s3") 2
  silentActor ! SilentMessage("whisper1")
  silentActor ! SilentMessage("whisper2")
  silentActor ! GetState(testActor) 3
  expectMsg(Vector("whisper1", "whisper2")) 4
}
```

- 1** A 'protocol' which keeps related messages together
- 2** The test system is used to create an actor
- 3** A message is added to the protocol to get the state
- 4** Used to check what message(s) have been sent to the testActor

The multi-threaded test uses the "testsystem" ActorSystem that is part of the TestKit to create a SilentActor actor. Since we now cannot just access the actor instance when using the multi threaded actor system, we'll have to come up with another way to see state change. For this a GetState message is added which takes an ActorRef . The TestKit has a testActor which you can use to receive messages that you expect. The GetState method we added is so we can simply have our SilentActor send its internal state there. That way we can call the expectMsg method, which expects one message to be sent to the testActor and asserts the message, in this case that it is a Vector with all the data fields in it.

SIDEBAR Timeout settings for the `expectMsg*` methods

The `TestKit` has several versions of the `expectMsg` and other methods for asserting messages. All of these methods expect a message within a certain amount of time, otherwise they timeout and throw an exception. The timeout has a default value that can be set in the configuration using the "akka.test.single-expect-default" key.

A *dilation factor* is used to calculate the actual time that should be used for the timeout (it is normally set to 1, which means the timeout is not dilated). Its purpose is to provide a means of leveling machines that can have vastly different computing capabilities: so on a slower machine, we should be prepared to wait a bit longer (common for developers to run tests on their fast workstations then commit and have probably slower continuous integration servers fail). Each machine can be configured with the factor needed to achieve (Check out Chapter 4 for more details on configuration) The max timeout can also be set on the method directly, but it is better to just use the configured values, and change the values across tests in the configuration if necessary.

Now all we need is the code for the silent actor that can also process `GetState` messages:

Listing 3.7 SilentActor implementation

```
object SilentActorProtocol {
  case class SilentMessage(data: String)
  case class GetState(receiver: ActorRef) ❶
}
class SilentActor extends Actor {
  import SilentActorProtocol._
  var internalState = Vector[String]()
  def receive = {
    case SilentMessage(data) =>
      internalState = internalState :+ data
    case GetState(receiver) => receiver ! internalState ❷
  }
}
```

- ❶ The `GetState` message is added for testing purposes
- ❷ The internal state is sent to the `ActorRef` in the `GetState` message

The internal state is sent back to the `ActorRef` in the `GetState` message, which in this case will be the `testActor`. Since the internal state is an immutable `Vector`,

this is completely safe. This is it for the `SilentActor` types: single and multi threaded variants. Using these approaches, we can construct tests that are very familiar to most programmers: state changes can be inspected and asserted upon by leveraging a few tools in the `TestKit`.

3.2.2 *SendingActor Example*

Returning to our ticketing example from Chapter 1, we need to test the fact that when we buy a `Ticket` to an `Event`, the count of available tickets is properly decremented. There's a `Lakers vs Bulls` game and we want to be able to support any number of requests for tickets. Since the `TicketingAgent` is designed to remove a ticket and pass it on to the next one, all we have to do is create a `SendingActor` and inject it into the chain as the next recipient, then we can see the state of the tickets collection and assert that there is one less than there was when we started.

Listing 3.8 Kiosk01 test

```
"A Sending Actor" must {
  "send a message to an actor when it has finished" in {
    import Kiosk01Protocol._

    val props = Props(new Kiosk01(testActor)) ❶
    val sendingActor = system.actorOf(props, "kiosk1")
    val tickets = Vector(Ticket(1), Ticket(2), Ticket(3))

    val game = Game("Lakers vs Bulls", tickets) ❷
    sendingActor ! game
    expectMsgPF() {
      case Game(_, tickets) => ❸
        tickets.size must be(game.tickets.size - 1) ❹
    }
  }
}
```

- ❶ The next `TicketingAgent` is passed to the constructor, in the test we pass in a `testActor`
- ❷ An `Event` message is created with three tickets
- ❸ the `testActor` should receive an `Event`
- ❹ the `testActor` should receive one ticket less

A game is sent to the `Agent01` Actor, which should process the `Event` and take off one `Ticket`, and send it off to the next `Agent`. In the test we pass in the `testActor` instead of another `Agent`, which is easily done, since the

`nextAgent` is just an `ActorRef`. Since we cannot exactly know which ticket was taken off, we can't use a `expectMsg(msg)` since we can't formulate an exact match for it. In this case we use `expectMsgPF` which takes a partial function just like the `receive` of the actor. Here we match the message that was sent to the `testActor`, which should be a `Event` with one less ticket. Of course, if we run the test now, it will fail because we have not implemented the message protocol in `Agent1`. Let's do that now.

Listing 3.9 Agent01 implementation

```
object Kiosk01Protocol {
  case class Ticket(seat: Int) ❷
  case class Game(name: String, tickets: Seq[Ticket]) ❸
}

class Kiosk01(nextKiosk: ActorRef) extends Actor {
  import Kiosk01Protocol._
  def receive = {
    case game @ Game(_, tickets) =>
      nextKiosk ! game.copy(tickets = tickets.tail) ❹
  }
}
```

- ❶ The next Agent is passed to the constructor, in the test we pass in a `testActor`
- ❷ The simplified `Ticket` message
- ❸ The `Event` contains tickets
- ❹ an immutable copy is made of the `Event` message with one less ticket

We once again create a protocol that keeps all related messages together. The `Agent1` actor matches the `Event` message and extracts the `tickets` out of it (it's not interested in the first field which is the name of the event), and assigns an *alias* to the message named `event`. Next it creates an immutable copy using the `copy` method that every case class has. The copy will only contain the `tail` of the list of tickets, which is an empty list if there were no tickets left, or everything except for the first ticket in the list. Once again we take advantage of the immutable property of case classes. The event is sent along to the `nextAgent`.

Let's look at some variations of the `SendingActor` type. Here are some common variations on the theme:

Table 3.1 SendingActor Types

Actor	Description
MutatingCopyActor	The actor creates a mutated copy and sends the copy to the next actor, which is the case that we have seen just now.
ForwardingActor	The actor forwards the message it receives, it does not change it at all.
TransformingActor	The actor creates a different type of message from the message that it receives.
SequencingActor	The actor creates many messages based on one message it receives and sends the new messages one after the other to another actor.

The `MutatingCopyActor`, `ForwardingActor` and `TransformingActor` can all be tested in the same way. We can pass in a `testActor` as the next actor to receive messages and use the `expectMsg` or `expectMsgPF` to inspect the messages. The `FilteringActor` is a bit different in that it addresses the question of how can we assert that some messages were *not* passed through? The `SequencingActor` needs a similar approach. How can we assert that we received the correct number of messages? The next test will show you how. Let's write a test for the `FilteringActor`. The `FilteringActor` that we are going to build should filter out duplicate events. It will keep a list of the last messages that it has received, and will check each incoming message against this list to find duplicates. (This is comparable to the typical elements of mocking frameworks that allow you to assert on invocations, counts of invocations, and absence.)

Listing 3.10 FilteringActor test

```

"filter out particular messages" in {
  import FilteringActorProtocol._
  val props = Props(new FilteringActor(testActor, 5))
  val filter = system.actorOf(props, "filter-1")

  filter ! Event(1) ❶
  filter ! Event(2)
  filter ! Event(1)
  filter ! Event(3)
  filter ! Event(1)
  filter ! Event(4)
  filter ! Event(5)
  filter ! Event(5)
  filter ! Event(6)

  val eventIds = receiveWhile() { ❷
    case Event(id) if id <= 5 => id
  }

  eventIds must be(List(1, 2, 3, 4, 5)) ❸
  expectMsg(Event(6))
}

```

- ❶ Send a couple of events, including duplicates
- ❷ Receive messages until the case statement does not match anymore
- ❸ Assert that the duplicates are not in the result

The test uses a `receiveWhile` method to collect the messages that the `testActor` receives until the case statement matches. In the test the `Event(6)` does not match the pattern in the case statement, which defines that all `Events` with an `id` less than or equal to 5 are going to be matched, popping us out of the while loop. The `receiveWhile` method returns the collected items as they are returned in the partial function as a list, which is not allowed to have any duplicates. Now let's write the `FilteringActor` that will guarantee this part of the specification:

Listing 3.11 FilteringActor implementation

```

object FilteringActorProtocol {
  case class Event(id: Long)
}

class FilteringActor(nextActor: ActorRef,
                    bufferSize: Int) extends Actor { ❶
  import FilteringActorProtocol._
  var lastMessages = Vector[Event]() ❷
  def receive = {
    case msg: Event =>
      if (!lastMessages.contains(msg)) {
        lastMessages = lastMessages :+ msg
        nextActor ! msg ❸
        if (lastMessages.size > bufferSize) {
          // discard the oldest
          lastMessages = lastMessages.tail ❹
        }
      }
  }
}

```

- ❶ A max size for the buffer is passed into the constructor
- ❷ A vector of last messages is kept
- ❸ The event is sent to the next actor if it is not found in the buffer
- ❹ The oldest event in the buffer is discarded when the max buffersize is reached

The above `FilteringActor` keeps a buffer of the last messages that it received in a `Vector` and adds every received message to it if it does not already exist in the list. Only messages that are not in the buffer are sent to the `nextActor`. The oldest message that was received is discarded when a max `bufferSize` is reached to prevent the `lastMessages` list from growing too large and possibly causing us to run out of space.

The `receiveWhile` method can also be used for testing a `SequencingActor`; you could assert that the sequence of messages that is caused by a particular event is as expected. Two methods for asserting messages that might come in handy when you need to assert a number of messages are `ignoreMsg` and `expectNoMsg`. `ignoreMsg` takes a partial function just like the `expectMsgPF` method, only instead of asserting the message it ignores any

message that matches the pattern. This can come in handy if you are not interested in many messages, but only want to assert that particular messages have been sent to the `testActor`. The `expectNoMsg` asserts that no message has been sent to the `testActor` for a certain amount of time, which we could have also used in between the sending of duplicate messages in the `FilteringActor` test. The test in 3.12 shows an example of using `expectNoMsg`.

Listing 3.12 FilteringActor implementation

```
"filter out particular messages using expectNoMsg" in {
  import FilteringActorProtocol._
  val props = Props(new FilteringActor(testActor, 5))
  val filter = system.actorOf(props, "filter-2")
  filter ! Event(1)
  filter ! Event(2)
  expectMsg(Event(1))
  expectMsg(Event(2))
  filter ! Event(1)
  expectNoMsg
  filter ! Event(3)
  expectMsg(Event(3))
  filter ! Event(1)
  expectNoMsg
  filter ! Event(4)
  filter ! Event(5)
  filter ! Event(5)
  expectMsg(Event(4))
  expectMsg(Event(5))
  expectNoMsg()
}
```

Since the `expectNoMsg` has to wait for a timeout to be sure that no message was received, the above test will run more slowly.

As we've seen the `TestKit` provides a `testActor` that can receive messages, which we can assert with `expectMsg` and other methods. A `TestKit` has only one `testActor` and since the `TestKit` is a class that you need to extend, how would you test an actor that sends messages to more than one actor? The answer is the `TestProbe` class. The `TestProbe` class is very much like the `TestKit`, only you can use this class without having to extend from it. Simply create a `TestProbe` with `TestProbe()` and start using it. The `TestProbe` will be used quite often in the tests that we are going to write in chapter 8, which covers topics like load balancing and routing.

3.2.3 SideEffectingActor Example

Remember the HelloWorld example? It does just one thing: its Greeter receives a message and outputs it to the console. The SideEffectingActor allows us to test scenarios such as these: where the effect of the action is not directly accessible. While many cases fit this description, this one sufficiently illustrates the final means of testing for an expected result.

Listing 3.13 Testing HelloWorld

```
import Greeter01Test._

class Greeter01Test extends TestKit(testSystem) ②
  with WordSpec
  with MustMatchers
  with StopSystemAfterAll {

  "The Greeter" must {
    "say Hello World! when a Greeting("World") is sent to it" in {
      val dispatcherId = CallingThreadDispatcher.Id
      val props = Props[Greeter].withDispatcher(dispatcherId) ③
      val greeter = system.actorOf(props)
      EventFilter.info(message = "Hello World!",
        occurrences = 1).intercept { ④
          greeter ! Greeting("World")
        }
    }
  }
}

object Greeter01Test {
  val testSystem = { ①
    val config = ConfigFactory.parseString(
      """akka.event-handlers = ["akka.testkit.TestEventListener"]""")
    ActorSystem("testsystem", config)
  }
}
```

- ① Create a system with a configuration that attaches a test event listener
- ② Use the testSystem from the Greeter01Test object
- ③ Single threaded environment
- ④ Intercept the log messages that were logged

The Greeter is tested by inspecting the log messages that it writes using the ActorLogging trait. The testkit module provides a TestEventListener

that you can configure to handle all events that are logged. The `ConfigFactory` can parse a configuration file from a `String`, in this case we only override the event handlers list.

The test is run in a single threaded environment because we want to check that the log event has been recorded by the `TestEventListener` when the greeter is sent the "World" Greeting . We use an `EventFilter` object, which can be used to filter log messages. in this case we filter out the expected message, which should only occur once. The filter is applied when the intercept code block is executed, which is when we send the message.

The above example of testing a `SideEffectingActor` shows that asserting some interactions can get complex quite quickly. In a lot of situations it is easier to adapt the code a little bit so that it is easier to test. Clearly, if we pass the listeners to the class under test, we don't have to do any configuration or filtering, we will simply get each message our Actor under test produces. The below example shows an adapted `Greeter` Actor which can be configured to send a message to a *listener* actor whenever a greeting is logged:

Listing 3.14 Simplifying testing of the Greeting Actor with a listener

```
class Greeter02(listener: Option[ActorRef] = None) ❶
  extends Actor with ActorLogging {
    def receive = {
      case Greeting(who) =>
        val message = "Hello " + who + "!"
        log.info(message)
        listener.foreach(_ ! message) ❷
    }
  }
```

- ❶ The constructor takes an optional listener, default set to `None`
- ❷ optionally sending to the listener

The `Greeter02` actor is adapted so that it takes an `Option[ActorRef]` , which is default set to `None`. After it successfully logs a message, it sends a message to the listener if the `Option` is not empty. When the actor is used normally without specifying a listener it runs as usual. Below is the updated test for this `Greeter02` actor.

Listing 3.15 Simpler Greeting Actor test

```
class Greeter02Test extends TestKit(ActorSystem("testsystem"))
  with WordSpec
  with MustMatchers
  with StopSystemAfterAll {

  "The Greeter" must {
    "say Hello World! when a Greeting("World") is sent to it" in {
      val props = Props(new Greeter02(Some(testActor))) ❶
      val greeter = system.actorOf(props, "greeter02-1")
      greeter ! Greeting("World")
      expectMsg("Hello World!") ❷
    }
    "say something else and see what happens" in {
      val props = Props(new Greeter02(Some(testActor)))
      val greeter = system.actorOf(props, "greeter02-2")
      system.eventStream.subscribe(testActor, classOf[UnhandledMessage])
      greeter ! "World"
      expectMsg(UnhandledMessage("World", system.deadLetters, greeter))
    }
  }
}
```

- ❶ Set the listener to the testActor
- ❷ Assert the message as usual

As you can see the test has been greatly simplified. We simply pass in a `Some(testActor)` into the `Greeter02` constructor, and assert the message that is sent to the testActor as usual.

In the next section we are going to look at two-way messages, and how these can be tested.

3.3 Two-way messages

We have already seen an example of two-way messages in the multi-threaded test for the `SendingActor` style actor, where we used a `GetState` message that contained an `ActorRef`. We simply called the `!` operator on this `ActorRef` to respond to the `GetState` request. As shown before the `tell` method has an implicit sender reference.

Two-way messages are quite easy to test in a black box fashion, a request should result in a response, which you can simply assert. In the following test we will test an `EchoActor`, an actor that echoes any request back in a response.

Listing 3.16 Testing Echoes

```
"Reply with the same message it receives without ask" in {
  val echo = system.actorOf(Props[EchoActor], "echo2")
  echo.tell("some message", testActor) ❶
  expectMsg("some message") ❷
}
```

- ❶ Call `tell` with the `testActor` as the sender
- ❷ Assert the message as usual

We just call the `tell` method with an explicit `sender`, which the `EchoActor` will use to send the response back to. The `EchoActor` stays exactly the same. It just sends a message back to the sender.

Listing 3.17 Echo Actor

```
class EchoActor extends Actor {
  def receive = {
    case msg =>
      sender ! msg
  }
}
```

The `EchoActor` reacts exactly the same way whether the `ask` pattern was used or of the `tell` method; the above is the preferred way to test two-way messages.

Our journey in this section has taken us through actor testing idioms that are offered by Akka's `TestKit`. They are all serving the same goal: making it easy to write unit tests that need access to results that can be asserted upon. The `testkit` provides both methods for single-threaded and multi-threaded testing. We can even 'cheat' a little and get at the underlying actor instance during testing. Categorizing actors by how they interact with others gives us a template for how to test the actor, which was shown for the `SilentActor`, `SendingActor` and `SideEffectingActor` types. In most cases the easiest way to test an actor is to pass a `testActor` reference to it, which can be used to assert expectations on the messages that are sent out by the actor under test. The `testActor` can be used to take the place of a sender in a request response or it can just act like the next actor that an actor is sending

messages to. Finally, we saw that in many cases it makes sense to prepare an actor for testing, especially if the actor is 'silent,' in which case it is beneficial to add an optional listener to the actor.

3.4 Summary

Test-Driven Development is fundamentally a way of working. Akka was designed to support that approach. Since the bedrock of regular unit testing is to invoke a method and get a response that can be checked for an expected result, we had to look at how to adopt a new mindset to go along with our message-based, asynchronous style. The TestToolkit provides us with those tools.

Actors also bring some new powers to the seasoned TDD programmer:

- Actors embody behavior, tests are fundamentally a means of checking behavior
- Message-based tests are cleaner: only immutable state goes back and forth, precluding the possibility of tests corrupting the state they are testing
- With the understanding of the core test actors, you can now write unit tests of actors of all kinds

As we go forward, developing additional examples, we will see these actors again, and the best aspects of TDD are preserved in Akka.

In the next chapter we are going to look at how actor hierarchies are formed and how supervision strategies and lifecycle monitoring can be used to build fault tolerant systems.

Fault tolerance



In this chapter

- Self healing systems
- Let it crash
- Actor life cycle
- Supervision
- Fault recovery strategies

This chapter covers the tools Akka provides to make applications more resilient. The first section describes the *let it crash* principle, including supervision, monitoring and actor lifecycle features. Of course, we will look at some examples that show how to apply these to typical failure scenarios.

4.1 What is fault tolerance (and what it isn't)

Let's first start with a definition of what we mean when we say a system is fault tolerant, and why we would write code to embrace the notion of failure. In an ideal world a system is always available and is able to guarantee that it will be successful with each undertaken action. The only two ways to this ideal are use components that can never fail or account for every single fault by providing a recovery action, that is, of course also assured of success. In most architectures, what we have instead is a catchall mechanism that will terminate as soon as an uncaught failure arises. Even if an application attempts to provide recovery strategies, testing them is hard, and being sure that the recovery strategies themselves work adds another layer of complexity. In the procedural world, each attempt to do something requires a return code that is checked against a list of possible faults. Exception handling came along and quickly became a fixture of modern languages, promising a less onerous path to providing the various required means of recovery, but while it has succeeded in yielding code that doesn't have to have fault checks on every line, the propagation of faults to ready handlers has not significantly improved.

The idea of a system that is free of faults sounds great in theory but the sad fact is that building one that is also highly available and distributed is simply not possible for any non-trivial system. The main reason for this is that large parts of any non-trivial system are not under our control and these parts can break. Then there is the prevalent problem of responsibility: as collaborators interact, using many times shared components (remember our example from the first chapter), it's not clear who is responsible for which possible faults. A good example of potentially unavailable resources is the network: it can go away at any time or be partly available, and if we want to continue operation, we will have to find some other way to continue communicating, or maybe disable communication for a while. We might depend on third party services that can misbehave, fail or simply be sporadically unavailable. The servers our software runs on can fail or can be unavailable or even experience total hardware failure. You obviously cannot magically make a server reappear out of it's ashes or automatically fix a broken disk to guarantee writing to it. This is why let it crash was born in the rack and stack world of the telcos where failed machines were common enough to make their availability goals impossible without a plan that accounted for them.

Since we cannot prevent all failures from happening we will have to prepared to adopt a strategy, keeping the following in mind:

- Things break. The system needs to be fault tolerant so that it can stay available and continue to run. Recoverable faults should not trigger catastrophic failures.
- In some cases it is acceptable if the most important features of the system stay available as long as possible, while in the mean time failing parts are stopped and cut off from the system so that they cannot interfere with the rest of the system, producing unpredictable results.
- In other cases certain components are so important that they need to have active backups (probably on a different server or using different resources) that can kick in when the main component fails so that the unavailability is quickly remedied.
- A failure in certain parts of the system should not crash the entire system so we need a way to isolate particular failures that we can deal with later.

Of course, the Akka toolkit does not include a fault tolerance silver bullet. We will still have to handle specific failures, but will be able to do it in a cleaner, more application-specific way. The following Akka features will enable us to build the fault tolerant behavior we need:

Table 4.1 Available Fault Avoidance Strategies

Fault containment or isolation	a fault should be contained to a part of the system and not escalate to a total crash.
Structure	isolating a faulty component means that some structure needs to exist to isolate it from; the system will need a defined structure in which active parts can be isolated.
Redundancy	a backup component should be able to take over when a component fails.
Replacement	If a faulty component can be isolated we can also replace it in the structure. The other parts of the system should be able to communicate with the replaced component just as they did before with the failed component.
Reboot	If a component gets into an incorrect state, we need to have the ability to get it back to a defined initial state. The incorrect state might be the reason for the fault and it might not be possible to predict all the incorrect states the component can get into because of dependencies out of our control.
Component Lifecycle	a faulty component needs to be isolated and if it cannot recover it should be terminated and removed from the system or re-initialized with a correct starting state. Some defined lifecycle will need to exist to start, restart and terminate the component.
Suspend	When a component fails we would like all calls to the component to be suspended until the component is fixed or replaced so that when it is, the new component can continue the work without dropping a beat. The call that was handled at the time of failure should also not disappear, it could be critical to our recovery, and further, it might contain information that is critical to understanding why the component failed. We might want to retry the call when we are sure that there was another reason for the fault.
Separation of Concerns	It would be great if the fault recovery code could be separated from the normal processing code. Fault recovery is a cross cutting concern in the normal flow. A clear separation between normal flow and recovery flow will simplify the work that needs to be done. Changing the way the application recovers from faults will be simpler if we have achieved this clean separation.

But wait a minute, you might say, why can't we just use plain old objects and exceptions to recover from failures? Normally exceptions are used to back out of a series of actions to prevent an inconsistent state instead of recovering from a failure in the sense we have discussed so far. But let's see how hard it would be to add fault recovery using exception handling and plain old objects in the next section.

4.1.1 Plain old objects and exceptions

Let's look at an example of an application that receives logs from multiple threads, 'parses' interesting information out of the files into row objects and writes these rows into some database. Some file watcher process keeps track of added files and informs many threads in some way to process the new files. The below figure gives an overview of the application and highlights the part that we will zoom into ("in scope"):

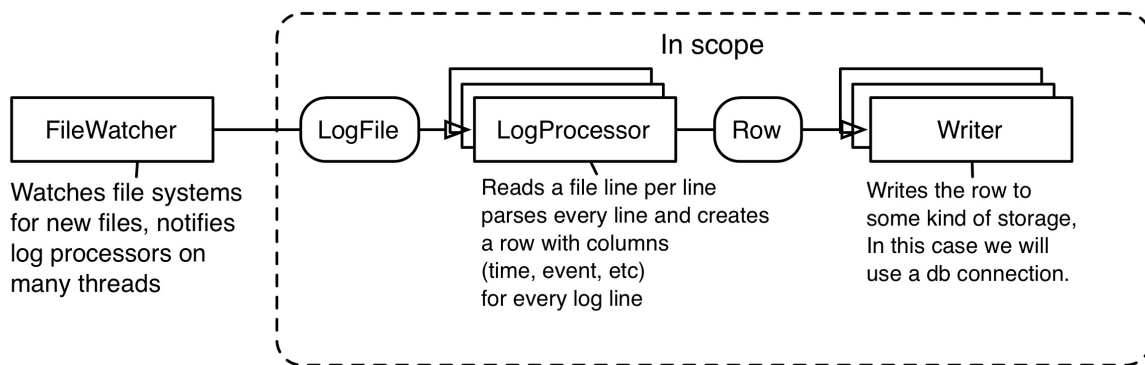


Figure 4.1 Process logs application

If the database connection breaks we want to be able to create a new connection to another database and continue writing, instead of backing out. If the connection starts to malfunction we might want to shut it down so that no part of the application uses it anymore. In some cases we want to just reboot the connection, hopefully to get rid of some temporary bad state in it. Pseudo code will be used to illustrate where the potential problem areas are. We'll look at the case where we would like to just get a new connection to the same database using standard exception handling.

First, all objects are setup that will be used from the threads. After setup, they will be used to process the new files that the file watcher finds. We setup a database writer that uses a connection. The below figure shows how the writer is created.

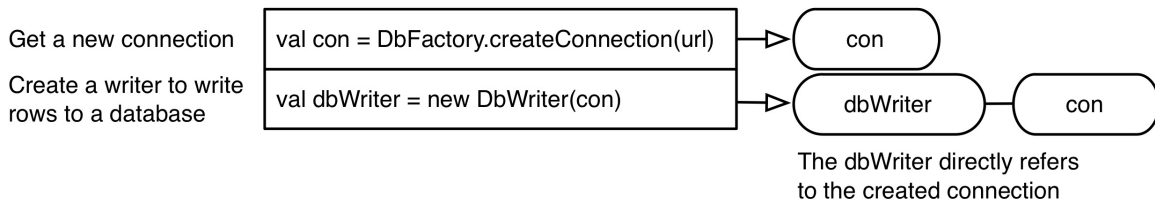


Figure 4.2 Create a writer

The dependencies for the writer are passed to the constructor as you would expect. The database factory settings, including the different urls, are passed in from the thread that creates the writer. Next we setup some log processors; each gets a reference to a writer to store rows, as shown in the figure below.

The dbWriter is made available to many log processors.

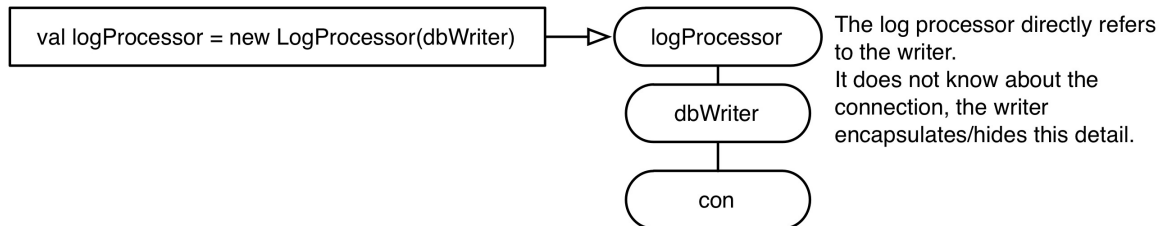
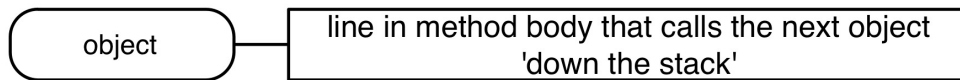


Figure 4.3 Create log processors

The below figure shows how the objects call each other in this example application:



Read the diagram top to bottom.

The diagram only shows where an object eventually calls another object down the call stack, the other details of the method are omitted. The below diagram shows a runnable that calls a logProcessor to process a file. Inside the process method the logProcessor eventually calls the dbWriter write method, which eventually calls the write on the connection object.

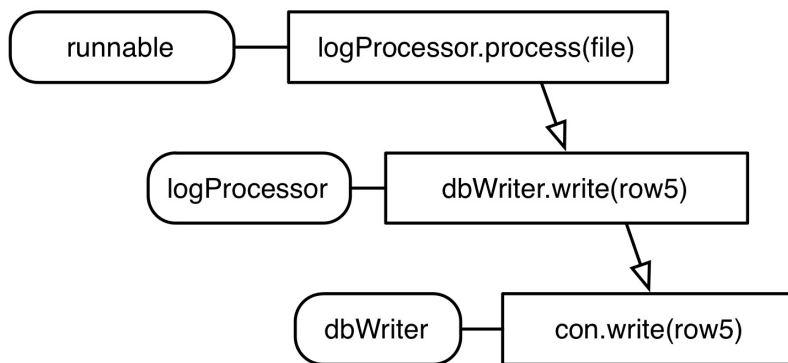


Figure 4.4 Call stack while processing log files

The above flow gets called from many threads to simultaneously process files found by the file watcher. The below figure shows a call stack where a `DbBrokenConnectionException` is thrown, which indicates that we should switch to another connection. The details of every method are omitted, the diagram only shows where an object eventually calls another object:

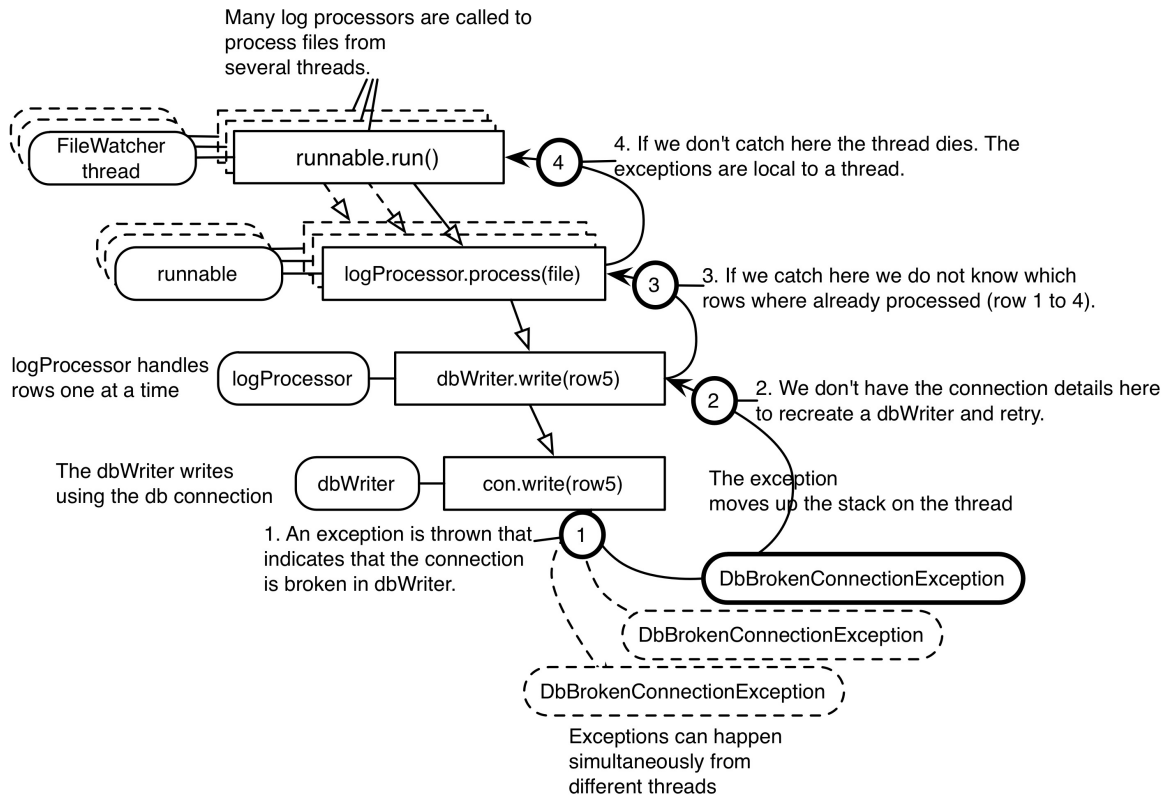


Figure 4.5 Call stack while processing log files

Instead of just throwing the exception up the stack, we would like to recover from the `DbBrokenConnectionException` and replace the broken connection with a working one. The first problem we face is that it is hard to add the code to recover the connection in a way that does not break the design. Also, we don't have enough information to recreate the connection: we don't know which lines in the file have already been processed successfully and which line was being processed when the exception occurred.

Making both the processed lines and the connection information available to all objects would break our simple design and violate some basic best practices like encapsulation, inversion of control, and single responsibility to name a few. (Good luck at the next code peer review with your clean coding colleagues!) We just want the faulty component replaced. Adding recovery code directly into the exception handling will entangle the functionality of processing log files with database connection recovery logic. Even if we find a spot to recreate the connection, we would have to be very careful that other threads don't get to use the faulty connection while we are trying to replace it with a new one because otherwise some rows would be lost. There are three connection pools in the Java world, after nearly two decades, only one even has a working implementation of dead

connection removal on another thread. This is clearly not easy with our existing tools.

Also, communicating exceptions between threads is not a standard feature, you would have to build this yourself. Let's look at the failure tolerant requirements to see if this approach even stands a chance.

- **Fault isolation;** Isolation is made difficult by the fact that many threads can throw exceptions at the same time. We'll have to add some kind of locking mechanism. It is hard to really remove the faulty connection out of the chain of objects, the application would have to be rewritten to get this to work. There is no standard support for cutting off the use of the connection in the future, so this needs to be built into the objects manually with some level of indirection.
- **Structure;** The structure that exists between objects is very simple and direct and by default does not provide any support for simply taking out an object. You would have to create a more involved structure yourself (again, with a level of indirection between the objects).
- **Redundancy;** When an exception is thrown it goes up the call stack. You might miss the context for making the decision which redundant component to use or lose the context of which input data to continue with, as seen in above example.
- **Replacement;** There is no default strategy in place to replace an object in a call stack, you would have to find a way to do it yourself. There are dependency injection frameworks that provide some features for this, but if any object simply referred directly to the old instance instead of through the level of indirection you are in trouble. If you intend to change an object in place you better make sure it works for multi-threaded access.
- **Reboot;** Similar to replacement, getting an object back to an initial state is not automatically supported and takes another level of indirection that you will have to build. All the dependencies of the object will have to be reintroduced as well. If these dependencies also need to be rebooted (lets say the log processor can also throw some recoverable error), things can get quite complicated with regard to ordering.
- **Component lifecycle;** an object only exists after it has been constructed or it is garbage collected and removed from memory. Any other mechanism is something you will have to build yourself.
- **Suspend;** The input data or some of its context is lost or not available when you catch an exception and throw it up the stack. You would have to build something yourself to buffer the incoming calls while the error has not been resolved. If the code is called from many threads you need to add locks to prevent multiple exceptions from happening at the same time. And you would need to find a way to store the associated input data to retry again later.
- **Separation of concerns;** The exception handling code is interwoven with the processing code and cannot be defined independently of the processing code.

So that's not looking very promising, getting everything to work correctly is going to be complex and a real pain. It looks like there are some fundamental features missing for adding fault tolerance to our application in an easy way:

- **Recreating objects and their dependencies and replacing these in the application structure**

is not available as a first-class feature.

- Objects communicate with each other directly so it is hard to isolate them.
- The fault recovery code and the functional code are tangled up with each other.

Luckily we have a simpler solution. We have already seen some of the actor features that can help simplify these problems. Actors can be (re)created from `Props` objects, are part of an actor system and communicate through actor references instead of direct references. In the next section, we will look at how actors provide a way to untangle the functional code from the fault recovery code and how the actor life-cycle makes it possible to suspend and restart actors (without invoking the wrath of the concurrency gods) in the course of recovering from faults.

4.1.2 Let it crash

In the previous section we learned that building a fault tolerant application with plain old objects and exception handling is quite a complex task. Lets look at how actors simplify this task. So what should happen when an Actor processes a message and encounters an exception? We already discussed why we don't want to just graft recovery code into the operational flow, so catching the exception inside an actor where the business logic resides, is out.

Instead of using one flow to handle both normal code and recovery code Akka provides two separate flows; one for normal logic and one for fault recovery logic. The normal flow consists of actors that handle normal messages, the recovery flow consists of actors that monitor the actors in the normal flow. Actors that monitor other actors are called supervisors. The below figure shows a supervisor monitoring an actor.

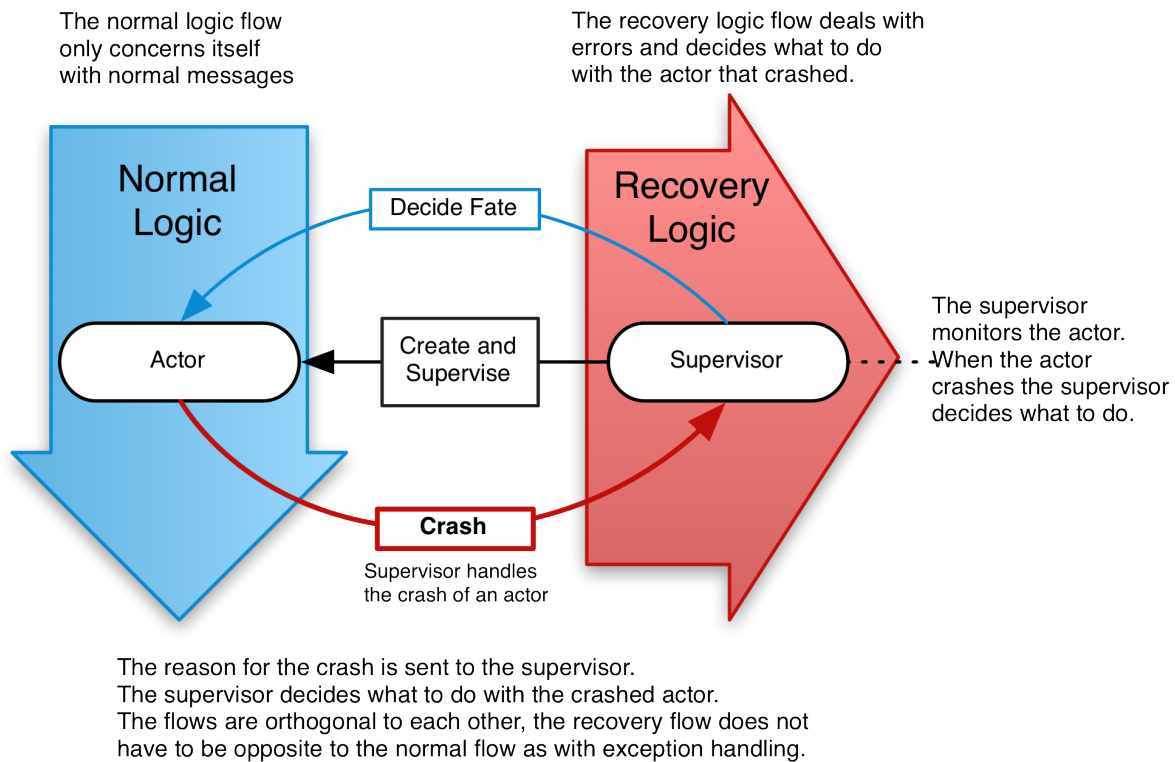


Figure 4.6 Normal and recovery flow

Instead of catching exceptions in an actor, we will just let the actor crash. The actor code only contains normal processing logic and no error handling or fault recovery logic, so its effectively not part of the recovery process, which keeps things much clearer. The mailbox for a crashed actor is suspended until the supervisor in the recovery flow has decided what to do with the exception. So how does an actor become a supervisor? Akka has chosen to enforce *parental supervision*, meaning that any actor that creates actors automatically becomes the supervisor of those actors. A supervisor does not 'catch exceptions,' rather it decides what should happen with the crashed actors that it supervises based on the cause of the crash. The supervisor does not try to fix the actor or its state. It simply renders a judgment on how to recover, then triggers the corresponding strategy. The supervisor has 4 options when deciding what to do with the actor:

- Restart; the actor must be recreated from its Props. after it is restarted (or rebooted if you will) the actor will continue to process messages. Since the rest of the application uses an ActorRef to communicate with the actor the new actor instance will automatically get the next messages.
- Resume; the same actor instance should continue to process messages, the crash is ignored.
- Stop; the actor must be terminated. It will no longer take part in processing messages.
- Escalate; the supervisor does not know what to do with it and escalates the problem to its parent, which is also a supervisor.

The below figure gives an example of the strategy that we could choose when we build the log processing application with actors. The supervisor is shown to take one of the possible actions when a particular crash occurs:

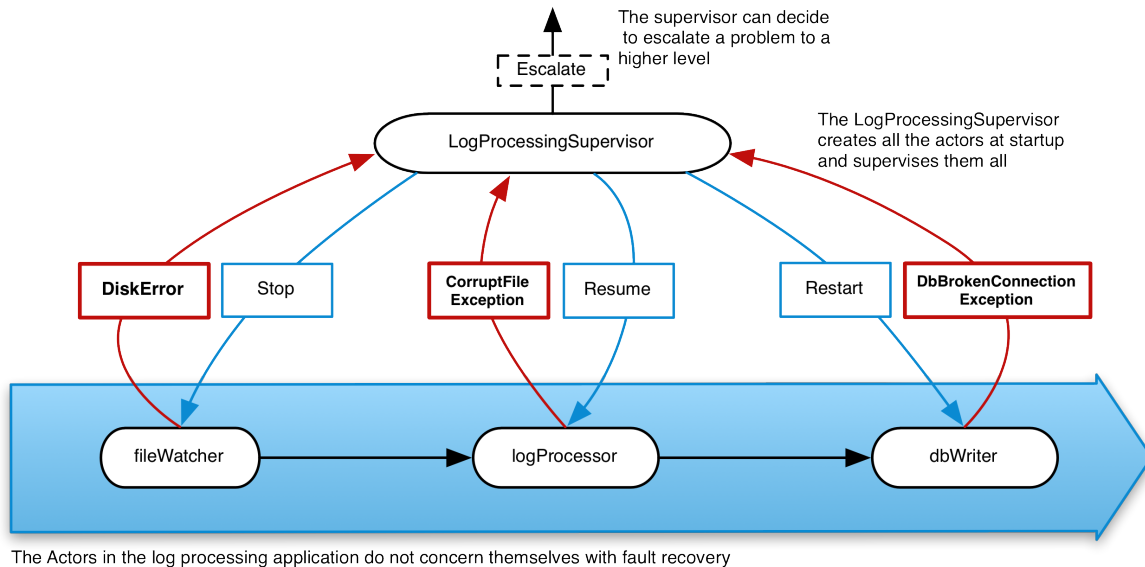


Figure 4.7 Normal and recovery flow in the logs processing application

The above figure shows a solution for making the log processing fault tolerant, at least for the broken connection problem. When the `DbBrokenConnectionException` occurs the `dbWriter` actor crashes and is replaced with a recreated `dbWriter` actor.

We will need to take some special steps to re-cover the failed message, which we will discuss in the details of how to implement a `Restart` in code later. Suffice it to say that in most cases you do not want to re-process a message because it probably caused the error in the first place. An example of that would be the case of the `logProcessor` encountering a corrupt file: reprocessing corrupt files could end up in what is called a *poisoned mailbox*, no other message will ever get processed because the corrupting message is failing over and over again. For this reason Akka chooses not to provide the failing message to the mailbox again after a restart, but there is a way to do this yourself if you are absolutely sure that the message did not cause the error, which we will discuss later. The good news is, if a job is processing tens of thousands of messages, and one is corrupt, default behavior will result in all the other messages being processed normally; the one corrupt file will not cause a catastrophic failure and erase all the other work done to that point (and prevent the remainder from occurring).

The below figure shows the how a crashed dbWriter actor instance is replaced with a fresh instance when the supervisor chooses to restart.

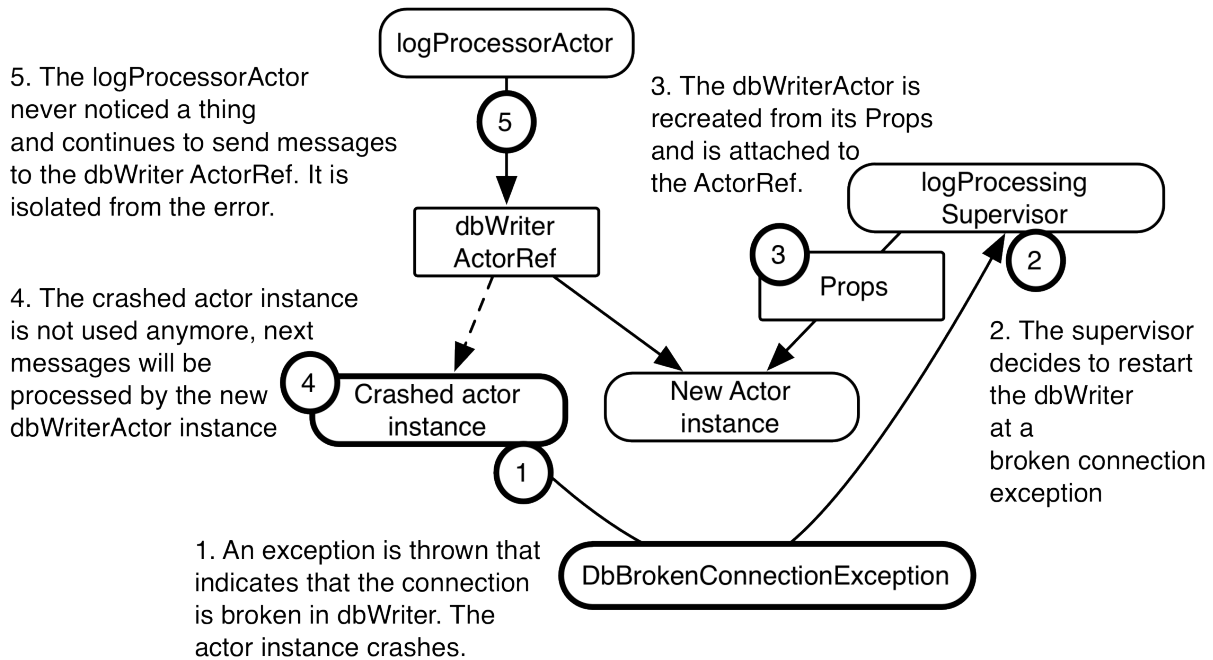


Figure 4.8 Handling the DbBrokenConnectionException with a restart

Let's recap the benefits of the let it crash approach:

- **Fault isolation;** A supervisor can decide to terminate an actor. The actor is removed from the actor system.
- **Structure;** the actor system hierarchy of actor references makes it possible to replace actor instances without other actors being affected.
- **Redundancy;** An actor can be replaced by another. In the example of the broken database connection the fresh actor instance could connect to a different database. The supervisor could also decide to stop the faulty actor and create another type instead. Another option would be to route messages in a load balanced fashion to many actors, which will be discussed in chapter 8.
- **Replacement;** An actor can always be recreated from its Props. A supervisor can decide to replace a faulty actor instance with a fresh one, without having to know any of the details for recreating the actor.
- **Reboot;** This can be done through a restart.
- **Component lifecycle;** An actor is an active component. It can be started, stopped and restarted. In the next section we will go into the details of how the actor goes through its life-cycle.
- **Suspend;** When an actor crashes its mailbox is suspended until the supervisor decides what should happen with the actor.
- **Separation of concerns;** The normal actor message processing and supervision fault recovery flows are orthogonal and can be defined and evolve completely independently of each other.

In the next sections we will get into the coding details of the actor life-cycle and supervision strategies.

4.2 Actor life-cycle

We have seen that an actor can restart to recover from a failure. But how can we correct the actor state when the actor is restarting? To answer that question we need to take a closer look at the actor life cycle. An actor is automatically started by Akka when it is created. The actor will stay in the Started state until it is stopped. From that moment the actor is in the Terminated state. When the actor is terminated it can't process messages anymore and will be eventually garbage collected. When the actor is in a Started state it can be restarted to reset the internal state of the actor. As we discussed in the previous section the actor instance is replaced by a fresh actor instance. The restart can happen as many times as necessary. During the life cycle of an actor there are three types of events:

1. The actor is created and started, for simplicity we will refer to this as the *Start* event.
2. The actor is restarted on the *Restart* event.
3. The actor is stopped by the *Stop* event.

There are several hooks in place in the `Actor` trait which are called when the events happen to indicate a life-cycle change. We can add some custom code in these hooks that can be used to recreate specific state in the fresh actor instance, to process the message that failed before the restart, or to cleanup some resources for instance. In the next sections we will look at the three events and how the hooks can be used to run custom code. The order in which the hooks occur is guaranteed although they are called asynchronously by Akka.

4.2.1 Start event

An actor is created and automatically started with the `actorOf` method. Top level actors are created with the `actorOf` method on the `ActorSystem`. A parent actor creates a child actor using the `actorOf` on its `ActorContext`.

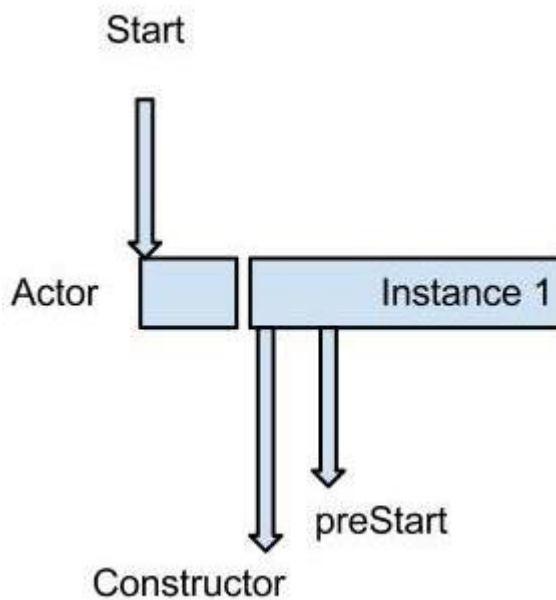


Figure 4.9 Starting an actor

After the instance is created the actor is going to be started by Akka. The `preStart` hook is called just before the actor is started. To use this trigger we have to override the `preStart` method.

Listing 4.1 `preStart` life cycle hook

```
override def preStart() {
  println("preStart")
}
```

1 do some work

This hook can be used to set the initial state of the actor.

4.2.2 Stop event

The next life-cycle event that we will discuss is the stop event. We will get back to the restart event later because its hooks have dependencies on the start and stop hooks. The stop event indicates the end of the actor life-cycle and occurs once, when an actor is stopped. An actor can be stopped using the `stop` method on the `ActorSystem` and `ActorContext` objects, or by sending a `PoisonPill` message to an actor.

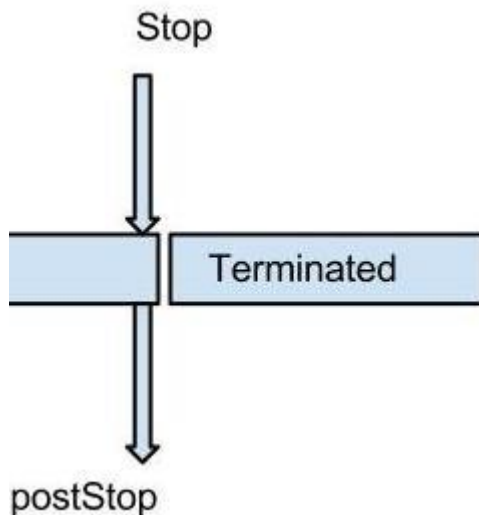


Figure 4.10 Stop an actor

The `postStop` hook is called just before the actor is terminated. When the actor is in the `Terminated` state the actor doesn't get any new messages to handle. The `postStop` method is the counterpart of the `preStart` hook.

Listing 4.2 `postStop` life cycle hook

```
override def postStop() {
  println("postStop")
}
```

1 do some work

Normally this hook implements the opposite function of the `preStart` and releases resources created in the `preStart` method and possibly stores the last state of the actor somewhere outside of the actor in the case that the next actor instance needs it. A stopped actor is disconnected from its `ActorRef`. After the actor is stopped, the `ActorRef` is redirected to the `deadLetters` `ActorRef` of the actor system, which is a special `ActorRef` that receives all messages that are sent to dead actors.

4.2.3 Restart event

During the life-cycle of an actor it is possible that its supervisor will decide that the actor has to be restarted. This can happen more than once, depending on the number of errors that occur. This event is a little bit more complex than the start or stop events. This is because the instance of an actor is replaced.

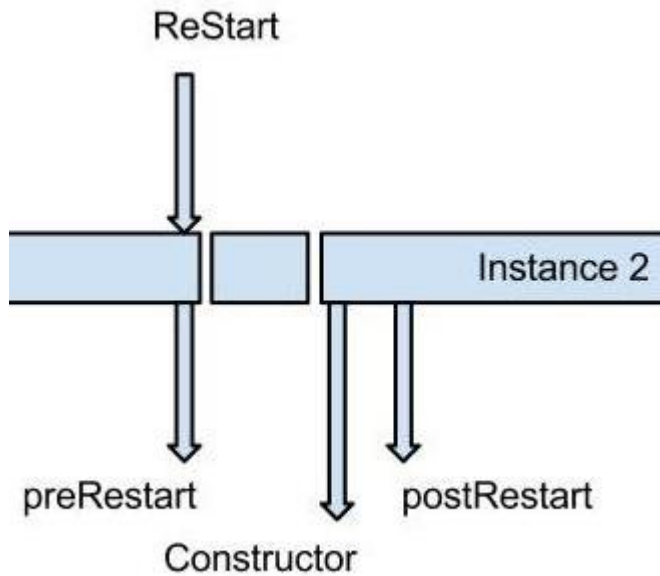


Figure 4.11 Restarting an actor

When a restart occurs the `preRestart` method of the crashed actor instance is called. In this hook the crashed actor instance is able to store its current state, just before it is replaced by the new actor instance.

Listing 4.3 `preRestart` life cycle hook

```

override def preRestart(reason: Throwable, 1
                        message: Option[Any]) { 1
  println("preRestart")
  super.preRestart(reason, message) 2
}

```

- ¹ exception which was thrown by the actor
- ² when the error occurred within the receive function than this is the message which the actor was trying to process
- ³ WARNING call the super implementation

Be careful when overriding this hook. The default implementation of the `preRestart` method stops all the child actors of the actor and then calls the `postStop` hook. If you forget to call `super.preRestart` this default behavior will not occur. Remember that actors are (re)created from a `Props` object. The `Props` object eventually calls the constructor of the actor. The actor

can create child actors inside its constructor. If the children of the crashed actor would not be stopped we could end up with more and more child actors when the parent actor is restarted.

It's important to note that a restart does not stop the crashed actor in the same way as the stop methods (described above in the stop event). As we will see later it is possible to monitor the death of an actor. A crashed actor instance in a restart does not cause a `Terminated` message to be sent for the crashed actor. The fresh actor instance, during restart, is connected to the same `ActorRef` the crashed actor was using before the fault. A stopped actor is disconnected from its `ActorRef` and redirected to the `deadLetters ActorRef` as described by the stop event. What both the stopped actor and the crashed actor have in common is that by default the `postStop` is called after they have been cut off from the actor system.

The `preRestart` method takes two arguments: the reason for the restart and optionally the message that was being processed when the actor crashed. The supervisor can decide what should (or can) be stored to enable state restoration as part of restarting. And of course this can't be done using local variables because after restarting a fresh actor instance will take over processing. One solution for keeping state beyond the death of the crashed actor is for the supervisor to send a message to the actor (will go in its mailbox). (Done by sending a message to its own `ActorRef`, which is available on the actor instance through the `self` value.) Other options are writing to something outside of the actor, like a database or the file system. This all depends completely on your system and the behavior of the actor.

Which brings us back to the log processing example, where we did not want to lose the `Row` message in the case of a `dbWriter` crash. The solution in that case could be to send the failed `Row` message to the `self ActorRef` so it would be processed by the fresh actor instance. One issue to note with this approach is that by sending a message back onto the mailbox, the order of the messages on the mailbox is changed. The failed message is pushed off the top of the mailbox and will be processed later than other messages that have been waiting in the mailbox. In the case of the `dbWriter` this is not an issue, but keep this in mind when using this technique.

After the `preStart` hook is called a new instance of the actor class is created and therefore the constructor of the actor is executed, through the `Props` object. After that the `postRestart` hook is called on this fresh actor instance.

Listing 4.4 postRestart life cycle hook

```

override def postRestart(reason: Throwable) { ❶
  println("postRestart")
  super.postRestart(reason) ❷
}

```

- ❶ exception which was thrown by the actor
- ❷ WARNING call the super implementation

Here too, we start with a warning. The super implementation of the `postRestart` is called because this will trigger the `preStart` function by default. The `super.postRestart` can be omitted if you are certain that you don't want the `preStart` to be called when restarting, in most cases though this is not going to be the case. The `preStart` and `postStop` are called by default during a restart and they are called during the start and stop events in the lifecycle, so it makes sense to add code there for initialization and cleanup respectively, killing two birds with one stone.

The argument `reason` is the same as received in the `preRestart` method. In the overridden hook, the actor is free to restore itself to some last known correct state, for example by using information stored by the `preRestart` function.

4.2.4 Putting the Life cycle Pieces Together

When we put all the different events together we get the full life-cycle of an actor. In this case only one restart is shown

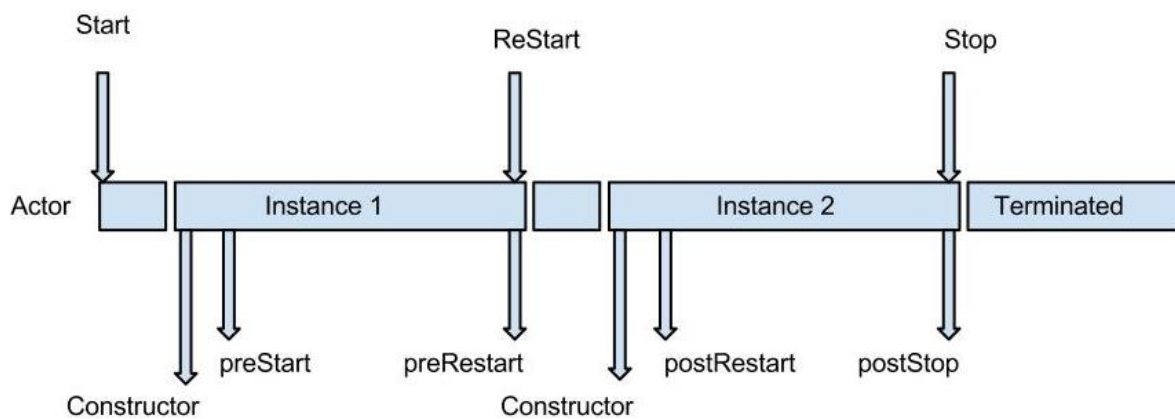


Figure 4.12 Full life cycle of an actor

Putting all the life cycle hooks together in one Actor we can see the different events occurring.

Listing 4.5 Example life cycle hooks

```
class LifecycleHooks extends Actor
with ActorLogging{
System.out.println("Constructor")
override def preStart() {println("preStart")}
override def
postStop() {println("postStop")}
override def
preRestart(reason: Throwable,
message: Option[Any]) {
println("preRestart")
super.preRestart (reason, message)
}
override def postRestart(reason: Throwable) {
println("postRestart")
super.postRestart(reason)
}
def
receive = {
case "restart" => throw
new
IllegalStateException("force restart")
case msg: AnyRef
=> println("Receive")
}
}
```

In the following 'test' we trigger all three life cycle events. The sleep just before the stop makes sure that we can see the postStop happening.

Listing 4.6 Test life cycle triggers

```
val testActorRef = system.actorOf(
  Props[LifecycleHooks], "LifecycleHooks")
testActorRef ! "restart"
testActorRef.tell("msg", testActor)
expectMsg("msg")
system.stop(testActorRef)
Thread.sleep(1000)
```

① start actor

② restart actor

③ stop actor

The result of the test is:

Listing 4.7 Output test life cycle hooks

Constructor	① Start event
preStart	①
preRestart force restart	② Restart event
postStop	②
Constructor	②
postRestart force restart	②
preStart	②
Receive	
postStop	③ Stop Event

Every actor goes through this life cycle; it is started and possibly restarted several times until the actor is stopped and terminated. The `preStart` , `preRestart` , `postRestart` and `postStop` hooks enable an actor to initialize and cleanup state and control and restore its state after a crash.

4.2.5 Monitoring the lifecycle

The lifecycle of an actor can be monitored. The lifecycle ends when the actor is terminated. An actor is terminated if the supervisor decides to stop the actor, or if the `stop` method is used to stop the actor or if a `PoisonPill` message is sent to the actor which indirectly causes the `stop` method to be called. Since the default implementation of the `preRestart` method stops all the child actors that the actor has with the `stop` methods, these children are also terminated in the case of a restart. The crashed actor instance in a restart is not terminated in this sense. It is removed from the actor system, but not by using the `stop` method, directly or indirectly. This is because the `ActorRef` will continue to live on after the restart, the actor instance has not been terminated but replaced by a new one. The `ActorContext` provides a `watch` method to monitor the death of an actor and an `unwatch` to de-register as monitor. Once an actor calls the `watch` method on an actor reference it becomes the monitor of that actor reference. A `Terminated` message is sent to the monitor actor when the monitored actor is terminated. The `Terminated` message only contains the `ActorRef` of the actor that died. The fact that the crashed actor instance in a restart is not terminated in the same way as when an actor is stopped now makes sense because otherwise you would receive many terminated messages whenever an actor restarts which would make it impossible to differentiate the final death of an actor from a temporary restart. The below example shows a `DbWatcher` actor that watches the lifecycle of a `dbWriter ActorRef`.

Listing 4.8

```
class DbWatcher(dbWriter: ActorRef) extends Actor with ActorLogging {
  context.watch(dbWriter) ❶
  def receive = {
    case Terminated(actorRef) => ❷
      log.warning("Actor {} terminated", actorRef) ❸
  }
}
```

- ❶ Watch the lifecycle of the `dbWriter`
- ❷ The `actorRef` of the terminated actor is passed in the `Terminated` message
- ❸ The watcher logs the fact that the `dbWriter` was terminated

As opposed to supervision, which is only possible from parent to child actors, monitoring can be done by any actor. As long as the actor has access to the `ActorRef` of the actor that needs to be monitored, it can simply call `context.watch(actorRef)`, after which it will receive a `Terminated` message when the actor is terminated. Monitoring and supervision can be combined as well and can be quite powerful as we will see in the next section.

We haven't discussed yet how a supervisor actually decides the fate of an actor, if the child should be terminated, restarted or stopped. This will be the main topic of the next section, where we will get into the details of supervision. In the next section we will first look at how the supervisor hierarchy is built up, followed by the strategies that a supervisor can use.

4.3 Supervision

In this section we are going to look at the details of supervision. We'll take the log processing example application and show you different types of supervision strategies. In this section we will focus on the supervisor hierarchy under the `/user` actor path which will also be referred to as the *user space*. This is where all application actors live. There is a supervisor hierarchy above the `/user` path which we will look at in chapter 4 regarding the shutdown sequence of an actor system. First we will discuss various ways to define a hierarchy of supervisors for an application and what the benefits and drawbacks are of each. Then we will look at how supervisor strategies can be customized per supervisor.

4.3.1 Supervisor hierarchy

The supervisor hierarchy is simply a function of the act of actors creating each other: every actor that creates another is the supervisor of the created child actor.

The supervision hierarchy is fixed for the lifetime of a child. Once the child is created by the parent it will fall under the supervision of that parent as long as it lives, there is no such thing as adoption in Akka. The parent terminating a child actor is the only way for the supervisor to cease its responsibilities. So it is important to choose the right supervision hierarchy from the start in your application, especially if you do not plan to terminate parts of the hierarchy to replace them with completely different subtrees of actors.

The most dangerous actors (i.e. actors that are most likely to crash) should be as low down the hierarchy as possible. Faults that occur far down the hierarchy can be

handled or escalated by more supervisors than a fault that occurs high up in the hierarchy. When a fault occurs in the top level of the actor system, it could restart all the top level actors or even shut down the actor system.

Lets look at the supervisor hierarchy of the log processing application as we intended in the previous section. The below figure shows how the supervisors and actors in the system communicate with each other and how the message flow from a new file to a stored row in the database takes place:

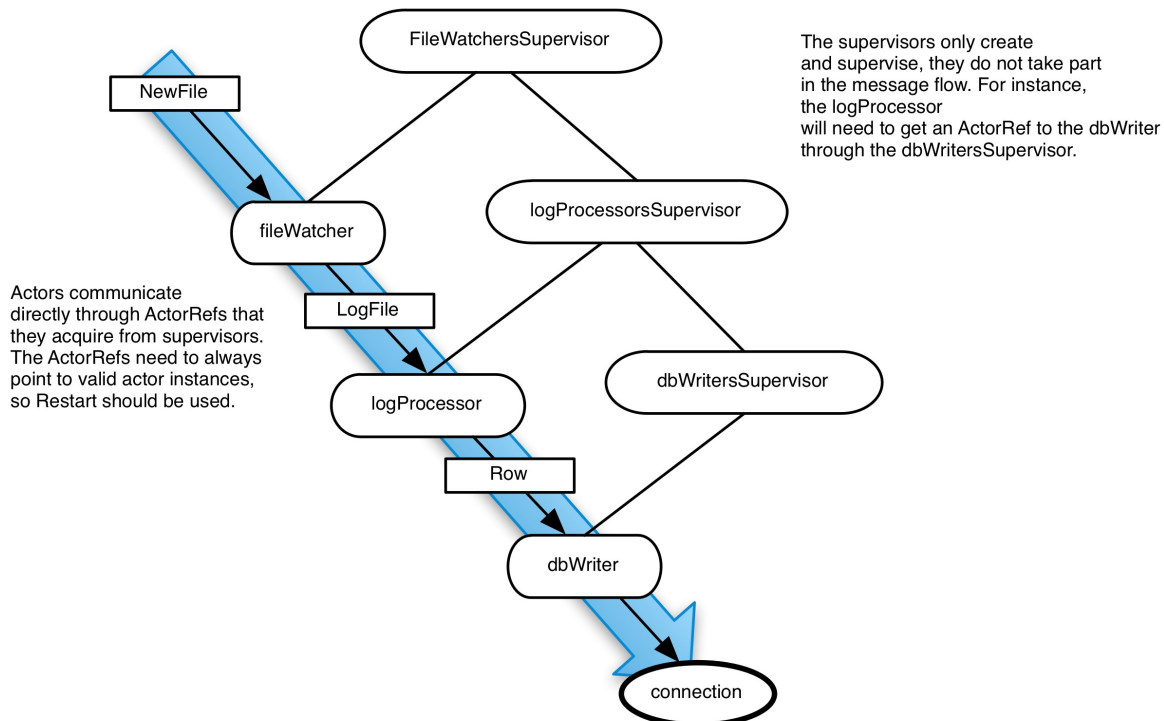


Figure 4.13 Message flow and supervisors separate

In this setup we connect the actors directly to each other using ActorRefs. Every actor knows the ActorRef of the next actor it sends messages to. The ActorRefs need to stay alive and always need to refer to a next actor instance. If an actor instance were to be stopped, the ActorRef would refer to the system's deadLetters which would break the application. A Restart will need to be used in all cases in the supervisor because of this so that the same ActorRef can be reused at all times.

The picture also shows that the fileWatcherSupervisor needs to both create the fileWatcher and the logProcessorsSupervisor. The fileWatcher needs a direct reference to the logProcessor but the logProcessor is created by the logProcessorsSupervisor at some point, so we can't just pass a logProcessor

ActorRef to the fileWatcher. We will have to add some messages that we can send to the logProcessorsSupervisor to request an ActorRef to the logProcessor so that we can give it to the fileWatcher. The benefit of this approach is that the actors talk to each other directly and the supervisors only supervise and create instances. The drawback is that we can only use Restart because otherwise messages will be sent to the deadLetters and get lost. Parental supervision makes decoupling the supervision from the message flow a bit harder. The logProcessor has to be created by the logProcessorsSupervisor, which makes it more difficult to pass the logProcessor ActorRef to the fileWatcher which in turn is created by the fileWatcherSupervisor.

The next picture shows a different approach. The supervisors don't merely create and supervise the actors in this case, they also forward all messages they receive to their children. From the perspective of the supervised children nothing has changed, forwarding is transparent because the original sender of the message is preserved. For instance the logProcessor will think that it received a message directly from the fileWatcher while in fact the logProcessorSupervisor forwarded the message.

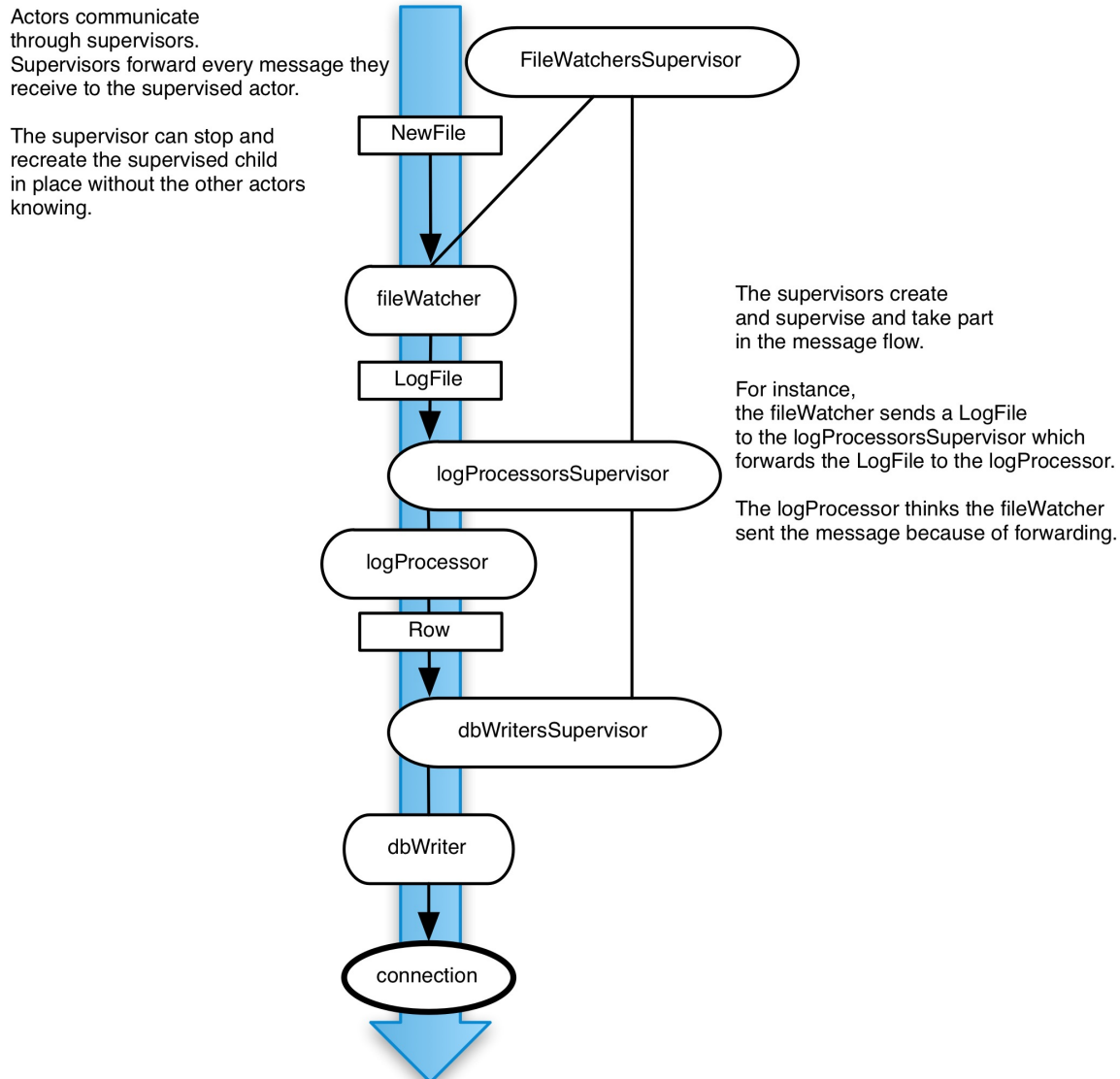


Figure 4.14 Supervisors forward messages in the message flow

One benefit of this approach is that the supervisors provide a level of indirection. A supervisor can terminate its children and spawn new ones without the other actors knowing about it. Compared to the previous approach this does not cause a gap in the message flow. This setup fits better in parental supervision because a supervisor can directly create its child actors using Props objects. The fileWatcherSupervisor for instance can create the logProcessingSupervisor and use its ActorRef when creating a fileWatcher. The forwarding of all messages to the children is how this approach works: the Actors continue to just send and receive messages, completely oblivious of the faults occurring.

The forwarding supervisors approach is more flexible and simpler than the

more separated approach because it fits in better with the idea of parental supervision. The following example shows how the hierarchy is built up in an application. In the next section we will look at the supervisors and the strategies they use in detail.

Listing 4.9 Build the supervisor hierarchy

```
object LogProcessingApp extends App {
  val sources = Vector("file:///source1/", "file:///source2/")
  val system = ActorSystem("logprocessing")
  // create the props and dependencies
  val con = new DbCon("http://mydatabase")
  val writerProps = Props(new DbWriter(con))
  val dbSuperProps = Props(new DbSupervisor(writerProps))
  val logProcSuperProps = Props(
    new LogProcSupervisor(dbSuperProps))
  val topLevelProps = Props(new FileWatchingSupervisor(
    sources,
    logProcSuperProps))
  system.actorOf(topLevelProps) ❶
}
```

- ❶ The top level supervisor for the log processing application is created. All actors lower in the hierarchy will be created by the supervisors below this actor using the Props objects that have been passed in.

The above code shows how the log processing application is built up. Props objects are passed as recipes to the actors so that they can create their children without knowing the details of the dependencies of the child actors. Eventually only one top level actor is created using `system.actorOf` because we want the rest of the actors in the system to be created by supervisors, so all other actors are created further down the line. In the next section we will revisit each actor and we will see how they exactly create their children, but in essence they simply use the Props object that they get passed and call `context.actorOf(props)` to build a child actor, making it the supervisor of each new instance.

Now that we know a bit more about how to structure the supervision hierarchy of an application, let's look at the different supervisor strategies that are available in the next section.

4.3.2 Predefined strategies

The top level actors in an application are created under the `/user` path and supervised by the `user guardian`. The default supervision strategy for the user guardian is to restart its children on any `Exception` except for when it receives internal exceptions that indicate that the actor was killed or when it failed during initialization, at which point it will stop the actor in question. This strategy is known as the `default strategy`. The strategy for the top level guardians can be changed which we will look at in chapter 4. Every actor has a default supervisor strategy which can be overridden by implementing the `supervisorStrategy` method. There are two predefined strategies available in the `SupervisorStrategy` object; the `defaultStrategy` and the `stoppingStrategy`. As the name implies, the default strategy is default for all actors, if you do not override the strategy an actor will always use the default. The default strategy is defined as follows in the `SupervisorStrategy` object:

Listing 4.10 Default Supervisor Strategy

```
final val defaultStrategy: SupervisorStrategy = {
  def
  defaultDecider: Decider = { //
    ①
  case _: ActorInitializationException => Stop //
    ②
  case _: ActorKilledException => Stop
  case _: Exception =>
  Restart
  }
  OneForOneStrategy()(defaultDecider) //
    ③
}
```

- ① The Decider chooses a Directive by pattern matching on the exceptions
- ② Stop, Start, Resume and Escalate are called Directives
- ③ A OneForOneStrategy is returned that uses the defaultDecider

The above code uses the `OneForOneStrategy` which we have not discussed yet. Akka allows you to make a decision about the fate of the child actors in two ways: all children share the same fate and the same recovery is applied to the lot, or a decision is rendered and the remedy is applied only to the crashed actor. In some cases you might want to only stop the child actor that failed. In other cases

you might want to stop all child actors if one of them fails, maybe because they all depend on a particular resource. If an exception is thrown that indicates that the shared resource has failed completely it might be better to immediately stop all child actors together instead of waiting for this to happen individually for every child. The `OneForOneStrategy` determines that child actors will not share the same fate, only the crashed child will be decided upon by the `Decider`. The other option is to use an `AllForOneStrategy` which uses the same decision for all child actors even if only one crashed. The next section will describe the `OneForOneStrategy` and `AllForOneStrategy` in more detail. The below example shows the definition of the `stoppingStrategy` which is defined in the `SupervisorStrategy` object:

Listing 4.11 Stopping Supervisor Strategy

```
final val stoppingStrategy: SupervisorStrategy = {
  def
  stoppingDecider: Decider = {
    case _: Exception => Stop //
    ❶
  }
  OneForOneStrategy()(stoppingDecider)
}
```

- ❶ Decides to stop on any `Exception`.

The stopping strategy will stop any child that crashes on any `Exception`. These built in strategies are nothing out of the ordinary. They are defined in the same way you could define a supervisor strategy yourself. So what happens if an `Error` is thrown, like a `ThreadDeath` or an `OutOfMemoryError` by an actor that is supervised using the above `stoppingStrategy`? Any `Throwable` that is not handled by the supervisor strategy will be escalated to the parent of the supervisor. If a fatal error reaches all the way up to the user guardian, the user guardian will not handle it since the user guardian uses the default strategy. In that case an uncaught exception handler in the actor system causes the actor system to shutdown. As we will see later in chapter 4 you can configure to exit the JVM when this happens or to just shutdown the actor system. In most cases it is good practice not to handle fatal errors in supervisors but instead gracefully shutdown the actor system since a fatal error cannot be recovered from.

4.3.3 Custom Strategies

Each application is going to have to craft strategies for each case that requires fault tolerance. As we have seen in the previous sections there are four different types of actions a supervisor can take to resolve a crashed actor. These are the building blocks we will use. In this section, we will return to the log processing and build the specific strategies it requires from these elements:

1. *Resume* the child, ignore errors and keep processing with the same actor instance
2. *Restart* the child, remove the crashed actor instance and replace it with a fresh actor instance
3. *Stop* the child, terminate the child permanently
4. *Escalate* the failure and let the parent actor decide what action needs to be taken

First we'll look at the exceptions that can occur in the log processing application. To simplify the example a couple of custom exceptions are defined:

Listing 4.12 Exceptions in the log processing application

```
@SerialVersionUID(1L)
class DiskError(msg: String)
    extends Error(msg) with Serializable ❶

@SerialVersionUID(1L)
class CorruptedFileException(msg: String, val file: File)
    extends Exception(msg) with Serializable ❷

@SerialVersionUID(1L)
class DbBrokenConnectionException(msg: String)
    extends Exception(msg) with Serializable ❸
```

- ❶ An unrecoverable Error that occurs when the disk for the source has crashed
- ❷ An Exception that occurs when the log file is corrupt and cannot be processed.
- ❸ An Exception that occurs when the database connection is broken.

The messages that the actors send to each other in the log processing application are kept together in a protocol object:

Listing 4.13 Log processing protocol

```
object LogProcessingProtocol {
  // represents a new log file
  case class LogFile(file: File) ❶
  // A line in the log file parsed by the LogProcessor Actor
  case class Line(time: Long, message: String, messageType: String) ❷
}
```

- ❶ The log file that is received from the FileWatcher. The log Processor will process these.
- ❷ A processed line in the LogFile. The database writer will write these to a database connection.

First lets start at the bottom of the hierarchy and look at the database writer that can crash on a `DbBrokenConnectionException` . When this exception happens the `dbWriter` should be restarted.

Listing 4.14 DbWriter crash

```
class DbWriter(connection: DbCon) extends Actor {
  import LogProcessingProtocol._
  def receive = {
    case Line(time, message, messageType) =>
      connection.write(Map('time -> time,
        'message -> message,
        'messageType -> messageType)) ❶
  }
}
```

- ❶ Writing to the connection could crash the actor.

The `DbWriter` is supervised by the `DbSupervisor`. The supervisor will forward all messages to the `DbWriter` as we discussed in the supervisor hierarchy section.

Listing 4.15 DbWriter supervisor

```
class DbSupervisor(writerProps: Props) extends Actor {
  override def supervisorStrategy = OneForOneStrategy() {
    case _: DbBrokenConnectionException => Restart ❶
  }
  val writer = context.actorOf(writerProps) ❷
  def receive = {
    case m => writer forward (m) ❸
  }
}
```

- ❶ Restart on DbBrokenConnectionException.
- ❷ The dbWriter is created by the supervisor from a Props object.
- ❸ All messages to the supervisor are forwarded to the dbWriter ActorRef.

If the database connection is broken, the database writer will be recreated with a new connection from the Props object. The line that was being processed when the DbBrokenConnectionException crashed the actor is lost. We will look at a solution for this later in this section. The next actor up the hierarchy in the logs application is the logProcessor. Let's look at how this actor is supervised. The below example shows the LogProcessor actor:

Listing 4.16 LogProcessor crash

```
class LogProcessor(dbSupervisor: ActorRef)
  extends Actor with LogParsing {
  import LogProcessingProtocol._
  def receive = {
    case LogFile(file) =>
      val lines = parse(file) ❶
      lines.foreach(dbSupervisor ! _) ❷
  }
}
```

- ❶ Parsing the file could crash the actor.
- ❷ Send the parsed lines to the dbSupervisor which in turn will forward the message to the dbWriter.

The logProcessor crashes when a corrupt file is detected. In that case we do not want to process the file any further, thus we ignore it. The logProcessor supervisor

resumes the crashed actor:

Listing 4.17 LogProcessor supervisor

```
class LogProcSupervisor(dbSupervisorProps: Props)
  extends Actor {
    override def supervisorStrategy = OneForOneStrategy() {
      case _: CorruptedFileException => Resume ❶
    }
    val dbSupervisor = context.actorOf(dbSupervisorProps) ❸
    val logProcProps = Props(new LogProcessor(dbSupervisor))
    val logProcessor = context.actorOf(logProcProps) ❷
    def receive = {
      case m => logProcessor forward (m) ❹
    }
  }
}
```

- ❶ Resume on `CorruptedFileException`. the Corrupted file is ignored.
- ❷ The `logProcessor` is created by the supervisor from a `Props` object.
- ❸ The database supervisor is created and supervised by this supervisor.
- ❹ All messages to the supervisor are forwarded to the `logProcessor ActorRef`.

The log processing supervisor does not need to know anything about the database supervisor dependencies, it simply uses the `Props` objects. If the log processing supervisor decides on a restart, it can just recreate the actors from their respective `Props` objects.

Up to the next actor in the hierarchy, the `FileWatcher`:

Listing 4.18 FileWatcher crash

```

class FileWatcher(sourceUri: String,
                  logProcSupervisor: ActorRef)
  extends Actor with FileWatchingAbilities {
  register(sourceUri) 1
  import FileWatcherProtocol._
  import LogProcessingProtocol._
  def receive = {
    case NewFile(file, _) => 2
      logProcSupervisor ! LogFile(file) 3
    case SourceAbandoned(uri) if uri == sourceUri =>
      self ! PoisonPill 4
  }
}

```

- 1** Registers on a source uri in the file watching API.
- 2** Sent by the file watching API when a new file is encountered.
- 3** Forwarding to the log processing supervisor.
- 4** The filewatcher kills itself when the source has been abandoned, indicated by the file watching API to not expect more new files from the source.

We'll not get into the details of the File watching API, it is provided in a FileWatchingAbilities trait. The FileWatcher does not take any 'dangerous' actions and will continue to run until the file watching API notifies the FileWatcher that the source of files is abandoned. The FileWatchingSupervisor monitors the FileWatchers for termination and it also handles the DiskError error that could have happened at any point lower in the supervisor hierarchy. Since the DiskError is not defined lower down the hierarchy it will automatically be escalated. Since this is an unrecoverable error, the FileWatchingSupervisor decides to stop all the actors in the hierarchy when this occurs. An AllForOneStrategy is used so that if any of the file watchers crashes with a DiskError all file watchers are stopped:

Listing 4.19 FileWatcher supervisor

```

class FileWatchingSupervisor(sources: Vector[String],
                             logProcSuperProps: Props)
  extends Actor {
  var fileWatchers: Vector[ActorRef] = sources.map { source =>
    val logProcSupervisor = context.actorOf(logProcSuperProps)
    val fileWatcher = context.actorOf(Props(
      new FileWatcher(source, logProcSupervisor))
    context.watch(fileWatcher) ❶
    fileWatcher
  }
  override def supervisorStrategy = AllForOneStrategy() {
    case _: DiskError => Stop ❷
  }
  def receive = {
    case Terminated(fileWatcher) => ❸
      fileWatchers = fileWatchers.filterNot(w => w == fileWatcher)
      if (fileWatchers.isEmpty) self ! PoisonPill ❹
  }
}

```

- ❶ Watch the file watchers for termination.
- ❷ Stop the file watchers on a `DiskError`.
- ❸ The `Terminated` message is received for a file watcher.
- ❹ When all file watchers are terminated the supervisor kills itself.

The `OneForOneStrategy` and `AllForOneStrategy` will continue indefinitely by default. Both strategies have default values for the constructor arguments `maxNrOfRetries` and `withinTimeRange`. In some cases you might like the strategy to stop after a number of retries or when a certain amount of time has passed. Simply set these arguments to the desired values. Once configured with the constraints, the fault is escalated if the crash is not solved within the time range specified or within a maximum number of retries. The below code gives an example of an impatient database supervisor:

Listing 4.20 Impatient database supervisor

```
class DbImpatientSupervisor(writerProps: Props) extends Actor {
  override def supervisorStrategy = OneForOneStrategy(
    maxNrOfRetries = 5,
    withinTimeRange = 60 seconds) { 1
    case _: DbBrokenConnectionException => Restart
  }
  val writer = context.actorOf(writerProps)
  def receive = {
    case m => writer forward (m)
  }
}
```

- 1** Escalate the issue if the problem has not been resolved within 60 seconds or it has failed to be solved within 5 restarts.

This mechanism can be used to prevent an actor from continuously restarting without any effect.

4.4 Summary

Fault tolerance is one of the most exciting aspects of Akka, and it's a critical component in the toolkit's approach to concurrency. The philosophy of 'Let it Crash' is not a doctrine of ignore the possible malfunctions that might occur, or the toolkit will swoop in and heal any faults, in fact, it's sort of the opposite: the programmer needs to anticipate recovery requirements, but the tools to deliver them without meeting a catastrophic end (or having to write a ton of code) are simply unparalleled. In the course of making our example log processor fault tolerant, we saw that:

- Supervision means we have a clean separation of recovery code
- The Actor model's being built on messages means that even when an actor goes away, we can still continue to function
- We can resume, abandon, restart: the choice is ours, given the requirements in each case
- We can even escalate through the hierarchy of Supervisors

Again, Akka's philosophy shines through here: pull the actual operational needs of the application up into the code, but do it in a structured way, with support from

the toolkit. The result is that sophisticated fault tolerance that would be either extremely difficult to achieve just running in a VM can be built and tested while the code is being written without a tremendous amount of extra effort.

Now that we know how Akka can help us implement functionality in a concurrent system by using actors and how to deal with errors within these actors, we can start building an application. In the next section we will build several different types of actor-based applications, and will look at how to provide services like configuration, logging, and deployment.

5 *Futures*

In this chapter

- Futures
- Composing Futures
- Recovering from Errors inside Futures
- Futures and Actors

In this chapter we're going to introduce Futures. The Future type that is explained in this chapter was initially part of the Akka toolkit. Since Scala 2.10 it is part of the standard Scala distribution. Like Actors, Futures are an important asynchronous building block in the Akka toolkit. Both Actors and Futures are great tools best used for different use cases. It's simply a case of the right tool for the right job. We will start with describing the type of use case that Futures are best suited for and work through some examples. Where Actors provide a mechanism to build a system out of concurrent *objects*, Futures provide a mechanism to build a system out of concurrent *functions*.

Futures make it possible to combine the results of functions without ever blocking or waiting. Exactly how you can achieve this will become clear in this chapter. We will focus on showing you examples of how to best use Futures instead of diving into the type system details that make many of the features of the Future type possible.

You don't have to choose for Futures or Actors, they can be used together. Akka provides common Actor and Future patterns which make it easy to work with both. You will learn about the pitfalls that you need to be aware of when using them together in the last section of this chapter.

5.1 The use case for Futures

In the chapters so far we've learned a lot about Actors. To contrast the best use cases for futures we will briefly think about use cases that *can* be implemented with Actors, but not without unwanted complexity.

Actors are great for processing many messages, capturing state and reacting with different behavior based on the messages they receive. They are resilient *objects* that can live on for a long time even when problems occur, using monitoring and supervision.

Futures are the tool to use when you would rather use *functions* and don't really need objects to do the job. A *Future* is a placeholder for a function result that will be available at some point in the future. It is effectively an asynchronous result. It gives us a way to reference the result that will eventually become available. The below figure shows the concept:

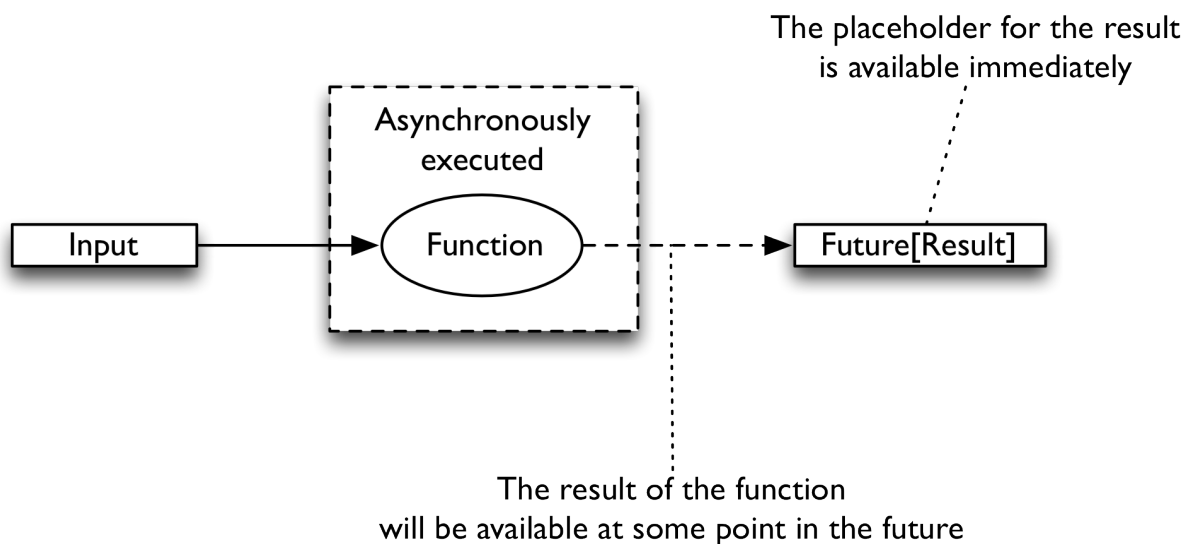


Figure 5.1 A placeholder for an asynchronous function result

A Future is a read-only placeholder. It can not be changed from the outside. A Future will contain a successful result or a failure once the asynchronous function is completed. After completion the result inside the Future cannot change and can be read many times, it will always give the same result. Having a placeholder for the result makes it easier to combine many functions that are executed asynchronously. It gives us a way to for instance call a web service without blocking the current thread and process the response at a later time.

NOTE**This is not POJF (Plain Old Java Future)**

To prevent any confusion, if you are familiar with the `java.util.concurrent.Future` class in Java 7 you might think that the `scala.concurrent.Future` discussed in this chapter is just a Scala wrapper around this Java class. This is not the case. The `java.util.concurrent.Future` class requires polling and only provides a way to get to the result with a blocking `get` method, while the Scala Future makes it possible to combine function results without blocking or polling as you will learn in this chapter.

To make the example more concrete we're going to look at another use case for the ticket system. We would like to create a web page with extra information about the event and the venue. The ticket would simply link to this web page so that customers can access it from their mobile device for instance. We might want to show a weather forecast for the venue when it is on open air event, route planning to the event around the time of the event (should I take public transport or drive by car?), where to park, or show suggestions for similar future events that the customer might be interested in. Futures are especially handy for *pipelining*, where one function provides the input for a next function. The `TicketInfo` service will find related information for an event based on the ticket number. In all these cases the services that provide a part of the information might be down and we don't want to block on every service request while aggregating the information. If services do not respond in time or fail, their information should just not be shown. To be able to show the route to the event we will first need to find the event using the ticket number, which is shown in the below figure.



Figure 5.2 Chain asynchronous functions

In this case `getEvent` and `getTraffic` are both functions that do asynchronous web service calls, executed one after the other. The `getTrafficInfo` web service call takes an `Event` argument. `getTrafficInfo` is called the moment the event becomes available in the `Future[Event]` result. This is very different from calling the `getEvent` method and polling and waiting for the event on the current thread. We simply

define a flow and the `getTrafficInfo` function will be called *eventually*, without polling or waiting on a thread. The functions execute as soon as they can. Limiting the amount of waiting threads is obviously *a good thing* because they should instead be doing something useful.

The below figure shows a simple example where calling services asynchronously is ideal. It shows a mobile device calling the TicketInfo service which in the below case aggregates information from a weather and traffic service:

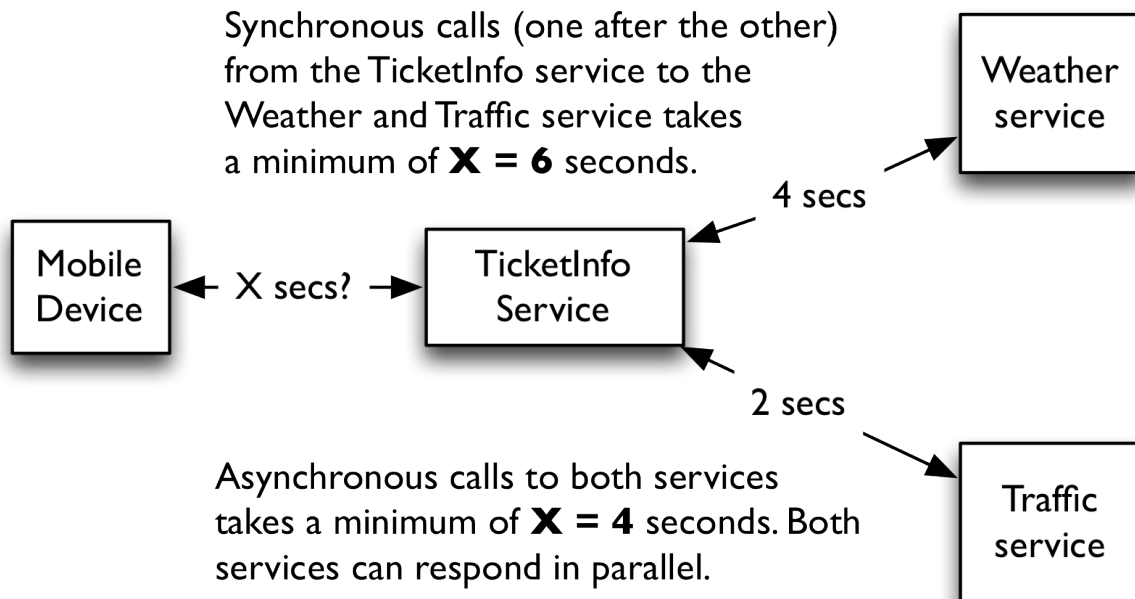


Figure 5.3 Aggregating results, sync vs async

Not having to wait for the weather service before calling the traffic service decreases the latency of the mobile device request. The more services need to be called the more dramatic the effect on latency will be since the responses can be processed in parallel. The next figure shows another use case. In this case we would like the fastest result of two competing weather services:

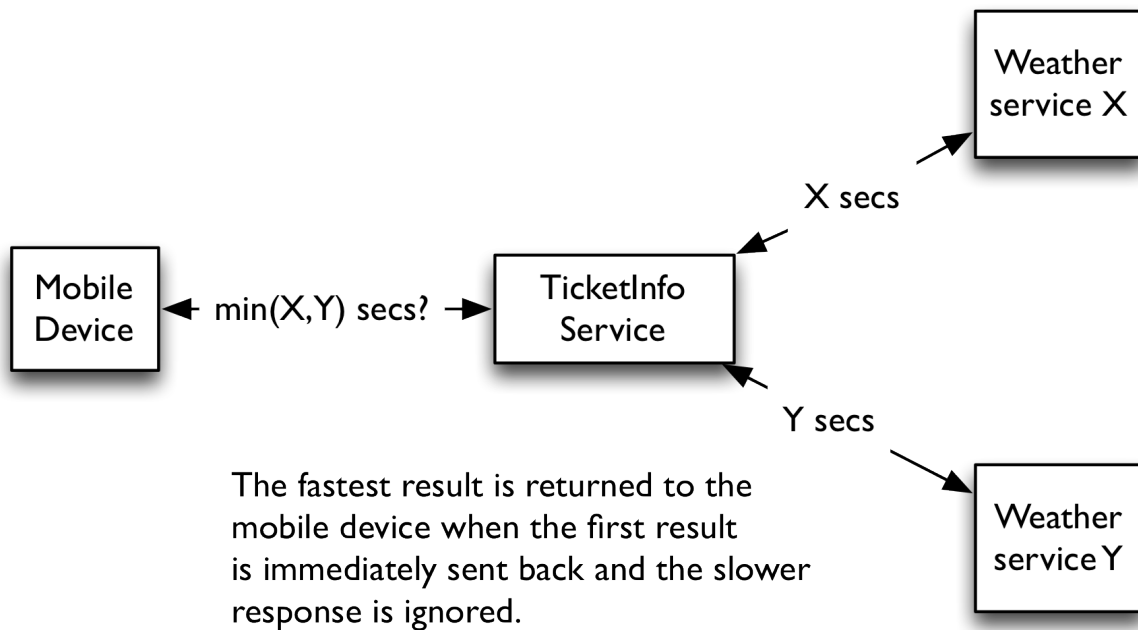
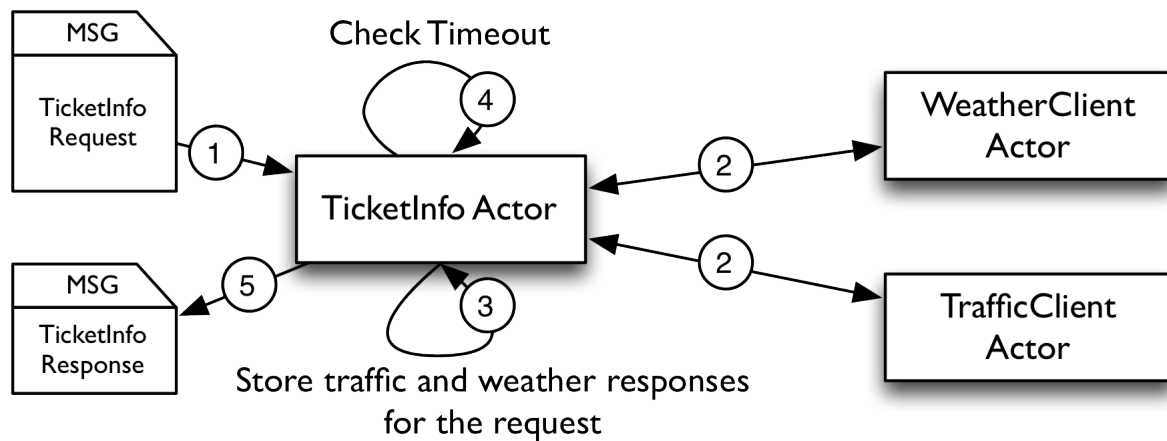


Figure 5.4 Respond with the fastest result

Maybe weather service X is malfunctioning and times out on the request. In that case you would not want to wait for this timeout but rather use the fast response of weather service Y which is working as expected.

It's not as if these scenarios are impossible to execute with Actors. It's just that we would have to do a lot of work for such a simple use case. Take the example of aggregating weather and traffic information. Actors have to be created, messages defined, receive functions implemented, as part of an ActorSystem. You would have to think about how to handle timeouts, when to stop the actors and how to create new actors for every web page request and combine the responses. The below figure shows how Actors could be used to do this.



- 1: Create TicketInfo Actor and send request
 - 2: create child actors and send request, correlate request with responses
 - 3: Store the responses of the child actors
 - 4: Send scheduled timeout message to TicketInfo Actor in case one of the child actors does not respond.
 - 5: Send back aggregated information response at timeout or when both responses have been received.
 6. Stop TicketInfo Actor and child Actors after sending.
- We only want to do this once per request.

Figure 5.5 Combine web service requests with Actors

We need two separate actors for the weather and traffic web service calls so that they can be called in parallel. How the web service calls are combined will need to be coded in the TicketInfo Actor for every specific case. That's a lot of work for just calling two web services and combining the results. Note however that actors are a better choice when fine-grained control over state is required or when actions need to be monitored or possibly retried.

So although actors are a great tool they are not the 'be all and end all' on our quest to never block again. In this case a tool specifically made for combining function results would be a lot simpler.

There are some variations on the above use case where futures are the best tool for the job. In general the use cases have one or more of the following characteristics:

- You do not want to block (wait on the current thread) to handle the result of a function.
- Call a function once-off and handle the result at some point in the future.
- Combine many once-off functions and combine the results.
- Call many competing functions and only use some of the results, for instance only the fastest response.
- Call a function and return a default result when the function throws an exception so the flow can continue.

- Pipeline these kind of functions, where one function depends on or more results of other functions.

In the next sections we're going to look at the details of implementing TicketInfo service with futures. We will start with just calling one web service asynchronously, after that we will combine many services in a pipeline of futures and we'll look at error handling.

5.2 In the Future nobody blocks

It's time to build the TicketInfoService and we're not explicitly going to sit on any thread waiting idly by. Unit testing is the only case where blocking on a future would be valid in our opinion because it can simplify validating results. And even then delaying any blocking all the way to the end of the test case is preferred. We're going to start with the TicketInfo service and try to execute the two steps in the below figure so that we can provide traffic information about the route to the event.

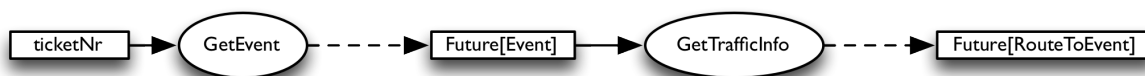


Figure 5.6 Get traffic information about the event

The first step is to get the event for the ticket number. The big difference between calling a function synchronously and asynchronously is the flow in which you define your program. The below listing shows an example of a synchronous web service call to get the event for the ticket number.

Listing 5.1 Synchronous call

```

val request = EventRequest(ticketNr) ①
val response:EventResponse = callEventService(request) ②
val event:Event = response.event ③
  
```

- ① Create the request
- ② Block main thread until the response is completed
- ③ Read the event value

The figure shows 3 lines of code executed on the main thread. The flow is very simple, a function is called and its return value is immediately accessible on the

same thread. The program obviously can't continue on the same thread before the value is accessible. Scala expressions are by default strict and the above code does not use any lazy evaluation. So every line in the above code has to produce a complete value.

Lets see what we need to change to this synchronous web service call into an asynchronous one. In the above case the `callEventService` is a blocking call to a web service, it needs to wait on a thread for the response. We'll first wrap the `callEventService` into a code block that is executed on a separate thread. The below figure shows the change in the code:

Listing 5.2 Asynchronous call

```
val request = EventRequest(ticketNr)
val futureEvent:Future[Event] = future {
    val response = callEventService(request)
    response.event
}
```

- ❶ Call code block asynchronously and return a Future Event result
- ❷ this is run on a separate thread.
- ❸ The event in the response can be accessed on the separate thread.

This asynchronous flow makes a small adjustment, it runs the `callEventService` in a separate thread. The separate thread is blocked by the `callEventService`, something that will be fixed later. The `future { ... }` is shorthand for a call to the `apply` method on the `Future` object with the code block as its only argument, `Future.apply(codeblock)`. This method is in the `scala.concurrent` package object so you only need to import `scala.concurrent._` to use it. It is a helper function to asynchronously call a 'code block' and get back a `Future[T]`, in this case a `Future[Event]`, because we need the `Event` for the next call to get traffic information. In case you are new to Scala, the last expression in a block is automatically the return value. The `Future.apply` method ensures that the last expression is turned into a `Future[Event]`. The type of the `futureEvent` value is explicitly defined for this example but can be omitted because of type inference in Scala.

The above code block is actually a *Closure*. A Closure is a special type of

function that can use values from its enclosing scope. You don't have to do anything special in Scala to create a closure, but it is important to note that this is a special type of function. The code block refers to the `request` value from the other thread, which is how we bridge between the main thread and the other thread and pass the request to the web service call.

Great, the web service is now called on a separate thread and we could handle the response right there. Lets see how we can chain the call to `callTrafficService` to get the traffic information for the event in the listing below. As a first step we will print the route to the event to the console:

Listing 5.3 Handling the event result

```
futureEvent.foreach { event => ❶
  val trafficRequest = TrafficRequest(destination = event.location,
                                     arrivalTime = event.time)
  val trafficResponse = callTrafficService(trafficRequest) ❷
  println(trafficResponse.route) ❸
}
```

- ❶ Asynchronously process the event result when it becomes available.
- ❷ Call the traffic service with a request based on the event.
- ❸ Print the route to the console.

The above listing uses the `foreach` method on `Future`, which calls the code block with the event result when it becomes available. The code block is only called when the `callEventService` is successful.

In this case we are expecting to use the `Route` later on as well, so it would be better if we could return a `Future[Route]`. The `foreach` method returns `Unit` so we'll have to use something else. The below listing shows how this is done with the `map` method.

Listing 5.4 Chaining the event result

```

val futureRoute:Future[Route] = futureEvent.map { event => ❶
    val trafficRequest = TrafficRequest(destination = event.location,
                                       arrivalTime = event.time)
    val trafficResponse = callTrafficService(trafficRequest)
    trafficResponse.route ❷
}

```

- ❶ handle the event and return a Future[Route]
- ❷ return the value to the map function which turns it into a Future[Route].

Both `foreach` and `map` should be familiar to you from using the `scala.collections` library and standard types like `Option` and `Either`. Conceptually the `Future.map` method is very similar to for instance `Option.map`. Where the `Option.map` method calls a code block if it contains some value and returns a new `Option[T]` value, the `Future.map` method eventually calls a code block when it contains a successful result and returns a new `Future[T]` value. In this case a `Future[Route]` because the last line in the code block returns a `Route` value. Once again the type of `futureRoute` is explicitly defined which can be omitted. The below code shows how you can chain both web service calls directly.

Listing 5.5 getRoute method with Future[Route] result

```

val request = EventRequest(ticketNr)

val futureRoute = future {
    callEventService(request).event
}.map { event => ❶

    val trafficRequest = TrafficRequest(destination = event.location,
                                       arrivalTime = event.time)

    callTrafficService(trafficRequest).route ❷
}

```

- ❶ Chain on the Future[Event]
- ❷ Return the route

If we refactor into a `getEvent` method that takes a `ticketNr` and a `getRoute` method that takes an event argument the code in the following listing would chain the two calls. The methods `getEvent` and `getRoute` respectively return a `Future[Event]` and `Future[Route]`.

Listing 5.6 Refactored version

```
val futureRoute = getEvent(ticketNr).map { event =>
  getRoute(event)
}
```

The `callEventService` and `callTrafficService` methods in the previous examples were blocking calls to show the transition from a synchronous to an asynchronous call. To really benefit from the asynchronous style the above `getEvent` and `getRoute` should be implemented with a non-blocking I/O API and return futures directly to minimize the amount of blocking threads. An example of such an API for HTTP is the `spray-client` library which is built on top of Akka actors and the *Java NIO library* which makes use of low-level OS facilities like *selectors* and *channels*. In the next sections you can assume that the web service calls are implemented with `spray-client`, which will be covered in the REST chapter [TODO correct reference].

A detail that has been omitted so far is that you need to provide an *implicit* `ExecutionContext` to use futures. If you do not provide this your code will not compile. The below snippet shows how you can import an implicit value for the global execution context.

Listing 5.7 Handling the event result

```
import scala.concurrent.Implicits.global
```

❶

❶ Use the global `ExecutionContext`

The `ExecutionContext` is an abstraction for executing tasks on some thread pooling implementation. If you are familiar with the `java.util.concurrent` package, it can be compared to a `java.util.concurrent.Executor` interface with extras. The global

ExecutionContext uses a *ForkJoinPool* if it is allowed according to security policy and otherwise falls back to a *ThreadPoolExecutor*. In the section *Futures and Actors* we will see that the dispatcher of an actor system can be used as an ExecutionContext as well.

So far we've only looked at chaining successful function results. The next section is going to show you how you can recover from error results.

5.3 Futuristic Errors

The future results in the previous section were expected to always succeed. Lets look at what happens if an `Exception` is thrown during the asynchronous execution. Start up a scala REPL session on the command line and follow along with the below listing:

Listing 5.8 Throwing an exception from the Future

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import scala.concurrent._
import ExecutionContext.Implicits.global

val futureFail = future { throw new Exception("error!") }
futureFail.foreach(value=> println(value)) ❶

// Exiting paste mode, now interpreting.

futureFail: scala.concurrent.Future[Nothing] =
  scala.concurrent.impl.Promise$DefaultPromise@193cd8e1

scala> ❷
```

- ❶ Try to print the value once the Future has completed
- ❷ Nothing gets printed since an Exception occurred

The above listing throws an `Exception` from another thread. The first thing you notice is that you do not see a stacktrace in the console which you would have seen if the exception was thrown directly. The `foreach` block is not executed. This is because the future is not completed with a successful value. One of the ways to get to the failure value is to use the `onComplete` method. This method also takes a

function like `foreach` and `map` but in this case it provides a `scala.util.Try` value to the function. The `Try` can be a `Success` or a `Failure` value. The below REPL session shows how it can be used to print the exception.

Listing 5.9 Using `onComplete` to handle Success and Failure

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import scala.util._ 1
import scala.util.control.NonFatal 2
import scala.concurrent._
import ExecutionContext.Implicits.global

val futureFail = future { throw new Exception("error!") }
futureFail.onComplete {
  case Success(value) => println(value)
  case Failure(NonFatal(e)) => println(e) 4
}

// Exiting paste mode, now interpreting.

java.lang.Exception: error! 5
```

- 1 Import statement for `Try`, `Success` and `Failure`
- 2 It is good practice to only catch non-fatal errors
- 3 Print the successful value
- 4 Print the non-fatal exception
- 5 The exception is printed

The `onComplete` method makes it possible to handle the success or failure result. Take note in the above example that the `onComplete` callback is executed even if the future has already finished, which is quite possible in the above case since a exception is directly thrown in the future block. This is true for all functions that are registered on a future.

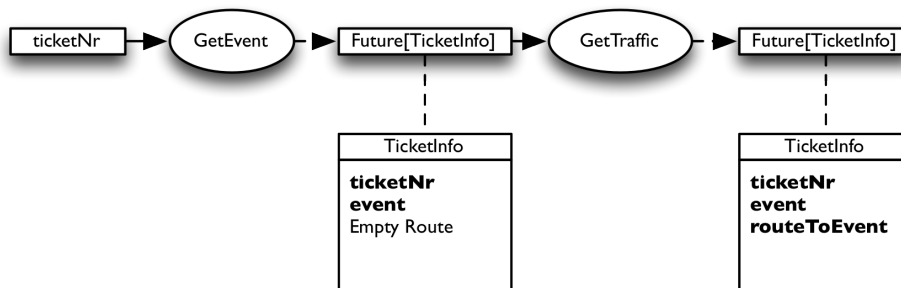
The `onComplete` method returns `Unit` so we cannot chain to a next function. Similarly there is a `onFailure` method which makes it possible to match exceptions. `onFailure` also returns `Unit` so we can't use it for further chaining. The below listing shows the use of `onFailure`.

Listing 5.10 Using onFailure to match on all non-fatal Exceptions

```
futureFail.onFailure { 1
  case NonFatal(e) => println(e) 2
}
```

- 1** Called when the function has failed
- 2** Match on all non-fatal exception types

We'll need to be able to continue accumulating information in the `TicketInfo` service when exceptions occur. The `TicketInfo` service aggregates information about the event and should be able to leave out parts of the information if the required service threw an exception. The below figure shows how the information around the event is going to be accumulated in a `TicketInfo` class for a part of the flow of the `TicketInfo` service.



Every step adds information to the `TicketInfo` contained in the `Future`.
 The route is initially empty.
 (**bold** means the element has been added to the `TicketInfo` value).

Figure 5.7 Accumulate information about the event in the `TicketInfo` class

The `getEvent` and `getTraffic` methods are modified to return a `TicketInfo` value (inside a `Future`) which will be used to accumulate information further down the chain. The `TicketInfo` class is a simple case class that contains optional values for the service results. The below listing shows the `TicketInfo` case class. In the next sections we'll add more information to this class like a weather forecast and suggestions for other events.

Listing 5.11 TicketInfo case class

```
case class TicketInfo(ticketNr:String, event:Option[Event]=None,
                    route:Option[Route]=None
                    ) ❶
```

- ❶ All extra information about the ticketNr is optional and default empty

It's important to note that you should always use immutable data structures when working with futures. Otherwise it would be possible to share mutable state between futures that possibly use the same objects. We're safe here since we're using case classes and Options which are immutable. When a service call fails the chain should continue with the TicketInfo that it had accumulated so far. The below figure shows how a failed GetTraffic call should be handled.

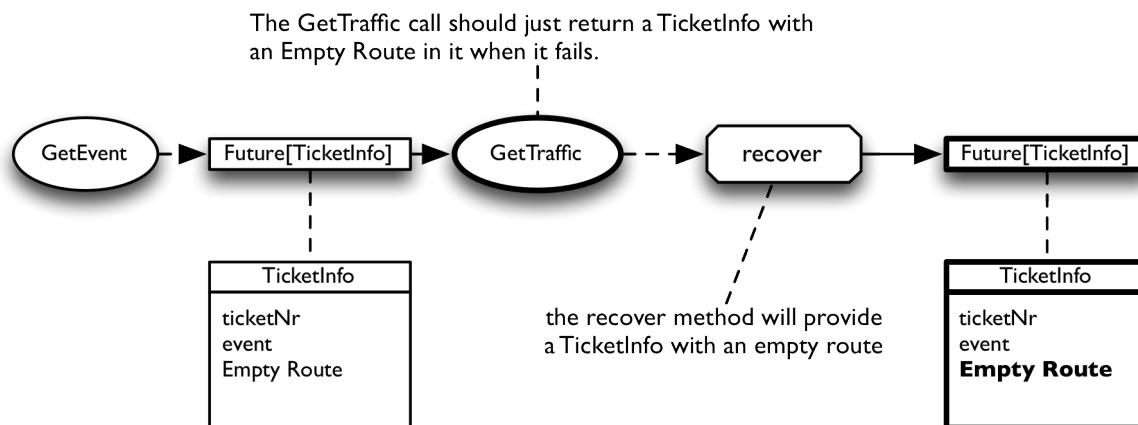


Figure 5.8 Ignore failed service response

The `recover` method can be used to achieve this. This method makes it possible to define what result should be returned when exceptions occur. The below snippet shows how it can be used to return the input `TicketInfo` when a `TrafficServiceException` is thrown.

Listing 5.12 Using `recover` to continue with an alternative `Future` result

```

val futureStep1 = getEvent(ticketNr) ❶
val futureStep2 = futureStep1.flatMap { ticketInfo => ❷
    getTraffic(ticketInfo).recover { ❸
        case e:TrafficServiceException => ticketInfo ❹
    }
}

```

- ❶ Get event returns a `Future[TicketInfo]`
- ❷ `flatMap` is used so we can directly return a `Future[TicketInfo]` instead of a `TicketInfo` value from the code block.
- ❸ `getTraffic` returns a `Future[TicketInfo]`.
- ❹ `recover` with a `Future` containing the initial `TicketInfo` value.

The `recover` method above defines that when a `TrafficServiceException` occurs that it should return the original `ticketInfo` that it received as an argument. The `getTraffic` method normally creates a copy of the `TicketInfo` value with the route added to it. In the above example we used `flatMap` instead of `map` on the future returned by `getEvent`. In the code block passed to `map` you would need to return a `TicketInfo` value which will be wrapped in a new `Future`. With `flatMap` you need to return a `Future[TicketInfo]` directly. Since `getTraffic` already returns a `Future[TicketInfo]` it is easier to use `flatMap`.

Similarly there is a `recoverWith` method where the code block would need to return a `Future[TicketInfo]`, instead of a `TicketInfo` value in the case of the `recover` method in this example. Be aware that the code block passed to the `recover` method call is executed *synchronously* after the error has been returned.

In the above code there is still a problem left. What will happen if the first `getEvent` call fails? The code block in the `flatMap` call will not be called because `futureStep1` is a failed `Future`, so there is no value to chain the next call on. The `futureStep2` value will be exactly the same as `futureStep1`, a failed future result. If we wanted to return an empty `TicketInfo` containing only the `ticketNr` we could recover for the first step as well which is shown in the below listing.

Listing 5.13 Using `recover` to return an empty `TicketInfo` if `getEvent` failed

```

val futureStep1 = getEvent(ticketNr)
val futureStep2 = futureStep1.flatMap { ticketInfo =>
  getTraffic(ticketInfo).recover {
    case e:TrafficServiceException => ticketInfo
  }
}.recover {
  case NonFatal(e) => TicketInfo(ticketNr) ❶
}

```

- ❶ Return an empty `TicketInfo` which only contains the `ticketNr` in case `getEvent` failed.

The code block in the `flatMap` call will not be executed. The `flatMap` will simply return a failed future result. The last `recover` call in the above listing turns this failed `Future` into a `Future[TicketInfo]` if the `Future` contains a non fatal `Throwable`. Now that you've learned how you can recover from errors in a chain of futures we're going to look at more ways to combine futures for the `TicketInfo` service.

5.4 Combining Futures

In the previous sections you were introduced to `map` and `flatMap` to chain asynchronous functions with futures. In this section we're going to look at more ways to combine asynchronous functions with futures. Both the `Future[T]` trait and the `Future` object provide *combinator methods* like `flatMap` and `map` to combine futures. These combinator methods are very similar to `flatMap`, `map` and others found in the *Scala Collections API*. They make it possible to create pipelines of transformations from one immutable collection to the next, solving a problem step by step. In this section we will only scratch the surface of the possibilities of combining futures in a functional style. If you would like to know more about *Functional Programming in Scala* we recommend *Functional Programming in Scala* by Paul Chiusano and Rúnar Bjarnason.

The `TicketInfo` service needs to combine several web service calls to provide the additional information. We will use the combinator methods to add information to the `TicketInfo` step by step using functions which take a `TicketInfo` and return a `Future[TicketInfo]`. At every step a copy of the `TicketInfo` case class is made which is passed on to the next function, eventually building a

complete `TicketInfo` value. The `TicketInfo` case class has been updated and is shown in the below listing, including the other case classes that are used in the service.

Listing 5.14 Improved `TicketInfo` class

```

case class TicketInfo(ticketNr:String,
                      userLocation:Location,
                      event:Option[Event]=None,
                      travelAdvice:Option[TravelAdvice]=None,
                      weather:Option[Weather]=None,
                      suggestions:Seq[Event]=Seq() ) ❶

case class Event(name:String,location:Location,
                 time:DateTime)

case class Weather(temperature:Int, precipitation:Boolean)

case class RouteByCar(route:String,
                     timeToLeave:DateTime,
                     origin:Location,
                     destination:Location,
                     estimatedDuration:Duration,
                     trafficJamTime:Duration) ❷

case class TravelAdvice(routeByCar:Option[RouteByCar]=None,
                       publicTransportAdvice: Option[PublicTransportAdvice]=None)

case class PublicTransportAdvice(advice:String,
                                 timeToLeave:DateTime,
                                 origin:Location, destination:Location,
                                 estimatedDuration:Duration) ❸

case class Location(lat:Double, lon:Double)

case class Artist(name:String, calendarUri:String)

```

- ❶ The `TicketInfo` case class collects travel advice, weather and event suggestions.
- ❷ To keep things simple in this example the route is just a string.
- ❸ To keep things simple in this example the advice is just a string.

All items are optional except the ticket number and the location of the user. Every step in the flow will add some information by copying the argument `TicketInfo` and modifying properties in the new `TicketInfo` value, passing it to the next function. The associated information will be left empty if a service call cannot be completed, as we've shown in the section on futuristic errors. The below figure

shows the flow of asynchronous web service calls and the combinators that we will use in this example:

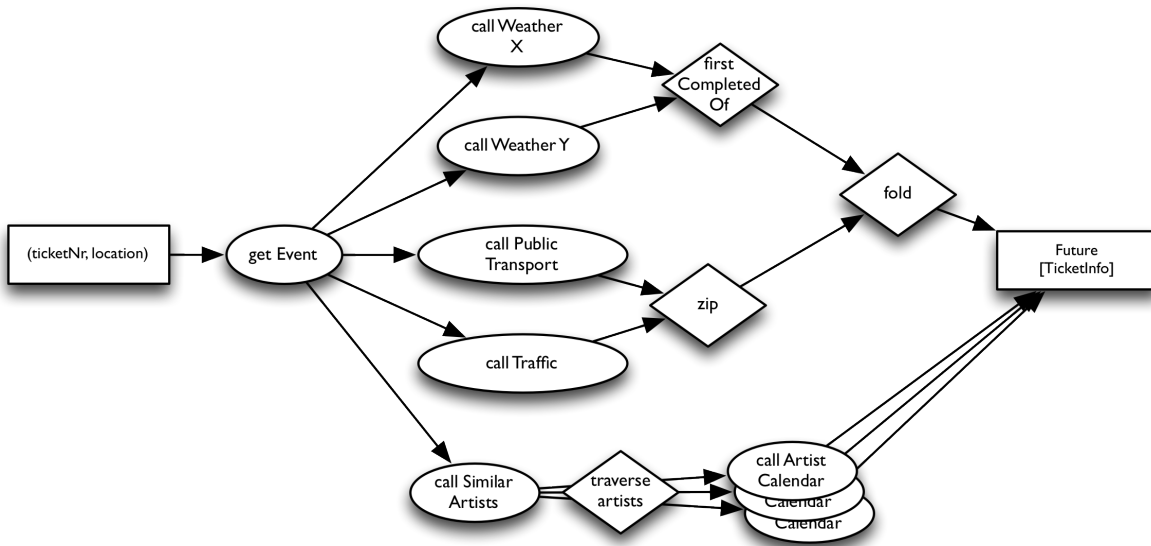


Figure 5.9 TicketInfoService Flow

The combinators are shown as diamonds in the above figure. We will look at every combinator in more detail. The flow starts with a `ticketNr` and a GPS location of the user of the ticket info service and eventually completes a `TicketInfo` future result. The fastest response of the weather services is used. Public transport and car route information are combined in a `TravelAdvice`. At the same time similar artists are retrieved and the calendar for each `Artist` is requested. This results in suggestions for similar events. All futures are eventually combined into a `Future[TicketInfo]`. Eventually this final `Future[TicketInfo]` will have an `onComplete` callback that completes the HTTP request with a response back to the client, which we will omit in these examples.

We'll start with combining the weather services. The `TicketInfo` service needs to call out to many weather services in parallel and use the quickest response. The below figure shows the combinators used in the flow.

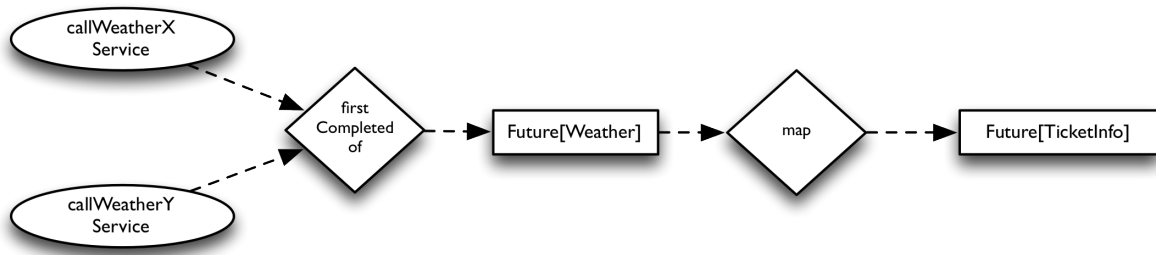


Figure 5.10 Weather flow

Both weather services return a `Future[Weather]`, which needs to be turned into a `Future[TicketInfo]` for the next step. If one of the weather services is not responsive we can still inform the client about the weather with the response of the other service. The below listing shows how the `Future.firstCompletedOf` method is used in the `TicketInfoService` flow to respond to the first completed service:

Listing 5.15 Using `firstCompletedOf` to get the fastest response

```
def getWeather(ticketInfo:TicketInfo):Future[TicketInfo] = {
  val futureWeatherX = callWeatherXService(ticketInfo)
    .recover(withNone) ❶
  val futureWeatherY = callWeatherYService(ticketInfo)
    .recover(withNone) ❶
  val futures = Seq(futureWeatherX, futureWeatherY)
  val fastestResponse = Future.firstCompletedOf(futures) ❷
  fastestResponse.map{ weatherResponse =>
    ticketInfo.copy(weather = weatherResponse) ❸
  } ❹
}
```

- ❶ The error recovery is extracted out into a `withNone` function omitted here. It simply recovers with a `None` value.
- ❷ The first completed `Future[Weather]`.
- ❸ Copy the weather response into a new `ticketInfo`. return the copy as the result of the map code block.
- ❹ the map code block transforms the completed `Weather` value into `TicketInfo`, resulting in a `Future[TicketInfo]`.

First two futures are created for the weather service requests. The `Future.firstCompletedOf` function creates a new `Future` out of the two provided weather service future results. It is important to note that `firstCompletedOf` returns the first *completed* future. A `Future` is completed with a successful value or a failure. With the above code the `ticketInfo` service will not be able to add weather information when for instance the `WeatherX` service fails faster than the `WeatherY` service can return a correct result. For now this will do since we will assume that a non-responsive service or a worse performing service will respond slower than a correct functioning service. We'll see later that a `firstSucceededOf` method is not too difficult to write ourselves. [We will get back to this once we explain `Promises` and implement our own `firstSucceededOf` method. [TODO this explanation forces us to explain `Promises` and get deeper, implementing our own `firstSucceededOf` not sure if this goes too far.]]

The public transport and car route service need to be processed in parallel and combined into a `TravelAdvice` when both results are available. The below figure shows the combinators used in the flow to add the travel advice.

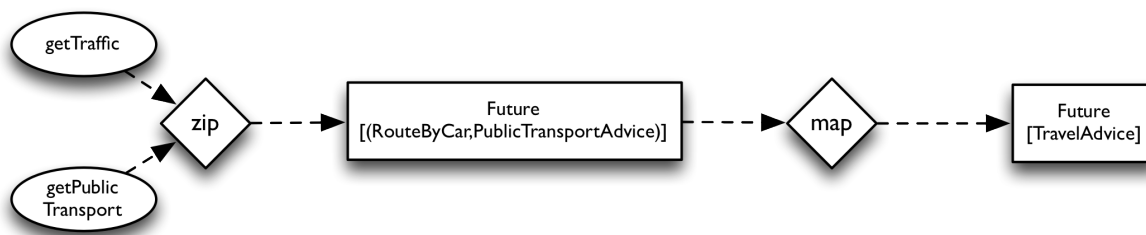


Figure 5.11 Travel advice flow

`getTraffic` and `getPublicTransport` return two different types inside a future, respectively `RouteByCar` and `PublicTransportAdvice`. These two values are first put together in a tuple value. The tuple is then mapped into a `TravelAdvice` value. The `TravelAdvice` class is shown in the below listing.

Listing 5.16 `TravelAdvice` class

```

case class TravelAdvice(routeByCar: Option[RouteByCar] = None,
  publicTransportAdvice: Option[PublicTransportAdvice] = None)
  
```

Based on this information the user can decide to travel by car or by public transport. The below listing shows how the `zip` combinator can be used for this.

Listing 5.17 Using zip and map to combine route and public transport advice

```
def getTravelAdvice(info:TicketInfo,
                   event:Event):Future[TicketInfo] = {
  val futureR = callTraffic(info.userLocation,
                           event.location,
                           event.time).recover(withNone)
  val futureP = callPublicTransport(info.userLocation,
                                    event.location,
                                    event.time).recover(withNone)

  futureR.zip(futureP) 1
    .map { 2
      case(routeByCar, publicTransportAdvice) =>
        val travelAdvice = TravelAdvice(routeByCar,
                                         publicTransportAdvice)
        info.copy(travelAdvice = Some(travelAdvice))
    }
}
```

- 1** Zip Future[RouteByCar] and Future[PublicTransportAdvice] into Future[(RouteByCar, PublicTransportAdvice)].
- 2** Transform the future route and public transport advice into a Future[TicketInfo]

The above code first zips the future public transport and route by car together into a new Future which contains both results inside a tuple value. It then maps over the combined future and turns the result into a Future[TicketInfo] so it can be chained further down the line. You can use a *for-comprehension* instead of using the map method. This can sometimes lead to more readable code. The below listing shows how it can be used, it does exactly the same thing as the zip and map in the above listing:

Listing 5.18 Using zip and map to combine route and public transport advice

```
for((route, advice) <- futureRoute.zip(futurePublicTransport); 1
    travelAdvice = TravelAdvice(route, advice)
) yield info.copy(travelAdvice = Some(travelAdvice)) 2
```

- 1** The future created by the zip method evaluates at some point into a routeByCar and publicTransportAdvice tuple.
- 2** The for-comprehension yields a TicketInfo, which is returned as a Future[TicketInfo] from the for comprehension, similar to how the map method does this.

If you are not too familiar to for-comprehensions, you could think of it as iterating over a collection. In the case of a Future we 'iterate' over a collection which eventually contains one value or nothing (in the case of an exception).

The next part of the flow we'll look at is the suggestion of similar events. Two web services are used; a similar artist service which returns information about artists similar to the one performing at the event. The artist information is used to call a specific calendar service per artist to request the next planned event close to the event location which will be suggested to the user. The below listing shows how the suggestions are built up.

Listing 5.19 Using for-comprehension and traverse to map

```
def getSuggestions(event:Event):Future[Seq[Event]] = {
  val futureArtists = callSimilarArtistsService(event)
  for(artists <- futureArtists
      events <- getPlannedEvents(event, artists)
  ) yield events
}
```

- ❶ returns a Future[Seq[Events]], a future list of planned events for every artist.
- ❷ returns a Future[Seq[Artist]], a Future to similar artists.
- ❸ 'artists' evaluates at some point to a Seq[Artist].
- ❹ 'events' evaluates at some point to a Seq[Events], a planned event for every called artist.
- ❺ The for comprehension returns the Seq[Event] as a Future[Seq[Event]].

The above example is a bit more involved. The code is split up over a couple of methods for clarity although this can obviously be in-lined. The `getPlannedEvents` is only executed once the artists are available. The `getPlannedEvents` uses the `Future.sequence` method to build a `Future[Seq[Event]]` out of a `Seq[Future[Event]]`. In other words it combines many futures into one single future which contains a list of the results. The code for `getPlannedEvents` is shown in the below listing.

Listing 5.20 Using sequence to combine many Future[[Event]] into one Future[Seq[Event]]

```
def getPlannedEvents(event:Event, artists:Seq[Artist]) = {
  val events = artists.map { artist=>
    callArtistCalendarService(artist, event.location)
  }
  Future.sequence(events)
}
```

- ❶ returns a Future[Seq[Event]], a list of planned events, one for every similar artist.
- ❷ map over the Seq[Artists]. for every artist call the calendar service. the 'events' value is a Seq[Future[Event]].
- ❸ Turns the Seq[Future[Event]] into a Future[Seq[Event]]. It eventually returns a list of events when the results of all the asynchronous callArtistCalendarService calls are completed.

The sequence method is a simpler version of the traverse method. The below example shows how getPlannedEvent looks when we use traverse instead.

Listing 5.21 Using traverse to combine many Future[[Event]] into one Future[Seq[Event]]

```
def getPlannedEventsWithTraverse(event:Event, artists:Seq[Artist]) = {
  Future.traverse(artists) { artist=>
    callArtistCalendarService(artist, event.location)
  }
}
```

- ❶ traverse takes a code block which is required to return a Future. It allows you to traverse a collection and at the same time create the future results.

Using sequence we first had to create a Seq[Future[Event]] so we could transform it into a Future[Seq[Event]]. With traverse we can do the same but without the intermediate step of first creating a Seq[Future[Event]].

It's time for the last step in the TicketInfoService flow. The TicketInfo value which contains the Weather information needs to be combined with the TicketInfo containing the TravelAdvice. We're going to use the fold method

to combine to `TicketInfo` values into one. The below listing shows how it is used:

Listing 5.22 Using `fold` to combine two `Future[[Event]]` into one `Future[Seq[Event]]`

```
val ticketInfos = Seq(infoWithTravelAdvice, infoWithWeather)

val infoWithTravelAndWeather = Future.fold(ticketInfos)(info) { ❷
  (acc, elem) => ❸

  val (travelAdvice, weather) = (elem.travelAdvice, elem.weather) ❹

  acc.copy(travelAdvice = travelAdvice.orElse(acc.travelAdvice),
    weather = weather.orElse(acc.weather)) ❺
}
```

- ❶ create a list of the `TicketInfo` containing the travel advice and the `TicketInfo` containing the weather.
- ❷ `fold` is called with the list and the accumulator is initialized with the `ticketInfo` that only contains the event information.
- ❸ `Fold` returns the result of the previously executed code block in the accumulator ('acc') value. it passes every element to the code block, in this case every `TicketInfo` value.
- ❹ extract the optional `travelAdvice` and `weather` properties out of the `ticketInfo`.
- ❺ copy the `travelAdvice` or the `weather` into the accumulated `TicketInfo`, whichever is filled. the copy is returned as the next value of 'acc' for the next invocation of the code block.

The `fold` method works just like `fold` on data structures like `Seq[T]` and `List[T]` which you are probably familiar with. It is often used instead of traditional for loops to build up some data structure through iterating over a collection. `fold` takes a collection, an initial value and a code block. The code block is fired for every element in the collection. The block takes two arguments, a value to accumulate state in and the element in the collection that is next. In the above case the initial `TicketInfo` value is used as the initial value. At every iteration of the code block a copy of the `TicketInfo` is returned that contains more information, based on the elements in the `ticketInfos` list.

The complete flow is shown in the below listing:

Listing 5.23 Complete TicketInfoService flow

```

def getTicketInfo(ticketNr:String,
                  location:Location):Future[TicketInfo] = {
  val emptyTicketInfo = TicketInfo(ticketNr, location)
  val eventInfo = getEvent(ticketNr, location) ❶
                  .recover(withPrevious(emptyTicketInfo))

  eventInfo.flatMap { info =>

    val infoWithWeather = getWeather(info) ❷

    val infoWithTravelAdvice = info.event.map { event =>
      getTravelAdvice(info, event) ❸
    }.getOrElse(eventInfo)

    val suggestedEvents = info.event.map { event =>
      getSuggestions(event) ❹
    }.getOrElse(Future.successful(Seq()))

    val ticketInfos = Seq(infoWithTravelAdvice, infoWithWeather)

    val infoWithTravelAndWeather = Future.fold(ticketInfos)(info) {
      (acc, elem) =>

        val (travelAdvice, weather) = (elem.travelAdvice, elem.weather)
        acc.copy(travelAdvice = travelAdvice.getOrElse(acc.travelAdvice),
                 weather = weather.getOrElse(acc.weather))
    } ❺

    for(info <- infoWithTravelAndWeather;
        suggestions <- suggestedEvents
        ) yield info.copy(suggestions = suggestions) ❻
  }

  // error recovery functions to minimize copy/paste ❼
  type Recovery[T] = PartialFunction[Throwable,T]

  // recover with None
  def withNone[T]:Recovery[Option[T]] = { case NonFatal(e) => None }

  // recover with empty sequence
  def withEmptySeq[T]:Recovery[Seq[T]] = { case NonFatal(e) => Seq() }

  // recover with the ticketInfo that was built in the previous step
  def withPrevious(previous:TicketInfo):Recovery[TicketInfo] = {
    case NonFatal(e) => previous
  }
}

```

- ① First call `getEvent` which returns a `Future[TicketInfo]`
- ② create a `TicketInfo` with Weather information
- ③ create a `TicketInfo` with `TravelAdvice` information
- ④ get a future list of suggested events
- ⑤ combine weather and travel into one `TicketInfo`
- ⑥ Eventually add the suggestions as well.
- ⑦ Error recovery methods that are used in the `TicketInfoService` flow.

That concludes the `TicketInfoService` example using futures. As you have seen, futures can be combined in many ways and the combinator methods make it very easy to transform and sequence asynchronous function results. The entire `TicketInfoService` flow does not make one blocking call. If the calls to the hypothetical web services would be implemented with an asynchronous HTTP client like the *spray-client* library the amount of blocking threads would be kept to a minimum for I/O as well. At the time of writing this book more and more asynchronous client libraries in Scala for I/O but also for database access are written that provide `Future` results.

In the next section we're going to look at how futures can be combined with Actors.

5.5 Futures and Actors

In the *Up and Running* chapter we used *Spray* for our first REST service which uses Actors for HTTP request handling. This chapter already showed that the `ask` method returns a `Future`. The below example was given:

Listing 5.24 Collecting event information

```

import akka.pattern.ask      ❶

import context._           ❷
implicit val timeout = Timeout(5 seconds)  ❸

val capturedSender = sender  ❹

def askEvent(ticketSeller:ActorRef): Future[Event] = {  ❺

  val futureInt = ticketSeller.ask(GetEvents).mapTo[Int]  ❶

  futureInt.map { nrOfTickets =>
    Event(ticketSeller.actorRef.path.name, nrOfTickets)
  }
}

val futures = context.children.map {
  ticketSeller => askEvent(ticketSeller)  ❸
}

Future.sequence(futures).map {
  events => capturedSender ! Events(events.toList)  ❷
}

```

- ❶ Import the ask pattern, which adds the ask method to ActorRef.
- ❷ The context contains an implicit def of the dispatcher of the actor. Importing the context imports the dispatcher of this actor as the execution context in implicit scope which will be used for the futures.
- ❸ A timeout needs to be defined for ask. If the ask does not complete within the timeout the future will contain a timeout exception.
- ❹ Capture the contextual sender reference into a value.
- ❺ A local method definition for asking GetEvents to a TicketSeller.
- ❻ The ask method returns a Future result. Because Actors can send back any message the returned Future is not typed. we use the mapTo method to convert the Future[Any] to a Future[Int]. If the actor responds with a different message than an Int the mapTo will complete with a failed Future.
- ❼ Send back information to the captured sender. The captured sender is the sender of the original GetEvents request which should receive the response.
- ❽ Ask all the children how many tickets they have left for the event.

This example should be more clear now than it was in chapter two. To reiterate, the example shows how the boxOffice actor can collect the number of tickets that every ticket seller has left.

The above example shows a couple of important details. First of all we *capture* the sender reference into a value. This is necessary because the sender is part of the actor context, which can differ at every message the actor receives. The sender of the message can obviously be different for every message the actor receives. Since the future callback is a Closure it *closes over* the values it needs to use. The sender could have a completely different value at the time the callback is invoked. This is why the sender is captured into a `capturedSender` value, which makes it possible to respond to the originator of the `GetEvents` request.

So be aware when using futures from Actors that the `ActorContext` provides a current view of the Actor. And since actors are stateful it is important to make sure that the values that you close over are not mutable from another thread. The easiest way to prevent this problem is to use immutable data structures and capture the reference to the immutable data structure before closing over it from a future, as shown above in the `capturedSender` example.

Another pattern that is used in the `Up and Running` example is `pipeTo`. The below listing shows an example of its use:

Listing 5.25

```
import akka.pattern.pipe ①
path("events") {
  get { requestContext =>
    val responder = createResponder(requestContext) ②
    boxOffice.ask(GetEvents).pipeTo(responder) ③
  }
}
```

- ① Imports the pipe pattern
- ② Create an actor for every request that completes the HTTP request with a response
- ③ The Future result of the ask is piped to the responder actor

The `RestInterface` uses a per-request actor to handle the HTTP request and provide a HTTP response. The `/events` URL is translated to a call to the `boxOffice` to collect the amount of tickets left per event. The result of the request to the `boxOffice` is a `Future[Events]`. The `Events` value that that future will eventually contain is piped to the responder actor when it becomes available. The responder actor completes the request when it receives this message, as is shown in below listing.

Listing 5.26

```

class Responder(requestContext:RequestContext,
                ticketMaster:ActorRef)
  extends Actor with ActorLogging {

  def receive = {

    case Events(events) =>
      requestContext.complete(StatusCodes.OK, events)
      self ! PoisonPill

    // other messages omitted..
  }
}

```

The `pipeTo` method is useful when you want an actor to handle the result of a `Future`, or when the result of asking an actor needs to be handled by another actor.

5.6 Summary

This chapter gave an introduction on futures. You have learnt how to use futures to create a flow out of asynchronous functions. The goal has been to minimize explicitly blocking and waiting on threads, maximize resource usage and minimize unnecessary latency.

A `Future` is a placeholder for a function result that will eventually be available. It is a great tool for combining functions into asynchronous flows. futures make it possible to define transformations from one result to the next. Since futures are all about function results it's no surprise that a functional approach needs to be taken to combine these results.

The combinator methods for futures provide a 'transformational style' similar to the combinators found in the `scala collections` library. Functions are executed in parallel and where needed in sequence, eventually providing a meaningful result. A `Future` can contain a successful value or a failure. Luckily failures can be recovered with a replacement value to continue the flow.

The value contained in a `Future` should be immutable to be sure that no accidental mutable state is shared. Futures can be used from Actors, but you need to be careful with what state you close over. The sender reference of an actor needs

to be captured into a value before it can be safely used for instance. Futures are used in the Actor API as the response of an `ask` method. Future results can also be provided to an actor with the `pipeTo` method.

Now that you know about futures we're going back to Actors in the next chapter. This time we'll scale the `goticks.com` app with Remote Actors.

Your first Distributed Akka App



In this chapter

- An introduction to scaling out
- Distributing the goticks.com App
- Remoting
- Testing distributed actor systems

So far we have only looked at building an Akka actor system on one node. This chapter will serve as an introduction to scaling out Akka applications. You will build your first distributed Akka App right here. We'll take the Up and Running App from chapter 2 and scale it out.

We start off with some common terminology and a quick look at the different approach Akka takes to scale out. You will be introduced to the *akka-remote* module and how it provides an elegant solution for communicating between actors across the network. We'll scale the goticks.com app out to two nodes; a frontend and a backend server. You'll find out how you can unit test the app using the Multi-JVM testkit.

This chapter will just get you acquainted to scaling out your apps. Chapter 7 will introduce you to clustering. Chapter 13 will dive into the details of scaling out a much more involved example once you are more familiar with how to build a real world Akka application, which is the main topic of the second part of this book.

6.1 Scaling out

You might have hoped that this was going to be a chapter about a silver bullet to make any application scale out to thousands of machines, but here is the truth: distributed computing is hard. Notoriously hard. Don't stop reading yet though! Akka will at least give you some really nice tools that make your life in distributed computing a little easier. Once again Akka is not promising a free lunch but just as actors simplify concurrent programming, we will also see that that they simplify the move to truly distributed computing. We will bring back our GoTicks.com project and make it distributed.

Most network technologies use a Remote Procedure Call (RPC) style of interaction for communicating with objects across the network, which tries to mask the difference between calling an object locally or remotely. The idea being that a local programming model is simplest, so let the programmer just work in that way, then transparently make it possible to remote some of the calls when and where required. This style of communication 'works' for point to point connections between servers but it is not a good solution for larger scale networks as we will see in the next section. Message-oriented middleware can solve this, but at the cost of having the application absorb the messaging system into the application layer. Akka takes a different approach when it comes to scaling out applications across the network, that gives us the best of both approaches: we have relative transparency of remoting collaborators, but we don't have to change our Actor code: you will see the top layer looks the same.

Before we dive in, we'll look at examples of network topologies and some common terminology in the following section, just in case you are not too familiar with these. If you're already an expert in the field you might want to skip right section 6.2.

6.1.1 Common network terminology

When we refer to a *Node* in this chapter we mean a running application which communicates across the network. It is a connection point in a network topology. It's part of a distributed system. Many nodes can run on one server or they can run on separate servers. Figure 6.1 shows some common network topologies:

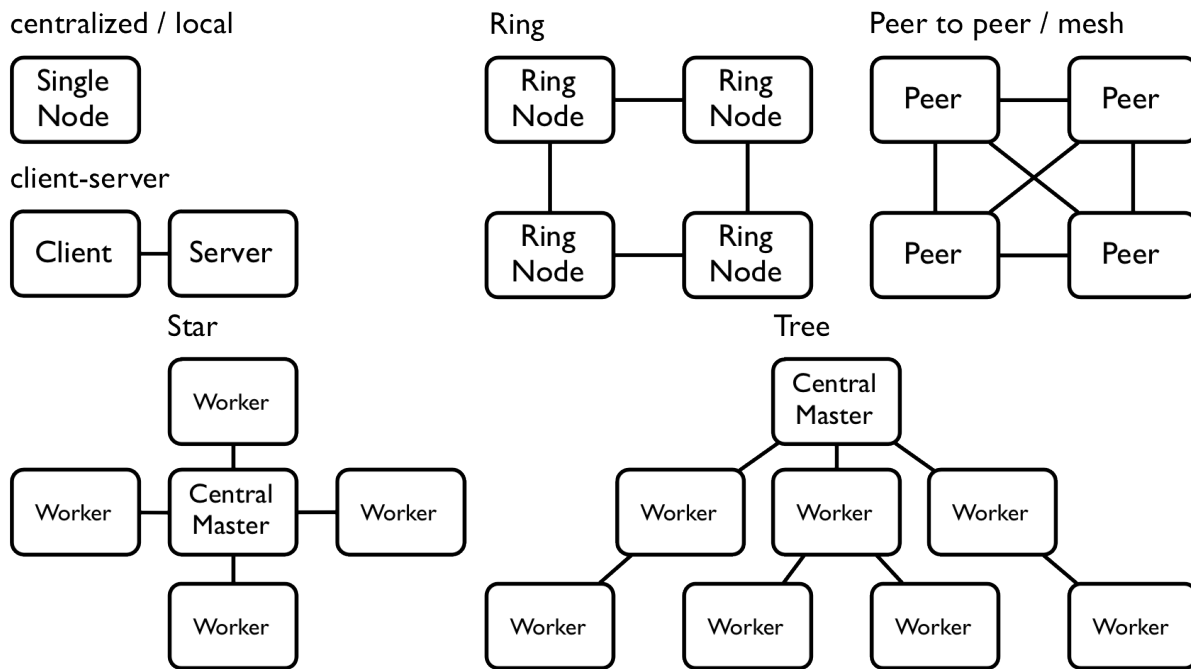


Figure 6.1 Common network topologies

A node has a specific *Role* in the distributed system. It has a specific responsibility to execute particular tasks. A node could for instance take part in a distributed database or it could be one of many web servers that fulfill frontend web requests.

A node uses a specific network *transport protocol* to communicate with other nodes. Examples of transport protocols are TCP/IP and UDP. Messages between the nodes are sent over the transport protocol and need to be encoded and decoded into network specific *protocol data units*. The protocol data units contain a stored representation of the messages as byte arrays. Messages need to be translated to and from bytes, respectively known as *serialization* and *deserialization*. Akka provides a serialization module for this purpose, which we will briefly touch on in this chapter.

When nodes are part of the same distributed system they share a *group membership*. This membership can be *static* or *dynamic* (or even a mix of both). In a static membership the number of nodes and the role of every node is fixed and cannot change during the lifetime of the network. A dynamic membership allows for nodes to take on different roles and for nodes to join and leave the network.

The static membership is obviously the simplest of the two. All servers hold a reference to the other nodes' network address at startup. It is also less resilient; a node cannot simply be replaced by another node running on a different network address.

The dynamic membership is more flexible and makes it possible for a group of nodes to grow and shrink as required. It also makes it possible to deal with failed nodes in the network, possibly automatically replacing them. It is also far more complex than the static membership. When a dynamic membership is properly implemented it needs to provide a mechanism to dynamically join and leave the cluster, detect and deal with network failures, identify unreachable/failed nodes in the network and provide some kind of *discovery* mechanism through which new nodes can find an existing group on the network since the network addresses are not statically defined.

Now that we have briefly looked at network topologies and common terminology the next section will look at the reason why Akka uses a distributed programming model for building both local and distributed systems.

6.1.2 Reasons for a distributed programming model

Our ultimate goal is to scale to many nodes and very often the starting point is a local app on one node: your laptop. So what changes exactly when we want to make the step to one of the distributed topologies in the previous section? Can't we abstract away the fact that all these nodes run on one 'virtual node' and let some clever tool work out all the details so we don't have to change the code running on the laptop at all? The short answer is no ¹. We can't simply abstract the differences between a local and distributed environment. Luckily you don't just have to take our word for it. According to the paper *A Note on Distributed Computing*² there are four important areas in which local programming differs from distributed programming which cannot be ignored. The four areas are latency, memory access, partial failure and concurrency. The below list briefly summarizes the differences in the four areas:

Footnote 1 Software suppliers that still sell you this idea will obviously disagree!

Footnote 2 1994 Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall

- *Latency*: having the network in between collaborators means a. much more time for each message, and b. delays due to traffic, resent packets, intermittent connections, etc.
- *Partial Failure*: Knowing if all parts of a distributed system are still functioning is a very hard problem to solve when parts of the system are not always visible, disappear and even reappear.
- *Memory Access*: Getting a reference to an object in memory in a local system cannot intermittently fail, which can be the case for getting a reference to an object in a distributed setting.
- *Concurrency*: no one 'owner' of everything, and the above factors mean the plan to interleave operations can go awry.

Using a local programming model in a distributed environment fails at scale because of these differences. Akka provides the *exact opposite*; a distributed programming model for both a distributed and a local environment. The above mentioned paper also mentions this choice and states that distributed programming would be simpler this way but also states that it could make local programming unnecessarily hard; as hard as distributed programming.

But times have changed. Almost 2 decades later we have to deal with many CPU cores. And more and more tasks simply need to be distributed in the cloud. Enforcing a distributed programming model for local systems has the advantage that it simplifies concurrent programming as we have seen in the previous chapters. We have already gotten used to asynchronous interactions, expect partial failures (even embrace it), and we use a shared nothing approach to concurrency, which both simplifies programming for many CPU cores and makes us more prepared for a distributed environment.

We will show you that this choice provides a solid foundation for building both local and distributed applications that are fit for the challenges of today. Akka provides both a simple API for asynchronous programming and the tools you need to test your applications locally and remotely. Now that you understand the reasoning for a distributed programming model for both local and distributed systems we're going to look at how we can scale out the goticks.com App that we built in Chapter 2 in the next sections.

6.2 Scaling Out with Remoting

Since this is an introduction to scaling out we're going to use the relatively simple example goticks.com app from chapter 2. In the next sections we will change the app so it runs on more than one node. Although the goticks.com app is an oversimplified example it will give us a feel of the changes we need to make to an app that has not made any accommodations for scaling yet.

We'll define a static membership between two nodes using a client-server network topology since it is the easiest path from local to distributed. The roles for the two nodes in this setup are frontend and backend. The REST Interface will run on a frontend node. The BoxOffice and all TicketSellers will run on a backend node. Both nodes have a static reference to each other's network addresses. Figure 6.2 shows the change that we're going to make:

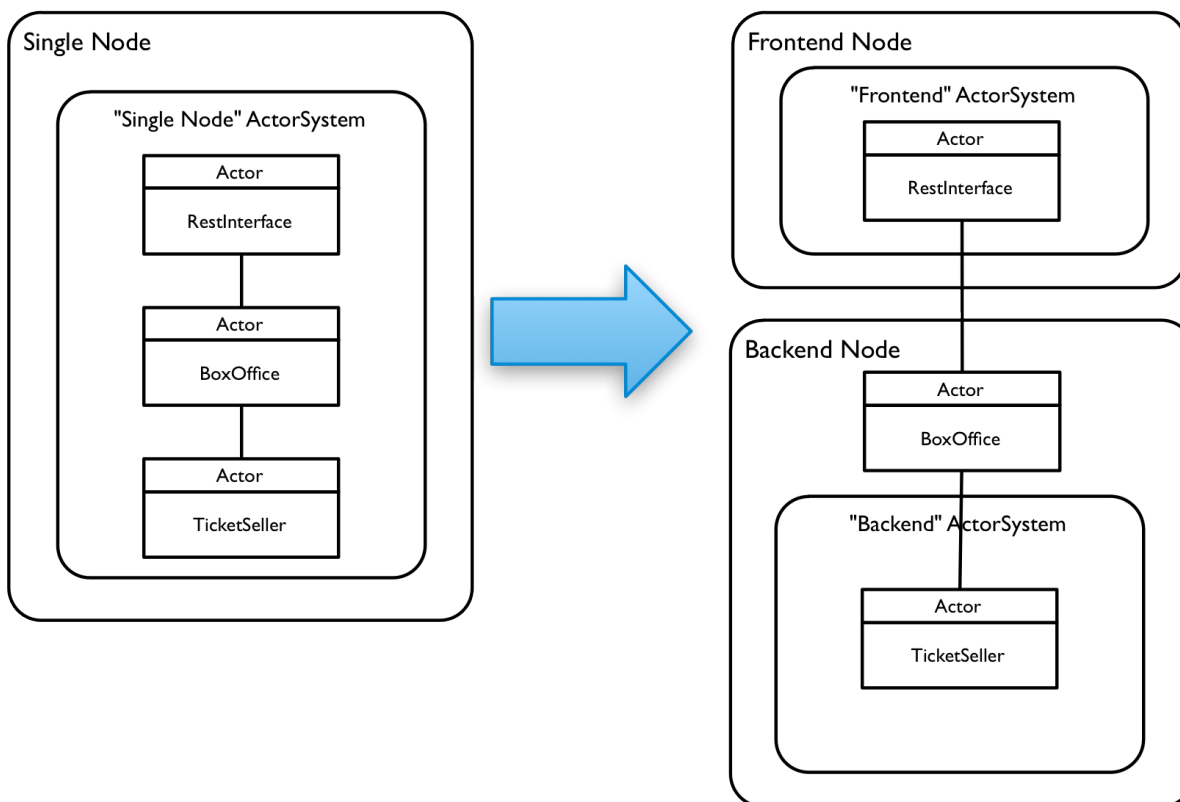


Figure 6.2 From single node to client-server

We'll use the akka-remote module to make this change. The BoxOffice Actor creates TicketSeller actors when new Events are created in the local version of the app. In the client server topology this will have to be done as well. As we will see the akka-remote module makes it possible to create and deploy actors remotely. The frontend is going to lookup the BoxOffice actor on a backend node on its known address which creates the TicketSeller actors. We'll also look at a variation on this where the frontend remotely deploys a BoxOffice actor on the backend node.

In the next section we're going to get our hands dirty with remoting. We'll start with looking at the changes that need to be made to the SBT build file and then look at the changes we have to make to the rest of the code.

6.2.1 Making the GoTicks App Distributed

The chapter6 folder in the akka-in-action contains a modified version of the chapter2 example. You can follow along by making the changes on top of the chapter2 sample as described here. The first thing we need to do is add the dependencies for akka-remote and the akka-multinode-testkit in the SBT build file:

Listing 6.1 Build File Changes for Distributed GoTicks

```

libraryDependencies += {
  val akkaV      = "2.2-M3"
  val sprayV    = "1.2-M8-SNAPSHOT"
  Seq(
    "com.typesafe.akka" %% "akka-actor"           % akkaV,
    "com.typesafe.akka" %% "akka-slf4j"          % akkaV,
    "com.typesafe.akka" %% "akka-remote"         % akkaV, ①
    "com.typesafe.akka" %% "akka-multi-node-testkit" % akkaV % "test", ②
    "com.typesafe.akka" %% "akka-testkit"         % akkaV % "test",
    "org.scalatest"     %% "scalatest"           % "1.9.1" % "test",
    "io.spray"          % "spray-can"            % sprayV,
    "io.spray"          % "spray-routing"        % sprayV,
    "io.spray"          %% "spray-json"          % "1.2.3",
    "com.typesafe.akka" %% "akka-slf4j"         % akkaV,
    "ch.qos.logback"   % "logback-classic"     % "1.0.10"
  )
}

```

- ① Dependency on akka-remote module
- ② Dependency on multi-node testkit for testing distributed actor systems

These dependencies are pulled in automatically when you start sbt or you can run `sbt update` to explicitly pull in the dependencies. Now that we have the dependencies updated and ready to go, let's look at the changes that we need to make for connecting frontend and backend. The actors on the frontend and backend will need to get a reference to their collaborator, which is the topic of the next section.

6.2.2 Remote REPL action

Akka provides two ways to get a reference to an actor on a remote node. One is to look up the actor by its path, the other is to create the actor, get its reference and deploy it remotely. We will start with the former option.

The REPL console is a great interactive tool for quickly exploring new scala classes. Let's get two actor systems up in two REPL sessions using the sbt console. Start a terminal in the `chapter6` folder using `sbt console`. We need to enable remoting so the first thing we need to do is provide some configuration. Normally an `application.conf` configuration file in your `src/main/resources` folder would contain this information but in the case of a REPL session we can just load it from

a String. Listing 6.2 contains the REPL commands to execute using the `:paste` command:

Listing 6.2 REPL commands for loading up Remoting

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val conf = """
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider" ❶
  }
  remote { ❷
    enabled-transport = ["akka.remote.netty.tcp"] ❸
    netty.tcp { ❹
      hostname = "0.0.0.0"
      port = 2551
    }
  }
}
"""
```

- ❶ Select the Remote ActorRef Provider to bootstrap remoting
- ❷ the configuration section for remoting
- ❸ Enable the TCP transport
- ❹ Settings for the TCP transport, the host and port to listen on

We're going to load this configuration string into an ActorSystem. Most notably it defines a specific *ActorRefProvider* for Remoting which bootstraps the akka-remote module. As the name suggests it also takes care of providing your code with ActorRefs to Remote Actors. Listing 6.3 first imports the required config and actor packages and then loads the config into an actor system:

Listing 6.3 Remoting Config

```
scala> import com.typesafe.config._
import com.typesafe.config._

scala> import akka.actor._
import akka.actor._

scala> val config = ConfigFactory.parseString(conf) ❶
config: com.typesafe.config.Config = ....

scala> val backend = ActorSystem("backend", config) ❷
[Remoting] Starting remoting
.....
[Remoting] Remoting now listens on addresses:
[akka.tcp://backend@0.0.0.0:2551]
backend: akka.actor.ActorSystem = akka://backend
```

- ❶ Parse the String into a Config object.
- ❷ Create the ActorSystem with the parsed Config object.

If you have typed along you just started your first remote-enabled ActorSystem from a REPL, it's that simple! Depending on your perspective that's five lines of code to bootstrap and start a server.

The "backend" ActorSystem is created with the config object which enables remoting. If you forget to pass the config to the ActorSystem you will end up with an ActorSystem that runs but is not enabled for remoting because the default application.conf that is packaged with Akka does not bootstrap remoting. The Remoting module now listens on all interfaces (0.0.0.0) on port 2551 for the backend actor system. Lets add a very simple Actor that just prints whatever it receives to the console so we can see that everything works. Listing 6.4 shows the code we need.

Listing 6.4 Configuring the Front End Actor

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class Simple extends Actor {

  def receive = {
    case m => println(s"received $m!")
  }
}
// Exiting paste mode, now interpreting.
scala> backend.actorOf(Props[Simple], "simple") ❶
```

- ❶ Create the simple actor in the backend actor system with the name "simple"

The Simple actor is now running in the backend actor system. It's important to note that the Simple actor is created with the name "simple". This will make it possible to find it on the other side by name. Time to start up another terminal, fire up sbt console and create another remoting-enabled actor system, the "frontend". We'll use the same commands as before except for the fact that we want to make sure that the frontend actor system runs on a different TCP port:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val conf = """
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "0.0.0.0"
      port = 2552 ❶
    }
  }
}
"""

import com.typesafe.config._

import akka.actor._

val config = ConfigFactory.parseString(conf)
```

```

val frontend= ActorSystem("frontend", config)
[Remoting] Starting remoting
.....
[Remoting] Remoting now listens on addresses:
[akka.tcp://backend@0.0.0.0:2552]
frontend: akka.actor.ActorSystem = akka://frontend

```

- 1 Run the frontend on a different port than the backend so they can both run on the same machine

The configuration is loaded into the frontend actorsystem. The frontend actorsystem is now also running and remoting has started. Lets get a reference to the `Simple` actor on the backend actor system from the frontend side. We're first going to construct an actor path. The belows figure shows how the path is built up:

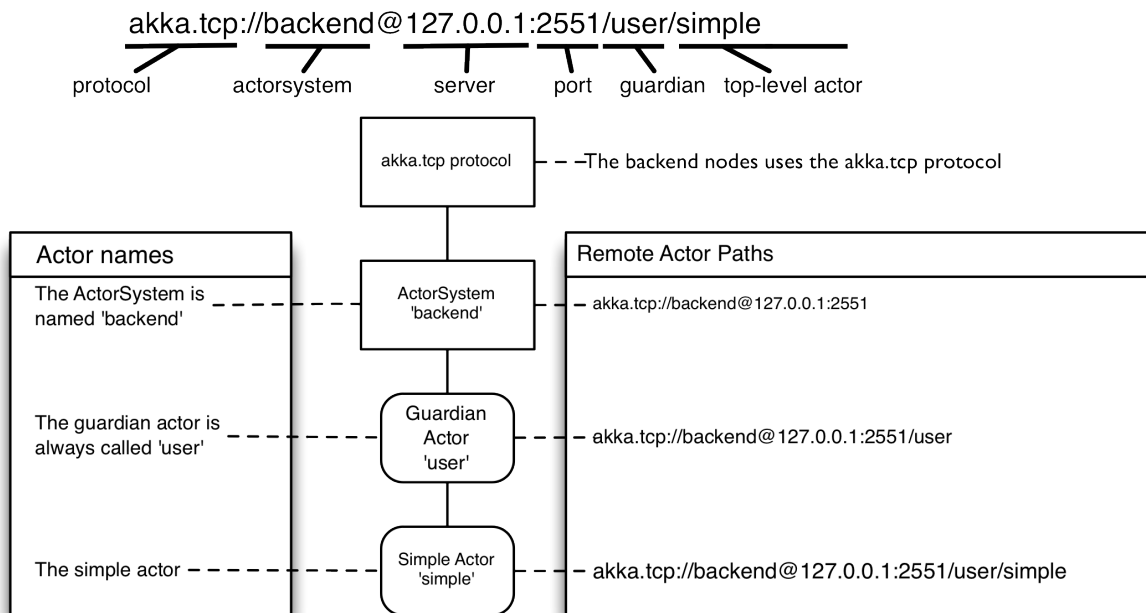


Figure 6.3 Remote actor paths

We can construct the path as a `String` and use the `actorSelection` method on the frontend actorsystem to find it:

```

scala> :paste
// Entering paste mode (ctrl-D to finish)

val path = "akka.tcp://backend@0.0.0.0:2551/user/simple"
val simple = frontend.actorSelection(path)

// Exiting paste mode, now interpreting.

```

```
path: String = akka.tcp://backend@0.0.0.0:2551/user/simple
simple: akka.actor.ActorSelection =
ActorSelection[Actor[akka.tcp://backend@0.0.0.0:2551/]/user/simple]
```

- ❶ The path to the remote Simple Actor
- ❷ Select the actor with an ActorSelection

Think of the *actorSelection* method as a query in the actor hierarchy. In this case the query is an exact path to a remote actor. The *ActorSelection* is an object that represents all the actors that have been found in the actor system with the *actorSelection* method. The Actor Selection can be used to send a message to all actors that match the query. We don't need the exact ActorRef of the Simple Actor for now, we only want to try and send a message to it so the ActorSelection will do. Since the backend actor system is already running in the other console you should be able to do the below:

```
scala> simple ! "Hello Remote World!"

scala>
```

When you switch to the terminal where you started the backend actor system you should see the following printed message:

```
scala> received Hello Remote World!!
```

The REPL console shows you that the message was sent from the frontend to the backend. Being able to interactively explore remoting systems using a REPL console is pure gold in our opinion so you can expect more of it in next chapters.

What happened under the covers is that the "Hello Remote World!" message was serialized, sent to a TCP socket, received by the remoting module, deserialized and forwarded to the Simple Actor running on the backend.

You probably noticed that we did not write any special code for serialization, so why did it work? It's because we sent a simple String ("Hello Remote World!"). Akka uses Java Serialization by default for any message that needs to be sent across the wire. Other serializers are also available and you can write your own custom

serializer as well which is a topic we will deal with in part 3. The Akka *remote message protocol* has a field which contains the name of the serializer that was used for the message so that the receiving remote module can de-serialize the payload bytes. The class that is used to represent a message needs to be Serializable and it needs to be available on the classpath on both sides. Luckily 'standard' case classes and case objects are serializable³ by default which are used as messages in the goticks.com app. Now that you have seen how you can lookup a remote actor and send a message to it in the REPL lets look at how we can apply it in the goticks.com app in the next section.

Footnote 3 Serializable is a marker interface and guarantees nothing. You need to verify that it works if you use 'non-standard' constructs.

6.2.3 Remote Lookup

Instead of directly creating a BoxOffice actor in the RestInterface actor we will look it up on the backend node. Figure 6.4 shows what we're going to try and achieve:

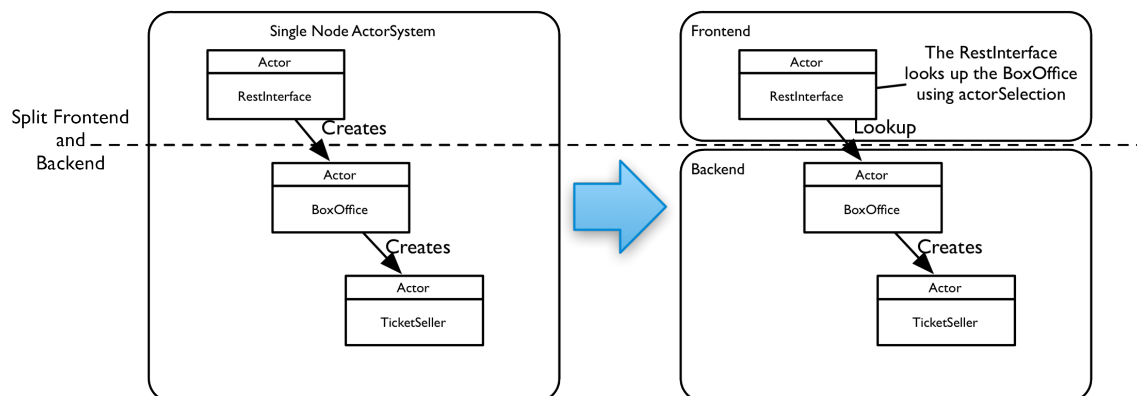


Figure 6.4 Remote Lookup of the BoxOffice Actor

In the previous version of the code the RestInterface directly created a child BoxOffice actor:

```
val boxOffice = context.actorOf(Props[BoxOffice], "boxOffice")
```

This call made the boxOffice a direct child of the RestInterface. To make the app a bit more flexible and to make it possible to run it both in single node and in

client server, we will move this code into a trait that we can mix in. This trait is shown in listing 6.5 as well as the change we need to make to the RestInterface code:

Listing 6.5 Creator of BoxOffice

```

trait BoxOfficeCreator { this: Actor => ❶
  def createBoxOffice:ActorRef = {
    context.actorOf(Props[BoxOffice], "boxOffice")
  } ❷
}

class RestInterface extends HttpServiceActor with RestApi { ❸
  def receive = runRoute(routes)
}

trait RestApi extends HttpService
  with ActorLogging
  with BoxOfficeCreator { actor: Actor => ❹
  val boxOffice = createBoxOffice ❺

  // rest of the code of the RestApi ...

```

- ❶ This trait has to be mixed into an Actor so it can use the actor context.
- ❷ The createBoxOffice method creates the BoxOffice actor and returns an ActorRef.
- ❸ The RestApi trait contains all the logic for the RestInterface actor.
- ❹ BoxOfficeCreator is mixed into the RestApi trait
- ❺ BoxOffice is created using the createBoxOffice method

We've now separated the code to create the BoxOffice in a separate trait and made it the default behavior of the RestInterface to create a local boxOffice. A RemoteBoxOfficeCreator trait will override the default behavior of which the details will follow shortly. A SingleNodeMain, FrontendMain and a BackendMain are created to start the app in single node mode or to start a frontend and backend separately. Listing 6.6 shows the interesting code (as snippets) of the three main classes:

Listing 6.6 Highlights from the Core Actors

```
//Snippet from SingleNodeMain
val system = ActorSystem("singlenode", config)

val restInterface = system.actorOf(Props[RestInterface],
                                   "restInterface") ❶

//Snippet from FrontendMain
val system = ActorSystem("frontend", config)

class FrontendRestInterface extends RestInterface
                               with RemoteBoxOfficeCreator ❷

val restInterface = system.actorOf(Props[FrontendRestInterface],
                                   "restInterface") ❸

//Snippet from BackendMain
val system = ActorSystem("backend", config)
val config = ConfigFactory.load("backend")
val system = ActorSystem("backend", config)

system.actorOf(Props[BoxOffice], "boxOffice") ❹
```

- ❶ Create the rest interface as before
- ❷ Mix the RemoteBoxOfficeCreator trait in
- ❸ Create the rest interface with a RemoteBoxOfficeCreator mixed in.
- ❹ Create a top-level boxOffice actor on the backend.

All main classes load their configuration from a specific configuration file, the SingleNodeMain, FrontendMain and BackendMain load from the files singlenode.conf, frontend.conf and backend.conf respectively. The frontend.conf file has an extra config section for looking up the boxoffice actor. The RemoteBoxOfficeCreator loads these extra configuration properties:

```
backend {
  host = "0.0.0.0"
  port = 2552
  protocol = "akka.tcp"
  system = "backend"
  actor = "user/boxOffice"
}
```

The path to the boxoffice actor is built from this configuration section. Getting an Actor Selection to the remote actor was fine in the REPL console, just to try out sending a message, when we were certain that the backend was present. In this case we would like to work with an ActorRef instead since the single node version used one. The RemoteBoxOfficeCreator trait is show in listing 6.7:

Listing 6.7 Trait for Creation of Remote BoxOffice

```
object RemoteBoxOfficeCreator {
  val config = ConfigFactory.load("frontend").getConfig("backend") ❶
  val host = config.getString("host")
  val port = config.getInt("port")
  val protocol = config.getString("protocol")
  val systemName = config.getString("system")
  val actorName = config.getString("actor")
}

trait RemoteBoxOfficeCreator extends BoxOfficeCreator { this:Actor =>
  import RemoteBoxOfficeCreator._

  def createPath:String = {
    s"$protocol://$systemName@$host:$port/$actorName" ❷
  }

  override def createBoxOffice = {
    val path = createPath
    context.actorOf(Props(classOf[RemoteLookup], path),
                     "lookupBoxOffice") ❸
  }
}
```

- ❶ Load the frontend.conf configuration and get the "backend" config section properties to build the path.
- ❷ Create the path to the boxoffice
- ❸ return an Actor that looks up the box office actor which we will cover next. The Actor is constructed with one argument; the path to the remote boxOffice.

The RemoteBoxOfficeCreator creates a separate RemoteLookup Actor to lookup the boxOffice. In previous versions of akka you could use the actorFor method to directly get an ActorRef to the remote actor. This method has been deprecated since the returned ActorRef did not behave exactly the same way as a local ActorRef in case the related actor died. An ActorRef returned by actorFor could point to a newly spawned remote actor instance while this was never the case

in a local context. At the time Remote Actors could not be watched for termination like local Actors which was another reason for the need to deprecate this method.

Which brings us to the reason for the RemoteLookup actor;

- The backend actor system might not have started up yet, or it could have crashed or it could have been restarted.
- The boxOffice actor itself could also have crashed and restarted.
- Ideally, we would start the backend node before the frontend, so the frontend could do the lookup once at startup.

The RemoteLookup actor will take care of these scenarios. Figure 6.5 shows how the RemoteLookup sits between the RestInterface and the BoxOffice. It transparently forwards messages for the RestInterface.

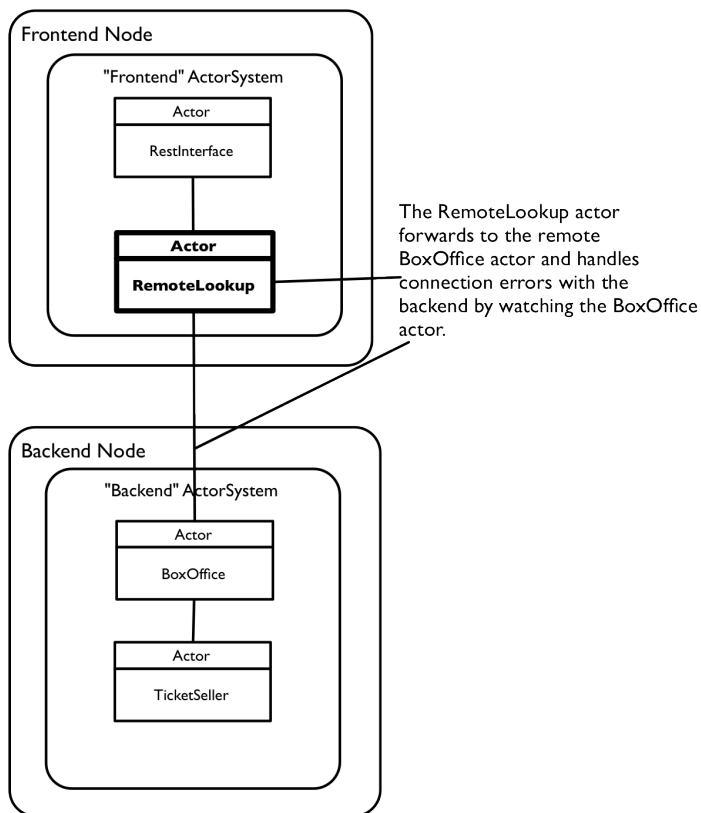


Figure 6.5 RemoteLookup actor

The RemoteLookup actor is a state machine that can only be in one of two states we have defined: identify or active. It uses the become method to switch its receive method to identify or active. The RemoteLookup tries to get a valid ActorRef to the BoxOffice when it does not have one yet in the identify state or it forwards all messages sent to a valid ActorRef to the BoxOffice in the active state.

If the `RemoteLookup` detects that the `BoxOffice` has been terminated it tries to get a valid `ActorRef` again when it receives no messages for a while. We'll use `RemoteDeathwatch` for this. Sounds like something new but from the perspective of API usage it's exactly the same thing as normal actor monitoring/watching. Listing 6.8 shows the code:

Listing 6.8 Remote Lookup

```

import scala.concurrent.duration._

class RemoteLookup(path:String) extends Actor with ActorLogging {
  context.setReceiveTimeout(3 seconds) ❶
  sendIdentifyRequest() ❷

  def sendIdentifyRequest(): Unit = {
    val selection = context.actorSelection(path) ❸
    selection ! Identify(path) ❷
  }

  def receive = identify ❺
  def identify: Receive = {
    case ActorIdentity(`path`, Some(actor)) => ❻
      context.setReceiveTimeout(Duration.Undefined) ❼
      log.info("switching to active state")
      context.become(active(actor)) ❽
      context.watch(actor) ❾

    case ActorIdentity(`path`, None) => ❿
      log.error(s"Remote actor with path $path is not available.")

    case ReceiveTimeout =>
      sendIdentifyRequest() ❶❶

    case msg:Any =>
      log.error(s"Ignoring message $msg, not ready yet.") ❶❷
  }

  def active(actor: ActorRef): Receive = { ❶❸
    case Terminated(actorRef) => ❶❹
      log.info("Actor $actorRef terminated.")
      context.become(identify)
      log.info("switching to identify state")
      context.setReceiveTimeout(3 seconds)
      sendIdentifyRequest()

    case msg:Any => actor forward msg ❶❺
  }
}

```

- ❶ Send a ReceiveTimeout message if no message has been received for 3 seconds
- ❷ Immediately start to request the identity of the actor
- ❸

- Select the actor by path
- ④ Send an Identify message to the actorSelection
- ⑤ The actor is initially in identify receive state
- ⑥ The actor has been identified and an ActorRef to it is returned
- ⑦ No longer send a ReceiveTimeout if the actor gets not messages since it is now active.
- ⑧ Change to active receive state
- ⑨ Watch the remote actor for termination
- ⑩ The actor is not (yet) available, the backend is unreachable or not started
- ⑪ Keep trying to identify the remote actor if no message is received.
- ⑫ No messages are sent in the identify receive state
- ⑬ The active receive state.
- ⑭ If the remote actor is terminated the RemoteLookup actor should change its behavior to the identify receive state
- ⑮ Forward all other messages when the remote actor is active.

As you can see Death watch / monitoring API which was described in chapter 3 is exactly the same for local and remote actors. Simply watching an ActorRef will make sure the actor gets notified of termination of an actor regardless if it's remote or local. Akka uses a very sophisticated protocol to statistically detect that a node is unreachable. We will look at this protocol in more detail in chapter 13. The ActorRef to the boxOffice is retrieved using a special `Identify` message which is sent to the ActorSelection. The remote module of the backend ActorSystem responds with an `ActorIdentity` message which contains the ActorRef to the remote actor.

That concludes the changes we had to make to the goticks.com app to move from a single node to a frontend and backend node. Apart from being able to communicate remotely, the frontend and backend can boot separately, the frontend will lookup the boxoffice and can communicate with it when it is available and can take action when it is not.

The last thing you could do is actually run the `FrontendMain` and `BackendMain` classes. We'll startup two terminals and use `sbt run` to run a main class in the project. You should get the following output in the terminals:

```
[info] ...
[info] ... (sbt messages)
[info] ...
Multiple main classes detected, select one to run:
[1] com.goticks.SingleNodeMain
[2] com.goticks.FrontendMain
[3] com.goticks.BackendMain
Enter number:
```

Select FrontendMain in one terminal and BackendMain in another. See what happens if you kill the sbt process that runs the BackendMain and restart it again. You can test if the app works with the same httpie commands as before, for instance `http PUT localhost:8000/events event=RHCP nrOfTickets:=10` to create an event with 10 tickets and `http GET localhost:5000/ticket/RHCP` to get a ticket to the event. If you did try to kill the backend process and start it up again you will see in the console that the RemoteLookup class switches from active to identify and back. You will also notice that Akka reports errors about the remote connection to the other node. If you are not interested in the logging of these remote lifecycle events you can switch the logging off by adding the below to the remote config section:

```
remote {
  log-remote-lifecycle-events = off
}
```

The remote lifecycle events are logged by default. This makes it easier to find problems when you start out with the remote module and for instance make a minor mistake in the actor path syntax. You can subscribe to the remote lifecycle events using the actor system's *eventStream* which is described in chapter 10 on Channels. Since remote actors can be watched like any local actor there is no need to act upon these events individually for the sake of connection management.

Lets review the changes:

- The BoxOfficeCreator trait was extracted from the code. A Remote version was added to lookup the BoxOffice on the backend.
- The RemoteBoxOfficeCreator adds a RemoteLookup Actor in between the RestInterface and the BoxOffice. It forwards all messages it receives to the Boxoffice. It identifies the ActorRef to the BoxOffice and remotely monitors it.

As said in the beginning of this section Akka provides two ways to get an ActorRef to a remote actor. In the next section we will look at the second option, namely remote deployment.

6.2.4 Remote Deployment

Remote deployment can be done programmatically or through configuration. We will start with the preferred approach: configured. Of course, this is preferred because changes to the cluster settings can be made without rebuilding the app. The standard `BoxOfficeCreator` trait creates the `boxOffice` as a child of the Actor it is mixed in with, namely the `RestInterface`:

```
val boxOffice = context.actorOf(Props[BoxOffice],
                               "boxOffice")
```

The local path to this actor would be `/restInterface/boxOffice`, omitting the user guardian actor. When we use configured remote deployment all we have to do is tell the frontend actor system that when an actor is created with the path `/restInterface/boxOffice` it should not create it locally but remotely. This is done with the piece of configuration in listing 6.9:

Listing 6.9 Configuration of the RemoteActorRefProvider

```
actor {
  provider = "akka.remote.RemoteActorRefProvider"
  deployment {
    /restInterface/boxOffice { ❶
      remote = "akka.tcp://backend@0.0.0.0:2552" ❷
    }
  }
}
```

- ❶ An actor with this path will be deployed remotely
- ❷ The remote address where the actor should be deployed. the ip address or host name has to match exactly with the interface the remote backend actor system is listening on.

Remote deployment can also be done programmatically which is shown for completeness sake. In most cases it is better to configure the remote deployment of actors through the configuration system (using properties), but in some cases, perhaps if you are referencing different nodes by CNAMEs (which are themselves configurable) for example, you might just do the configuration in code. Fully dynamic remote deployment makes more sense when using the akka-cluster

module since it is built specifically to support dynamic membership. An example of programmatic remote deployment is shown in listing 6.10.

Listing 6.10 Programmatic Remote Deploy Configuration

```
val uri = "akka.tcp://backend@0.0.0.0:2552"
val backendAddress = AddressFromURIString(uri) ❶

val props = Props[BoxOffice].withDeploy(
  Deploy(scope = RemoteScope(backendAddress))
) ❷

context.actorOf(props, "boxOffice")
```

- ❶ Create an address to the backend from the uri
- ❷ Create a Props with a remote deployment scope

The above code creates and deploys the `boxOffice` remotely to the backend as well. The Props configuration object specifies a remote scope for deployment.

It is important to note that remote deployment does not require that Akka automatically deploys the actual class file(s) for the `BoxOffice` actor into the remote actor system in some way; the code for the `BoxOffice` needs to already be present on the remote actor system for this to work and the remote actor system needs to be running. If the remote backend actorsystem crashes and restarts the `ActorRef` will not automatically point to the new remote actor instance. Since the actor is going to be deployed remotely it cannot already be started by the backend actor system as we did in the `BackendMain`. Because of this a couple of changes have to be made. The following Main classes are defined:

```
// the main class to start the backend node.
object BackendRemoteDeployMain extends App {
  val config = ConfigFactory.load("backend")
  val system = ActorSystem("backend", config) ❶
}

object FrontendRemoteDeployMain extends App {
  val config = ConfigFactory.load("frontend-remote-deploy")
  val host = config.getString("http.host")
  val port = config.getInt("http.port")
  val system = ActorSystem("frontend", config)

  val restInterface = system.actorOf(Props[RestInterface], ❷
    "restInterface")
  Http(system).manager ! Bind(listener = restInterface,
    interface = host,
```

```
port =port)
}
```

- ❶ Not creating the boxOffice actor anymore
- ❷ Not mixing in a specific trait anymore, using the default BoxOfficeCreator

When you run these main classes with two terminals like before and create some events with `httpie` you will see something similar to the below message in the console of the frontend actor system:

```
// very long message, formatted in a couple of lines to fit.
INFO [RestInterface]: Received new event Event(RHCP,10), sending to
Actor[akka.tcp://backend@0.0.0.0:2552/remote/akka.tcp/
  frontend@0.0.0.0:2551/user/restInterface/boxOffice#-1230704641]
```

Which shows that the frontend actor system is actually sending a message to the remote deployed boxOffice. The actor path is different than you would expect. It keeps track of where the actor was deployed from. The remote daemon that listens for the backend actorsystem uses this information to communicate back to the frontend actorsystem.

What we have worked up so far works, but there is one problem with this approach. If the backend actor system is not started when the frontend tries to deploy the remote actor the deployment obviously fails, but what is maybe not so obvious is that the ActorRef is still created. Even if the backend actor system is started later the created ActorRef does not work. This is correct behavior since it is not the same actor instance. (As distinguished from the prior failure cases we saw, where only the actor itself is restarted, in which case the ref will still point to the recreated Actor.)

If we want to do something when the remote backend crashes or the remote boxOffice actor crashes we will have to make some more changes. We'll have to watch the boxOffice ActorRef like we did before and take actions when this happens. Since the RestInterface has a val reference to the boxOffice we will need to once again put an Actor in between the way we did with the RemoteLookup actor. This in between actor will be called `RemoteBoxOfficeForwarder`.

The configuration needs to be changed slightly since the boxOffice now has the path `restInterface/forwarder/boxOffice` because of the

`RemoteBoxOfficeForwarder` in between. instead of the `/restInterface/boxOffice` path in the deployment section it should now read as `/restInterface/forwarder/boxOffice`.

Listing 6.11 shows the `ConfiguredRemoteBoxOfficeDeployment` trait and the `RemoteBoxOfficeForwarder` that will watch the remote deployed actor.

Listing 6.11 Watch Mechanisms for Remote Actors

```

trait ConfiguredRemoteBoxOfficeDeployment
  extends BoxOfficeCreator { this:Actor =>

  override def createBoxOffice = {
    context.actorOf(Props[RemoteBoxOfficeForwarder],
      "forwarder") ❶
  }
}

class RemoteBoxOfficeForwarder extends Actor with ActorLogging {
  context.setReceiveTimeout(3 seconds)

  deployAndWatch() ❷

  def deployAndWatch(): Unit = {
    val actor = context.actorOf(Props[BoxOffice], "boxOffice")
    context.watch(actor) ❸
    log.info("switching to maybe active state")
    context.become(maybeActive(actor)) ❹
    context.setReceiveTimeout(Duration.Undefined)
  }

  def receive = deploying

  def deploying:Receive = {

    case ReceiveTimeout =>
      deployAndWatch()

    case msg:Any =>
      log.error(s"Ignoring message $msg, not ready yet.")
  }

  def maybeActive(actor:ActorRef): Receive = {
    case Terminated(actorRef) => ❺
      log.info("Actor $actorRef terminated.")
      log.info("switching to deploying state")
      context.become(deploying)
      context.setReceiveTimeout(3 seconds)
      deployAndWatch()
    case msg:Any => actor forward msg
  }
}

```

- ❶ Create a forwarder that watches and deploys the remote BoxOffice
- ❷ Remotely deploy and watch the BoxOffice
- ❸ watch the remote BoxOffice for termination
- ❹

- ☞ Switch to 'maybe active' once the actor is deployed. We can't be sure without using lookup if the actor is deployed.
- 5 The deployed boxoffice is terminated so it's certain that a retry deployment is needed.

The above `RemoteBoxOfficeForwarder` looks very similar to the `RemoteLookup` class in the previous section in that it is also a state machine, in this case it is in one of two states; 'deploying' or 'maybe active'. Without doing a actor selection lookup we can't be sure that the remote actor is actually deployed. The exercise to add remote lookup with `actorSelection` to the `RemoteBoxOfficeForwarder` is left to the reader, for now the 'maybe' active state will do.

The `Main` class for the frontend needs to be adapted to mix in the `ConfiguredRemoteBoxOfficeDeployment` into the `RestInterface`. The `FrontendRemoteDeployWatchMain` class shows how this trait is mixed in:

```
class RestInterfaceWatch extends RestInterface
with ConfiguredRemoteBoxOfficeDeployment

val restInterface = system.actorOf(Props[RestInterfaceWatch],
"restInterface")
```

Running the `FrontendRemoteDeployWatchMain` and the `BackendRemoteDeployMain` on two `sbt` console terminals shows how the remote deployed actor is watched and how it is redeployed when the backend process is killed and restarted again, or when the frontend is started before the backend.

In case you just read over the previous paragraph and though 'meh,' read that paragraph again. The app is automatically redeploying an actor when the node it runs on reappears and continues to function. This is cool stuff and we've only scratched the surface!

That concludes this section on remote deployment. We've looked at both remote lookup and remote deployment and what is required to do this in a resilient way. Even in the situation where you only have two servers it's a major benefit to have resilience built in from the start. In both lookup and deployment examples the nodes are free to startup in any order. The remote deployment example could have been done purely by changing the deployment configuration but we would have

ended up with a too naive solution which did not take node or actor crashes into consideration and would have required a specific startup order.

In the section 6.26 we're going to look at the multi-jvm sbt plugin and the akka-multi-node-testkit which makes it possible to test the frontend and backend nodes in the goticks app.

6.2.5 Multi-JVM testing

The sbt multi-jvm plugin makes it possible to run tests across multiple JVMs, which we will want to do now that we are making the app distributed. The sbt multi-jvm plugin needs to be registered with sbt in the project/plugins.sbt file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-multi-jvm" % "0.3.5")
```

We also have to add another sbt build file to use it. the Multi-JVM plugin only supports the scala DSL version of SBT project files so we need to add a GoTicksBuild.scala file in the chapter6/project folder. SBT merges the build.sbt and the below file automatically, which means that the dependencies don't have to be duplicated in listing 6.12.

Listing 6.12 Multi-JVM Configuration

```

import sbt._
import Keys._
import com.typesafe.sbt.SbtMultiJvm
import com.typesafe.sbt.SbtMultiJvm.MultiJvmKeys.{ MultiJvm }

object GoTicksBuild extends Build {

  lazy val buildSettings = Defaults.defaultSettings ++
    multiJvmSettings ++
    Seq(
      crossPaths := false
    )

  lazy val goticks = Project(
    id = "goticks",
    base = file("."),
    settings = buildSettings ++ Project.defaultSettings
  ) configs(MultiJvm)

  lazy val multiJvmSettings = SbtMultiJvm.multiJvmSettings ++
    Seq(
      compile in MultiJvm <=<
        (compile in MultiJvm) triggeredBy (compile in Test) ❷
      parallelExecution in Test := false, ❸
      executeTests in Test <=<
        ((executeTests in Test), (executeTests in MultiJvm)) map {
          case (_, testResults), (_, multiJvmResults) =>
            val results = testResults ++ multiJvmResults
            (Tests.overall(results.values), results)
        }
    )
}

```

- ❶ Make sure our tests are part of the default test compilation
- ❷ Turn off parallel execution
- ❸ Make sure executed as part of default test target

If you're not an SBT expert, don't worry about the details of this build file here. The above basically configures the multi-jvm plugin and makes sure that multi-jvm tests are executed along with the normal unit tests. *SBT in Action* (<http://www.manning.com/suereth2/>) does a great job at explaining the details of SBT if you would like to know more about it.

Multi JVM tests need to be added to the `src/multi-jvm/scala` folder by default. Now that our project is setup correctly for multi-jvm tests we can start with

a unit test for the frontend and backend of the goticks.com app. First a `MultiNodeConfig` needs to be defined which describes the roles of the nodes that are tested. The below listing shows the multi node config for the client server (frontend and backend) configuration:

```
object ClientServerConfig extends MultiNodeConfig {
  val frontend = role("frontend") ❶
  val backend = role("backend") ❷
}
```

- ❶ The frontend role
- ❷ The backend role

Two roles have been defined, the frontend and the backend as you would expect. The roles will be used to identify the node for unit testing and to run specific code on each node for testing purposes. Before we start to write a test we need to write some infrastructure code to hookup the test into scalatest:

```
import akka.remote.testkit.MultiNodeSpecCallbacks
import org.scalatest.{BeforeAndAfterAll, WordSpec}
import org.scalatest.matchers.MustMatchers

trait STMultiNodeSpec extends MultiNodeSpecCallbacks ❶
  with WordSpec with MustMatchers with BeforeAndAfterAll { ❷

  override def beforeAll() = multiNodeSpecBeforeAll() ❸

  override def afterAll() = multiNodeSpecAfterAll()
}
```

- ❶ Get callbacks by extending TestKit's class
- ❷ Get the rest of the test traits we need
- ❸ Make all our tests use our before and after methods

This trait is used to startup and shutdown the multi node test which you can reuse for all your multi node tests. It is mixed into the unit test specification. Now for the test which is shown below. It's quite a bit of code so let's break it down. The first thing we need to do is create a `MultiNodeSpec` which mixes in the

STMultiNodeSpec we just defined. Two versions of the ClientServerSpec will need to run on two separate JVMs. The code in listing 6.13 shows how two ClientServerSpec classes are defined for this purpose.

Listing 6.13 Spec Classes for Multi Node Tests

```
class ClientServerSpecMultiJvmFrontend extends ClientServerSpec ❶
class ClientServerSpecMultiJvmBackend extends ClientServerSpec ❷

class ClientServerSpec extends MultiNodeSpec(ClientServerConfig) ❸
with STMultiNodeSpec with ImplicitSender {
  def initialParticipants = roles.size ❹
}
```

- ❶ The Spec that will run on the 'frontend' JVM
- ❷ The Spec that will run in the 'backend' JVM
- ❸ The number of nodes that participate in the test.
- ❹ The Spec which describes what both nodes should do.

The ClientServerSpec uses the STMultiNodeSpec and also an ImplicitSender trait. The ImplicitSender trait sets the testActor as the default sender for all messages, which makes it possible to just call expectMsg and other assertion functions without having to set the testActor as the sender of messages every time. The code in listing 6.14 shows how we make this happen.

Listing 6.14 Configuring the TestActor

```
import ClientServerConfig._ ❶

trait TestRemoteBoxOfficeCreator
  extends RemoteBoxOfficeCreator { this:Actor => ❷

  override def createPath: String = { ❸
    val actorPath = node(backend) / "user" / "boxOffice" ❹
    actorPath.toString
  }
}
```

- ❶ Import the config so we can access the backend role

- 2 The `TestRemoteBoxOfficeCreator` will be used in the test instead of the `RemoteBoxOfficeCreator`.
- 3 override the `createPath` method so it can return a path to the test system on the backend node for testing
- 4 The `node()` method returns the address of the backend role node during test. the expression here creates an `ActorPath`.

The backend and frontend role nodes run on a random port by default. The `TestRemoteBoxOfficeCreator` replaces the `RemoteBoxOfficeCreator` in the test since it creates a path from a configured host, port and actor name in the `frontend.conf` file. Instead we want to use the address of the backend role node during testing and lookup a reference to the `boxOffice` actor on that node. The above code achieves this. Listing 6.15 shows tests of our distributed architecture.

Listing 6.15 Testing the Distributed Architecture

```

"A Client Server configured app" must {
  "wait for all nodes to enter a barrier" in {
    enterBarrier("startup") ❶
  }

  "be able to create an event and sell a ticket" in { ❷

    runOn(frontend) { ❸
      enterBarrier("deployed") ❹

      val restInterface = system.actorOf(
        Props(new RestInterfaceMock
              with TestRemoteBoxOfficeCreator)) ❺

      val path = node(backend) / "user" / "boxOffice"
      val actorSelection = system.actorSelection(path) ❻

      actorSelection.tell(Identify(path), testActor) ❼

      val actorRef = expectMsgPF() {
        case ActorIdentity(`path`, ref) => ref ❽
      }

      restInterface ! Event("RHCP", 1)

      expectMsg(EventCreated) ❾

      restInterface ! TicketRequest("RHCP")

      expectMsg(Ticket("RHCP", 1))
    }

    runOn(backend) { ❿
      system.actorOf(Props[BoxOffice], "boxOffice") ⓫
      enterBarrier("deployed") ❸
    }

    enterBarrier("finished") ⓬
  }
}

```

❶

- Let all nodes startup
- ② Test scenario for the frontend and backend node
- ③ Run the code in this block on the frontend JVM
- ④ Wait for the backend node to deploy.
- ⑤ Create a mock Rest Interface.
- ⑥ get an actor selection to the remote box office
- ⑦ Send an Identify message to the actor selection
- ⑧ Wait for the boxOffice to report that it is available. The RemoteLookup class will go through the process of getting an ActorRef to the boxOffice.
- ⑨ Expect messages as usual with the TestKit.
- ⑩ Run the code in this block on the backend JVM
- ⑪ Create the boxOffice with name "boxOffice" so the RemoteLookup class can find it.
- ⑫ Signal that the backend is deployed.
- ⑬ Indicate that the test has completed.

There is quite a lot going on here. The unit test can be broken up in four pieces. First it waits for all nodes to start by using the `enterBarrier("startup")` call which executes on both nodes. The actual test then continues to specify what code should be run on the frontend node and the backend node. The frontend node waits for the backend node to signal that it is deployed and executes a test.

The backend node only starts the boxOffice so it can be used from the frontend node. Since we would have to add HTTP client requests if we would use the real RestInterface for now we'll use a RestInterfaceMock class. This actor mixes in the TestRemoteBoxOfficeCreator trait which confers behavior that is nearly identical to the RemoteBoxOfficeCreator trait, except that it gets the path from the backend node under test. Since the RemoteLookup actor is still used (the createBoxOffice method is not overridden) we will need to wait for the remote actorRef to have been identified. The actorSelection is used for this purpose and we expect a ActorIdentity message before we start sending messages to the remote boxOffice for testing.

After that we can finally test the interactions between the frontend and the backend node. We can use the same methods that we used in chapter 2 for expecting messages. This multi-jvm test can be run by executing the `multi-jvm:test` command in sbt, give it a try.

Figure 6.6 shows how the test actually flows. Note that the coordination of the various collaborators, and their runtimes, is made pretty much automatic by the multi-jvm test kit. Doing this with your own hand hewn code would be a lot of work.

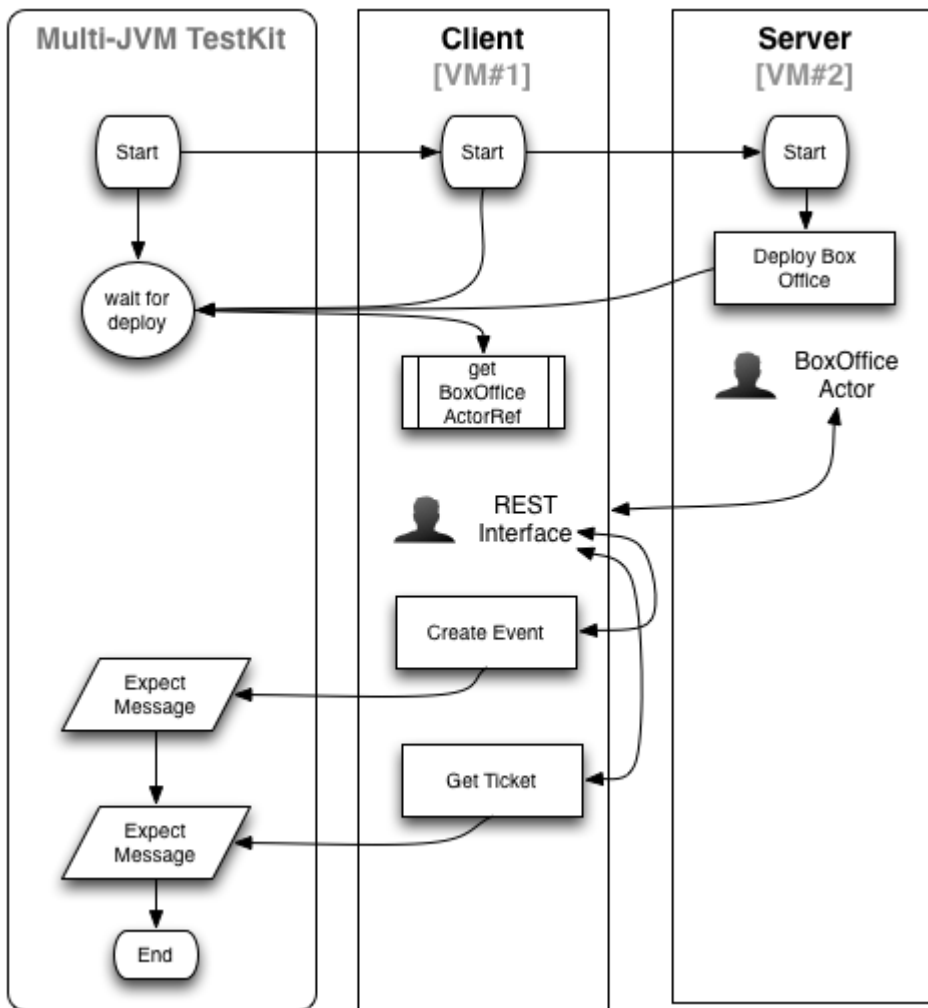


Figure 6.6 Multi-JVM Test Flow

The chapter6 project also has a unit test for a single node version of the app and apart from some of the infrastructure setup the test is basically the same. The example multi-jvm test here just shows how an app that was initially built for a single node can be adapted to run on two nodes. The big difference between the single node and the client server setup is how the actor reference to the remote system is found; is it looked up or deployed remotely. Having a Remote Lookup in between the RestInterface and the boxOffice gave as some flexibility and the ability to survive crashes. It gave an interesting problem to solve in the example unit test; how do we wait for the remote ActorRef to the boxOffice to become available? the actorSelection and Identity message mechanism were the answer for this.

This concludes our first look at the multi-node-testkit module. We'll see more of it in the chapters to come. The test above shows an example of how the goticks.com app can be unit tested in a distributed environment. In this case it runs

on two JVM's on a single machine. As we will see later in chapter 13 the multi-node-testkit can also be used to run unit tests on several servers.

6.3 Summary

Do you remember the reason we gave at the beginning of the chapter for why we couldn't just flip a switch and have our app work in a distributed fashion (using remoting)? It was because we have circumstances that we have to account for in the network world that our local only app is able to completely ignore. As you would expect, much of what we ended up having to actually do in this chapter boiled down to just accounting for those new circumstances, and as predicted, Akka made it easy.

Despite the fact that we had to make some changes, we also found a lot of constancy:

- We benefited from the fact that an ActorRef behaves the same whether the Actor is local or remote.
- The monitoring API for death watch of distributed systems is exactly the same as for local systems.
- Despite the fact that collaborators were now separated by the network, by simply using forwarding (in the RemoteLookup and RemoteBoxOfficeForwarder), we transparently allowed the RestInterface and BoxOffice to communicate with each other.

This is important because taking our app to the next level does not require that we either unlearn what we have learned, or learn a whole new load of stuff; the basic operations remain largely the same, which is the hallmark of a well designed toolkit.

We also learned a some new things:

- REPL provides us an easy, interactive means of getting our stuff going in the distributed topology of our choice.
- The multi-node-testkit which makes it incredibly easy to test distributed actor systems no matter if they are built with akka-remote, akka-cluster or even both. (Akka is rather unique in providing proper unit testing tools for a distributed environment.)

We have intentionally not dealt with the fact that messages will get lost in the `RemoteLookup` and `RemoteBoxOfficeForwarder` when the backend node is not available. In upcoming chapters, we will:

- see how a `Reliable Proxy` can be used for messaging between peer nodes
- fix goticks to deal with the fact that the state of the `TicketSellers` is lost when a backend node crashes
- how state can be replicate across a cluster

But before we get there lets look at dynamic node memberships with the `akka-cluster` module in the next chapter.

Configuration, Logging and Deployment

In this chapter

- configuration
- logging
- stand-alone applications
- web applications
- deployment

Thus far, we have focused on creating actors and working with the actor system. To create an application which can actually be run, we will need several other things to be bundled with it before it's ready for deployment. First, we'll dive into how Akka supports configuration, then we will look at logging, including how you can use your own logging framework. Then we will go through two deployment examples: the first will be a stand-alone application and the second a web-based one.

7.1 Configuration

Akka uses the Typesafe Config Library, which sports a pretty state-of-the-art set of capabilities. Of course, the typical features are there: the ability to define properties in different ways and then reference them in the code (job one of configuration is to grant us runtime flexibility by making it possible to use variables outside the code). There is also a sophisticated means of merging multiple configuration files based on simple conventions that determine how overrides will occur. One of the most important requirements of a configuration system is providing us a means of targeting multiple environments (e.g. development, testing, production), without having to explode the bundle. We will see how that is done as well.

7.1.1 Trying Out Akka Configuration

Like other Akka libraries, the Typesafe Config Library also takes pains to minimize the dependencies that are needed; it has no dependencies on other libraries. We start with a quick tour of how to use the configuration library.

First the library uses a hierarchy of properties.

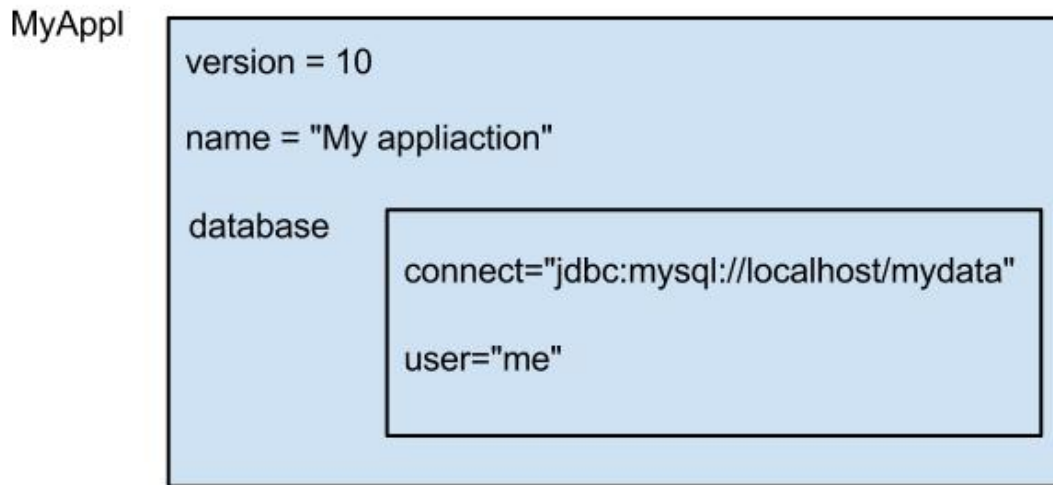


Figure 7.1 Configuration example

To get the configuration, the `ConfigurationFactory` is used. The library of course also supports the ability to specify which configuration file is used, and in the next sections, we will be looking at the configuration files in more detail, but for now we will start by using the default one.

Listing 7.1 Getting configuration

```
val config = ConfigFactory.load()
```

When using the default, the library will try to find the configuration file. Since the library supports a number of different configuration formats, it looks for different files, in the following order:

- **application.properties**
This file should contain the configuration properties in the java property file format.
- **application.json**
This file should contain the configuration properties in the json style
- **application.conf**

This file should contain the configuration properties in the HOCON format. This is a format based on json but easier to read..

It is possible to use all the different files at the same time. For the example below, in listing 7.2 we use the last file:

Listing 7.2 application.conf

```
MyAppl {
  version = 10
  description = "My application"
  database { ❶
    connect="jdbc:mysql://localhost/mydata"
    user="me"
  }
}
```

❶ Nesting is done by simply grouping with {}s

For simple applications, this file will often suffice. The format looks kind of like JSON. The primary advantage is that it's more readable and it's easy to see how properties are being grouped. JDBC is a perfect example of properties most apps will need that are better grouped together. In the dependency injection world, you would group items like this by controlling the injection of the properties into objects (e.g. DataSource). This is a simpler approach. Let's look at how we can make use of these properties, now that we've define them.

There are several methods to get the values as different types and the "." is used as the separator in the path of the property.

Listing 7.3 Getting properties

```
val applicationVersion = config.getInt("MyAppl.version")
val databaseConnectSting = config.getString("MyAppl.database.connect") ❶
```

❶ we can use the connect string from inside the database {}s from the prior listing

Sometimes, an object doesn't need much configuration. What if we have an object that is creating the database connection. It needs only the connect string and the user. When we pass the configuration, the object needs to know the path of the

property. But when you want to reuse that object a problem rises. The start of the path is "MyAppl" another application has probably another configuration root. And therefore, the path to the property has changed. This can be solved by using the functionality of getting a subtree as configuration.

Listing 7.4 Getting a configuration subtree

```
val databaseCfg = configuration.getConfig("MyAppl.database") ❶
val databaseConnectSting = databaseCfg.getString("connect") ❷
```

- ❶ First get the subtree by name
- ❷ Then reference the property as relative to the subtree root

Using this approach, you give the databaseCfg to the object and it doesn't need the full path of the property, only the last part, the name of the property. This means the object can be reused without introducing path problems.

It is also possible to perform substitutions when you have a property that is used multiple times in your configuration, for example the host name of the database connect string.

Listing 7.5 substitution

```
hostname="localhost" ❶
MyAppl {
  version = 10
  description = "My application"
  database {
    connect="jdbc:mysql://${hostname}/mydata" ❷
    user="me"
  }
}
```

- ❶ Simple variable definition, no types needed of course (note quotes though)
- ❷ Then the familiar \${} substitution syntax

Config file variables are often used for things like the application name, or for version numbers, as repeating them in many places in the file could potentially be dangerous. It is also possible to use system properties or environment variables in the substitutions as well.

Listing 7.6 system property or environment variable substitution

```
hostname=${?HOST_NAME} ❶
MyAppl {
  version = 10
  description = "My application"
  database {
    connect="jdbc:mysql://${hostname}/mydata"
    user="me"
  }
}
```

- ❶ The ? signifies getting the value from an environment variable

But the problem with these properties is that you never know for sure that these properties exist. To solve this we can make use of the possibility that redefinition of a property overrules the previous definition. And the substitution of a system property or environment variable definition simply vanishes if there's no value for the specified property `HOST_NAME`. Listing 7.7 shows how to do this.

Listing 7.7 system property or environment variable substitution

```
hostname="localhost" ❶
hostname=${?HOST_NAME} ❷
MyAppl {
  version = 10
  description = "My
  application"
  database {
    connect="jdbc:mysql://${hostname}/mydata"
    user="me"
  }
}
```

- ❶ Define the usual simple way first
 ❷ If there is an env var, override, otherwise, leave it with the value we just assigned

It's pretty easy to see what's going on here. Defaults are important in configuration because we want to force the user to do as little configuration as

possible. Furthermore, it's often the case that apps should run with no configuration until they really need to be pushed into a production environment; development usage can often be done with nothing but defaults.

7.1.2 Using Defaults

Let's continue with our simple JDBC configuration. It's generally safe to assume that developers will be connecting to a database instance on their own machine, referenced as 'localhost.' As soon as someone wants to see a demo, we will be scrambling to get the app working on an instance somewhere, that will no doubt have different names and the database will likely be on another machine. The laziest thing we could do is just make a copy of the whole config file and give it a different name, then have some logic in the app that says 'use this file in this environment, and this one in another.' The problem with this is that now we have all our configuration in 2 places. Makes more sense to just override the 2 or 3 values that are going to be different in the new target environment. The defaulting mechanism will allow us to do that easily. The configuration library contains a fall-back mechanism; the defaults are placed into a configuration object which is then handed over to the configurator as the fall-back configuration source. Figure 7.2 shows a simple example of this.

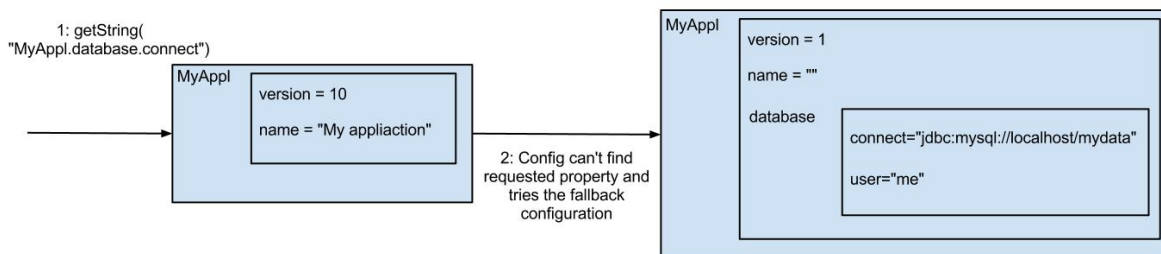


Figure 7.2 Configuration fall-back

SIDEBAR Preventing Null Properties

The defaulting mechanism prevents cases of the values being different depending on where they are being used. As a result of this principle, when a configuration property is read, the value should always be set. If the framework were to allow the property to be empty, again the code would behave differently based on how (and where) configuration was done. Therefore, when getting a property that isn't set, an exception is thrown.

This fall-back structure grants us a lot of flexibility. But of course, for it to provide the defaults we need, we have to know how to configure them. They are

configured in the file `reference.conf` and placed in the root of the jar file; the idea is that every library contains its own defaults. The configuration library will find all the `reference.conf` files and integrate these settings into the configuration fall-back structure. This way all the needed properties of a library will always have a default and the principle of having always getting some value back will be preserved. (Later, we'll see that we can also explicitly stipulate defaults programmatically as well.)

We already mentioned that the configuration library supports multiple formats. There's nothing stopping you from using multiple formats in a single application. Each file can be used as the fall-back of another file. And to be able to support the possibility of overruling properties with system properties, the higher ranking configuration contains these. The structure is always the same, so the relationships between defaults and overrides is likewise always the same. Figure 7.3 shows the files the config library uses to build the complete tree, in priority order.

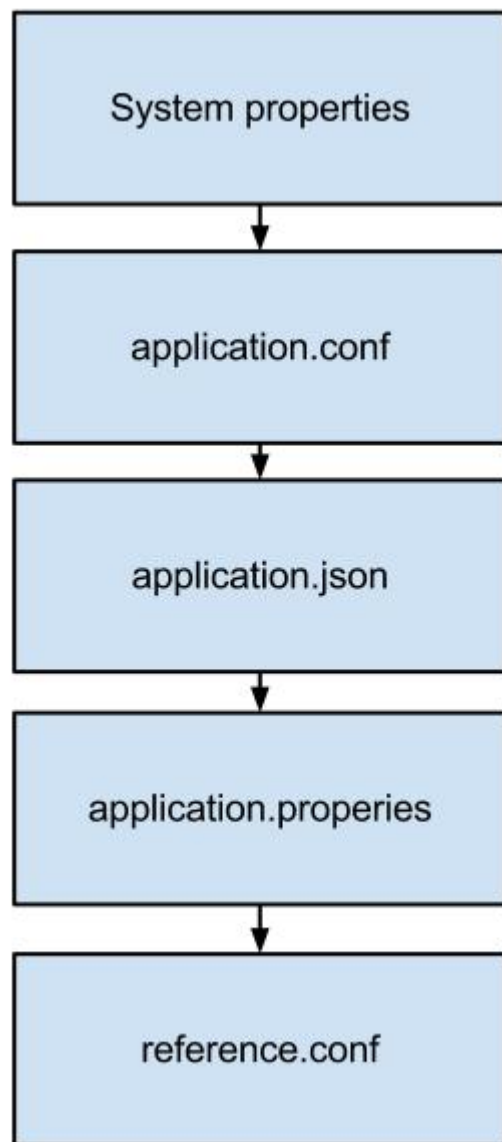


Figure 7.3 Priority of the Configuration fall-back structure

Most applications will use only one of these application file types. But if you want to provide on set of application defaults, then override some of them, as we want to do with our JDBC settings, we can do that and just following this guide, realize that the upper configurations shown in figure 7.3 will overrule the values defined in the lower configurations.

By default, the file `application.{conf,json,properties}` is used to read the configuration. There are two ways to change the name of the configuration file. The first option is to use an overloaded load function on the `ConfigurationFactory`. When loading the configuration, the name of the base should be supplied, as shown in figure 7.8.

Listing 7.8 Changing configuration file

```
val config = ConfigFactory.load("myapp") ❶
```

❶ Simply ask the factory to load our new name

This way it doesn't try to load `application.{conf,json,properties}`, but `myapp.{conf,json,properties}`. (This is required if you need to have multiple configurations in a single JVM.)

Another option is to use system properties. Sometimes, this is the easiest thing because you can just create a bash script and set a property and the app will pick it up and start using it (better than exploding jars or wars to monkey with the files inside).

- `config.resource` specifies a resource name - not a base-name, i.e. `application.conf` not `application`
- `config.file` specifies a file system path, again it should include the extension
- `config.url` specifies a URL

System properties can be used for the name of the configuration file, when using the `load` method without arguments. When using one of these properties the default behavior of searching for the different formats `conf`, `json` and `properties` is skipped.

7.1.3 Akka Configuration

OK we have seen how we can use the configuration library for our application's properties, but what do we need to do when we find ourselves wanting to change some of Akka's configuration options? How is Akka using this library? It is possible to have multiple `ActorSystems` which have their own configuration. When no configuration is present at creation, the actor system will use create the configuration using the defaults. Listing 7.9 shows what this looks like.

Listing 7.9 Default configuration

```
val system = ActorSystem("mySystem") ❶
```

❶

- Creating uses internally `ConfigFactory.load()`

But it is also possible (and useful) to supply the configuration while creating an `ActorSystem`. Listing 7.10 shows a simple way of accomplishing this.

Listing 7.10 Use specified configuration

```
val configuration = ConfigFactory.load("mysystem") ❶
val systemA = ActorSystem("mysystem", configuration) ❷
```

- ❶ First, load the configuration, providing our name
- ❷ then pass it to the `ActorSystem` constructor

The configuration is within your application; it can be found in the settings of the `ActorSystem`.

Listing 7.11 Access to the configuration from the running app

```
val mySystem = ActorSystem("myAppl")
val config = mySystem.settings.config ❶
val applicationDescription = config.getString("myAppl.name") ❷
```

- ❶ Once the `ActorSystem` is constructed, we can get the config just by referencing it using this path
- ❷ Then we just get a property as we would ordinarily

By this point, we have seen how we can use the configuration system for our own properties, and how to use the same system to configure the `ActorSystem` that is the backbone of Akka. The presumption through these first two sections has been that we have but one Akka app on the given system that is hosting us. In the next section, we will discuss configuring systems that share a single instance.

7.1.4 Multiple systems

Depending on your requirements, it may be necessary to have different configurations, say for multiple subsystems, on a single instance (or machine). There are several ways Akka supports this. Let's start by looking at cases where you are using several JVMs, but they run in the same environment using the same files. We already described the first option: the use of system properties. When starting a new process, a different configuration file is used. But most of the time a lot of the configuration is the same for all the subsystems and only a small part differs. This problem can be solved by using the include option.

Let us look at an example. Let's say we have a baseConfig file like the one in listing 7.12.

Listing 7.12 baseConfig.conf

```
MyAppl {
  version = 10
  description = "My application"
}
```

For this example, we start with this simple configuration root, which would most likely have one shared and one differing property: the version number is likely to be the same across subsystems, but we will probably want different names and descriptions for each subsystem.

Listing 7.13 subAppl.conf

```
include "baseConfig" ①
MyAppl {
  description = "Sub Application" ②
}
```

- ① simply name the config file we want to include (no extension)
- ② then provide the new description

Because the include is before the rest of the configuration, the value for description is overridden just as it was in a single file. This way, you can have one basic configuration and only the differences needed in the specific configuration

files for each subsystem.

But what if the sub systems are running in the same JVM. Then we can't use the system properties to read other configuration files. How should we do the configuration then? We have already discussed what's needed for this next case: we can use an application name when loading the configuration. And of course we can also use the include method to group all the configuration that is the same. The only drawback is the possible number of configuration files. If that's a concern, there is another solution that leverages the ability to merge configuration trees using the fall-back mechanism.

We start by combining the two configurations into one (see listing 7.14)

Listing 7.14 combined.conf

```
MyAppl {
  version = 10
  description = "My application"
}

subApplA {
  MyAppl { ❶
    description = "Sub application"
  }
}
```

❶ by lifting this, we get the shared property (version) and override the description

The trick we are using is that we take a subtree within subApplA of the configuration and put that in front of the configuration chain. This is called lifting a configuration, because the configuration path is shortened. Figure 7.4 shows how this is done.

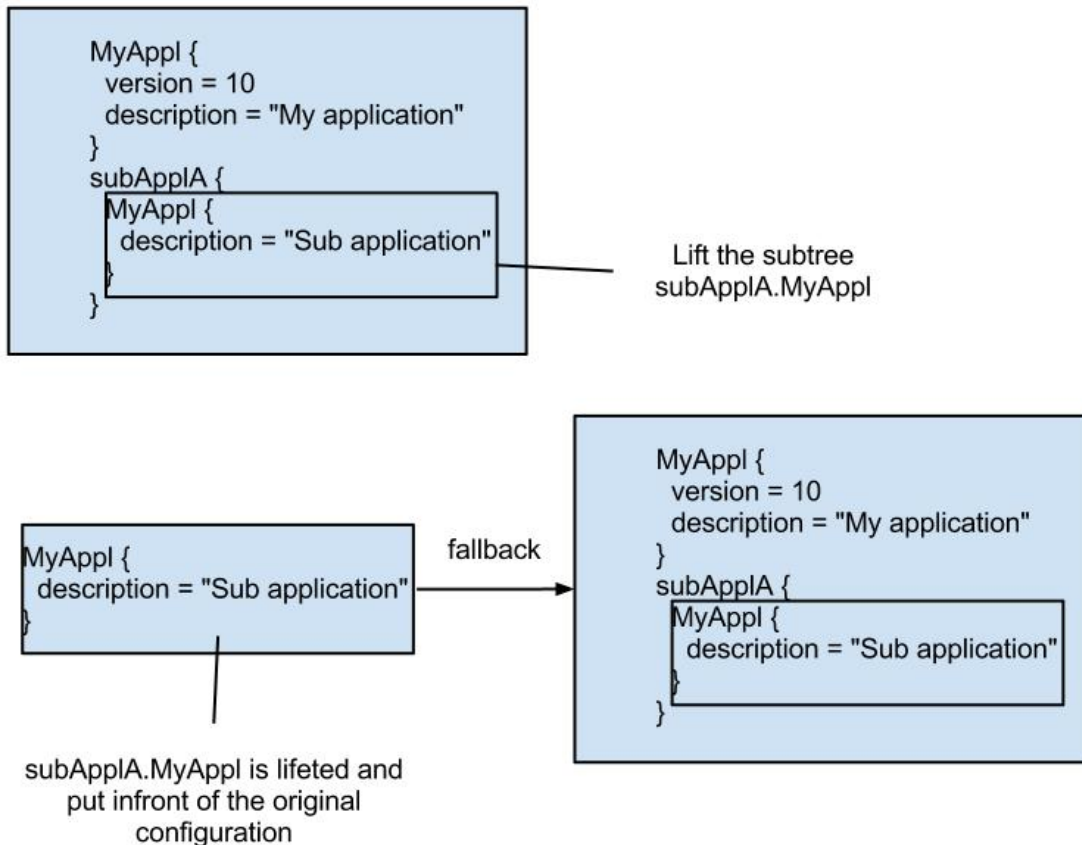


Figure 7.4 Lifting configuration part

When we request the property `MyAppl.description`, we get the result of "Sub application" because that was set in the configuration at the highest level and when we ask for `MyAppl.version` we get the value 10 because this wasn't defined in the higher configuration, so it uses the normal fall-back mechanism to get the value. Listing 7.15 shows how we load the configuration to have both the lift and the callback. Notice, the fallback is chained programmatically here (not relying on the file conventions we covered above).

Listing 7.15 Lift example with fallback

```

val configuration = ConfigFactory.load("combined")
val subApplAcf = configuration.getConfig("subApplA")
val config = subApplAcf.withFallback(configuration)

```

- 1 Select the subtree `subApplA`
- 2 Add the configuration as fall-back

Configuration is a crucial part of delivering applications: though it starts out with meager requirements that are usually easily met, invariably, demands appear that can complicate matters, and quite often the configuration layer of an application becomes very tangled and complex. The Typesafe Config Library gives us a number of powerful tools to prevent this from happening:

- Easy defaulting based on convention (with overrides)
- Sophisticated defaulting that allows us to require the least amount of configuration necessary
- Several syntax options from traditional, through Java, JSON, and HOCON.

We have not come near exhausting this topic, but we have shown you enough that you can deal with a pretty broad range of typical requirements that will come up as you start to deploy Akka solutions. In the next section, we tackle logging, which is not only critical, but developers tend to have strong opinions and they tend to want to use what they are used to. We will address how Akka allows this, through configuration.

7.2 Logging

Another function each application needs is to be able to write messages to a log file. Because everyone has their own preferences regarding which logging library to use, the Akka toolkit has implemented a logging adapter to be able to support all kinds of logging frameworks and also minimize the dependencies on other libraries. As was the case with configuration, there are two sides to logging: how you use the log for your application-level logging needs, and how you can control what Akka puts into the logs (which is a critical part of debugging). We'll cut the same path starting with the application use of logging.

7.2.1 Logging in an Akka Application

Just as you would in normal Java or Scala code, you are going to have to create a logger instance inside any Actor that needs to put messages into the log. Listing 7.16 shows how we do that.

Listing 7.16 Creating logging adapter

```
class MyActor extends Actor {
  val log = Logging(context.system, this)
  ...
}
```

The first thing that is notable is that the ActorSystem is needed. This is done so there is a separation of the logging from the used framework. The logging adapter uses the system eventStream to send the log messages to the eventhandler. The eventStream is the publish-describe system of Akka (which is described later). But for now the eventhandler receives these message and uses the preferred logging framework to log the message. This way all the actors can log and only one actor has a dependency on the specific logging framework implementation. Which eventHandler is used can be configured. Another advantage is that logging means IO and IO is always slow, and in a concurrent environment, this can be even worse because you have to wait until another thread is done writing its log messages. So in a high performance application you don't want to wait until the logging is done. Using the eventHandler, the actors logging don't have to wait. Listing XREF default-akka-logger-config shows the configuration required for the default event-handler to be created.

Listing 7.17 Configure eventHandler

```
akka {
  # Event handlers to register at boot time
  # (Logging$DefaultLogger logs to STDOUT)
  event-handlers = ["akka.event.Logging$DefaultLogger"]
  # Options: ERROR, WARNING, INFO, DEBUG
  loglevel = "DEBUG"
}
```

This eventHandler doesn't use a log framework, but logs all the received messages to standard out. When you want to create your own eventhandler you have to create an Actor which handles several messages. An example of such handler is

Listing 7.18 My eventHandler

```
import akka.event.Logging.InitializeLogger
import akka.event.Logging.LoggerInitialized
import akka.event.Logging.Error
import akka.event.Logging.Warning
import akka.event.Logging.Info
import akka.event.Logging.Debug
class MyEventListener extends Actor
{
  def receive = {
    case InitializeLogger(_) =>
      sender ! LoggerInitialized ❶
    case Error(cause, logSource, logClass, message) =>
      println( "ERROR " + message) ❷
    case Warning(logSource, logClass, message) =>
      println( "WARN " + message) ❸
    case Info(logSource, logClass, message) =>
      println( "INFO " + message) ❹
    case Debug(logSource, logClass, message) =>
      println( "DEBUG " + message) ❺
  }
}
```

- ❶ Upon receipt of this message, the initialization of your handler can be done, and when complete, a `LoggerInitialized` should be sent to the sender
- ❷ An error message is received, log this message or not. Here you can add some logic of filtering log records when your log framework doesn't support this
- ❸ A warning message is received
- ❹ An information message is received
- ❺ An debug message is received

This is a very simple example and is only showing the message protocol. In real life, this Actor will of course be more complex. The Akka toolkit has two implementations of this logging eventHandler. The first is already mentioned and that is the default logger to `STDOUT`. The second implementation is using `SLF4J`. This can be found in the `akka-slf4j.jar`. To use this handler add the following configuration to your `application.conf`

Listing 7.19 Use slf4j eventHandler

```
akka {
    event-handlers =
    [ "akka.event.slf4j.Slf4jEventHandler" ]
    # Options: ERROR,
    WARNING,
    INFO, DEBUG
    logLevel = "DEBUG"
}
```

7.2.2 Using Logging

Let's revisit the creation of the Akka logger instance that we first saw in listing 7.16. We discussed the first part of the creation process (the ActorSystem). If you recall, there was a second parameter; here it is again in listing 7.20.

Listing 7.20 Revisiting the creation of the logger

```
class MyActor extends Actor {
    val log = Logging(context.system, this)
    ...
}
```

The second parameter of the Logging is used to as the source of this logging channel. In this case, it is the class instance. The source object is translated to a String to indicate the source of the log messages. The translation to a string is done according to the following rules:

- if it is an Actor or ActorRef, its path is used
- in case of a String, the string is used
- in case of a class an approximation of its simpleName is used

For convenience you can also use the ActorLogging trait to mix-in the log member into actors. This is provided because most of the time you want to use the logging as shown in listing 7.21.

Listing 7.21 Creating logging adapter

```
class MyActor extends Actor with
  ActorLogging {
  ...
}
```

The adapter also supports the ability to use placeholders in the message. Placeholders prevent you from having to check logging levels. If you construct messages with concatenation, the work will be done each time, even if the level precludes the insertion of the message in the log! Using placeholders, you don't have to check the level (e.g. `if (logger.isDebugEnabled())`), and the message will only be created if it would be included in the log given the current level. The placeholder is the string "{}" in the message. Listing 7.22

Listing 7.22 Using place holders

```
log.debug("two parameters: {}, {}", "one", "two")
```

Nothing too disorienting here, most people who have been doing logging in Java or a VM language will find this fairly familiar. One of the other common logging challenges that can cause developers headaches is learning how to get control of the logging of the various toolkits or frameworks your app is using. For example, if you have a persistence provider, eventually you will need to be able to turn up its logging to see what it's doing. In the next section we will show how this is done with Akka.

7.2.3 Controlling Akka's logging

While developing an application, one needs sometimes very low debug level of logging. Akka is able to log records when certain internal events happen or the log has processed certain messages. These log messages are intended for developers and aren't meant for operations. Thankfully, given the architecture that we have discussed already, we don't have to worry about the possibility that our chosen logging framework and the one Akka uses are not the same, or worse, conflict with each other. Akka provides a simple configuration layer that allows you to exert some control over what it outputs to the log, and as we are both using the pubsub attached to a single adapter, once we change these settings, we will see the results in whatever our chosen appenders are (console, file, etc.). The Listing below (7.23) shows the settings we can manipulate to elicit more or less information from Akka in the logs.

Listing 7.23 Akka's Logging configuration file

```
akka {
  # logging must be set to
  # DEBUG to use any of the options below
  loglevel =      DEBUG
  # Log the complete configuration at INFO level when the actor
  # system is started. This is useful when you are uncertain of
  # what configuration is used.
  log-config-on-start = on
  debug {
    # logging of all user-level messages that are processed by
    # Actors that use akka.event.LoggingReceive enable function of
    # LoggingReceive, which is to log any received message at
    # DEBUG level
    receive = on 1
    # enable DEBUG logging of all AutoReceiveMessages
    # (Kill, PoisonPill and the like)
    autoreceive = on
    # enable DEBUG logging of actor lifecycle changes
    # (restarts, deaths etc)
    lifecycle = on
    # enable DEBUG logging of all LoggingFSMs for events,
    # transitions and timers
    fsm = on
    # enable DEBUG logging of subscription (subscribe/unsubscribe)
    # changes on the eventStream
    event-stream = on
  }
  remote {
    # If this is "on", Akka will log all outbound messages at
    # DEBUG level, if off then they are not logged
    log-sent-messages = on
    # If this is "on," Akka will log all inbound messages at
    # DEBUG level, if off then they are not logged
    log-received-messages = on
  }
}
```

- 1** Log messages received by your Actors, requires the use of `akka.event.LoggingReceive` when processing messages

The comments explain most of these options (see the annotation for the `receive` property and its additional requirement). Notice also that we are shielded from one of the major annoyances of having to tweak the configuration of a framework or toolkit: knowing which packages to change levels on. This is another inherent

advantage of message-based systems (the notion that they are pretty self-explanatory, by just watching the message traffic flow between the collaborators).

Listing 7.24 Using LoggingReceive

```
class MyActor extends Actor with ActorLogging { ❶
  def receive = LoggingReceive {
    case ... => ...
  }
}
```

- ❶ add the `LoggingReceive` trait so we can see actor messages as log traces

Now when you set the property `akka.debug.receive` to on, the messages received by our actor will be logged.

Again, we have not exhausted the topic of logging, but we have shown you enough to really get going, and to ease your understandable anxiety about whether you will be expected to use some other approach, or to have to wrangle with two different loggers (yours and the Akka's). Logging is a critical tool which you could argue is even more useful in message-passing systems, where the process of just stepping along a single line of executing code in a debugger is often not possible. In the next section, we will discuss the last requirement of application delivery: deployment.

7.3 Deploying Actor-based Applications

We have already seen how we can use the `ActorSystem` and `Actors`, do the configuration and logging. But it takes more to create an application. Everything has to come together, the system should be started and a deployment has to be created. In this section we show two possible ways to create an application. The first is a simple stand-alone application. The other application will be a web-based application using `Play-mini`. These are simple examples to give you an idea how easy it is to create an application.

7.3.1 Stand-alone application

To create a stand alone application we use the `MicroKernel` of Akka combined with the `akka-plugin` to create a distribution. We start with the `HelloWorld Actor`. This actor is a simple actor that receives a message and replies with a hello message.

Listing 7.25 HelloWorld Actor

```
class HelloWorld extends Actor
  with ActorLogging { ❶

  def receive = {
    case msg:String =>
      val hello = "Hello %s".format(msg)
      sender ! hello
      log.info("Sent response {}",hello)
  }
}
```

- ❶ Using the ActorLogging trait to be able to log messages

Next we need an Actor that calls the HelloWorld actor. Lets call this the HelloWorldCaller

Listing 7.26 HelloWorldCaller

```
class HelloWorldCaller(timer:Duration, actor:ActorRef)
  extends Actor with ActorLogging {

  case class TimerTick(msg:String)

  override def preStart() {
    super.preStart()
    context.system.scheduler.schedule( ❶
      timer, ❷
      timer, ❸
      self, ❹
      new TimerTick("everybody")) ❺
  }

  def receive = {
    case msg: String => log.info("received {}",msg)
    case tick: TimerTick => actor ! tick.msg
  }
}
```

- ❶ Using the Akka scheduler to send messages to yourself
 ❷ The duration before the schedule is triggered for the first time
 ❸

- The duration between the scheduled triggers
- ④ The ActorRef where the messages are to be sent
- ⑤ The message which is sent

This Actor is using the built-in scheduler to generate messages regularly. The scheduler is used to send repeatedly the created TimerTick to us. And every time we receives this TimerTick, we send a message to the actor reference given in the constructor. This will be the HelloWorld Actor in our application. When it receives a String as a message it is just logged. This String will be the reply of the HelloWorld in our application.

To create our application we need to build the actor system at startup. We are using the Akka kernel which contains a Bootable interface and we can implement this interface to create our system. This implementation will be called when starting.

Listing 7.27 BootHello

```
import akka.actor.{ Props, ActorSystem }
import akka.kernel.Bootable
import scala.concurrent.duration._

class BootHello extends Bootable {
  ①
  val system = ActorSystem("hellokernel")
  ②

  def startup = {
    ③
    val actor = system.actorOf(
      Props[HelloWorld])
    ④
    val config = system.settings.config
    ⑤
    val timer = config.getInt("helloWorld.timer")
    system.actorOf(Props(
      new HelloWorldCaller(
        ⑥
        timer millis,
        ⑦
        actor)))
    ⑧
  }

  def shutdown = {
    ⑨
    system.shutdown()
  }
}
```

① Extends the Bootable trait to be able to be called when starting the application

②

- ☐ Creating our ActorSystem
- ③ The method called when the system starts
- ④ Create our HelloWorld Actor
- ⑤ Get the timer duration from our configuration
- ⑥ Create the Caller Actor
- ⑦ Create a Duration from an Integer. This works because we have imported akka.util.duration._
- ⑧ Passes the reference of the HelloWorld Actor to our caller
- ⑨ The method called when the system stops

So now we have built our system and need some resources to make our application work properly. We use configuration to define the default value for our timer.

Listing 7.28 reference.conf

```
helloWorld {
    timer=5000
}
```

Our default is 5000 milliseconds. Be sure that this reference.conf is placed inside our jar file. Next we have to setup the logger event handler and this is done in the application.conf

Listing 7.29 application.conf

```
akka {
    event-handlers =
    [ "akka.event.slf4j.Slf4jEventHandler" ]

    # Options: ERROR,
    WARNING,
    INFO, DEBUG
    loglevel = "DEBUG"
}
```

At this point we have all our code and resources and need to create a distribution. In this example we are using the akka-sbt-plugin to create the complete distribution. The first step is to create a plugins.sbt file

Listing 7.30 project/plugins.sbt

```
resolvers += "Typesafe Repository"
            at "http://repo.akka.io/releases/"

            addSbtPlugin("com.typesafe.akka"
                % "akka-sbt-plugin" % "2.0.1")
```

Just for the readers who are not familiar with SBT. The first line is to add the location of a repository where the plugin can be found. Then there is a white line; this is necessary because it indicates that the previous line is ended. The next line defines the plugin we want to use.

The last part we need before we are done is the SBT build file for our project.

Listing 7.31 project/HelloKernelBuild.scala

```

import sbt._
import Keys._
import akka.sbt.AkkaKernelPlugin
import akka.sbt.AkkaKernelPlugin.{ Dist,
  outputDirectory,
  distJvmOptions
}
object HelloKernelBuild extends Build {
  lazy val HelloKernel = Project(
    id = "hello-kernel-book",
    base = file("."),
    settings = defaultSettings
      ++ AkkaKernelPlugin.distSettings ④
      ++ Seq(
        libraryDependencies ++= Dependencies.helloKernel, ②
        distJvmOptions in Dist := "-Xms256M -Xmx1024M",
        outputDirectory in Dist := file("target/helloDist") ①
      )
  )

  lazy val buildSettings = Defaults.defaultSettings
    ++ Seq(
      organization := "com.manning",
      version := "0.1-SNAPSHOT",
      scalaVersion := "2.9.1",
      crossPaths := false,
      organizationName := "Mannings",
      organizationHomepage :=
        Some(url("http://www.mannings.com"))
    )

  lazy val defaultSettings = buildSettings ++ Seq(
    resolvers += "Typesafe Repo" at
      "http://repo.typesafe.com/typesafe/releases/",
    // compile options
    scalacOptions ++= Seq("-encoding", "UTF-8",
      "-deprecation",
      "-unchecked"),
    javacOptions ++= Seq("-Xlint:unchecked",
      "-Xlint:deprecation")
  )
}
// Dependencies
object Dependencies {
  import Dependency._
  val helloKernel = Seq(akkaActor,
    akkaKernel,
    akkaSlf4j,
    slf4jApi,
    slf4jLog4j, ③
    Test.junit,

```

```

        Test.scalatest,
        Test.akkaTestKit)
    }
    object Dependency {
      // Versions
      object V {
        val Scalatest = "1.6.1"
        val Slf4j = "1.6.4"
        val Akka = "2.0"
      }

      // Compile
      val commonsCodec = "commons-codec" %
        "commons-codec" % "1.4"
      val commonsIo = "commons-io"
        % "commons-io" % "2.0.1"
      val commonsNet = "commons-net"
        % "commons-net" % "3.1"
      val slf4jApi = "org.slf4j"
        % "slf4j-api" % V.Slf4j
      val slf4jLog4j = "org.slf4j"
        % "slf4j-log4j12" % V.Slf4j
      val akkaActor = "com.typesafe.akka"
        % "akka-actor" % V.Akka
      val akkaKernel = "com.typesafe.akka"
        % "akka-kernel" % V.Akka
      val akkaSlf4j = "com.typesafe.akka"
        % "akka-slf4j" % V.Akka
      val scalatest = "org.scalatest" %% "scalatest"
        % V.Scalatest
      object Test {
        val junit = "junit" % "junit" %
          "4.5" % "test"
        val scalatest = "org.scalatest" %% "scalatest" %
          V.Scalatest % "test"
        val akkaTestKit = "com.typesafe.akka"
          % "akka-testkit" % V.Akka % "test"
      }
    }
  }
}

```

- ❶ Define where the created distribution should be put
- ❷ Define the application dependencies
- ❸ Use log4j as logging framework
- ❹ Here we add the Akka plugin functionality to our project

SIDEBAR Simple Build Tool: More Information

At some point, you will no doubt want more details on SBT and what all it can do. You can read the documentation. The project is hosted on GitHub (<https://github.com/sbt/sbt>). Included with the documentation is a demo from ScalaDays that is quite extensive. Manning also has a book that has been recently released, *SBT in Action* that goes into great detail, working through not only what you can do, but what makes SBT a next generation build tool.

Now we have defined our project in SBT and are ready to create our distribution. Listing 7.32 shows how we can start SBT and run the dist command.

Listing 7.32 Create distribution

```
sbt
[info] Loading project
definition from J:\boek\manningAkka\
listings\kernel\project
[info]
Set current project to hello-kernel-book (in build
file:/J:/boek/manningAkka/listings/kernel/)
> dist ①
```

- ① Once sbt is done loading, type dist and press return

After this, SBT has created a distribution in the directory `/target/helloDist`. This directory contains 4 subdirectories

- **bin**
This contains the start script. One for windows and one for Unix
- **config**
This directory contains the configuration files needed to run our application.
- **deploy**
This directory is where our jar file placed
- **lib**
This directory contains all the jar files our application depends upon.

Now we have a distribution and all that is left now, is to run our application. The scripts need our boot class as an argument

Listing 7.33 run application

```
start.bat ch04.BootHello

        ./start
        ch04.BootHello
```

And when we look in the log file we see that every 5 seconds the helloWorld actor is receiving messages, and the caller receives its messages. Of course, this application has no real utility. In the next section, we will build an app that can communicate with others through the web.

7.3.2 Akka with a web application

The remarkable part of what we'll do in this section is that it's really no more work than the prior section, yet we will have an app that can expose a services interface, opening up great possibilities. Play-mini is an extension created on top of Play! (a web framework also created by typesafe). It's a match because we are interested in making an application that has no user interface, but rather just a way to expose a service via the web. There are a number of options for deploying Akka in a webapp, we are showing play-mini because it is very simple and lightweight.

Listing 7.34 extend application

```
object PlayMiniHello extends Application {
    ...
}
```

Just by extending Application, we bring a lot of functionality in here. Just as we did in the kernel example, we start by creating our actor system. Listing 7.35 shows the creation of the ActorSystem.

Listing 7.35 create application system

```
object PlayMiniHello extends Application {
    lazy val system = ActorSystem("webhello")
    lazy val actor = system.actorOf(Props[HelloWorld])
}
```

So far, our web app looks the same as the kernel one. First thing we're going to need in the web app that the kernel one did not have is to create routes; these are the URLs our application supports. Our example supports two URLs, both are GET actions. The first is /test. This is an example of a response we can directly create. Ok is one of the results we can return; other examples are NotFound, BadRequest, ServiceUnavailable, etc. Listing 7.36 shows the route creation.

Listing 7.36 create simple route

```
object PlayMiniHello extends Application {
  def route = {
    case GET(Path("/test")) => Action { ❶
      Ok("TEST @ %s\n".format(System.currentTimeMillis))
    }
  }
}
```

- ❶ This is all we have to do to map our REST path to an action
- ❷ Single line sends back our HTTP response code and a message (we include the time)

In newer, service-oriented architectures, whole applications can be built from a few such service mappings. Remember, this is all we've done so far beyond our simple kernel example; the required plumbing and message handling came in with Akka's Application class (and the ActorSystem). The second route is our hello request. To service this request we need a parameter name from the request and when it isn't supplied, we will use the name defined in our configuration. Listing 7.37 shows how we get a parameter while mapping a REST path.

Listing 7.37 define form

```
val writeForm = Form("name" -> text(1,10))
```

To get the name parameter from the request, we use the play Form class. We define the form with our name parameter, which should be a string and the length of the string should be between 1 and 10. To get the value of the name parameter we have to bind the form with the request. When this fails we use the configuration parameter `helloWorld.name`.

Listing 7.38 get request parameter

```

case GET(Path("/hello")) => Action {
  implicit request => ❶
  val name = try {
    writeForm.bindFromRequest.get ❷
  } catch {
    case ex:Exception => {
      log.warning("no name specified")
      system.settings.config.getString("helloWorld.name") ❸
    }
  }
  ...
}

```

- ❶ define the request as implicit to allow binding later
- ❷ Bind our form to the implicit request and get the result
- ❸ Use our configuration to get the default name

Now we have our name, which we can send to our actor. But we have to wait for our result before we can create a response. And we don't want to block our thread. Instead of returning our response directly, we create an `AsyncResult`. This `AsyncResult` needs a promise. A promise is closely linked to a `Future`, but the `Future` is used at the consumer site. With the `Promise`, the result is waited for at the producer site; it supplies the result when it is done. More details about `Futures` and `Promises` are covered in a later chapter.

Listing 7.39 AsyncResult

```

AsyncResult {
  val resultFuture = actor ? name ❶
  val promise = resultFuture.asPromise ❷
  promise.map { ❸
    case res:String => {
      Ok(res)
    }
  }
}

```

- ❶ Send our request using the ask method
- ❷ Create a `Promise` using the future
- ❸

- Create a response when the result of the HelloWorld Actor is received.

To create a response using the response of our HelloWorld actor, we need to use the ask method. We have seen that the ask method returns a Future. We are going to use this Future to create a Promise based on the received Future. The last step is to fill the Promise with our result. This is executed when the future has a result. When the response of the HelloWorld actor is received, the code within map is executed. This way we don't have to wait for the result in our thread. All the other code is executed directly.

This works fine when we get a response in time, but what happens when the response isn't received in time and we get an AskTimeoutException. Then the map part isn't called and our Promise will not contain a result. To solve this problem, we extend the Future to also create a String when it fails. To do this we use the recover method of the Future.

Listing 7.40 recover Future

```
val
    resultFuture = actor ? name recover {
    case ex:AskTimeoutException
    => "Timeout"
    ①
    case ex:Exception => {
    log.error("recover from "+ex.getMessage)
    "Exception:" + ex.getMessage
    }
    }
```

- ① Translate the AskTimeoutException into the string "Timeout"

With the recover we create a new Future which will return the message of the initial Future when successful and otherwise the following code will be executed. So when we get a response from the HelloWorld Actor, the new Future will return the same message. But when the initial Future fails because there is a AskTimeoutException, the exception is replaced by the String "Timeout". This way the future will always return a String even when it fails. When we use the new Future to create a Promise, the map code block will be called even when there is an exception during the ask.

When we put all the code parts together we get the following class. We have seen all the parts in the prior sections, so there is nothing new here. What is notable

is how little code is required overall to not just spit back a message, but to handle:

- Defaulting through a property (from a resource file)
- Retrieving a parameter
- Gracefully handling timeout
- Doing it all concurrently

Use this comprehensive (albeit simple) example (in listing 7.41) as a map to what has gone before in this chapter.

Listing 7.41 PayMiniHelloHello

```

object PlayMiniHello extends Application {
  lazy val system = ActorSystem("webhello")
  lazy val actor = system.actorOf(Props[HelloWorld])
  implicit val timeout = Timeout(1000 milliseconds)
  val log = Logging(system, PlayMiniHello.getClass)

  def route = {
    case GET(Path("/test")) => Action {
      Ok("TEST @ %sn".format(System.currentTimeMillis))
    }

    case GET(Path("/hello")) => Action {
      implicit request =>

      val name = try {
        writeForm.bindFromRequest.get
      } catch {
        case ex:Exception => {
          log.warning("no name specified")
          system.settings.config.getString("helloWorld.name")
        }
      }

      AsyncResult {
        val resultFuture = actor ? name recover {
          case ex:AskTimeoutException => "Timeout"
          case ex:Exception => {
            log.error("recover from "+ex.getMessage)
            "Exception:" + ex.getMessage
          }
        }

        val promise = resultFuture.asPromise
        promise.map {
          case res:String => {
            log.info("result "+res)
            Ok(res)
          }
          case ex:Exception => {
            log.error("Exception "+ex.getMessage)
            Ok(ex.getMessage)
          }
          case _ => {
            Ok("Unexpected message")
          }
        }
      }
    }
  }
}

```

```

    }
    val writeForm = Form("name" -> text(1,10))
  }

```

- ❶ Implement the Application trait
- ❷ Create our system and actors
- ❸ Create a route and create a direct result
- ❹ Create a request using our helloworld actor
- ❺ In this action, we need the request and make it implicit for getting the parameter name
- ❻ Get the parameter name, for more details look in the play documentation. Using the request as implicit
- ❼ Because we do not want to block, we return an AsyncResult
- ❽ The request to our HelloWorld Actor. Because we could receive a timeout, we create a recover that returns a string when the ask fails
- ❾ Return the string Timeout when the ask fails with a timeout
- ❿ Create a string when another exception has happened
- ⓫ Create a promise from the ask Future
- ⓬ This is the actual creation of our result. This is executed when the response of the actor is received
- ⓭ Definition of our name parameter. It should be a text with a minimal size of 1 to a maximum of 10 characters, look in the Play! documentation for more details

There are still a few minor things we need to do outside the app. The next one is to define which class should be used to start the application. In our case "PlayMiniHello." To do this, we have to create the class Global.

Listing 7.42 Global

```

object Global extends com.typesafe.play.mini.Setup(
  ch04.PlayMiniHello)

```

This class extends indirectly the `play.api.GlobalSettings` trait. This allows us to use the `onStart` and `onStop` methods which create and stop the Actor system. We didn't use these methods in the example; preferring to do it in our Application class, which will be the only place we address the ActorSystem.

To be able to run our application, we need configuration files, similar to what we saw in the kernel example, starting with the `reference.conf`

Listing 7.43 reference.conf

```
helloWorld {  
  name=world  
}
```

And the application.conf (which is where the property we're using to provide a default name is coming from).

Listing 7.44 application.conf

```
helloWorld {  
  name="world!!!"  
}  
  
akka {  
  event-handlers = ["akka.event.slf4j.Slf4jEventHandler"]  
  
  # Options: ERROR, WARNING, INFO, DEBUG  
  loglevel = "DEBUG"  
}
```

Now that we have all our code and resource files, we can make the SBT build script

Listing 7.45 Build.scala

```

import sbt._
import Keys._
import PlayProject._

object Build extends Build {
  lazy val root = Project(id = "playminiHello",
    base = file("."), settings = Project.defaultSettings).settings(
    resolvers += "Typesafe Repo" at
      "http://repo.typesafe.com/typesafe/releases/",
    resolvers += "Typesafe Snapshot Repo" at
      "http://repo.typesafe.com/typesafe/snapshots/",
    libraryDependencies ++= Dependencies.hello,
    mainClass in (Compile, run) := 1
      Some("play.core.server.NettyServer"))
}

object Dependencies {
  import Dependency._
  val hello = Seq(akkaActor,
    akkaSlf4j,
    // slf4jLog4j,
    playmini
  )
}

object Dependency {

  // Versions
  object V {
    val Slf4j      = "1.6.4"
    val Akka      = "2.0"
  }

  // Compile
  val slf4jLog4j  = "org.slf4j"           % "slf4j-log4j12" % V.Slf4j
  val akkaActor  = "com.typesafe.akka" % "akka-actor"      % V.Akka
  val playmini   = "com.typesafe" %% "play-mini" % "2.0-RC3"
  val akkaSlf4j  = "com.typesafe.akka" % "akka-slf4j"      % V.Akka
}

```

1 Line added to support testing within SBT

SBT supports testing of the webapplication within SBT. By adding the line

Listing 7.46 SBT run

```
mainClass in (Compile, run) :=  
    Some("play.core.server.NettyServer")
```

It is possible to start the application with the command run within SBT.

Listing 7.47 SBT run

```
sbt > run
```

At this moment the application is running and listening on port 9000. To test this you can do a request using a browser or using curl or wget.

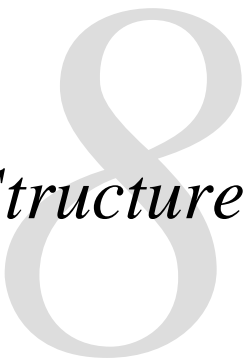
7.4 Summary

Like so much in development, Deployment looks like it's going to be a piece of cake as you approach. Yet in practice, it often turns into a vortex of each component configuring itself from its own resources, with no rhyme or reason behind the overall system approach. As is the case in all things design-wise, Akka's approach is to provide state of the art tools, but with the emphasis on simple conventions that are easy to implement. These made making our first app ready to run rather easy. But more importantly, we have seen that we can carry this simplicity forward into much more complex realms.

- File conventions for simple overriding of configuration
- Intelligent defaulting means apps can supply most of what's needed
- Yet granular control over injecting config
- State of the art logging through an adapter and single dependency point
- Lightweight application bundling, including for the web
- Using a build tool that also bundles and runs your app

The age of the release engineer as the hardest working member of the team may be ending. As we go forward with more complex examples in the book, you will see that we will not see the deployment layer blow up on us. This is a huge part of the Akka story: it is delivering not only a power-packed runtime with messaging and concurrency built in, but the means to get solutions running in it more rapidly than in the less powerful application environments most of us are accustomed to.

System Structure



In this chapter

- Pipe and Filter Pattern
- Scatter-Gather Pattern
- Routing
- Recipient list
- Aggregator
- Become/unbecome

One of the immediate implications of Actor based programming is how do we model code that requires collaborators to work together if each unit of work is done in parallel? Collaboration implies some notion of process, which, though there can be parallel processes, there will also be cases where it's essential that certain steps happen after a required prior step has been completed. By implementing a few of the classic Enterprise Integration Patterns, we'll take the next step in showing how Akka allows us to use these design approaches while still making use of its inherent concurrency.

- integration tools and platforms
- messaging systems
- WSO2, and SOA and Web-service based solutions

We are going to focus primarily on the most relevant Enterprise Integration patterns to show different ways of connecting actors to solve problems. The architectural Enterprise Integration Patterns will get the most attention in this chapter, as we are considering system structure.

We start with the simple Pipes and Filters pattern. This is the default pattern for

most message passing systems and is very straightforward, but, of course, the classical version is sequential. (We will adapt it to work in our concurrent, message-based architecture.) Next will be the Scatter-Gather Pattern, which does provide a means of parallelizing tasks. Actor implementations of these patterns are not only remarkably compact and efficient, but they are free of a lot of the implementation details that quickly seep into patterns that have to deal with messaging (as most of these do).

8.1 Pipes and Filters

The concept of piping refers to the ability for one process or thread to pump its results to another processor for additional processing. Most people know it from some exposure to Unix, where it originated. The set of piped components is often referred to as a 'pipeline,' and most people's experience of it is of each step occurring in sequence with no parallelism. Yet, we will quickly see that there are often good reasons to want to see non-dependent aspects of a process occur in parallel. That's what we will show here. First, a description of this pattern's applicability, and its form, then a look at how we can implement it using Akka.

8.1.1 Enterprise integration pattern Pipes and Filters

In many systems a single event will trigger a sequence of tasks. For example our camera in chapter 3. It receives the photo and before the event is sent to central processing, a number of checks are done. When no license plate is found in the photo, the system is unable to process the message any further and therefore, it will be discarded. In this example we also discard the message when the speed is below the maximum speed. Which means that only messages that contain the license plate of a speeding vehicle end up getting to the central processor. You can probably already see how we will apply the Pipes and Filters Pattern here: the constraints are filters, and the interconnects are the pipes in this case.

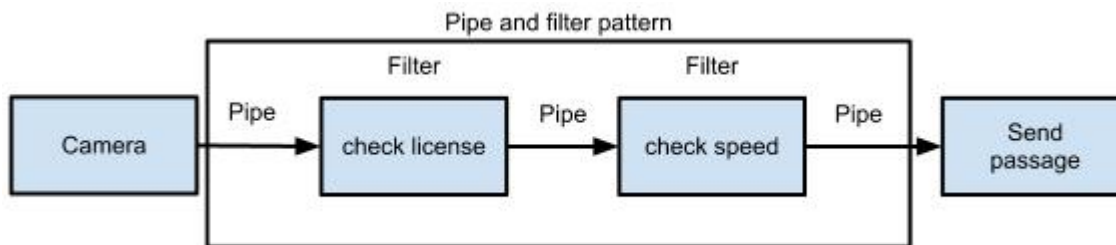


Figure 8.1 Example of Pipes and Filters

Each Filter consists of three parts, the inbound pipe where the message is

received, the processor of the message, and finally the outbound pipe where the result of the processing is published.

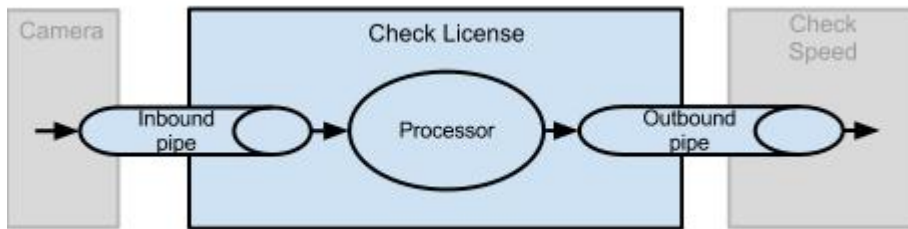


Figure 8.2 Three parts of a Filter

In Figure 8.2 these parts are shown. The two pipes are drawn partly outside the Filter because the outbound pipe of the check license filter is also the inbound pipe of the check speed filter. An important restriction is that each filter must accept and send the same messages, because the outbound pipe of a filter can be the inbound pipe of any other filter in the pattern. This means that all the filters need to have the same interface. This includes the inbound and the outbound pipe. This way it is very easy to add new processes, change the order of processes or remove them. Because the filters have the same interface and are independent, nothing has to be changed, other than potentially adding additional pipes.

8.1.2 Pipes and filters in Akka

The filters are the processing unit of the messages system, so when we apply the pattern to Akka we use actors to implement our filters. Thanks to the fact that the messaging is supplied behind the scenes, we can just connect a number of Actors and the pipes are already there. So it would seem to be quite simple to implement this pattern with Akka. Are we done here? Not quite. There is a small requirement which is crucial for implementing the Pipes and Filter pattern and that is that the interface is the same for all the filters and that these steps are independent. This means that all the messages received by the different Actors should be the same, because the message is part of the interface of the filter as shown in Figure 8.3. If we were to use different messages, the interface of the next actor would differ and our uniformity requirement would be violated, preventing us from being able to indiscriminately apply filters.

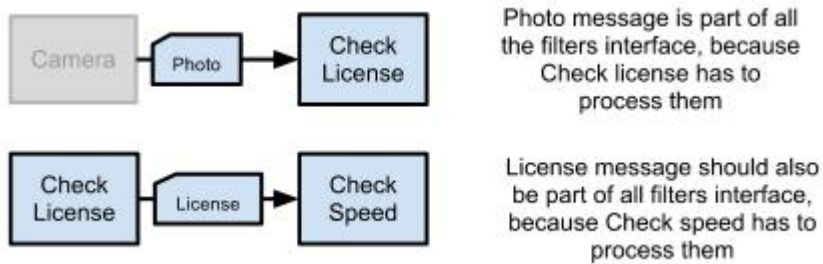


Figure 8.3 Messages send by different actors

Given the requirement that the input and output to the pipe need be the same, both actors must accept and send the same messages.

Let us create a small example with a Photo message and two filters the License and speed filter.

Listing 8.1 A Pipe with Two Filters Example

```
case class Photo(license: String, speed: Int)
```

```
class SpeedFilter(minSpeed: Int, pipe: ActorRef) extends Actor {
  def receive = {
    case msg: Photo =>
      if (msg.speed > minSpeed)
        pipe ! msg
  }
}
```

```
class LicenseFilter(pipe: ActorRef) extends Actor {
  def receive = {
    case msg: Photo =>
      if (!msg.license.isEmpty)
        pipe ! msg
  }
}
```

1 The message which will be filtered

2 Filter all Photos which have a speed lower than the minimal speed

3 Filter all Photos which have an empty license

There is nothing special about these Actor filters. We used actors with one way messages in section 2.1.2 and other examples. But because the two actors process and send the same message type, we can construct a pipeline from them, which allows for either one to feed the other its results, meaning the order in which we apply the filters does not matter. In the next example, we'll show how this gives us flexibility that comes in handy when we find that the order will have a marked influence on the execution time. Let's see how this works.

Listing 8.2 Pipe and filter test

```

val endProbe = TestProbe()
val speedFilterRef = system.actorOf(
  Props(new SpeedFilter(50, endProbe.ref)))
val licenseFilterRef = system.actorOf(
  Props(new LicenseFilter(speedFilterRef)))
val msg = new Photo("123xyz", 60)
licenseFilterRef ! msg
endProbe.expectMsg(msg)
licenseFilterRef ! new Photo("", 60)
endProbe.expectNoMsg(1 second)
licenseFilterRef ! new Photo("123xyz", 49)
endProbe.expectNoMsg(1 second)

```

- 1 Construct the pipeline
- 2 Test a message which should be passed through
- 3 Test a message without a license
- 4 Test a message with a low speed

The license filter uses a lot of resources. It has to locate the letters and numbers on the plate, which is CPU-intensive. When we put the camera on a busy road, we find that the recognize filter can't keep up with pace of new photos arriving. Our investigations reveal that 90% of the messages are approved by the License Check and 50% of the messages are approved by the speed filter.

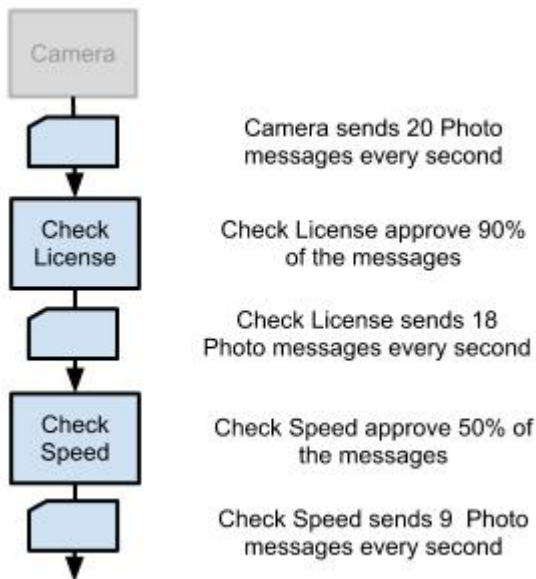


Figure 8.4 Number of processed messages for each filter for initial configuration

In this example shown in Figure 8.4 the Check license has to process 20 message each second. To improve performance, it would be better to reorder the filters. Since most of the messages are filtered by the speed filter, the load on the

license filter will be decreased significantly.

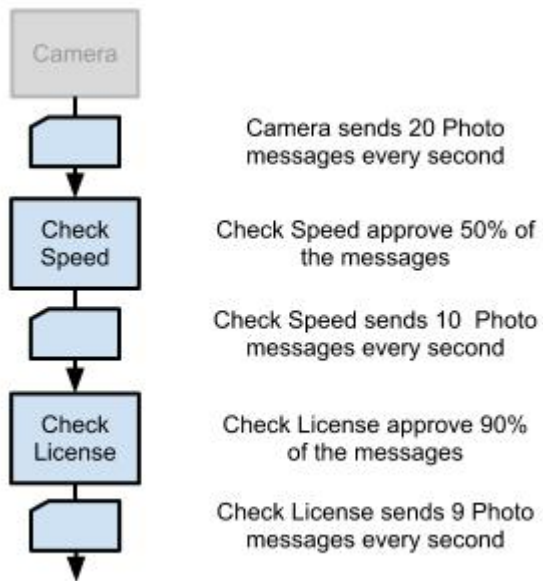


Figure 8.5 Number of processed messages for each filter for altered configuration

As we can see in figure 8.5, when we switch the order of filters, check license is asked to evaluate 10 licenses per second; reordering halved the load of the check license filter. And because the interfaces are the same and the processes are independent we can easily change the order of the actors without changing the functionality or the code. Before the pipes and filters pattern, we had to change both components to get this to work. Using this pattern, the only change is when building the chain of actors at startup time, which can be easily made configurable.

Listing 8.3 Changed order of filters

```

val endProbe = TestProbe()
val licenseFilterRef = system.actorOf(
  Props(new LicenseFilter(endProbe.ref)))
val speedFilterRef = system.actorOf(
  Props(new SpeedFilter(50, licenseFilterRef)))
val msg = new Photo("123xyz", 60)
speedFilterRef ! msg
endProbe.expectMsg(msg)
speedFilterRef ! new Photo("", 60)
endProbe.expectNoMsg(1 second)
speedFilterRef ! new Photo("123xyz", 49)
endProbe.expectNoMsg(1 second)
  
```

1 Construct the pipeline in another order

We see that it doesn't matter which order we use, the pipeline gives us the same functionality; this flexibility is the strength of this pattern. In our example we used actual filters, but this pattern can be extended; the processing pipeline is not limited to filters. As long as the process accepts and produces the same types of messages, and is independent of the other processes, this pattern applies. In the next section, we will see a pattern that enables a divide and conquer approach, which of course, requires concurrency, and Akka again makes it easy. We scatter units of work amongst a number of processing Actors and then gather their results into a single set, allowing the consumer of the work product to just make a request and get a single response.

8.2 Scatter-Gather Pattern

In the previous section, we created a pipeline of tasks which were executed sequentially. The ability to execute tasks in parallel is often preferable. We'll look at the Scatter-Gather Pattern next and will see how we can accomplish this. Akka's inherent ability to dispatch work to actors asynchronously provides most of what we need to make this pattern work. The processing tasks (filters in the previous example) are the gather parts; the Recipient List is the scatter component. We'll use the Aggregator for the gather part (provided by Akka).

8.2.1 Applicability

The pattern can be applied in two different scenarios. The first case is when the tasks are functionally the same, but only one is passed through to the gather component as the chosen result. The second scenario is when work is divided for parallel processing and each processor submits its results which are then combined into a result set by the aggregator. We will see the benefits of the pattern clearly in both of our Akka implementations in the following section.

COMPETING TASKS

Let's start with the following problem. A client buys a product, let's say a book at a web shop, but the shop doesn't have the requested book in stock, so it has to buy the book from a supplier. But the shop is doing business with three different suppliers and wants to pay the lowest price. Our system needs to check if the product is available, and at what price. This has to be done for each supplier, and only the supplier with the lowest price will be used. In figure 8.6 we show how the Scatter-Gather Pattern can help here.

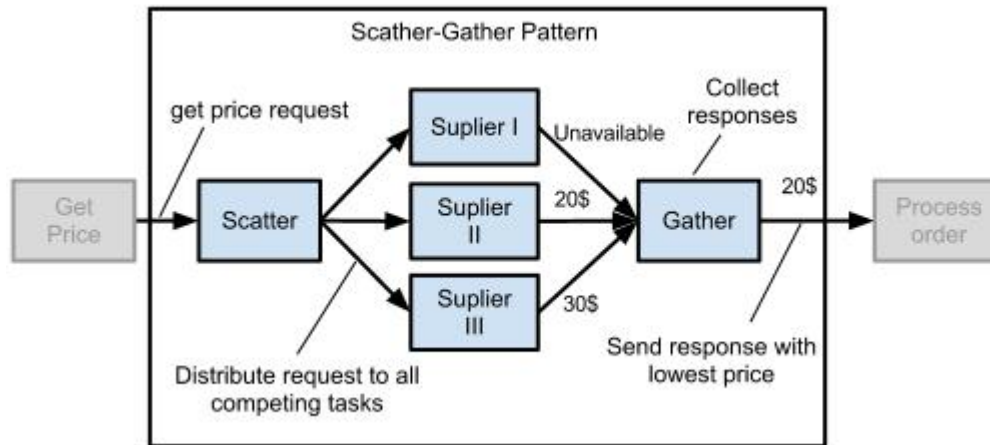


Figure 8.6 Scatter-Gather Pattern with competing tasks

The message of the client is distributed over three processes and each process checks the availability and price of the product. And the gather process will collect all the results and only pass the messages with the lowest price (in the example \$20). The processing tasks are all focused on one thing: getting the price of the product, but they may be doing it in different ways, because there are multiple suppliers. In the pattern parlance, this is the competing tasks aspect, as only the best result is going to be used. For our example, it's the lowest price, but the selection criteria could be different in other cases. Selection in the Gather component is not always based on the content of the message. It is also possible that you only need one solution, in which case, the competition is merely determining which is the quickest response. For example, the time of sorting a list depends greatly on the algorithm used and the initial unsorted list. When performance is critical, we sort the list in parallel using different sorting algorithms. If we did such a thing with Akka, we would have one Actor doing a bubble sort, one a quicksort, maybe one doing a heap sort. All tasks will result in the same sorted list, but depending on the unsorted list, one of them will be the fastest. In this case the gather will select the first received message and tell the other actors to stop. This is also an example of using the Scatter-Gather Pattern for competing tasks.

PARALLEL COOPERATIVE PROCESSING

Another case where the Scatter-Gather Pattern can be used is when the tasks are performing a sub task. Let us go back to our Camera example. While processing a Photo, different information has to be retrieved from the photo and added to the Photo messages. For example the time the photo was created and the speed of the vehicle. Both actions are independent of each other and can be performed in parallel. When both tasks are ready, the results must be joined together into a single message containing the time and the speed. Figure 8.7 shows the use of Scatter-Gather for this problem. This pattern starts with scattering a message to multiple tasks: GetTime and GetSpeed. And the results of both tasks should be combined into a single message which can be used by other tasks.

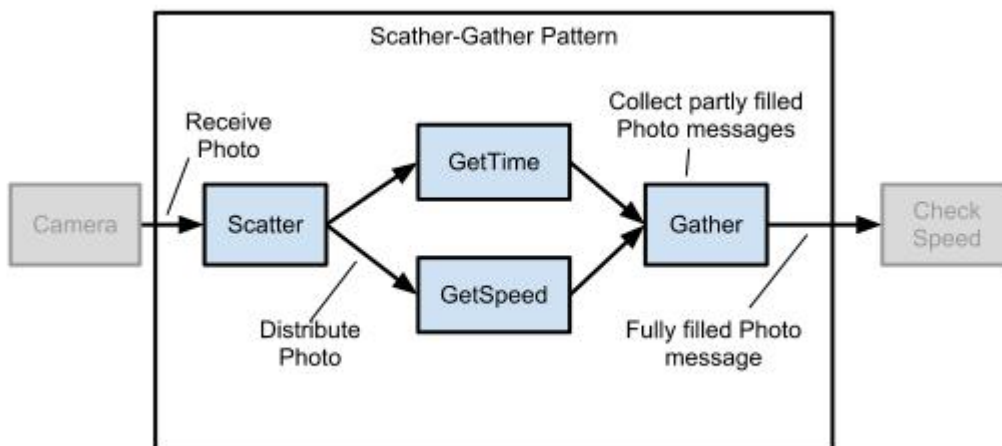


Figure 8.7 Scatter-Gather Pattern for task parallelization

8.2.2 Parallel tasks with Akka

Let's see how we can implement the Scatter-Gather Pattern in the second scenario with Akka actors. We are going to use the Photo example. Each component in this pattern is implemented by one actor. In this example we use one type of message, which is used for all tasks. And each task can add the data to the same type message when processing has completed. The requirement that all tasks should be independent can't always be met. This only means that the order of both tasks can't be switched. But all the other benefits of adding, removing or moving the tasks apply.

We start by defining the message that will be used. This message is received and sent by all components in this example.

```
case class PhotoMessage(id: String,
  photo: String,
  creationTime: Option[Date] = None,
  speed: Option[Int] = None)
```

For our example message, we mock the traffic cameras and image recognition tools by just providing the image. Note, the message has an ID, which can be used by the Aggregator to associate the messages with their respective flows. The other attributes are the creation time and speed; they start empty and are provided by the GetSpeed and GetTime tasks. The next step is to implement the two processing tasks GetTime and GetSpeed.

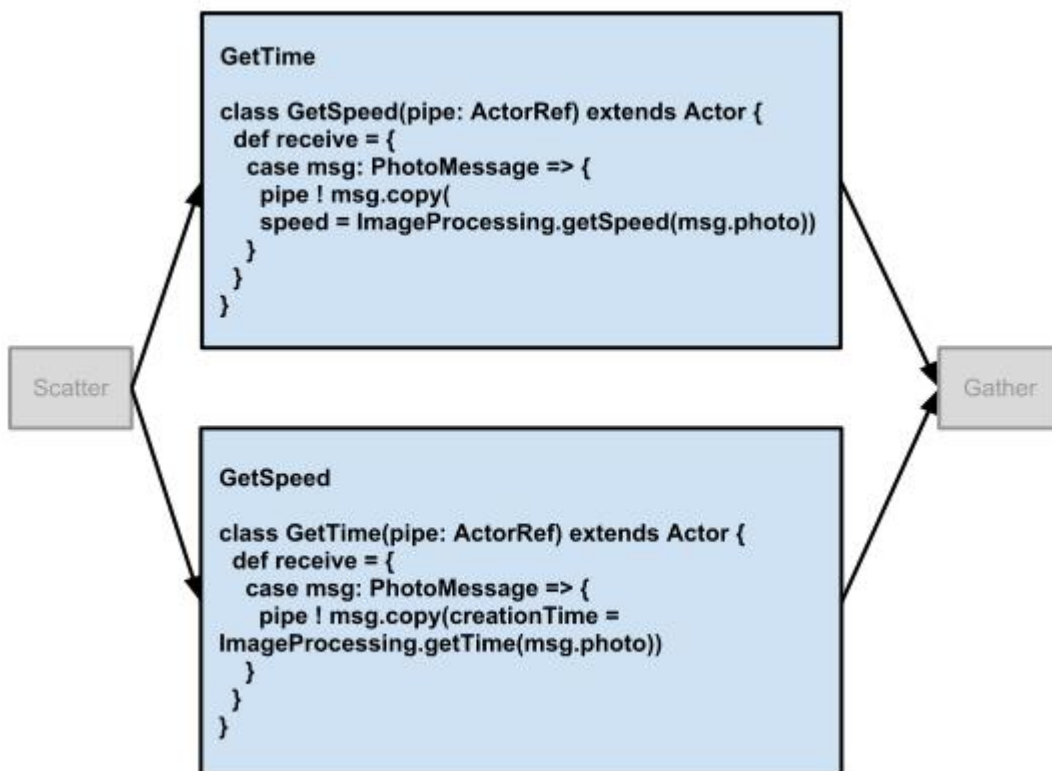


Figure 8.8 Listing of the two processing tasks GetTime and GetSpeed

As is shown in Figure 8.8, the two actors have the same structure, the difference being which attribute is extracted from the image. So these actors are doing the actual work, but we need an actor that implements the scatter functionality that will dispatch the images for processing. In the next section, we will use the recipient list to scatter the tasks, then the results are combined with the Aggregator Pattern.

8.2.3 Implement the scatter component using the Recipient list

When a PhotoMessage enters the Scatter-Gather Pattern, the scatter component has to send the message to the processors (the GetTime and GetSpeed actors from the prior section). We use the simplest implementation of the scatter component and that is the Recipient list. (The scattering of messages can be implemented in a number of ways; any approach that creates multiple messages from one message and distributes it, will do.)

The Recipient list is a simple pattern because it is one component; its function is to send the received message to multiple other components. Figure 8.9 shows that the received messages are sent to the GetTime and GetSpeed Tasks.

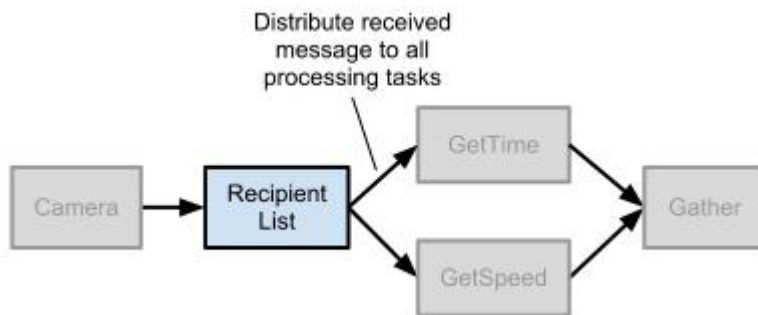


Figure 8.9 Recipient list pattern

Given that we have to perform the same two extractions on every message, the RecipientList is static and the message is always sent to the GetTime and GetSpeed tasks. Other implementations might call for a dynamic recipient list where the receivers are determined based on the message content, or the state of the list.



Figure 8.10 Listing of the Recipient list

In Figure 8.10 the simplest implementation of a recipient list is shown; when a message is received, it is sent to members. Let's put our RecipientList to work. We start by creating it with Akka testProbes (we first saw these in Chapter 3).

Listing 8.4 Recipient list test

```

val endProbe1 = TestProbe()
val endProbe2 = TestProbe()
val endProbe3 = TestProbe()
val list = Seq(endProbe1.ref, endProbe2.ref, endProbe3.ref)
val actorRef = system.actorOf(
  Props(new RecipientList(list)))
val msg = "message"
actorRef ! msg
endProbe1.expectMsg(msg)
endProbe2.expectMsg(msg)
endProbe3.expectMsg(msg)

```

① Create the recipient list

② Send the message
 ③ All the recipients
 ③ have to receive the
 ③ message

And when we send a message to the `RecipientList` actor, the message is received by all probes.

This pattern isn't mind-blowing, but used in the Scatter-Gather Pattern, it is quite useful.

8.2.4 Implementing the gather component with the Aggregator pattern

The recipient list is scattering one message into two message flows to the `GetSpeed` and `GetLicense`. Both flows are doing a part of the total processing. So when the time and speed have both been retrieved, the messages need to be joined into a single result. This is done in the gather component. Figure 8.11 shows the Aggregator pattern which is used as the gather component. Just as the `RecipientList` is used as a scatter component.

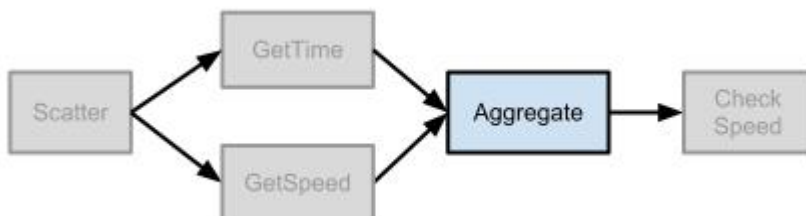


Figure 8.11 Example of Aggregator pattern as gather component

The Aggregator pattern is used to combine multiple messages into one. This can be a selection process when the processing tasks are competing with each other, or merely combining several messages into one as we are doing here. One of the characteristics of the Aggregator is that the messages have to be stored somehow and when all messages have been received, the Aggregator can process

them. To keep it simple we are going to implement an Aggregator that combines two PhotoMessages into one.

Listing 8.5

```
class Aggregator(timeout:Duration, pipe:ActorRef) extends Actor {
  val messages = new ListBuffer[PhotoMessage] ❶
  def receive = {
    case rcvMsg: PhotoMessage => {
      messages.find(_.id == rcvMsg.id) match {
        case Some(alreadyRcvMsg) => { ❷
          val newCombinedMsg = new PhotoMessage(
            rcvMsg.id,
            rcvMsg.photo,
            rcvMsg.creationTime.getOrElse(alreadyRcvMsg.creationTime),
            rcvMsg.speed.getOrElse(alreadyRcvMsg.speed) )
          pipe ! newCombinedMsg
          //cleanup message
          messages -= alreadyRcvMsg ❸
        }
        case None => messages += rcvMsg ❹
      }
    }
  }
}
```

- ❶ The buffer to store the messages which can't be processed yet
- ❷ This is the second (of two) messages so we can start combining them
- ❸ Remove the processed message from the list
- ❹ Received the first message, so store it for processing later

The first thing when receiving a message, is to check if it is the first message or the second. When it is the second we can process the messages. Processing in this Aggregator is to combine the messages into one and send the result to the next process. When it is the first message the message is stored in the messages buffer.

Listing 8.6 Aggregator test

```

val endProbe = TestProbe()
val actorRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref)))
val photoStr = ImageProcessing.createPhotoString(new Date(), 60)
val msg1 = PhotoMessage("id1",
  photoStr,
  Some(new Date()),
  None)
actorRef ! msg1
val msg2 = PhotoMessage("id1",
  photoStr,
  None,
  Some(60))
actorRef ! msg2
val combinedMsg = PhotoMessage("id1",
  photoStr,
  msg1.creationTime,
  msg2.speed)
endProbe.expectMsg(combinedMsg)

```

1 Send the first message

2 Send the second

3 Expect the combined message

The Aggregator works as expected. Two messages are sent to it, whenever they are ready, and one combined message is then created and sent on. But because we have state in our actor we need to assure that the state is always consistent. So what happens when one task fails? When this happens the first message is stored forever in the buffer and no one would ever know what happened to this message. As occurrences pile up, our buffer size increases and eventually it might consume too much memory, which can cause a catastrophic fault. There are many way to solve this. In this example we are going to use a timeout. We expect that both processing tasks need about the same amount of time to execute, therefore both messages should be received about the same time. This time can differ because of the availability of resources needed to process the message. When the second message isn't received within the stipulated timeout, it is presumed lost. The next decision we have to make is how the Aggregator should react to the loss of a message. In our example, the loss of a message is not catastrophic so we want to continue with a message which is not complete. So, in our implementation, the Aggregator will always send a message even when one of them was not received.

To implement the timeout we will use the scheduler. Upon receipt of the first message, we schedule a `TimeoutMessage` (providing self as the recipient). The message is only still in the buffer when the second message was not received. This

can be used to detect what action should be taken.

Listing 8.7 Implementing the Timeout

```

case class TimeoutMessage(msg:PhotoMessage)

def receive = {
  case rcvMsg: PhotoMessage => {
    messages.find(_.id == rcvMsg.id) match {
      case Some(alreadyRcvMsg) => {
        val newCombinedMsg = new PhotoMessage(
          rcvMsg.id,
          rcvMsg.photo,
          rcvMsg.creationTime.getOrElse(alreadyRcvMsg.creationTime),
          rcvMsg.speed.getOrElse(alreadyRcvMsg.speed) )
        pipe ! newCombinedMsg
        //cleanup message
        messages -= alreadyRcvMsg
      }
      case None => {
        messages += rcvMsg

        context.system.scheduler.scheduleOnce( ❶
          timeout,
          self,
          new TimeoutMessage(rcvMsg))
      }
    }
  }
}

case TimeoutMessage(rcvMsg) => { ❷
  messages.find(_.id == rcvMsg.id) match {
    case Some(alreadyRcvMsg) => {
      pipe ! alreadyRcvMsg ❸
      messages -= alreadyRcvMsg
    }

    case None => //message is already processed ❹
  }
}
}

```

- ❶ Schedule the timeout
- ❷ Timeout has expired
- ❸ Send the first message when the second isn't received
- ❹ Both messages are already processed, so do nothing

We have implemented the timeout, now let's see if it is received when the Aggregator fails to receive two message in the allowable time.

```
val endProbe = TestProbe()
```

```

val actorRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref)))
val photoStr = ImageProcessing.createPhotoString(
  new Date(), 60)
val msg1 = PhotoMessage("id1",
  photoStr,
  Some(new Date()),
  None)
actorRef ! msg1
endProbe.expectMsg(msg1)

```

1 Create the message

2 Send only one message
3 the timeout and receive the message

As you can see, when we send only one message the timeout is triggered, we detect a missing message and send the first message as the combined message.

But this isn't the only problem that can occur. In chapter 3 we have seen that we have to be careful when using state. When the Aggregator fails somehow we are losing all the messages which are already received, because the Aggregator is restarted. So how can we solve this problem? Before the actor is restarted the `preRestart` method is called. This method can be used to preserve our state. For this Aggregator we can use the simplest solution: have it resend the messages to the itself before restarting. Because we don't depend on the order of the received messages, this should be fine even when failures occur. By resending the messages from our buffer, the messages are stored again when the new instance of our actor is started. The complete Aggregator becomes:

Listing 8.8 Aggregator

```

class Aggregator(timeout: FiniteDuration, pipe: ActorRef)
  extends Actor {

  val messages = new ListBuffer[PhotoMessage]
  implicit val ec = context.system.dispatcher
  override def preRestart(reason: Throwable, message: Option[Any]) {
    super.preRestart(reason, message)
    messages.foreach(self ! _)
    messages.clear()
  }

  def receive = {
    case rcvMsg: PhotoMessage => {
      messages.find(_.id == rcvMsg.id) match {
        case Some(alreadyRcvMsg) => {
          val newCombinedMsg = new PhotoMessage(
            rcvMsg.id,
            rcvMsg.photo,
            rcvMsg.creationTime.getOrElse(alreadyRcvMsg.creationTime),
            rcvMsg.speed.getOrElse(alreadyRcvMsg.speed))
          pipe ! newCombinedMsg
          //cleanup message
          messages -= alreadyRcvMsg
        }
        case None => {
          messages += rcvMsg
          context.system.scheduler.scheduleOnce(
            timeout,
            self,
            new TimeoutMessage(rcvMsg))
        }
      }
    }
    case TimeoutMessage(rcvMsg) => {
      messages.find(_.id == rcvMsg.id) match {
        case Some(alreadyRcvMsg) => {
          pipe ! alreadyRcvMsg
          messages -= alreadyRcvMsg
        }
        case None => //message is already processed
      }
    }
    case ex: Exception => throw ex
  }
}

```

1 Send all the received messages to our own mailbox

2 Added for testing purposes

We added the ability to throw an exception to trigger a restart for testing purposes. But when we receive the same message type twice, how will our timeout

mechanism work? Because we do nothing when the messages are processed, it isn't a problem when we get the Timeout twice. And because it is a timeout, we don't want the timer to be reset. And in this example only the first timeout will take action when this is necessary. So this simple mechanism will work.

So does our change solve the problem? Let's test it by sending the first message and make the Aggregator restart before sending the second message. Is the Aggregator still able to combine the two messages despite the restart?

Listing 8.9 Aggregator missing a message

```

val endProbe = TestProbe()
val actorRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref)))
val photoStr = ImageProcessing.createPhotoString(new Date(), 60)
val msg1 = PhotoMessage("id1",
  photoStr,
  Some(new Date()),
  None)
actorRef ! msg1 1
actorRef ! new IllegalStateException("restart") 2
val msg2 = PhotoMessage("id1",
  photoStr,
  None,
  Some(60))
actorRef ! msg2 3
val combinedMsg = PhotoMessage("id1",
  photoStr,
  msg1.creationTime,
  msg2.speed)
endProbe.expectMsg(combinedMsg)

```

- 1** Send the first message
- 2** Restart the Aggregator
- 3** Send the Second message

The test passes, showing that the Aggregator was able to combine the message even after a restart. In messaging, durability refers to the ability to maintain messages in the midst of service disruptions. We implemented it simply by having the Actor resend any messages it might be holding to itself, and we verified that it works with a unit test (so if some aspect of the durable implementation is changed, our test will let us know before we suffer a runtime failure).

8.2.5 Combining the components into the Scatter-Gather Pattern

With each component tested and ready, we can now make a complete implementation of the pattern. Note, by developing each piece in isolation with unit tests, we enter this final assembly phase confident that each collaborator will do its job successfully.

Listing 8.10 Scatter-Gather implementation

```

val endProbe = TestProbe()
val aggregateRef = system.actorOf(
  Props(new Aggregator(1 second, endProbe.ref))) ❶
val speedRef = system.actorOf(
  Props(new GetSpeed(aggregateRef))) ❷
val timeRef = system.actorOf(
  Props(new GetTime(aggregateRef))) ❸
val actorRef = system.actorOf(
  Props(new RecipientList(Seq(speedRef, timeRef))) ❹
val photoDate = new Date()
val msg = PhotoMessage("id1",
  ImageProcessing.createPhotoString(photoDate, 60))
actorRef ! msg ❺
val combinedMsg = PhotoMessage(msg.id,
  msg.photo,
  Some(photoDate),
  Some(60))
endProbe.expectMsg(combinedMsg) ❻

```

- ❶ Create the Aggregator
- ❷ Create the GetSpeed actor and pipe it to the Aggregator
- ❸ Create the GetTime actor and pipe it to the Aggregator
- ❹ Create the recipient list of the GetTime and GetSpeed actors
- ❺ Send the message to the recipient list
- ❻ Receive the combined message

In this example we send one message to the first actor the recipient list. This actor creates two message flows which can be processed in parallel. Both results are sent to the Aggregator and when both messages are received, a single message is sent to the next step: our probe. This is how the scatter gather pattern works. In our example we had two tasks, but this pattern doesn't restrict the number of tasks.

The Scatter-Gather Pattern can also be combined with the Pipe and Filter Pattern. This can be done in two ways: the first is to have the complete Scatter-Gather Pattern as part of a pipe line. This means that the complete

Scatter-Gather Pattern is implementing one filter. The scatter component accepts the same messages as the other filter components in the filter pipeline. And the gather component sends only those interface messages.

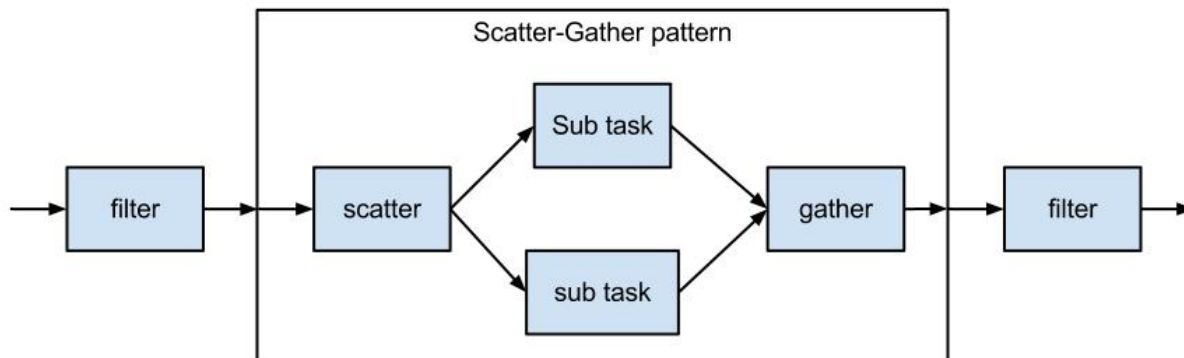


Figure 8.12 Use scatter gather pattern as filter

In Figure 8.12 we see the filter pipeline and one of the filters is implemented using the Scatter-Gather Pattern. This results in a flexible solution where we can change the order of filters and add or remove filters without disrupting the rest of the processing logic.

Another possibility is that the pipeline is part of the scattered flow. This means that the messages are sent through the pipeline before they are gathered.

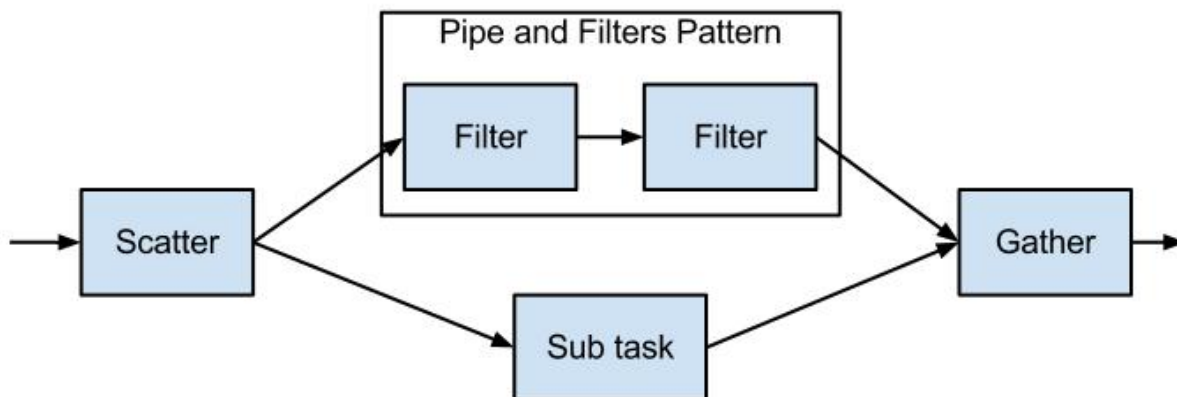


Figure 8.13 Use of Pipe and Filter Pattern in a Scatter-Gather Pattern

In Figure 8.13 you can see that the Scatter-Gather Pattern results in the message being scattered into two streams. One of the streams is a pipeline while the other is just a single processing task (per the prior example). Combining the patterns can be handy as systems grow larger; we are keeping the parts flexible and reusable. Now we'll turn our attention to applications that

8.3 Summary

In this chapter, we tackled the design of flexible collaborative solutions in Akka using some common enterprise patterns. By combining the patterns, we are able to create complex systems. Some of takeaways:

- Scaling processing requires that we spread work amongst concurrent collaborators
- Patterns give us a starting point on standard ways of doing that
- The Actor model allowed us to focus on the design of our code, not messaging and scheduling implementation details
- Patterns are building blocks that can be combined to build bigger system parts

Through all of these implementations, Akka has made it easy to adapt to more complex requirements without having to change the fundamental approach to constructing our components. We are still just sending messages to Actors, and sometimes those messages are part of a sequential process and at others, they are enabling concurrency. In the next chapter we will focus on the different functions of the tasks itself. After completing these chapters, you will be able ready to start implementing a complete system.

10

Message Channels

In this chapter

- Point to point
- Publish subscribe
- EventBus
- Dead letter
- Guaranteed delivery
- Reliable-proxy

In this chapter we are taken a closer look at the message channels which can be used to send messages from one Actor to another. We start with the two types of channels. The point-to-point channel and the Publish-subscribe. The point-to-point is the channel, which we used in all our examples until now. But to explain the differences between the two channels we included this type here too. Sometimes we need a more flexible method to send messages to receivers. In the publish-subscribe section we describe a method to send messages to multiple receivers and without knowing which receivers need the message. The receivers are kept by the channel and can change during the operation of the application. Other names which are often used for these kind of channels are EventQueue or EventBus. Akka has an EventStream which implements a publish-subscribe channel. But when this implementation isn't sufficient, then Akka has a collection of traits which helps to implement an custom publish subscribe channel.

Next we describe two special channels. The first is the Dead Letter channel, which contain message that couldn't be delivered. This is sometimes also called a dead message queue. This channel can help when debugging, why some messages aren't processed or to monitor where there are problems. In the last section we

describe the Guaranteed delivery channel. We describe that we can't create a reliable system without at least some guaranties of delivering messages. But we done;t need always the full guaranteed delivery. Akka doesn't have the full guaranty delivery, but we describe the level of guaranty delivery, the Akka framework supports, which differ for sending messages to local and remote actors.

10.1 Channel types

We start this chapter with describing the two types of channels. The first one is the point-to-point channel. The name describes it characteristics and connect one point, the sender to one other point, the receiver. Most of the time this is sufficient, but there are cases that we want to send a message to a number of receivers. In this case we need we need multiple channels or we use the second type channel the Publish subscribe channel. Another advantage of this channel is that the number of receivers can dynamically change when the application is operational. To support this kind of channel Akka has implemented the EventBus.

10.1.1 Point to point

A channel transports the message from the sender to the receiver. The point-to-point channel sends the message to one receiver. We have already used this kind of channel in all our previous examples. Since we already used this type of channel in the previous chapters, we will recap the important parts here to describe the differences between the two types of channels.

In the previous examples the sender knows the next step of the process and can decide which channel to use to send it's message to the next step. Some time it is just one like the "Pipe and Filter" examples of section 7.1. In these examples the sender has one AkkaRef where is sends the message when the actor has finished processing. But in other cases like the RecipientList of section 7.2.3, the actor has multiple channels and decide which channel or use multiple channels to send the message. This way the connectivity between the actors are more of a static nature.

Another characteristic of the channel is that when multiple messages are send the order of these messages are not changed. A point-to-point channel delivers the message to exactly one Receiver as shown in Figure 10.1.

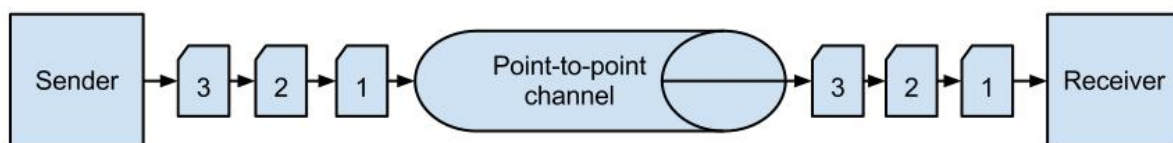


Figure 10.1 Point-to-point channel

It is possible for a point-to-point channel to have multiple Receivers, but the channel makes sure that only one Receiver receives the message. The round-robin Router in section 7.3.1 is an example of the channel having multiple receivers. The processing of the messages can be done concurrently by different Receivers, but only one Receiver consumes any one message. This is shown in figure 10.2.

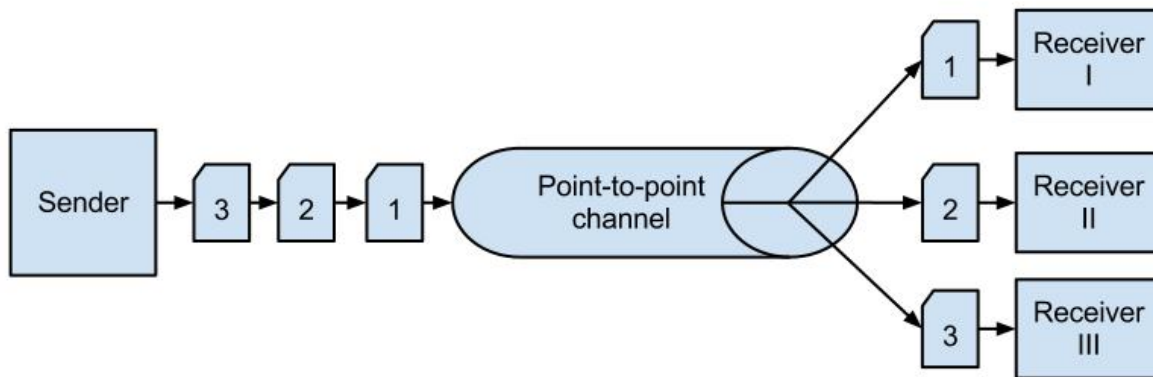


Figure 10.2 Point-to-point channel with multiple receivers

The channel has multiple receivers, but every message is delivered to one receiver. This type channel is used when the connection between sender and receiver is more of a static nature. The sender knows which channel it has to use to reach the receiver.

This type of channel is the most common channel when using Akka. Because in Akka the ActorRef is the implementation of a point-to-point channel. Because all the messages send will be delivered to one Actor. And the order of the message send to the ActorRef will not change when delivered to the receiving Actor.

10.1.2 Publish subscribe

We have seen in the previous section that the point-to-point channel each message delivers to only one Receiver. In these cases the sender knows where the message has to be send to. But some times the sender doesn't know who is interested in the message. This is the greatest difference between the point-to-point channel and the Publish-subscribe channel. The channel is responsible for keeping track of the receivers who need the message instead of the sender. The channel can also deliver the same message to multiple receivers.

In our web shop the first step is receiving the order. After this first step the system needs to take the next step in processing, which is deliver the book to the customer. Therefor the receiving step sends a message to the delivery step. But to

keep the inventory up to date we also need the order message in this component. At this point the received order needs to be distributed to two parts of the system as shown in Figure 10.3.

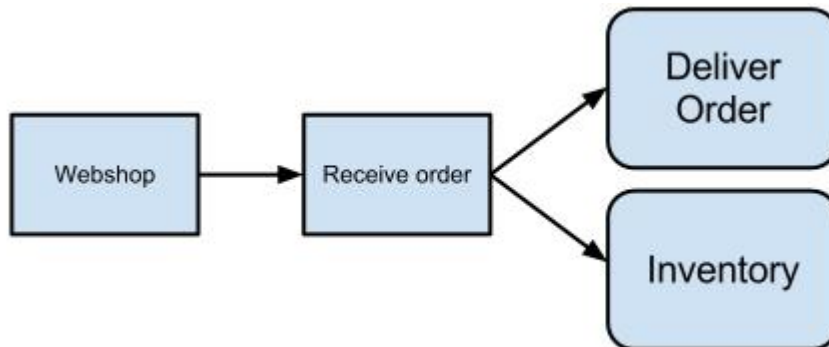


Figure 10.3 Web shop processing the order message

As an advertisement action we want to send a present when a customer buys a book. Therefore we extend our system with a gift module. And again the order message is needed. So every time we add a new subsystem we need to change the first step to send the message to more receivers. To solve this problem we can use the Publish-subscribe channel. The channel is able to send the same message to multiple receivers, without the sender knows which receiver. Figure 10.4 shows that the published messages are sent to the Delivery and the Inventory subsystem.

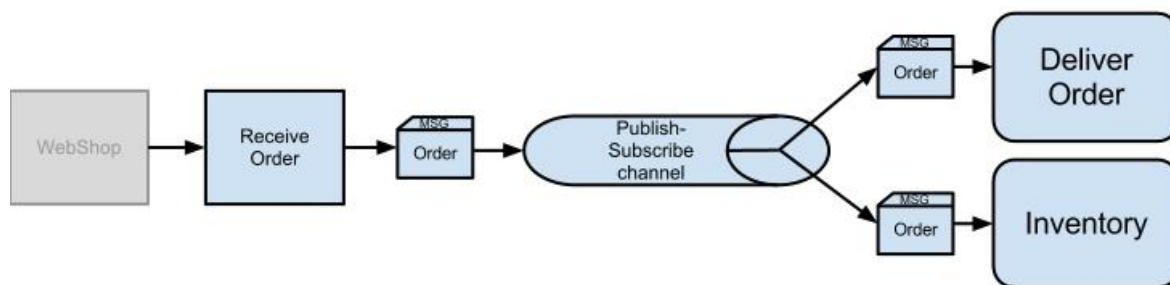


Figure 10.4 Using the publish-subscribe channel to distribute the order message

When we want to add the gift functionality we subscribe to the channel and doesn't need to modify the "Receive Order". Another benefit of this channel is that the number of receivers can differ during the operation and isn't static. For example we don't want to send always a present, only on the action days. When using this channel we are able to add the gift module to the channel only during the action period and remove the module from the channel when there is no gift action. This is shown in figure 10.5.

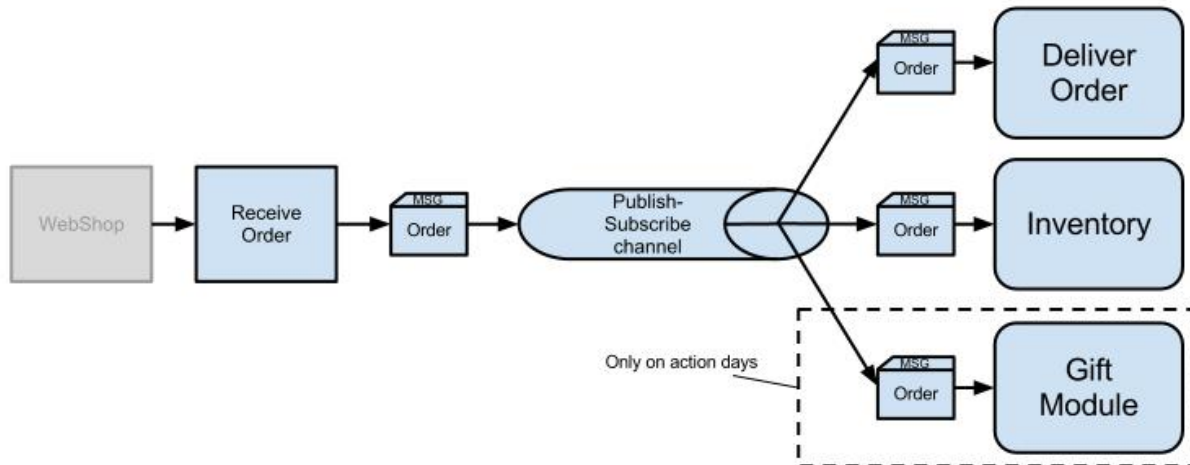


Figure 10.5 Gift module is receiving only messages on action days

When a receiver is interested in a message of the publisher, it subscribes itself to the channel. When the publisher sends a message through the channel, the channel makes sure that all the subscribers gets the message. And when the Gift Module doesn't need the order messages it unsubscribes itself from the channel. This makes that the channel methods can be divided into two usages. The first usage is done at the send side. Here one must be able to publish the messages. The other usage is at the receiver side. At this end the receivers must be able to Subscribe and Unsubscribe to the channel. Figure 10.6 shows the two usages.

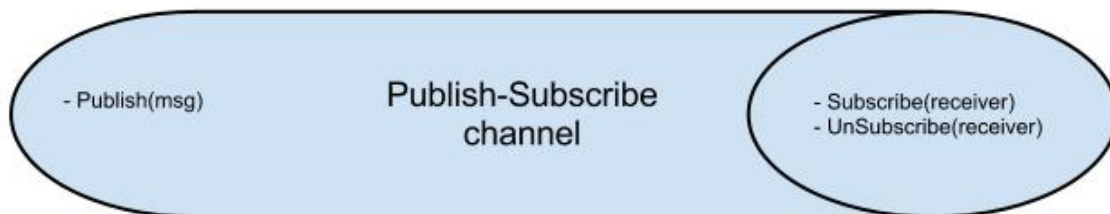


Figure 10.6 Usages of a publish-subscribe channel

Because the Receivers can subscribe itself to the channel, this solution is very flexible. The publisher doesn't need to know how many receivers it has. It is even possible that it has no receivers at some point, because the number of subscribers can variate during the operation of the system.

AKKA EVENTSTREAM

Akka has also support for this kind of channels. The most easiest when needed a publish-subscribe channel, is to use the EventStream. Every ActorSystem has one and is therefor also available in all Actors. The EventStream can be seen as a manager of multiple Publish-Subscribe channels. Because the Actor can be subscribed to specific message type and when someone publishes a message of that specific type the actor receives that message. The actor doesn't need any modifications to receive the messages received from the EventStream. And isn't any different as all the previous examples we have shown.

```
class DeliverOrder() extends Actor {
  def receive = {
    case msg: Order => ...//Process message
  }
}
```

The only difference is how the message is send. It isn't even necessary that the Actor does the subscribing itself. It is possible to make the subscription by anyone that has the actor reference and the EventStream. Figure 10.7 shows the subscribe interface of Akka. To subscribe an Actor to receive the Order messages, you need to call the subscribe method of the EventStream.

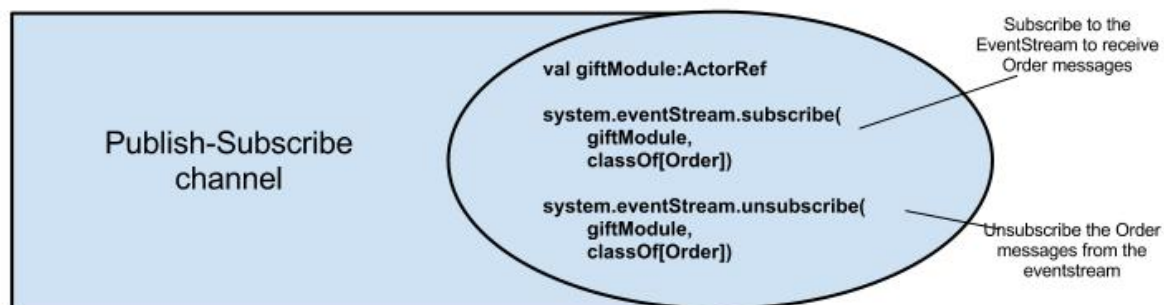


Figure 10.7 Subscribe Interface of EventStream

And when the Actor isn't interested anymore, for example our the gift action ends, than the method Unsubscribe can be used. In the example we Unsubscribe the GiftModule and after this method call the Actor doesn't receives any Order messages which are published.

This is all which have to be done when subscribing the GiftModule to receive

the Order message. After calling the subscribe method the GiftModule will receive all the Order messages, which are published to the EventStream. This method can be called for different Actors which need these Order messages. And when an Actor needs multiple message types, the subscribe method can be called multiple times with different message types.

To publish a message to the EventStream is also very easy, just call the publish method as shown in Figure 10.8

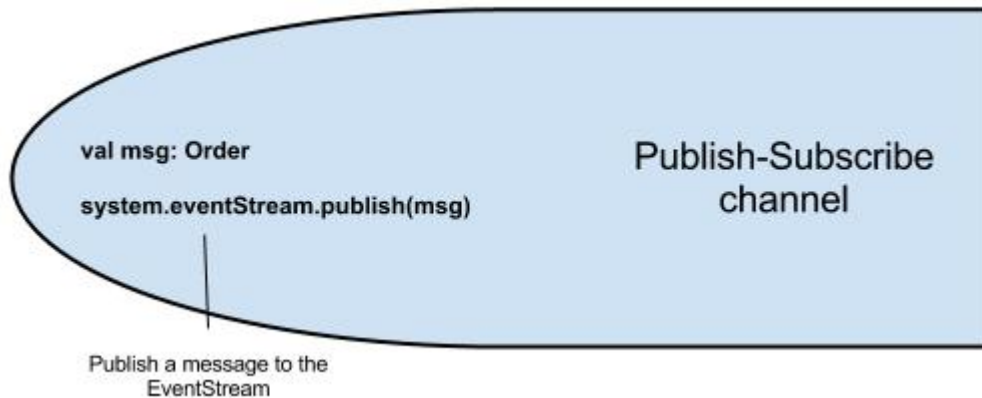


Figure 10.8 Publish Interface of EventStream

After this call the message "msg" is sent to all subscribed Actors. Which can do their processing. This is the complete Akka implementation of the Publish-Subscribe channel.

In Akka it is possible to subscribe for multiple message types. For example our GiftModule needs also the messages when an order is canceled, because the gift shouldn't be sent also. In this case the GiftModule has subscribed to the EventStream to receive the Order and Cancel messages. But when calling the Unsubscribe for the Orders, the subscription for the cancellations is still valid and these messages are still received. When stopping the GiftModule we need to Unsubscribe for all subscriptions. This can be done with one call

```
system.eventStream.unsubscribe(giftModule)
```

After this call the GiftModule isn't subscribed to any message type anymore. These set of four methods is the Akka interface of the publish-subscribe channel, which is quite simple. Listing 10.1 shows how we can use the Akka EventStream to receive Order messages.

Listing 10.1 EventStream in Action

```

val DeliverOrder = TestProbe() ❶
val giftModule = TestProbe() ❶
system.eventStream.subscribe( ❷
  DeliverOrder.ref,
  classOf[Order])
system.eventStream.subscribe( ❷
  giftModule.ref,
  classOf[Order])
val msg = new Order("me", "Akka in Action", 3)
system.eventStream.publish(msg) ❸
DeliverOrder.expectMsg(msg) ❹
giftModule.expectMsg(msg) ❹
system.eventStream.unsubscribe(giftModule.ref) ❺
system.eventStream.publish(msg)
DeliverOrder.expectMsg(msg)
giftModule.expectNoMsg(3 seconds) ❻

```

- ❶ Creating our Receiver Actors
- ❷ Subscribe our Receiver Actors to receive Order messages
- ❸ Publish an Order
- ❹ The message is received by both Actors
- ❺ Unsubscribe the GiftModule
- ❻ GiftModule doesn't receive the message anymore

We use the TestProbes as the receivers of the messages. And both receivers are subscribed to receive the Order messages. After publishing one message to the EventStream both receivers have received the message. And after unsubscribing the GiftModule, only the DeliverOrder is receiving the messages, just as we expected.

We already mentioned the benefit of decoupling the receivers and the sender and the dynamic nature of the publish-subscribe channel, but because the EventStream is available for all actors is also a nice solution for messages which can be send from all over the system and needs to be collected at one or more Actors. A good example is logging. Logging can be done throughout the system and needs to be collected at one point and be written to a log file. Internally the ActorLogging is using the EventStream to collect the log lines from all over the system.

This EventStream is very useful but sometimes we need more control and want

to write our own publish-subscribe channel. In the next sub section, we show how we can do that.

CUSTOM EVENTBUS

Let assume that we only want to send a gift when someone ordered more than one book. When implementing this our GiftModule only needs the message when the amount is higher than 1. When using the EventStream we can't do that filtering with the EventStream. Because the EventStream works on the class type of the message. We can do the filtering inside the GiftModule, but lets assume that this consumes resources we don't want. In that case we need to create our own publish-subscribe channel and Akka has also support to do that.

Akka has defined a generalized interface the EventBus, which can be implemented to create a publish-subscribe channel. An EventBus is generalized so that it can be used for all implementations of a publish-subscribe channel. In the generalized form there are three entities.

- Event
This is the type of all events published on that bus. In the Akka EventStream all uses AnyRef as event and therefor supports all type of messages
- Subscriber
This is the type of subscribers allowed to register on that event bus. In the Akka EventStream the subscribers are ActorRef's
- Classifier
This defines the classifier to be used in selecting subscribers for dispatching events. In the Akka EventStream the Classifier is the class type of the messages

By changing the definition of these entities, it is possible to create any publish-subscribe channel possible. The interface has place holders for the three entities and different publish and subscribe methods which are also available at the EventStream. In Listing 10.2 the complete interface of the EventBus is shown.

Listing 10.2 EventBus interface

```

package akka.event

trait EventBus {
  type Event
  type Classifier
  type Subscriber

  /**
   * Attempts to register the subscriber to the specified Classifier
   * @return true if successful and false if not (because it was
   * already subscribed to that Classifier, or otherwise)
   */
  def subscribe(subscriber: Subscriber, to: Classifier): Boolean

  /**
   * Attempts to deregister the subscriber from the specified Classifier
   * @return true if successful and false if not (because it wasn't
   * subscribed to that Classifier, or otherwise)
   */
  def unsubscribe(subscriber: Subscriber, from: Classifier): Boolean

  /**
   * Attempts to deregister the subscriber from all Classifiers it may
   * be subscribed to
   */
  def unsubscribe(subscriber: Subscriber): Unit

  /**
   * Publishes the specified Event to this bus
   */
  def publish(event: Event): Unit
}

```

The whole interface has to be implemented and because most implementations needs the same functionality Akka has also a set of composable traits implementing the EventBus interface, which can be used to easily create your own implementation of the EventBus.

Lets implement a custom EventBus for our GiftModule to be able to receive only the Orders which have multiple books. With our EventBus we can send and receive Orders, therefore the Event we are using in our EventBus will be the Order class. To define this in our OrderMessageBus we simply set de event type defined in the EventBus

```

class OrderMessageBus extends EventBus {
  type Event = Order
}

```

}

Another entity which we needed to define is the Classifier. In our example we want to make a difference between a single book orders and orders with multiple books. We have chosen to classify the Order messages on the criteria "is Multiple Book Order" and use a Boolean as classifier and therefore we have to define the Classifier as a Boolean. This is defined just as the event.

```
class OrderMessageBus extends EventBus {
  type Event = Order
  type Classifier = Boolean
}
```

We skip the subscriber entity for now, because we are going to define that a little different. We have defined our Classifier and need to keep track of the subscribers for each Classifier. In our case for the two values of "is Multiple Book Order" true and false. Akka has three composable traits which can help in keeping track of the subscribers. All these traits are still generic. So they can be used with any Entities you have defined. This is done by introducing new abstract methods.

- **LookupClassification**
This trait uses the most basic classification. It maintain a set of subscribers for each possible classifier and extract a classifier from each event. How it extract a classifier is done with the classify method which should be implemented by the custom EventBus implementation.
- **SubchannelClassification**
This trait is used when classifiers form a hierarchy and it is desired that subscription can be possible not only at the leaf nodes, but also to the higher nodes. This trait is used in the EventStream implementation, because classes have a hierarchy and it is possible to use the superclass to subscribe to extended classes.
- **ScanningClassification**
This trait is a more complex one, it can be used when classifiers have an overlap. This means that one Event can be part of more classifiers, for example if we give more gifts when ordering more books. When ordering more than 1 book you get a book marker, but when you order more than 10, you get also an coupon for your next order. So when I order 11 copies, the order is part of the classifier more than 1 book and more than 10 books. When this order is published the subscribers of "more than one book" need the message, but also the subscribers of "more than 10 books" needs this order. For this situation the ScanningClassification trait can be used.

In our implementation we are going to use the LookupClassification. The other two are similar to this one. These traits implements the subscribe and Unsubscribe

methods of the `EventBus` interface. But they also introduce new abstract methods which need to be implemented in our class. When using the `LookupClassification` trait we need to implement the following

- `classify(event: Event): Classifier`
This is used for extracting the classifier from the incoming events.
- `compareSubscribers(a: Subscriber, b: Subscriber): Int`
This method must define a order over the subscribers, to be able to compare them just as the `java.lang.Comparable.compare` method.
- `publish(event: Event, subscriber: Subscriber)`
This method will be invoked for each event for all subscribers which registered themselves for the events classifier.
- `mapSize: Int`
This returns the expected number of the different classifiers. This is used for the initial size of an internal data structure.

We use the "is Multiple Book Order" as classifier. And this has two possible values, therefore we use the value 2 for the `mapSize`.

```
import akka.event.{LookupClassification, EventBus}

class OrderMessageBus extends EventBus with LookupClassification {
  type Event = Order
  type Classifier = Boolean

  def mapSize = 2 ❶

  protected def classify(event: StateEventBus#Event) = {
    event.number > 1 ❷
  }
}
```

- ❶ Set the `mapSize` to 2
- ❷ Return true when the number is greater than 1 and otherwise false, which is used as classifier

And we mentioned that the `LookupClassification` must be able to get a classifier from our event. This is done by the `classify` method. In our case we just return the result of the check `event.number > 1`. All we need to do now is to define the subscriber, for this we are using the `ActorEventBus` trait. This is probably the trait that will be used most of the time in a Akka message system, because this trait defines that the subscriber is an `ActorRef`. It also implements the `compareSubscribers` method needed by the `LookupClassification`. The only method

we still need to implement is the publish method, before we are done. The complete implementation is shown in Listing 10.3.

Listing 10.3 Complete implementation of the OrderMessageBus

```
import akka.event.ActorEventBus
import akka.event.{ LookupClassification, EventBus }

class OrderMessageBus extends EventBus
  with LookupClassification
  with ActorEventBus {

  type Event = Order
  type Classifier = Boolean
  def mapSize = 2

  protected def classify(event: OrderMessageBus#Event) = {
    event.number > 1
  }

  protected def publish(event: OrderMessageBus#Event,
    subscriber: OrderMessageBus#Subscriber) {
    subscriber ! event
  }
}
```

- 1 Extends our class with the two support traits of Akka
- 2 Define the entities
- 3 Implement the classify method
- 4 Implement the publish method by sending the event to the subscriber

At this moment we are done implementing our own EventBus and can be used to subscribe to and publish messages. In listing 10.4 we see an example how this EventBus can be used.

Listing 10.4

```

val bus = new OrderMessageBus ❶
val singleBooks = TestProbe()
bus.subscribe(singleBooks.ref, false) ❷
val multiBooks = TestProbe()
bus.subscribe(multiBooks.ref, true) ❸
val msg = new Order("me", "Akka in Action", 1)
bus.publish(msg) ❹
singleBooks.expectMsg(msg)
multiBooks.expectNoMsg(3 seconds) ❺
val msg2 = new Order("me", "Akka in Action", 3)
bus.publish(msg2) ❻
singleBooks.expectNoMsg(3 seconds)
multiBooks.expectMsg(msg2)

```

- ❶ Create the `OrderMessageBus`
- ❷ Subscribe the `singleBooks` to the single book classifier (`false`)
- ❸ Subscribe the `multiBooks` to the multi book classifier (`true`)
- ❹ Publish a order with one copy
- ❺ Only the `singleBooks` receives the message
- ❻ When publishing a order with multiple copies only the `multiBooks` receives the message

As you can see our custom `EventBus` works exactly as the `EventStream`, only the used classifier is different. Akka has several other traits which can be used. More details about these traits can be found in the Akka documentation.

As we have seen in this section Akka has support for publish-subscribe channels. In most cases the `EventStream` will be sufficient, when in need for a publish-subscribe channel. But when you need more specialized channels, it is possible to create your own, by implementing the `EventBus` interface. This is a generalized interface, which can be implemented in any way you need. To support the implementation of an custom `EventBus` Akka has several traits which can be used to implement a part of the `EventBus` interface.

In this section we have seen the two basic types of channels. In the next section we take a look at some special channels.

10.2 Specialized channels

In this section we take a look at two special channels. The first one is the DeadLetter channel. This isn't a really a different type of channel as shown in the previous section. The use of this channel is specific, because one doesn't send messages to it. Only failed message are put on this channel. Listening on this channel can help to find problems in your system. The second channel is the Guaranteed deliver channel. Again this isn't a new type, but this channel guaranties all messages which are send are also delivered. But we explain in that section that there are several levels of Guaranties of the delivery, which is important to know when creating a reliable system. Especially because Akka doesn't support the full guaranty delivery.

10.2.1 Dead letter

The Enterprise Integration patterns also describe a "dead letter channel" or "dead letter queue". This is a channel which contain all the messages which can't be processed or delivered. This channel is also called "dead message queue". This is a normal channel but you don't normally send any messages using this channel. Only when there are problems with the message for example it can't be delivered, the message is placed on this channel. This is shown in figure 10.9.

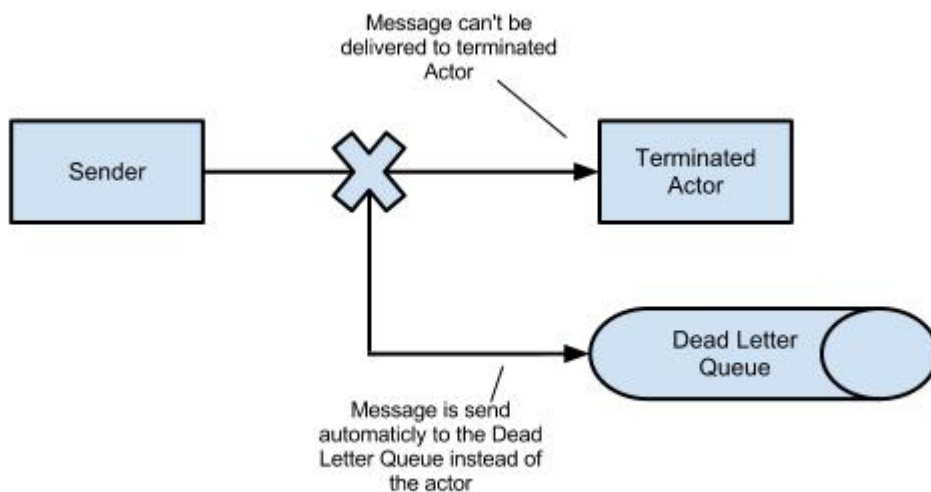


Figure 10.9 DeadLetter queue

By monitoring this channel you know which messages aren't processed and can take corrective actions. Especially when testing your system, this queue can be

helpful why some messages aren't processed. When creating a system that isn't allowed to drop any messages, this queue can be used to reinsert the messages when the initial problems are solved.

Akka is using the `EventStream` to implement the dead letter queue. This way only the actors which are interested in the failed messages are receiving them. When a message is queued in a mailbox of an actor that Terminates or is send after the Termination, the message is send to the `EventStream` of the `ActorSystem`. The message is wrapped into a `DeadLetter` object. This Object contains the original message, the sender of the message and the intended receiver. This way the Dead letter queue is integrated in the `EventStream`. To get these dead letter messages you only need to subscribe your actor to the `EventStream` with the `DeadLetter` class as the `Classifier`. This is the same as described in the previous section, but only we using here another messages type, the `DeadLetter`.

```
val deadLetterMonitor: ActorRef

system.eventStream.subscribe(
    deadLetterMonitor,
    classOf[DeadLetter])
```

After this subscribe the `deadLetterMonitor` is getting all the messages that fails to be delivered. Let take a look at a small example. We create a simple `Echo` actor which sends the message receive back to the sender and after starting the actor we send directly a `PoisonPill`. This will result in the actor to be terminated. In Listing 10.5 we show that we receive the message when we subscribed to the `DeadLetter` queue.

Listing 10.5 Catching messages which can't be delivered

```

val deadLetterMonitor = TestProbe()
system.eventStream.subscribe( ❶
    deadLetterMonitor.ref,
    classOf[DeadLetter])
val actor = system.actorOf(Props[EchoActor], "echo")
actor ! PoisonPill ❷
val msg = new Order("me", "Akka in Action", 1)
actor ! msg ❸
val dead = deadLetterMonitor.expectMsgType[DeadLetter] ❹
dead.message must be(msg)
dead.sender must be(testActor)
dead.recipient must be(actor)

```

- ❶ Subscribe to the DeadLetter channel
- ❷ Terminate the Echo Actor
- ❸ Send a message to the terminated Actor
- ❹ Expect a DeadLetter message in the DeadLetterMonitor

Messages send to a Terminated Actor can't be processed anymore and the ActorRef of this actor should not be used anymore. When there are messages send to a terminated Actor, these message will be send to the DeadLetter queue. And we see that our message is indeed received by our deadLetterMonitor.

Another use of the DeadLetter queue is when the processing fails. This is a Actor specific decision. An actor can decide that a received message couldn't be processed and that it doesn't know what to do with it. In this situation the messages can be send to the dead letter queue. The ActorSystem has a reference to the DeadLetter Actor. When a message need to be send to the dead letter queue, you can send it to this Actor.

```
system.deadLetters ! msg
```

When sending a message to the DeadLetter, it is wrapped also into a DeadLetter object. But the initial receiver become the DeadLetter Actor. When creating an auto correcting system, information is lost when sending the message this way to the DeadLetter Queue. For example the original sender is lost, The only information you got is the Actor which has send the message to the queue. This can be sufficient, but when you need also the original sender, it is possible to send

a `DeadLetter` Object instead of the original message. When this message type is received, the wrapping is skipped and the message send is put on the queue without any modification. In Listing 10.6 we send a `DeadLetter` Object and see that this message isn't modified.

Listing 10.6 Sending `DeadLetter` messages

```
val deadLetterMonitor = TestProbe()
val actor = system.actorOf(Props[EchoActor], "echo") ❶
system.eventStream.subscribe(
  deadLetterMonitor.ref,
  classOf[DeadLetter])
val msg = new Order("me", "Akka in Action", 1)
val dead = DeadLetter(msg, testActor, actor) ❷
system.deadLetters ! dead
deadLetterMonitor.expectMsg(dead) ❸
system.stop(actor)
```

- ❶ Create a Actor reference which will be used as initial recipient
- ❷ Create the `DeadLetter` message and send it to the `DeadLetter` Actor
- ❸ The `DeadLetter` message is received in the monitor

As shown in the example the `DeadLetter` message is received unchanged. This makes it possible to handle all the messages, which are not processed or couldn't be delivered, the same. What to do with the messages is completely depended on the system our are creating. Sometimes it isn't even important to know that messages were dropped, but when creating a highly robust system, you may want to resend the message again to the recipient like it was send initially.

In this section we described how to catch message which failed to be processed. In the next section we describe another specialize channel, according the Enterprise Integration patterns, the `Guaranteed delivery` channel

10.2.2 *Guaranteed delivery*

The guaranteed delivery channel is point-to-point channel with the guaranty that the message is always delivered to the receiver. This means that the delivery is done even when all kind of errors occurs. This means that the channel must have all kind of mechanism and checks to be able to guaranty the delivery, for example the message has to be saved on disk in case the process crashes. Don't we need always the guaranteed delivery channel, when creating a system? Because how can we create a reliable system, when it isn't guarantied that messages are delivered? Yes we need some guaranties, but we don't need always the maximum available guaranty.

Actually, implementations of a guaranteed delivery channel, aren't able to guaranty the delivery in all situations. For example when a message is send from one location and that location burns down. In that situation no possible solution can found to send the message anywhere, because it is lost in the fire. The question we need to ask is, is the level of guaranty sufficient for our purpose. What would happened when the message was delivered? Probably the receiver would fail because of the same reason that most of the system isn't available anymore.

When creating a system, we need to know what guaranties the channel has and if that is sufficient for your system. Let's take a look at the guaranties Akka provides.

The general rule of message delivery is that messages are delivered at-most-once. This means that Akka promise that messages are delivered once or fails to deliver, Which means that the message is lost. This doesn't look good to build a reliable system. Why doesn't Akka implement a full guarantied delivery. The first reason is that while implementing this, you have to address several challenges, which make it complex and needs a lot of overhead to send one message. This results in a performance penalty even when you don't need that level of guaranty delivery.

Secondly nobody needs just Reliable Messaging. One wants to know if the request was successful processed, which is done by receiving a business-level acknowledgement message. This is not something Akka could make up, because this is system depended. The last reason why Akka doesn't implements a full guaranty delivery, is that it is always possible to add stricter guarantees on top of basic ones, when needed. The other way around to lose guaranties, to improve performance is not possible.

But when Akka can't guaranty that a message is delivered, makes it very hard to

create a reliable message system. But this is the basic rule for delivery of messages to local and remote actors. When we look these two situations separately, we see that Akka isn't as bad as it sounds.

Sending local messages will not likely fails, because it is like a normal method call. This fails only when there are catastrophic VM errors, like `StackOverflowError`, `OutOfMemoryError` or a memory access violation. In all of these cases, the actor was very likely not in a position to process the message anyway. So the guaranties when sending a message to a local actor, are pretty good and reliable.

The problem of losing the messages is when using remote actors. When using remote actors, it is a lot more likely for a message delivery failure to occur. Especially when an intermediate unreliable network is involved. If someone unplugs an Ethernet cable, or a power failure shuts down a router, messages will be lost and the actors would be able to process them just fine if it was received. To solve this the `ReliableProxy` is created. This makes sending messages as reliable as sending local messages. The only consideration is that both JVM's of the sender and receiver are influencing the reliability of this channel.

How does the `ReliableProxy` work? When starting the `ReliableProxy` creates a tunnel between the two `ActorSystems` on the different Nodes.

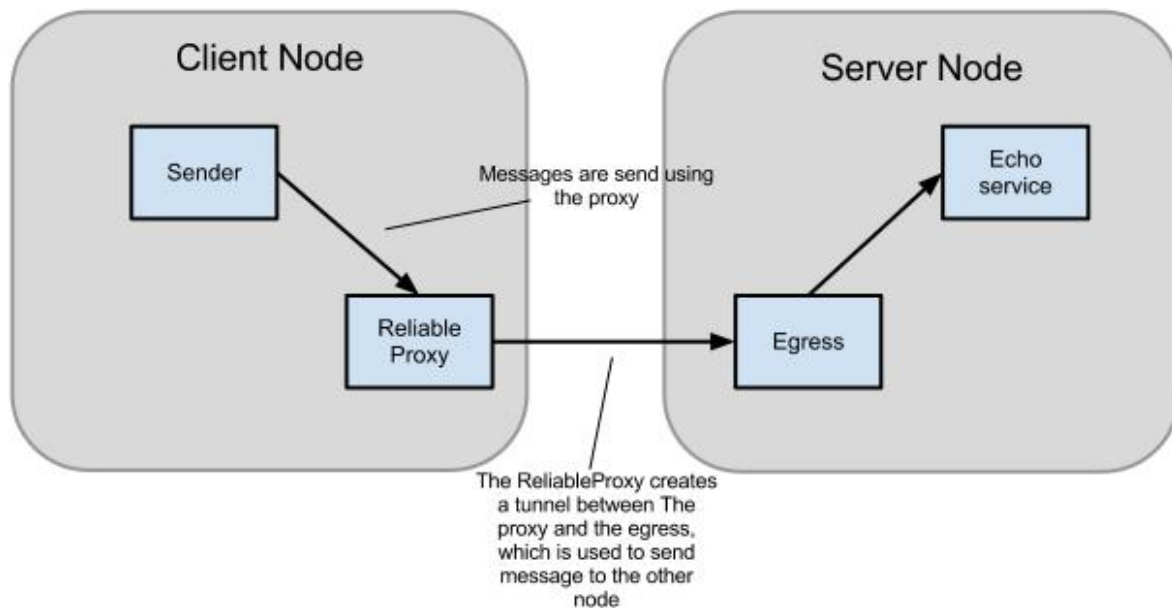


Figure 10.10 `ReliableProxy`

As shown in Figure 10.10 this tunnel has an entry, which is the `ReliableProxy`

and an exit the Egress. The Egress is an Actor which is started by the ReliableProxy and both Actors implements the checks and resend functionality to be able to keep track of which of the messages are delivered to the remote receiver. When the delivery fails the ReliableProxy will retransmit the messages until it succeed. When the egress has received the message it checks if it was already was received and send it to the actual receiver. But what happens when the target actor is terminated. When this happens it is impossible to deliver the message. This is solved by the ReliableProxy to terminate also when the target terminates. This way the system behaves the same way as using a direct reference. On the receiver side the difference is also not visible between sending messages direct or using the proxy. One restriction of using the ReliableProxy is that the tunnel is only one-way and for one receiver. This means that when the receiver replies to the sender the tunnel is NOT used. When the reply has to be also reliable, than another tunnel has to be made between the Receiver and the Sender.

Now let us see this in action. To create a Reliable proxy is simple all we need is a reference to the remote target Actor

```
import akka.contrib.pattern.ReliableProxy

val echo = system.actorFor(node(server) / "user" / "echo")

val proxy = system.actorOf(
  Props(new ReliableProxy(echo, 500.millis)), "proxy")
```

In the example we create a proxy using the echo reference. We also add a retryAfter value of 500 milliseconds. When failing to send a message it is retried after 500 milliseconds. This is all we have to do to use the Reliable Proxy. To show the result we create a Multi-node test with two nodes, the client and server node. On the server Node we create a EchoActor as receiver and on the client node we run our actual test. Just as in Chapter 5 we need the multitude configuration and the STMultinodeSpec for our ReliableProxySample test class.

```
import akka.remote.testkit.MultiNodeSpecCallbacks
import akka.remote.testkit.MultiNodeConfig
import akka.remote.testkit.MultiNodeSpec

trait STMultinodeSpec
  extends MultiNodeSpecCallbacks
  with WordSpec
  with MustMatchers
```

```

with BeforeAndAfterAll {
  override def beforeAll() = multiNodeSpecBeforeAll()
  override def afterAll() = multiNodeSpecAfterAll()
}

object ReliableProxySampleConfig extends MultiNodeConfig {
  val client = role("Client")
  val server = role("Server")
  testTransport(on = true)
}

class ReliableProxySampleSpecMultiJvmNode1 extends ReliableProxySample
class ReliableProxySampleSpecMultiJvmNode2 extends ReliableProxySample

```

1 Define client Node
2 Define server Node
3 We want to simulate Transport failures

Because we want to demonstrate that the message is send even when the network is down for a while, we need to turn on the testTransport. As we mentioned we need to run an EchoService on the server node.

```

system.actorOf(Props(new Actor {
  def receive = {
    case msg:AnyRef => {
      sender ! msg
    }
  }
}), "echo")

```

This service echo's every message it receives back to the sender. When this is running we can do the actual test on the client node. To create the full environment where we can do our test is shown in Listing 10.7.

Listing 10.7 Setup of the environment for the ReliableProxySample test

```

import akka.remote.testconductor.Direction
import scala.concurrent.duration._
import concurrent.Await
import akka.contrib.pattern.ReliableProxy

class ReliableProxySample
  extends MultiNodeSpec(ReliableProxySampleConfig)
  with STMultiNodeSpec
  with ImplicitSender {

  import ReliableProxySampleConfig._

  def initialParticipants = roles.size

  "A MultiNodeSample" must {

    "wait for all nodes to enter a barrier" in {
      enterBarrier("startup")
    }

    "send to and receive from a remote node" in {
      runOn(client) {
        enterBarrier("deployed")

        val echo = system.actorFor(node(server) / "user" / "echo") ❶
        val proxy = system.actorOf(
          Props(new ReliableProxy(echo, 500.millis)), "proxy") ❷

        ... Do the actual test
      }

      runOn(server) {
        system.actorOf(Props(new Actor { ❸
          def receive = {
            case msg:AnyRef => {
              sender ! msg
            }
          }
        }), "echo")
        enterBarrier("deployed")
      }

      enterBarrier("finished")
    }
  }
}

```

❶ Create reference to echo service

❷

- ☺ Create ReliableProxy tunnel
- ③ Implement the echo service

Now that we have our complete test environment we can implement the actual test. In Listing 10.8 we show that the message which is send while there was no communication between the nodes is only processed when we use the proxy. When using the direct actor reference the message is lost.

Listing 10.8 Implementation of the ReliableProxySample

```

proxy ! "message1"           ①
expectMsg("message1")
Await.ready(
  testConductor.blackhole( client, server, Direction.Both), ②
  1 second)

echo ! "DirectMessage"     ③
proxy ! "ProxyMessage"
expectNoMsg(3 seconds)

Await.ready(
  testConductor.passThrough( client, server, Direction.Both), ④
  1 second)

expectMsg("ProxyMessage")  ⑤
echo ! "DirectMessage2"    ⑥
expectMsg("DirectMessage2")

```

- ① Test the proxy under normal conditions
- ② Turn off the communication between the two nodes
- ③ Send a message using both references
- ④ Restore the communication
- ⑤ The message send using the proxy is received
- ⑥ Final test that the direct send messages are received when communication is restored

Using the ReliableProxy gives you the same guaranties for remote Actors as local actors. Which is that as long there are no critical VM errors in the JVM runtime on all Nodes of the system, the message is delivered one time to the destination Actor.

In this chapter we have seen that Akka doesn't have a full Guaranteed delivery channels, but there is a level of guaranties Akka can give. For local actors the delivery is guaranteed as long there are no critical VM errors. For remote actors the at-most-once the delivery is guaranteed. But this can be improved by using the

ReliableProxy when sending the message to across the JVM boundaries. Using this proxy gives the same guaranties as sending messages locally.

These guaranties of delivery is enough for most system, but when a system needs more Guaranties you can create mechanism on top of the Akka delivery system to get those guaranties. This isn't implemented by Akka because this is often system specific and take a performance hit which isn't necessary on most cases.

10.3 Summary

We have seen in this chapter that there are two types of messaging channels. The point-to-point channel, which sends a message to one receiver and the publish-subscribe channel, which can send a message to multiple receivers. Receivers can subscribe itself to the this last channel, which makes the receivers dynamic. At any time the number can variate. Akka has a the EventStream which is the default implementation of a publish-subscribe channel. Which use the class types of the messages as classifier. Akka has several traits which can be used to make your own publish-subscribe channel, when the EventStream isn't sufficient.

We have also seen that Akka has a DeadLetter channel, which uses the EventStream. This channel contains all the message that could not be delivered to the requested Actor. And can be used when debugging your system, when message sometimes get lost.

In the last section, we took a closer look at the delivery guaranties of Akka. And seen that there is a difference between messages send to local Actors and remote Actors. And when we need the same delivery guaranties we can use the ReliableProxy. But be careful this is only one-way. When the receiver sends a message to the sender, the ReliableProxy isn't used.

In this chapter we have seen how we can send messages between Actors. But the Actors implements the functionality of the system. When implementing these functionality, it is possible that the actor needs state to be implement this functionality. Next chapter we see how we can implement actors with state.

Finite State Machines and Agents



In this chapter

- Finite State Machine
- Agents
- Shared state

This book has advanced many reasons for stateless components when implementing a system. This is to avoid all kinds of problems, like restoring state after an error. But in most cases there are components within a system which need state to be able to provide the required functionality. In chapter 7, we saw two possible ways to keep state in an Actor. The first was to use class attributes, which we showed in our aggregator example. This is the simplest way. The second solution was to use the become/unbecome functionality, which we used in our state-dependent router. These two mechanisms are the more basic ways to implement state. But in some situations, these solutions are insufficient.

In this chapter, we show you two other solutions for dealing with state. We start with how to design dynamic behavior, depending on the actor's state, using Finite State Machine Modeling. We create an example model which will be implemented in the second section, where we show that Akka has support for easily implementing a Finite State Machine. In the last section, we show how we can share state between different threads by using Akka agents. Using these agents eliminates the need to use locking mechanisms, because the state of the Agents can be changed only asynchronously using events, but the state can be read synchronously, without any performance penalty.

11.1 Using a Finite State Machine

Finite-state machine (FSM), also called a state machine, is a common, language-independent modeling technique. FSMs can model a large number of problems, common applications are communication protocols, language parsing, and even business problems like Purchase Orders, Quotes, and Orders. What they encourage is isolation of state; we will see our Actors called on mostly to transition things from one state to another, in atomic operations, thus no locks will be needed. For those who have not encountered them, we start with a short description. After this introduction, we move on to an FSM example, which we will implement with Akka in the next section.

11.1.1 Quick introduction of Finite State Machine

The simplest example of a Finite State Machine is a device whose operation proceeds through several states, transitioning from one to the next as certain events occur. The washing machine is usually the classic example used to explain FSMs: there is a process that requires initiation steps, then once the machine takes over, it progresses through a sequence of specific states (filling the tub, agitation, draining, spinning). The transitions in the washing machine are all triggered by a program that wants a certain amount of each stage, based on the User's desires (light/heavy loads, prewash, etc.). The machine is only ever in one state at a time. The PO process mentioned above is a similar example from business: there is an established protocol for two parties to define an exchange of goods or services. With the example of the business documents, we see that for each stage of the machine, there is a state representation (a PO, or a Quote or a Request for Quote). Modeling software this way allows us to deal with state in an atomic, isolated way, which is a core principal of the Actor model.

An FSM is called a machine because it can only be in one of a finite number of states. Changing from one state to another is triggered by an event or condition. This state change is called a transition. A particular FSM is defined by a number of states and the different triggers for all the possible transitions. There are a lot of different ways to describe the FSM, but most of the time the FSM is described in some kind of a diagram. In Figure 11.1 we show a simple diagram to illustrate how we describe the FSM, because there are a number of different notations when creating an FSM diagram.

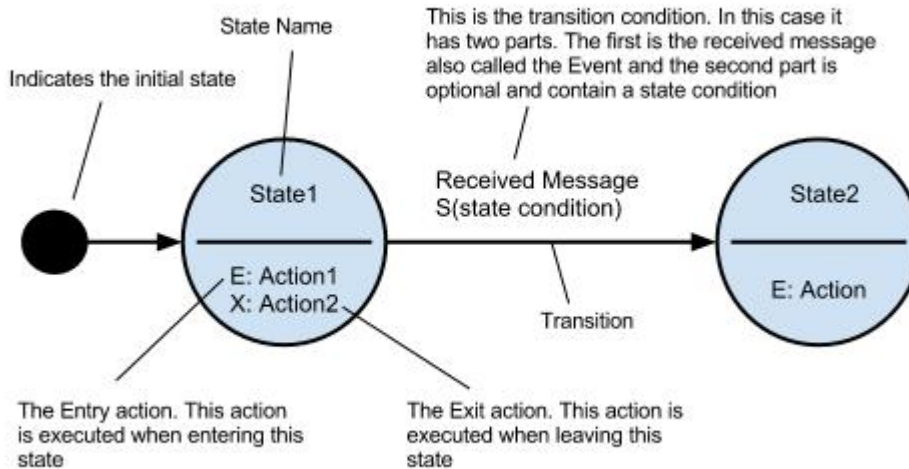


Figure 11.1 Diagram example of a Finite State Machine

In this example we show an FSM with two states, State1 and State2. When instantiating the machine, we start in State1, which is shown in the diagram by the black dot. State1 has two different actions. An Entry action and an Exit action. (Although we won't use the exit action in this chapter, we show it so you'll understand how the model works.) Just as the name says, the first action is executed when the machine sets the state State1 and the second when the machine changes from State1 to another state. In this example we have only two states so this action is only executed when it goes to State2. In the next examples, we only use the Entry actions, because this is a simple FSM. Exit actions can do some cleaning or restore some state, so they don't embody part of the logic of the machine. It can be seen more like a finally clause in a try-catch statement, which must always be executed when exiting the try block.

Changing state, which is called a transition, can only happen when the machine is triggered by an event. In the diagram this transition is shown by the arrow between State1 and State2. The arrow indicates the Event and optionally a state condition (for instance, we might only transition to the spin cycle when the tank is empty). The events in the Akka FSM are the messages the Actor receives. That's it for the introduction, now let's see how an FSM can help us implement a solution to a real problem.

11.1.2 Creating an FSM model

The example we are going to see to show how we can use FSM support in Akka is the inventory system of the bookstore. The inventory Service gets requests for specific books and sends a reply. When the book is in inventory, the order system gets a reply that a book has been reserved. But it is possible that there aren't any books left and that the inventory will have to ask the publisher for more books, before it can service the order. These messages are shown in Figure 11.2.

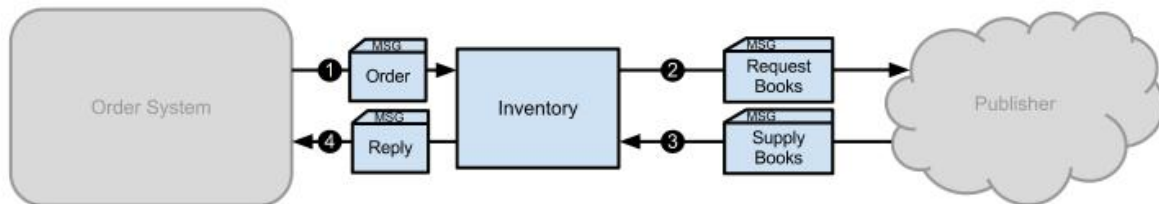


Figure 11.2 The Inventory example

To keep the example simple, we have only one type of book in our inventory and we support ordering only one book at the time. When an order is received the inventory checks if it has any copies of that book. When there are copies, the reply is created that the book is reserved. But when there aren't any copies of the requested book left, the processing has to wait and request more books from the Publisher. The publisher can respond by supplying more books or with a sold out message. During the wait for more books, other orders can be received.

To describe the situation we can use an FSM, because the inventory can be in different states and expect different messages before it can proceed to the next step. Figure 11.3 shows our problem using the FSM.

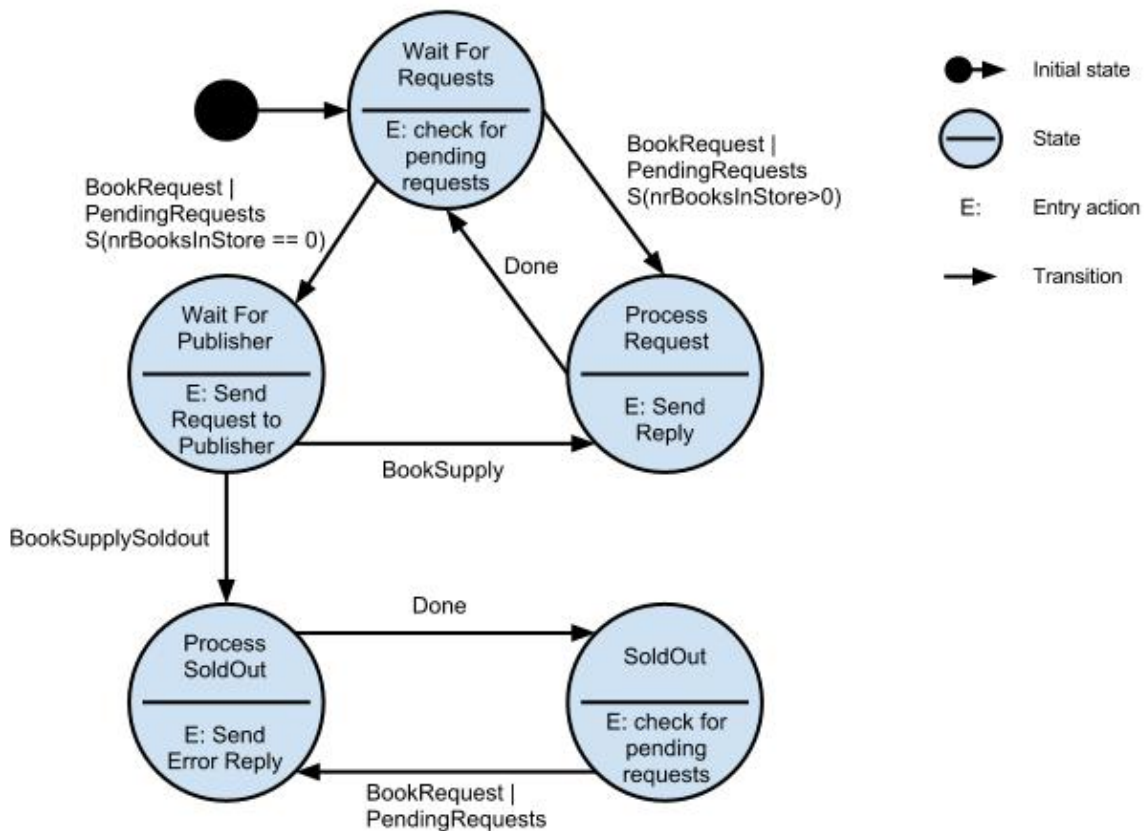


Figure 11.3 FSM of the Inventory example

One thing the diagram does not depict is the fact that we can still receive `BookRequests`, which will be added to the `PendingRequest` List, while in our wait state. This is important because it represents the preservation of needed concurrency. Note that when we get back to the wait state, it is possible that there are pending requests. The entry action is to check and if there are, trigger one or both transitions depending on the number of books in the store. When the books are sold out, the state becomes 'Process SoldOut.' This state sends an error reply to the order requester and triggers the transition to the state 'SoldOut.' FSMs give us the ability to describe complex behaviour in a clear, concise manner.

Now that we have described our solution using an FSM, let's see how Akka can help to implement our FSM model.

11.2 Implementation of an FSM model

In section 7.3.3 we saw the become/unbecome mechanism. This can help in implementing an FSM; just as we did in the State base router: we can map behaviors to states. It is possible to use become/unbecome mechanism for small and simple FSM models. But when there are multiple transitions to one state, the implementation of the Entry action has to be implemented in different become/receive methods, which can be hard to maintain for more complex FSMs. Therefore, Akka provides an FSM trait, which we can use when implementing an FSM Model. This results in clearer and more maintainable code. In this section, we explain how to use this FSM trait. We start by implementing the transitions of our inventory FSM, and in the next section we implement the entry actions to complete the implementation of the inventory FSM. At this point, we implement the designed FSM, but Akka FSM also has support for using timers within the FSM, which is described next. We end with the Termination of the Akka FSM, which enables us to do some cleanup when needed.

11.2.1 Implementing transitions

To start implementing an FSM model using Akka, we create an Actor with the FSM trait. (The FSM trait may only be mixed into an Actor.) Akka has chosen the self type approach instead of extending Actor to make it obvious that an actor is actually created. When implementing an FSM, we need to take several steps before we have a complete FSM Actor. The two biggest ones are defining the state and then the transitions. So let's get started creating our Inventory FSM, by making an Actor with the FSM trait mixed in.

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

class Inventory() extends Actor with FSM[State, StateData] {
  ...
}
```

The FSM trait takes two type parameters:

1. State
The super type of all state names
2. StateData
The type of the state data which are tracked by the FSM.

The super type is usually a sealed trait with case objects extending it, because it

doesn't make sense to create extra states without creating transitions to those states. So let's start to define our states. We'll do that in the next section.

DEFINING THE STATE

The state definition process starts with a single trait (appropriately named) 'State,' with cases for each of the specific states our object can be in (note: this helps make the FSM code self-documenting).

```
sealed trait State
case object WaitForRequests extends State
case object ProcessRequest extends State
case object WaitForPublisher extends State
case object SoldOut extends State
case object ProcessSoldOut extends State
```

The defined states are the same as shown in the previous section. Next we have to create our state data.

```
case class StateData(nrBooksInStore:Int,
                    pendingRequests:Seq[BookRequest])
```

This is the data that we use when we need a State condition to decide which transition is fired. So it contains all the pending requests and the number of books in store. In our case we have one class which contains the StateData (which is used in all states) but this isn't mandatory. It is possible to use a trait for the StateData as well. And create different StateData classes that extend the basic state trait. The first step in implementing the FSM is we define the initial state and the initial StateData. This is done using the startWith method

```
class Inventory() extends Actor with FSM[State, StateData] {
  startWith(WaitForRequests, new StateData(0,Seq()))
  ...
}
```

Here we define that our FSM starts in the state WaitForRequests and the stateData is empty. Next we have to implement all the different state transitions. These state transitions only occurs when there is an event. And in the FSM trait we define for each state which events we expect and what the next state will be. By

defining the next state we have designated a transition. So we start with the Events of the state `WaitForRequests`. In the next section, we will define the actual transitions and see how we go from plan to working code.

DEFINING THE TRANSITIONS

Lets look at Figure 11.4 where we have our state and the two possible transitions.

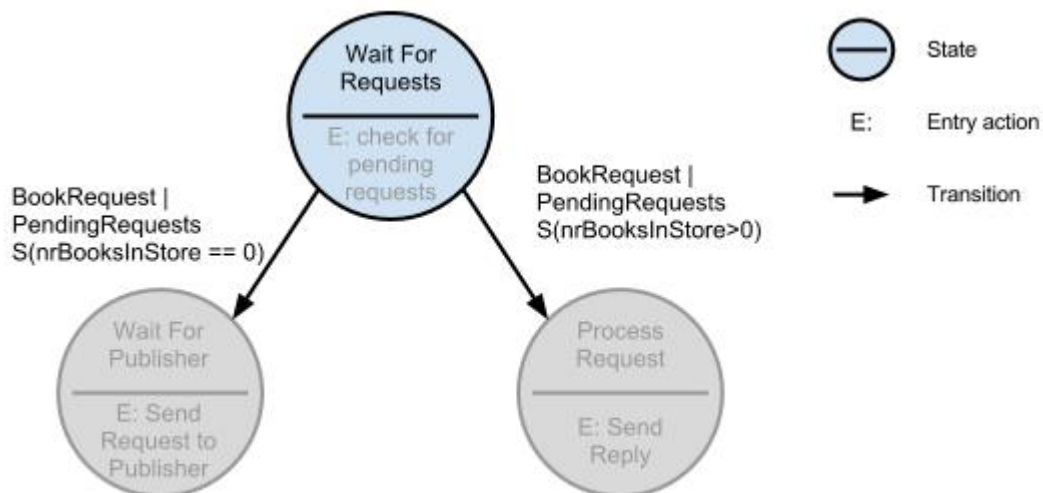


Figure 11.4 State transitions of state `Wait for Requests`

We see that we can expect two possible events. The `BookRequest` or the `PendingRequests` message. And depending on the state `nrBooksInStore` the state changes to `ProcessRequest` or `WaitForPublisher`, which are the transitions. We need to implement these transitions in our `Inventory` FSM. We do that with the "when" declaration.

```

class Inventory() extends Actor with FSM[State, StateData] {
  startWith(WaitForRequests, new StateData(0,Seq()))

  when(WaitForRequests) { ❶
    case Event(request:BookRequest, data:StateData) => { ❷
      .....
    }
    case Event(PendingRequests, data:StateData) => { ❸
      ...
    }
  }
  ...
}
  
```

❶ Declare the transitions for state `WaitForRequests`

- ② Declare the possible Event when a BookRequest messages occur
- ③ Declare the possible Event when a PendingRequests messages occur

We start with the "when" declaration for the WaitForRequests state. Which is a partial function to handle all the possible Events, in the specified State. In our case we can have two different Events. When we are in the WaitForRequests state, a new BookRequest or a PendingRequests message can arrive. Next we have to implement the transition.

Either we are going to remain in the same state or we are going to transition to another one. This can be indicated by the following two methods

```
goto(WaitForPublisher) ①
stay ②
```

- ① Declare that the next state is WaitForPublisher
- ② Declare that the state doesn't change

Another responsibility of this transition declaration is updating the StateData. For example when we receive a new BookRequest Event we need to store the request in our PendingRequests. This is done by the "using" declaration. When we implement the complete transition declaration for the WaitForRequests State we get the following

```
when(WaitForRequests) {
  case Event(request:BookRequest, data:StateData) => {
    val newStateData = data.copy( ①
      pendingRequests = data.pendingRequests :+ request)
    if (newStateData.nrBooksInStore > 0) {
      goto(ProcessRequest) using newStateData ②
    } else {
      goto(WaitForPublisher) using newStateData
    }
  }
  case Event(PendingRequests, data:StateData) => {
    if (data.pendingRequests.isEmpty) {
      stay ③
    } else if (data.nrBooksInStore > 0) {
      goto(ProcessRequest) ④
    } else {
      goto(WaitForPublisher)
    }
  }
}
```

- ❶ Create a new state, by appending the new request
- ❷ declare the next state and update the stateData
- ❸ Use the stay when there are not any pending requests.
- ❹ Use the goto without updating the stateData

In this example, we used the stay without updating the stateData, but it is possible to update the state with "using" too, just like the goto declaration. This is all we have to do to declare the transitions of our first state. The next step is to implement the transitions for all our states. When we examine the possible Events more closely, we see that the Event BookRequest in most states has the same effect: we generally want to just add the request to our pending requests and do nothing else. For these events, we can declare the "whenUnhandled." This Partial function is called when the state partial function doesn't handle the event. Here we can implement the default behaviour when a BookRequest is received. The same declarations can be used as we did in the "when" declaration.

```
whenUnhandled {
  // common code for all states
  case Event(request:BookRequest, data:StateData) => {
    stay using data.copy(
      pendingRequests = data.pendingRequests :+ request)
  }
  case Event(e, s) => {
    log.warning("received unhandled request {} in state {}/{}",
      e, stateName, s)
    stay
  }
}
```

❶ Only update the stateData

❷ Log when the event isn't handled

In this partial function we can also log unhandled events, which can be helpful with debugging this FSM implementation. Now we can implement the rest of the states.

Listing 11.1 Implementation of the transition of the other states

```

when(WaitForPublisher) {
  case Event(supply:BookSupply, data:StateData) => {
    goto(ProcessRequest) using data.copy(
      nrBooksInStore = supply.nrBooks)
  }
  case Event(BookSupplySoldOut, _) => {
    goto(ProcessSoldOut)
  }
}
when(ProcessRequest) {
  case Event(Done, data:StateData) => {
    goto(WaitForRequests) using data.copy(
      nrBooksInStore = data.nrBooksInStore - 1,
      pendingRequests = data.pendingRequests.tail)
  }
}
when(SoldOut) {
  case Event(request:BookRequest, data:StateData) => {
    goto(ProcessSoldOut) using new StateData(0,Seq(request))
  }
}
when(ProcessSoldOut) {
  case Event(Done, data:StateData) => {
    goto(SoldOut) using new StateData(0,Seq())
  }
}

```

1 The transition declaration of the state **WaitForPublisher**

2 The transition declaration of the state **ProcessRequest**

3 The transition declaration of the state **SoldOut**

4 The transition declaration of the state **ProcessSoldOut**

Now we have defined all our transitions for every possible state. This was the first step in creating an Akka FSM Actor. At this moment, we have an FSM and react to events and change state but the actual functionality of the model, the entry actions, are not implemented yet. This is covered in the next section.

11.2.2 Implementing the entry actions

The actual functionality is done by the entry and exit actions. At this point we are going to implement these actions. In our FSM model, we had defined several entry actions. Just as declaring the transitions for each state, the actions are also implemented for each state. In Figure 11.5 we show the initial state `WaitForRequests` again, to see the Entry action we have to implement. The discreet structure of the implementation code, as we will see, also lends itself to unit testing.

ACTIONS ON TRANSITIONS

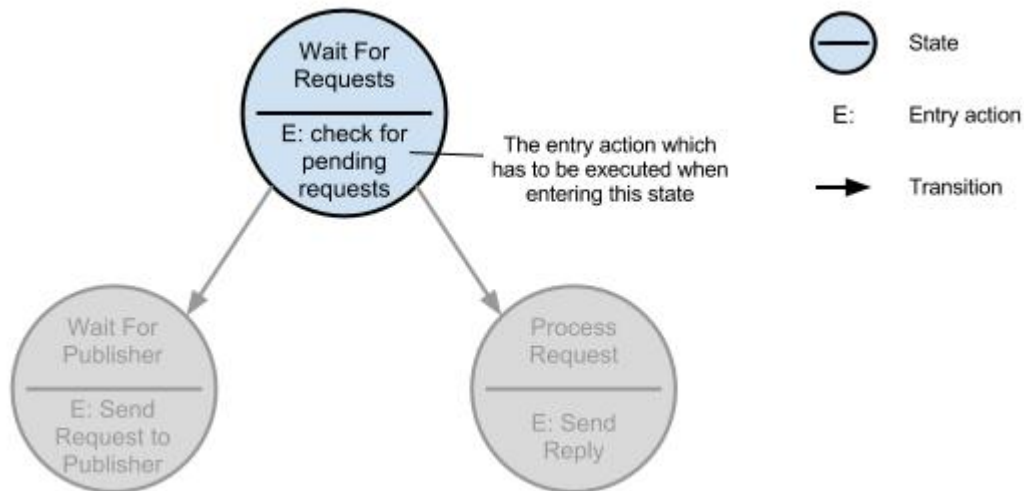


Figure 11.5 The entry Action of the WaitForRequests state

The entry action can be implemented in the `onTransition` declaration. It is possible to declare every possible transition because the transition callback is also a partial function and takes as input the current and the next state.

```

onTransition {
  case WaitForRequests -> WaitForPublisher => {
    ...
  }
}
  
```

In this example we defined the action which has to be executed when the transition occurs from 'WaitForRequests' to 'WaitForPublisher.' But it is also possible to use wild cards. In our example, we don't care which state we are coming from, so we use the wild card on the original state. When implementing the action, one would probably need the `stateData` because this is called when a transition occurs, both the before state and the state after the transition are available and can be used. The new state is available via the variable `nextStateData` and the old `stateData` is available via the variable `stateData`. In our example, we only use the newly created state, because we have only entry actions and our state always contains the complete state. In listing 11.2 we implement all the Entry actions of our FSM.

Listing 11.2 Implementation of the entry actions

```
class Inventory(publisher:ActorRef) extends Actor
  with FSM[State, StateData] {

  startWith(WaitForRequests, new StateData(0,Seq()))

  when...

  onTransition {
    case _ -> WaitForRequests => {
      if (!nextStateData.pendingRequests.isEmpty) {
        // go to next state
        self ! PendingRequests
      }
    }
    case _ -> WaitForPublisher => {
      publisher ! PublisherRequest
    }
    case _ -> ProcessRequest => {
      val request = nextStateData.pendingRequests.head
      reserveId += 1
      request.target !
        new BookReply(request.context, Right(reserveId))
      self ! Done
    }
    case _ -> ProcessSoldOut => {
      nextStateData.pendingRequests.foreach(request => {
        request.target !
          new BookReply(request.context, Left("SoldOut"))
      })
      self ! Done
    }
  }
}
```

- 1 The entry action to check for pending requests
- 2 The entry action to send request to publisher
- 3 The entry action to send a reply to the sender and signal that the processing is done
- 4 The entry action to send a error reply to all the PendingRequests and signal that the processing is done

If you look closely you see that we don't have a declaration for the state `SoldOut` and that is because that state doesn't have an entry action. Now that we have defined our complete FSM, we need to call one important method "initialize". This method is needed to initialize and startup the FSM.

```
class Inventory(publisher:ActorRef) extends Actor
  with FSM[State, StateData] {

  startWith(WaitForRequests, new StateData(0,Seq()))

  when...

  onTransition...
```

```

    initialize
  }

```

The FSM is ready, all we need is a mock up implementation for the publisher and we can test our FSM. This implementation will supply a predefined number of books. And when all the books are gone, the SoldOut reply is sent.

TESTING THE FSM

```

class Publisher(totalNrBooks: Int, nrBooksPerRequest: Int)
  extends Actor {

  var nrLeft = totalNrBooks
  def receive = {
    case PublisherRequest => {
      if (nrLeft == 0)
        sender ! BookSupplySoldOut
      else {
        val supply = min(nrBooksPerRequest, nrLeft)
        nrLeft -= supply
        sender ! new BookSupply(supply)
      }
    }
  }
}

```

1 No more books left

2 Supply a number of books

Now we are ready to test the FSM. We can test the FSM by just sending messages and checking if we get the expected result. But while debugging this component, there is additional available information. Akka's FSM has another helpful feature. It is possible to subscribe to the state changes of the FSM. This can prove useful in programming the application functionality, but it can also be very helpful when testing. It will allow you to closely check if all the expected states were encountered. And if all transitions occur at the correct time. To subscribe to the transition Event, all you have to do is to send a 'SubscribeTransitionCallBack' message to the FSM. In our test, we want to collect these transition events within a testprobe.

```

val publisher = system.actorOf(Props(new Publisher(2)))
val inventory = system.actorOf(Props(new Inventory(publisher)))
val stateProbe = TestProbe()

```

1 First we create the publisher when creating the inventory actor

3 The probe is


```
inventory ! new SubscribeTransitionCallBack(stateProbe.ref)
stateProbe.expectMsg(new CurrentState(inventory, WaitForRequests))
```

4 The process is subscribed to get a transition notifications

When subscribing to an FSM the request, the FSM responds with a `CurrentState` message. Our FSM starts in the `WaitForRequests` just as we expected. Now that we are subscribed to the transitions, we can send a `BookRequest` and see what happens

```
inventory ! new BookRequest("context1", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))
stateProbe.expectMsg(
  new Transition(inventory, WaitForPublisher, ProcessRequest))
stateProbe.expectMsg(
  new Transition(inventory, ProcessRequest, WaitForRequests))
replyProbe.expectMsg(new BookReply("context1", Right(1)))
```

1 Sending this message should trigger state actor which will go through 3 states to handle our preliminary book request. Finally, we will get our reply.

2 The previous transition through 3 states to handle our preliminary book request.

3 Finally, we will get our reply.

As you see the FSM goes through different states before sending a reply. First it has to get books from the publisher. The next step is to actually process the request. And finally the state returns into the 'WaitForRequests' state. But we know that the Inventory got two copies, so when we send another request, the FSM goes through different states than the first time.

```
inventory ! new BookRequest("context2", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, ProcessRequest))
stateProbe.expectMsg(
  new Transition(inventory, ProcessRequest, WaitForRequests))
replyProbe.expectMsg(new BookReply("context2", Right(2)))
```

Because there was a book available, it skipped the 'WaitForPublisher' state. At this point, all the books have been sold so what happens when we send another `BookRequest`?

```
inventory ! new BookRequest("context3", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))
stateProbe.expectMsg(
  new Transition(inventory, WaitForPublisher, ProcessSoldOut))
replyProbe.expectMsg(
  new BookReply("context3", Left("SoldOut")))
stateProbe.expectMsg(
  new Transition(inventory, ProcessSoldOut, SoldOut))
```

1 Each test requires merely that we send the same message

2 Different outcome this time: we are

sold out

Now we get the 'SoldOut' message just as we designed. This is basically the functionality of the FSM, but because a lot of times FSM models use timers to generate events and trigger transitions. Therefore Akka also supports timers within its FSM.

11.2.3 Timers within FSM

As we mentioned earlier, an FSM can model a large number of problems and a lot of solutions for these problems depend on timers. For example to detect an idle connection or a failure because the reply isn't received within a specified time. To demonstrate the use of timers, we are going to change our FSM a little. When it is in the state 'WaitingForPublisher,' we don't wait forever for the publisher to reply. If the publisher fails to respond, we want to send the request again. Figure 11.6 shows the changed FSM.

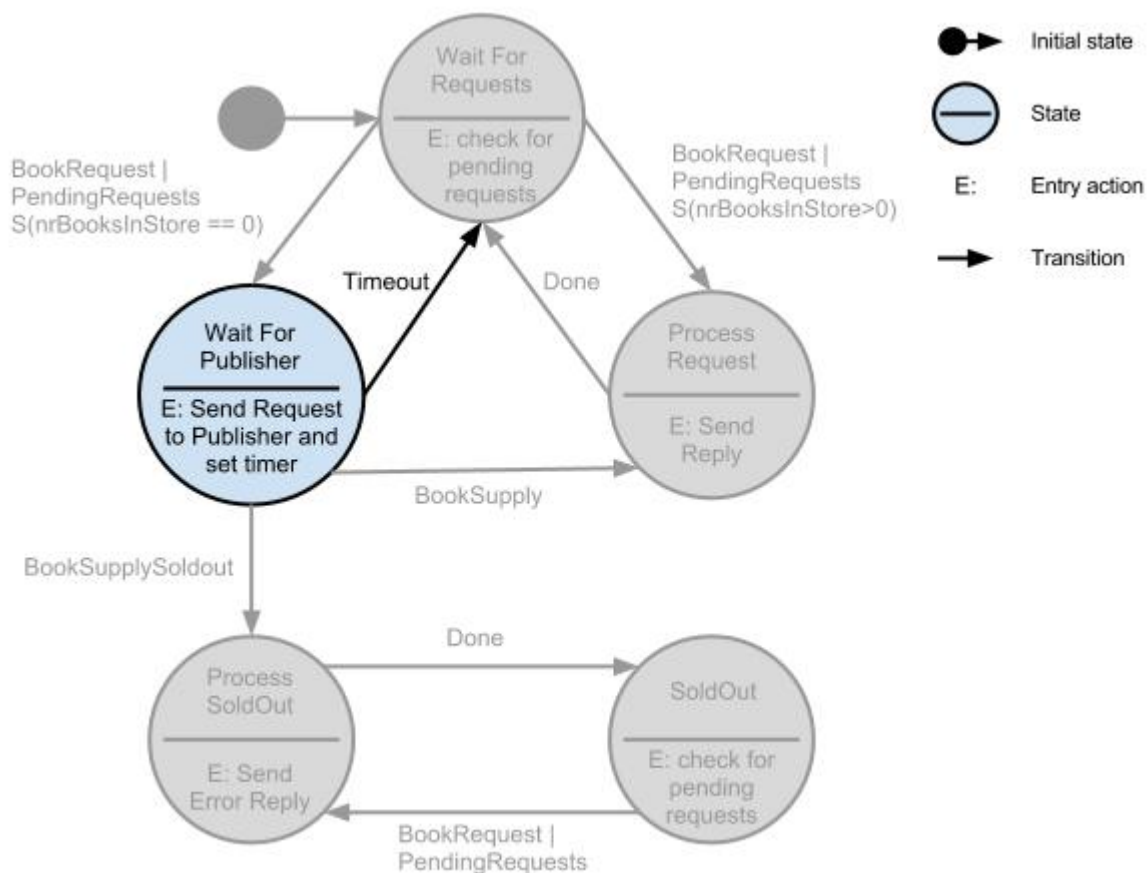


Figure 11.6 FSM using timers

The only change is that a timer is set as part of the entry action and when this timer expires the state changes to the WaitForRequests state. When this happens

the `WaitForRequests` checks if there are `PendingRequests` (and there must be, otherwise the FSM wouldn't have been in the state `'WaitForPublisher'` in the first place). And because there are `PendingRequests`, the FSM goes to `WaitForPublisher` state again. which triggers the entry action again and a message is sent to the publisher.

The changes we need to make here are minor. First, we have to set the timeout. This can be done setting the `stateTimeout` when declaring the state transitions of the `WaitForPublisher` state, and the second change is to define the transition when the timer expires. The changed "when" declaration becomes:

```
when(WaitForPublisher, stateTimeout = 5 seconds) {
  case Event(supply:BookSupply, data:StateData) => {
    goto(ProcessRequest) using data.copy(
      nrBooksInStore = supply.nrBooks)
  }
  case Event(BookSupplySoldOut, _) => {
    goto(ProcessSoldOut)
  }
  case Event(StateTimeout,_) => goto(WaitForRequests)
}
```

1 set the stateTimeout

2 Define the timeout transition

That is all we need to do to be able to retransmit to the publisher using a timer. This timer is canceled upon reception of any other message while in the current state. You can rely on the fact that the `StateTimeout` message will not be processed after an intervening message. Let's see how this works in action by executing the following test in Listing 11.3.

Listing 11.3 Testing Inventory with timers

```

val publisher = TestProbe()
val inventory = system.actorOf(
  Props(new InventoryWithTimer(publisher.ref)))
val stateProbe = TestProbe()
val replyProbe = TestProbe()
inventory ! new SubscribeTransitionCallBack(stateProbe.ref)
stateProbe.expectMsg(
  new CurrentState(inventory, WaitForRequests))
//start test
inventory ! new BookRequest("context1", replyProbe.ref)
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))
publisher.expectMsg(PublisherRequest)
stateProbe.expectMsg(6 seconds,
  new Transition(inventory, WaitForPublisher, WaitForRequests))
stateProbe.expectMsg(
  new Transition(inventory, WaitForRequests, WaitForPublisher))

```

1 Wait more than 5 seconds for this transition

As you can see, when the publisher doesn't respond with a reply, the state changes after 5 seconds to the `WaitForRequests` state. There is another way to set the `stateTimer`. The timer can also be set by specifying the next state using the method `forMax`. For example, when we want to set the `stateTimer` differently, coming from another state. In the next snippet, we see an example how we can use the `forMax` method

```
goto(WaitForPublisher) using (newData) forMax (5 seconds)
```

When using this method, it will overrule the default timer setting specified in the `WaitForPublisher` when declaration. With this method it is also possible to turn off the timer by using `Duration.Inf` as the value in the `forMax` method.

Beside the state timers, there is also support for sending messages using timers within FSM. The usage is not complex and therefore we just need a quick summary of the API. There are three methods to deal with FSM timers. The first one is to create a timer.

```

setTimer(name: String,
  msg: Any,
  timeout: FiniteDuration,
  repeat: Boolean)

```

All the timers are references with their name. With this method we create a timer and define: the name, the message to send when the timer expires, the interval of the timer, and if it is a repeating timer.

The next method is to cancel the timer

```
cancelTimer(name: String)
```

This will cancel the timer immediately and even when the timer has already fired and enqueued the message, the message will not be processed after this `cancelTimer` call. The last method can be used to get the status of the timer at any time

```
isTimerActive(name: String): Boolean
```

This method will return true when the timer is still active. This can be that the timer didn't fire yet or that the timer has the repeat set to true.

11.2.4 Termination of FSM

Sometimes we need to do some cleanup when an Actor finishes. The FSM has a specific handler for these cases: `onTermination`. This handler is also a partial function and takes a `StopEvent` as an argument.

```
StopEvent(reason: Reason, currentState: S, stateData: D)
```

There are three possible reasons this can be received.

- Normal
This is received when there is a normal termination.
- Shutdown
This is received when the FSM is stopped due to a shutdown.
- Failure(cause: Any)
This reason is received when the termination was caused by a failure

A common termination handler would look something like this

```
onTermination {
  case StopEvent(FSM.Normal, state, data) // ...
```

```

case StopEvent(FSM.Shutdown, state, data) // ...
case StopEvent(FSM.Failure(cause), state, data) // ...
}

```

An FSM can be stopped from within the FSM. This can be done using the `stop` method, which takes the reason why the FSM is to be stopped. When the `ActorRef` is used to stop the actor, the shutdown reason is received in the termination handler.

Using the Akka FSM trait gives a complete toolkit to implement any FSM, without much extra effort. There is a clean separation between the actions of a state and the state transitions. The support of timers make it easy to detect idle state or failures. And there is an easy translation from the FSM model to the actual implementation.

In all the examples about state in chapter 7 and in this section, the state is contained within one actor. But what can we do when we need some state amongst multiple actors? In the next section we are going to look at how we can do this using agents.

11.3 Implement Shared state using agents

The best way to deal with state is to use that state only within one actor, but this is not always possible. Sometimes we need to use the same state within different actors and as we mentioned before, using shared state needs some kind of locking. And locking is hard to do correctly. For these situations, Akka has agents, which eliminate the need for locking. An agent guards the shared state and allows multiple threads to get the state and is responsible for updating it on behalf of the various threads. And because the agent does the updating, the threads don't need to know about locking. In this section we are going to describe how these agents are able to guard the state and how we can them to share state. We start by addressing the question what are agents then we show their basic usage. After that, we show extra agent functionality to track state updates.

11.3.1 Simple Shared state with agents

How can the state of the agent be retrieved by using synchronous calls while updates to the state are done asynchronously? Akka accomplishes this by sending actions to the agent for each operation, where the messaging infrastructure will preclude a race condition (by assuring that only one send action is running in a given `ExecutionContext`, at a time). For our example, we need to share the number of copies sold for each book, so we will create an Agent that contains this value.

```

case class BookStatics(val nameBook: String, nrSold: Int)
case class StateBookStatics(val sequence: Long,
                             books: Map[String, BookStatics])

```

The `StateBookStatics` is the state object and it contains a sequence number, which can be used to check for changes and the actual book statistics. For each book, a `BookStatics` instance is created which is put into a map using the title as the key. In Figure 11.7 we show that getting this state from the agent we can use a simple method call.

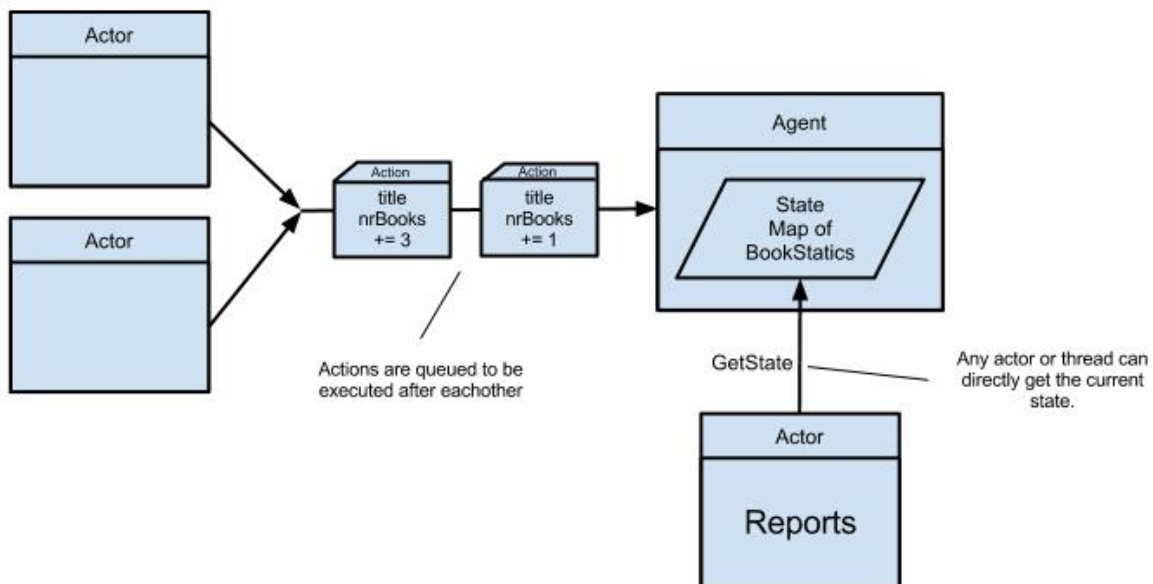


Figure 11.7 Updating and retrieving state using an agent

But when we need to update the number of books, we have to send the update action to the agent. In the example, we show the first update is added with one and the second action is to update the state with 3 copies. These Actions can be sent from different Actors or threads, but are queued like messages sent to actors. And just as messages sent to an Actor, the actions are executed one at the time, which makes locking unnecessary.

To make this work there is one important rule: that all updates to the state are done within the agent's execution context. This means that the state object

contained by the agent must be immutable. In our example, we can't update the content of the map. To be able to change it you need to send an Action to the Agent to change the actual state. Let's see how we are doing this in the code.

We start by creating an Agent. When creating an Agent we have to supply the initial state, in this case, an empty instance of StateBookStatics

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent

val stateAgent = new Agent(new StateBookStatics(0,Map()))
```

When creating the agent we need to provide an implicit ExecutionContext which is used by the agent. We use the global ExecutionContext defined by the `import scala.concurrent.ExecutionContext.Implicits.global`. At this point, the Agent is guarding the state. As we mentioned earlier, the state of the agent can be simply retrieved by using synchronous calls. There are two ways to do that. The first is to call

```
val currentBookStatics = stateAgent()
```

Or, one could use the second method the "get" method, which is doing exactly the same thing.

```
val currentBookStatics = stateAgent.get
```

Both methods return the current state of the BookStatics. So far nothing special, but updating the BookStatics can only be done by asynchronously sending actions to the agent. To update the state, we use the send method of the agent; we send the new state to the Agent.

```
val newState = StateBookStatics(1, Map(book -> bookStat ))
stateAgent send newState
```

But be very careful with sending a complete, new state; this is only correct when the new state is independent of the previous state. In our case, the state depends on the previous state because other threads may have added new numbers

or even other books, before us. So we shouldn't use the method shown. To make sure that when updating the state we end up with the correct state, we invoke a function on the agent instead.

```
val book = "Akka in Action"
val nrSold = 1

stateAgent send( oldState => {
  val bookStat = oldState.books.get(book) match {
    case Some(bookState) =>
      bookState.copy(nrSold = bookState.nrSold + nrSold)
    case None => new BookStatics(book, nrSold)
  }
  oldState.copy(oldState.sequence+1,
    oldState.books + (book -> bookStat ))
})
```

We use the same "send" method, but instead of the new state we send a function. This function is translating the old state into the new state. The function is updating the nrSold attribute with one and when there isn't already a BookStatics present for the book a new object is created. The last step is to update the map.

Because the actions are executed one at any time, we don't need to worry that during this function the state will be changed, and therefore, we don't need a locking mechanism. We have seen how we can get the current state and how we can update the state, this is the basic functionality of an agent. But because the updates are asynchronous it is sometimes necessary to wait for the update to be finished. This functionality is described in the next section.

11.3.2 Waiting for the state update

In some cases, we need to update shared state and use the new state. For example, we need to know which book is selling the most, and when a book becomes popular, we want to notify the authors. To do this, we need to know when our update has been processed before we can check whether the book is the most popular. For this, Agents have the "alter" method, which can be used for updating the state. It works exactly as the send method only it returns a Future, which can be used to wait for the new state.

```
implicit val timeout = Timeout(1000)
val future = stateAgent alter( oldState => {
  val bookStat = oldState.books.get(book) match {
    case Some(bookState) =>
      bookState.copy(nrSold = bookState.nrSold + nrSold)
```

- 1 Since we'll be
- 2 ~~waiting~~ ~~alter~~ ~~give~~ ~~time~~ ~~to~~ ~~wait~~ ~~on~~
- 3 This is where we update the value

```

    case None => new BookStatics(book, nrSold)
  }
  oldState.copy(oldState.sequence+1,
    oldState.books + (book -> bookStat ))
})
val newState = Await.result(future, 1 second)

```

4 Our new state will be returned here when it's available

In this example, we performed the update using a function, but just as was the case with the send method, it is also possible to use the new state within the alter method. As you can see, the changed status is returned within the supplied Future. But this doesn't mean that this is the last update. It is possible that there are still pending changes for this state. We know that our change is processed and that the result of this change is returned, but it is possible that there are multiple changes at nearly the same time and we want the final state or another thread needs the final state and only knows that the process before it may have updated the state. So this thread doesn't have any reference from the alter method; it needs to wait. The Agent provides us a "Future" for this. This future finishes when the pending state changes are all processed.

```

val future = stateAgent.future
val newState = Await.result(future, 1 second)

```

This way, we can be sure of the latest state at this moment. In the next section, we show that agents also can be used with monadic notation. This enables you to create simple and powerful constructions. But keep in mind that new agents are created when using monads, leaving the original agents untouched. They are called 'persistent' for this reason. An example with a map shows the creation of a new Agent.

```

import scala.concurrent.ExecutionContext.Implicits.global
val agent1 = Agent(3)

val agent2 = agent1 map (_ + 1)

```

When using this notation, agent2 is a newly created Agent that contains the value 4 and agent1 is just the same as before (it still contains the value 3).

We showed that when shared state was needed, we could use the Agents to manage the state. The consistency of the state is guaranteed by allowing updates

only be done in the Agents context. These updates are triggered by sending actions to the agent.

11.4 Summary

Clearly, writing applications that never hold state is an unattainable goal. In this chapter, we saw several approaches to state management that Akka provides. The key takeaways are:

- FSMs, which can seem specialized and perhaps daunting, are pretty easy to implement with Akka and the resulting code is clean and maintainable, their implementation as a trait results in code where the actions are separated from the code that defines the transitions
- Agents provide another means of state that's especially useful when several Actors need access.
- Both techniques, FSMs and agents, allow us to employ some shared state without falling back into having to manage locks.
- Timers, in the FSMs, and the use of Futures with Agents, provide a level of orchestration in implementing state changes.

This chapter took us through examples that showed the implementation of complex, dependent interactions modifying shared state. We accomplished this while still keeping the spirit of our stateless, messaging principles in tact, by using mechanisms that allow us to coordinate multiple actors around a set of shared states. Typically, when we make systems like these, we will want all actions to succeed or all to fail. In other words we need transactions. In the next chapter, we describe how we can use transactions within the Akka toolkit.

12

Working with Transactions

In this chapter

- Transactions
- ACID
- Software Transactional Memory
- Optimistic locking
- Transactor's

In chapter 3, we described the "let it crash" philosophy and the supervisor model for implementing it with Actors. This works well, but when an actor has to execute several tasks to service one request, this can result in partially processed requests. Let's take the classic example of transferring money from one account to another. To do this, two actions have to be taken. The first is to withdraw the amount from the first account and the second action is to deposit the amount in the other account. When a failure occurs after the withdrawal has succeeded, but before the deposit has finished, our "let it crash" approach will not result in correctly restored state. To solve this we need to make sure that both actions succeed or that they both fail. Typically, transactions are used for such problems. The transaction makes a group of actions act like one action. This is done by addressing assuring the following properties:

- atomic
This means that all the actions are successful or none of them. In our example, the withdrawal and the deposit must fail. When one of the actions fails, the result of the all prior actions must be reversed. (This is called a "rollback.")
- consistent
After a transaction finishes, the system must be left in a consistent state. So when transferring money, the sum of the amounts on both accounts should be the same before

and after the transaction.

- isolated

Any changes done before the transaction has succeeded or failed must be invisible to other users of the system who might be attempting to implement similar functionality.^β

- durable

Durability means that the system cannot fail to honor the requirements of the transaction because a fault, like a power outage, occurred: it must be resilient.

These properties are often referenced by the acronym ACID. The primary goal of a transaction is to keep the shared state consistent. We already discussed some of these same issues earlier, and how Akka addresses them, but multiple state transformations require additional capabilities.

Akka agents and actors can be used within transactional systems. To be able to accomplish this, Akka uses Software Transactional Memory (STM). STM implements the first three properties of a transaction (Atomic, Consistent, and Isolated of ACID), but with better performance than traditional locking solutions. We start this chapter by describing the basics of STM and how it achieves superior performance. Once we know the basics, we describe how we can use the Akka agents within STM transactions. And finally, we show two different approaches to using STM transactions with actors. The first approach is to distribute the transaction over multiple actors. The second approach is to use transactors. These transactors are special actors which implement the use of the distributed transaction for you so you need only implement the functionality. Let's get started with the basics of STM.

12.1 Software Transactional Memory

To explain what the Software Transactional Memory model is, we need to forget for a moment that we have Akka. In this section we want to share memory between multiple threads. Let's go to our Kiosk example from chapter 1. We have an event and we have a number of seats that multiple threads want to lay claim to. Our shared data is a list of seats

```
case class Seat(seatNumber: Int)

val availableSeats: Seq[Seat]
```

For this example our seat has only a seat number. When we want to get a seat from the list we need to get the first available seat and update the list.

```
val head = availableSeats.head
availableSeats = availableSeats.tail
```

But when we have multiple threads and they execute at the same time, those threads might claim the first seat, if the second thread looks before the first has updated the list. And when this happens, the seat is sold twice. We need some way to prevent one seat from being sold multiple times.

We have seen already a solution in chapter 1, which is to use immutable messages. But what can we do, when we can't or want to use immutable messages and just want to protect the shared data from becoming inconsistent. In this section, we describe how STM protects shared data, which is implemented differently than the traditional locking mechanism (and also provides better performance). After we describe how STM works, we will describe the common functionality of STM to be able to create transactions.

12.1.1 *Protecting shared data*

The most common solution to protecting shared data is that when a thread wants to access the shared data, we block all other threads from accessing the shared structure. This is called locking. Here's a classic example: `synchronized` assures only one thread at a time into the section of code that updates the `reservedSeat` value:

```
val reservedSeat = availableSeats.synchronized {
  head = availableSeats.head
  availableSeats = availableSeats.tail
  head
}
```

Synchronization is the most common way to prevent others from entering 'the critical section' while we are in process on our own changes. Before the `synchronized` block is executed, the system has to obtain a lock. Figure 12.1 shows how this is done. Thread 1 tries to read the list of seats and then update it, while another competing thread is trying to do the same.

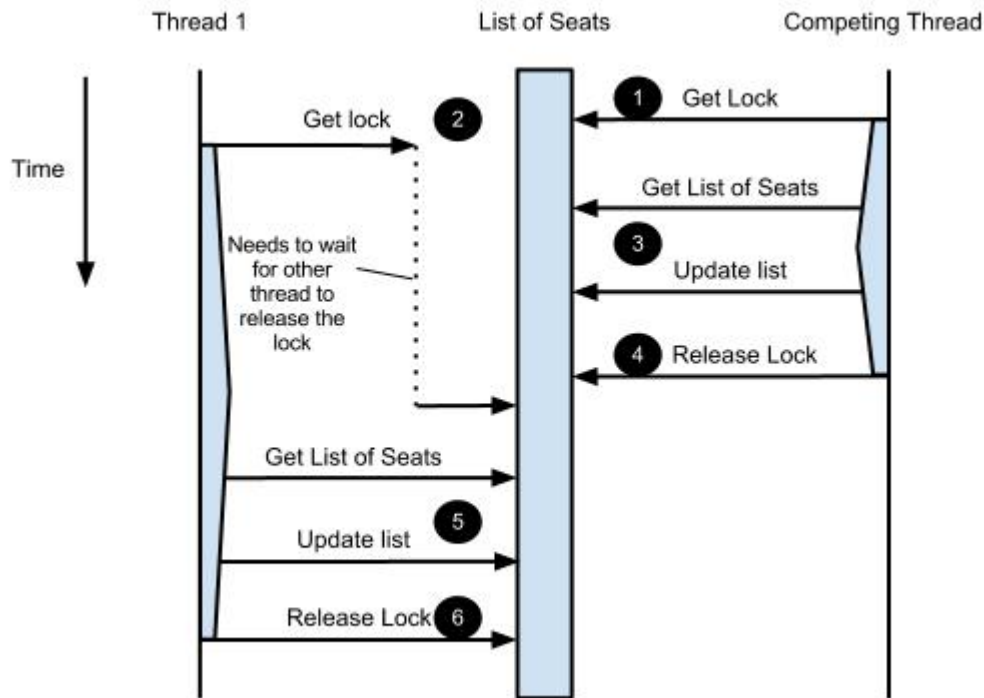


Figure 12.1 Locking shared data to keep data consistency

When another competing thread has a lock already, the current thread has to wait until the lock is released. After the lock is obtained, the critical section can be executed and when it's done, the lock is released. Thus, the critical section cannot be entered by more than one thread at a time.

A problem with this is that when a thread only wants to read all available seats it still has to lock the list too. All this locking decreases the performance of the system. And most of the time, the locking is done even when there isn't any other thread trying to access the shared data. This is called "pessimistic locking." We assume that we are going to access the shared data simultaneously. The word "synchronized" gives you the impression that multiple threads need to synchronize their actions to avoid problems, but as we stated, most of the time we don't have the data update collisions.

Clearly, since there is 'pessimistic locking,' there must also be 'optimistic locking.' As the name would imply, this approach assumes that there is no problem with accessing the shared data, so we will just execute the code without any locking. But before leaving the critical section, the system checks for possible update collisions (this is shown in Figure 12.2). If there were no collisions, we simply commit the transaction. When a collision is detected, the changes are discarded (rolled back) and the critical section is retried.

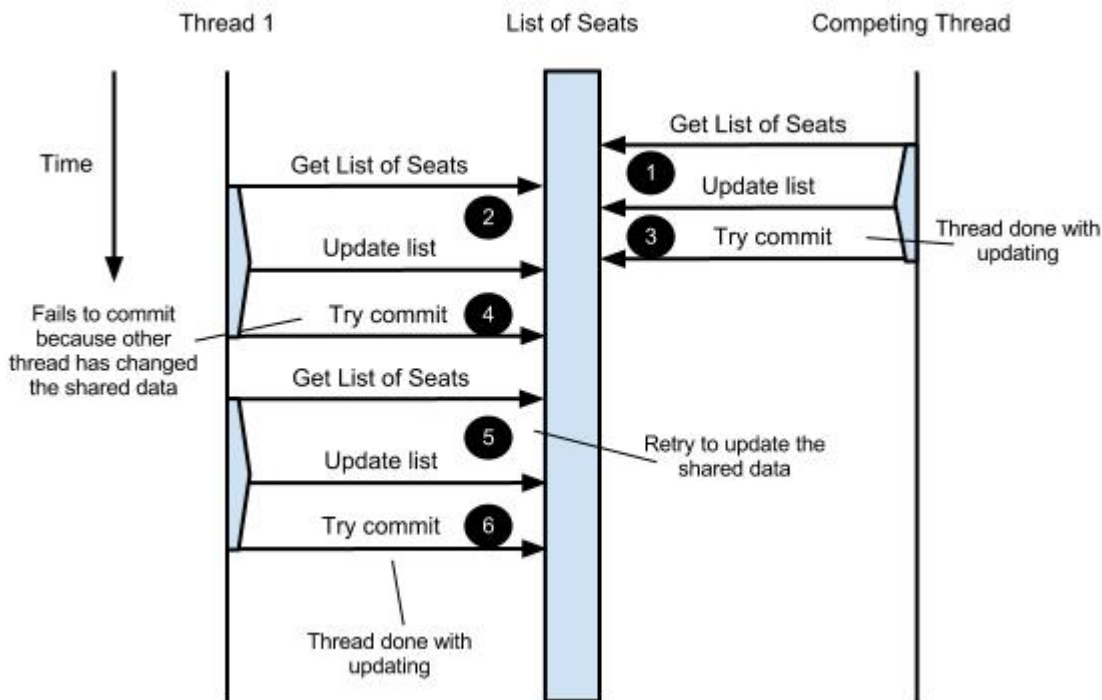


Figure 12.2 Optimistic Locking used by STM to keep data consistency

This is how STM works: by using optimistic locking. It prevents inconsistency of the shared data caused by multiple threads accessing that data. The key to this process is being able to know if the shared data was changed during the transaction. To detect this, we will wrap all the shared data in an STM Ref class.

```
import concurrent.stm.Ref

val availableSeats = Ref(Seq[Seat]())
```

To get or set the value of the shared data we can simply write `availableSeats()`. For example to update our `availableSeats` we can write:

```
availableSeats() = availableSeats().tail
```

The value of a Ref can only be accessed within an atomic block. An atomic block defines a group of code to be executed as one atomic command. The check in the Ref is used within an atomic block is checked at compile time by requiring that an implicit `InTxn` value be available.

When we want to protect the seat list, we get the following (similar to the

synchronized example):

```
import concurrent.stm._

val availableSeats = Ref(seats)

val reservedSeat = atomic {implicit txn => {
  val head = availableSeats().head
  availableSeats() = availableSeats().tail
  head
}}
```

- 1 Available seats
- 2 We get the last item that was at head

This code sample is functionally doing the same as the synchronized example but the locking mechanism works completely differently. And because we use "optimistic" locking there is a small but important difference: the critical section will be executed only once when using synchronized, but using the STM atomic, the critical section can be executed more than once. This is because at the end of the block's execution, a check is done to see if there was a collision. Let's see this collision check in action. We start by defining our shared data: a list of seats.

```
val seats = (for (i <- 0 until 15) yield Seat(i))
val availableSeats = Ref(seats)
```

To demonstrate a data write collision we need multiple threads. In this example we create a future which removes 10 seats with a waiting time of 50 milliseconds between each one.

```
implicit val ec = ExecutionContext.global
val f = Future {
  for (i <- 0 until 10) {
    atomic { implicit txn => {
      availableSeats() = availableSeats().tail
    }}
    Thread.sleep(50)
  }
}
```

- 1 Create transaction to be able to update the availableSeats

Now that we have a thread to update the shared list of seats, we start to create a critical section which takes a lot longer to finish than the competing thread. normally in our code, we wouldn't want to add a counter of any sort, but we'll do so here with `nrRuns` to show you how many times the critical section is executed. We

have added a `nrRuns` to be able to count how many times the critical section is executed.

```
var nrRuns = 0
val mySeat = atomic { implicit txn => {
  nrRuns += 1
  val reservedSeat = availableSeats().head
  Thread.sleep(100)
  availableSeats() = availableSeats().tail
  reservedSeat
}}
```

- 1 Update the counter we added to see
- 2 ~~the delay allows~~ the competing thread to update, causing a collision.

The collision triggers the STM to rollback and retry the critical section. This will continue until the competing thread has finished getting the first 10 seats (0 to 9) and doing this in 500ms. So we would expect that the critical section will reserve seat number 10 (next available seat) after running 6 times, because the first 5 runs the competing thread updates the availableSeats before the main thread was able to complete its transaction.

```
Await.ready(f, 2 seconds)
nrRuns must be (6)
mySeat.seatNumber must be (10)
```

When running this test, we see that the `nrRuns` are indeed 6 and the seat number is 10. This multiple execution of the critical section is the most important difference for a developer when using "optimistic" locking instead of the traditional "pessimistic" locking. In our example, we used a `var nrRuns` and updated it in our atomic block. In normal code you should never do this, we did this to show that it is updated multiple times. In normal code you don't want or need to know how many times the section is executed only that the result is committed once. So once again, you never update variables or classes which are defined outside the atomic code block, because it is possible that it is updated more than once.

We stated that the collision check was done at the end of the atomic block, but for performance optimization, the check is done every time the reference is used. Therefore it is also possible that the atomic block isn't executed completely, but only the first half. When it detects a collision, the rollback is done immediately and doesn't waste time with executing lines that are going to be rolled back anyway. This is a second reason to not update classes or variables defined outside the

atomic block, because you never know which part will be retried and which part will be executed only once.

This is the basic functionality of STM. STM is able to implement the first part of the ACID properties by using "optimistic" locking. The first three properties Atomic, Consistent and Isolated are all addressed by STM. Only the last property "Durable" doesn't apply to STM. This is because STM is all about in memory transactions and in memory is never durable.

12.1.2 Using the STM transactions

We have described the basic functionality of STM, but there is more. As we stated the only way to reference the shared state is within an atomic block. But when we want to do a simple read of shared data we need to create an atomic block and this means writing a lot of code just for a single, simple read. When using only one reference, you could also use the View of a reference. The Ref.View enables you to execute one action on one Reference. By using the Ref.View, you improve the performance and minimize the code. This view can be retrieved by using the "single" method. This View supports several functions, which can be applied to the shared data of the Ref. To get the value of the View we use the get method. To get all current available seats we can write the following:

```
availableSeats.single.get
```

This code doesn't need to be in an atomic block. This View can also be used to update the value and has several methods to do that. These View updates can be used instead of an atomic block when an the atomic block only accesses one single Ref. And it might be more concise and more efficient to use a Ref.View in those situations. In our example we use only one Ref (the availableSeats), so to improve performance it was better to use the Ref.View. But to be able to show how STM works, we used atomic instead of the Ref.View. Here's what the code looks like when rewritten to use the View:

```
val mySeat = atomic {implicit txn => {
  val head = availableSeats().head
  availableSeats() = availableSeats().tail
  head
}}
}
```

1 Using the atomic block

```
val myseat = availableSeats.single.getAndTransform(_.tail).head
```

2 Using the Ref.View

Using the Ref.View method makes the code a little bit more compact and also makes the critical section smaller, which decreases the chance of a collision, improving the total performance of the system.

Now we know how we can read and update shared data, but what about when we run into trouble? For example, we didn't check if the available seats is empty. What happens when there is an exception within an atomic block? The code will do what you expect: it will roll back all changes before handling it. In listing 12.1 we show that the exception causes the atomic block to rollback. We do this by modifying our example to attempt to get two seats when there is only one left.

Listing 12.1 Example of Rollback when an exception occurs

```
val availableSeats = Ref(Seq(Seat(1)))
evaluating {
  val mySeat = atomic { implicit txn => {
    var reservedSeats = Seq[Seat]()
    reservedSeats = reservedSeats :+ availableSeats().head
    availableSeats() = availableSeats().tail
    Thread.sleep(100)
    reservedSeats = reservedSeats :+ availableSeats().head
    availableSeats() = availableSeats().tail
    reservedSeats
  }}
} must produce [Exception]
availableSeats.single.get.size must be (1)
```

1 Generate the exception

2 List is still containing one seat

As you see, the exception caused the transaction to be rolled back and the list is unchanged. But it is also possible to check conditions and indicate that the code reached a dead end. For example, when you have a pool of resources (e.g. database connections), and in the atomic block you need one, but all of them are in use, you end up at a dead end, until one of the other threads releases a connection. For cases like this, you can use the retry method inside the atomic block.

```
val pool = Ref(connectionPool)
atomic { implicit txn => {
  if(pool().size == 0)
    retry
  ...
}}
```

When you call `retry` all the changes are rolled back and the critical section is retried, even when all the references are unchanged. But does this mean it retries to execute the critical section over and over again until the conditions are changed? No, STM keeps track which references are used and only when one of them has been changed, the next `retry` is executed. Internally the `retry` is implemented using blocking constructs, so there is no busy-waiting when a `retry` is called. So it has a better performance than when you try to solve this within the atomic block.

The last functionality of STM we want to describe here are alternative atomic blocks. For example when all the seats have been sold, we can wait all we want, a seat will never become available. For this, we can create an alternative atomic block. This is done with the `orAtomic` method. When we take our seat example it would look like listing 12.2.

Listing 12.2 Using alternative atomic block to implement a stop criteria

```
val availableSeats = Ref(Seq[Seat]())
val mySeat = atomic { implicit txn => {
  val allSeats = availableSeats()
  if (allSeats.isEmpty)
    retry
  val reservedSeat = allSeats.head
  availableSeats() = allSeats.tail
  Some(reservedSeat)
}}.orAtomic {implicit txn => {
  //else give up and do nothing
  None
}}
mySeat must be (None)
```

1 Call `retry` when list is empty

2 Define an alternative section

3 give up and return `None`

When the `availableSeat` list is empty, we call the `retry`, which triggers to execute the alternative atomic block. In this block we do nothing but return a `None` to indicate we were unable to get a seat.

The alternative block isn't limited by one. You can use as many blocks you need. These alternative blocks can be chained to implement the alternative solutions to the problem. If the first alternative calls `retry` then the second will be tried, if the second calls `retry` then the third will be tried, and so on. Keep in mind that the alternative block is only tried when a `retry` is called, not when the transactions reads are inconsistent.

We have shown how we can create shared data and that we only reference them within an atomic block, or when we can use the `Ref.View`. This option gives us

better performance, but can only be used when we have one reference within an atomic block. Optimistic locking outperforms locking because when there was no collision, we paid no price, and even when there was, the price paid was slight: a retry. That retry mechanism can also prove useful when we have additional reasons to execute the critical section again.

There is more to STM, but we have covered the most common functionality. The only disadvantage might be that you need to consider that the atomic block can be executed more than once and that all the shared data needs to be wrapped into STM references, but the way to look at this is it's consistent with the one of the most important tenets of Akka's thinking: you must prepare for failures.

Now that we know how STM works we can go back to how Akka makes it possible to implement STM.

12.2 Agents within transactions

Let's look at how we can take the Akka Agent from section 10.3 and have it participate within a transaction. We make a distinction between reading and updating the shared data. This is to provide additional clarity with regards to how the Agent behaves within the transaction, because it might not work as you expect at first sight. We start with reading within a transaction.

12.2.1 Reading from Agents within a transaction

When an Agent is used within a transaction, it isn't necessary to wrap it with an STM reference to be able to use its data. Let's look again at the seat example from the previous section, but we will make a small change because we don't want to do the update in the transaction (yet).

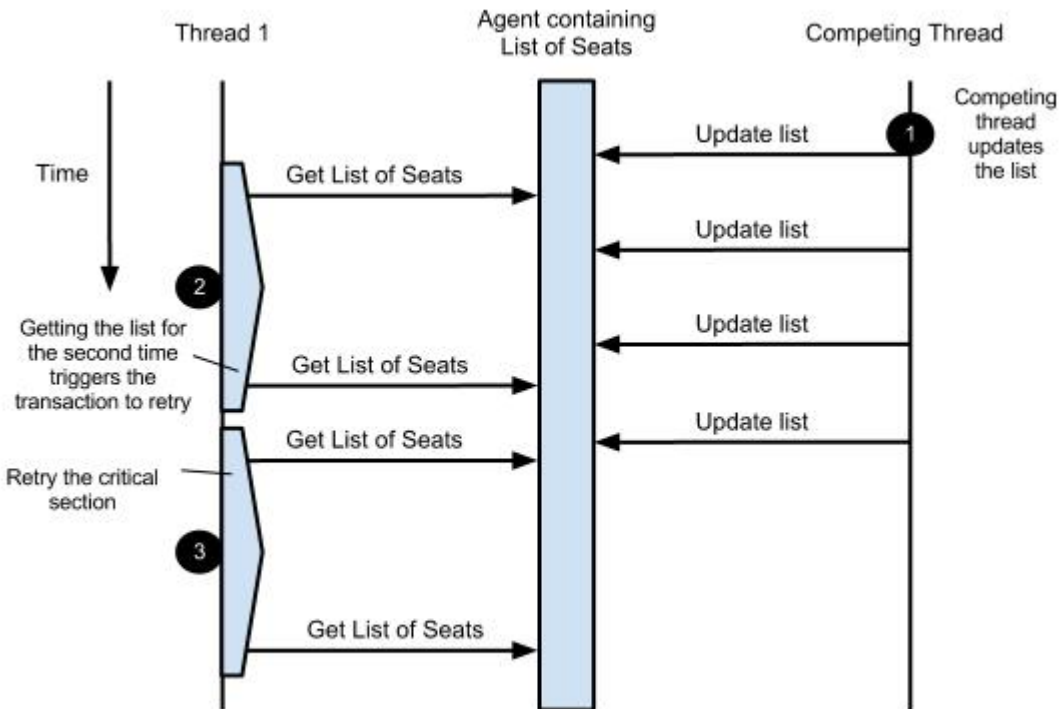


Figure 12.3 Example reading an Agent's state within a transaction

Our competing thread will update the agent every 50 ms and our test thread tries to read the Agent's state twice within a transaction. When the Agent's state has changed in the meantime, the transaction has to be retried. This happens as long as the competing thread is updating the agent. So when we rewrite our seat example using an agent we get listing 12.3.

Listing 12.3 Competing thread updating the agent

```
val seats = (for (i <- 0 until 15) yield Seat(i))
val availableSeats = Agent(seats)
val future = Future {
  for (i <- 0 until 10) {
    availableSeats send (_.tail)
  }
  Thread.sleep(50)
}
```

We create an Agent with a list of available seats and a competing thread which updates the list by removing 10 seats from it. This will cause the example transaction to retry. In our example transaction shown in listing 12.4 we do two

reads of the agent to detecting the change of the shared data while waiting for 100ms to simulate doing some processing on the data.

Listing 12.4 Reading an agents content withing an atomic block

```
var nrRuns = 0
val firstSeat = atomic { implicit txn => {
  nrRuns += 1
  val currentList = availableSeats.get ❶
  Thread.sleep(100) ❷
  availableSeats.get.head ❸
}}
Await.ready(future, 1 second)
nrRuns must be > (1)
firstSeat.seatNumber must be (10)
```

- ❶ Do the first read of the agent
- ❷ Simulate some processing with the availableSeat list
- ❸ Trigger the check if the availableSeat list has changed

In this example we see that the critical section is executed more than once, because the value of the agent has changed during the transaction. And the first available seat, which is returned is the expected seat with number 10. So far the Agent behaves just as the STM reference version does, but the difference is in the update.

12.2.2 Updating Agents within a transaction

In chapter 10.3 we saw that updating the state of an Agent is done by sending actions to it. We did that also in the previous example using the competing thread. But when we use agents within a transaction, the update of an Agent is done differently than we might expect. The actual update isn't done within the transaction, but by sending the action as part of the transaction. This means that when we send an action, the action is held until the transaction is committed and when if the transaction is rolled back, the action sent to the Agent is also rolled back. Let's look at an example that demonstrates this.

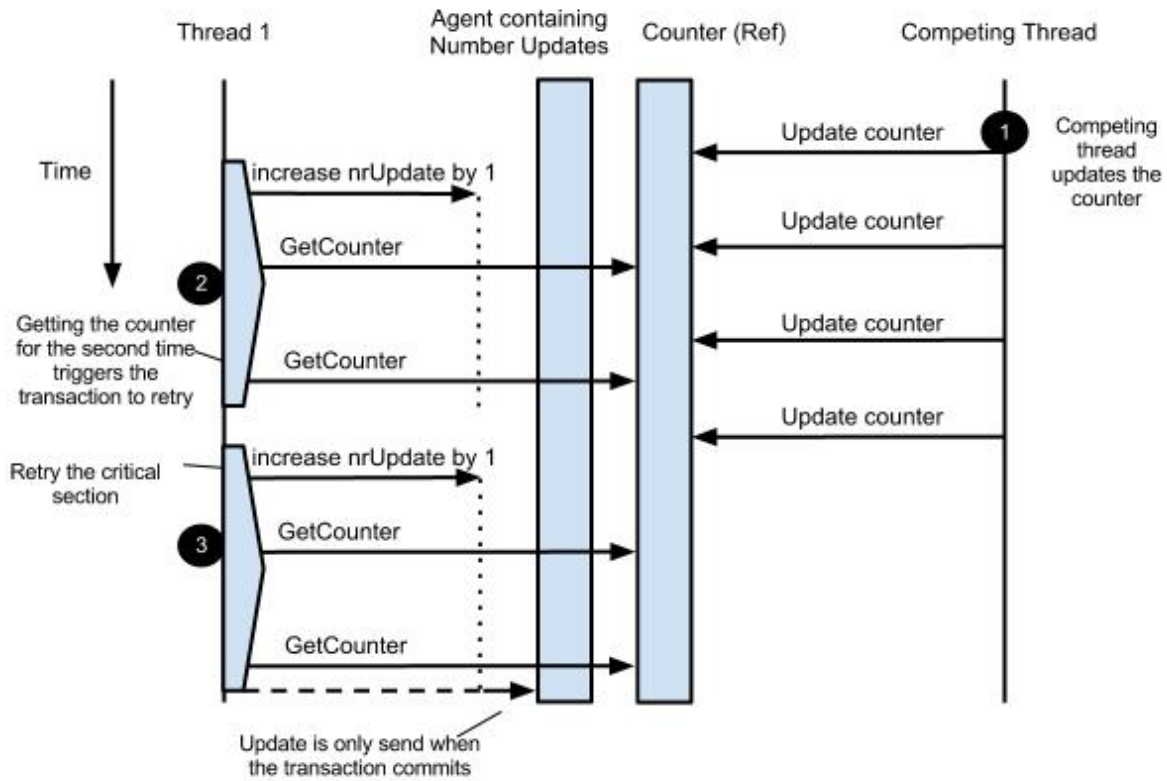


Figure 12.4 Example of sending an Agent update within a transaction

In example 12.4 we use two counters. One to trigger the retry of the transaction and the other to count the number of agent updates. The counter "numberUpdates" is using an agent. Another counter is implemented by using an STM reference. This counter is used to trigger the data write collision. When we implement this test example we get the Listing 12.5

Listing 12.5 Update an agent within a transaction

```

val numberUpdates = Agent(0)
val count = Ref(5)
Future {
  for (i <- 0 until 10) {
    atomic { implicit txn => {
      count() = count() +1
    }}
    Thread.sleep(50)
  }
}
var nrRuns = 0
val myNumber = atomic { implicit txn => {
  nrRuns += 1
  numberUpdates send (_ + 1)
  val value = count()
  Thread.sleep(100)
  count()
}}
nrRuns must be > (1)
myNumber must be (15)
Await.ready(numberUpdates.future(), 1 second)
numberUpdates.get() must be (1)

```

- 1 Send the update to the agent
- 2 Trigger the retry of the atomic block
- 3 The agent is only one time updated

As you can see the atomic block is executed more than once but the agent is only updated once. This is because the update action is only sent when the transaction commits and that is only once. A common mistake is that the actual update is also done within the transaction. This is even implied when looking at the Agent example given in the Akka documentation in versions lower than 2.2. In listing Listing 12.6 we have copied the example from the Akka documentation. We show a different example, quickly, to illustrate this, one that is something of a hello world of the transaction realm: the transfer of money from one account to another.

Listing 12.6 Copy of the agent example used in the Akka documentation

```
def transfer(from: Agent[Int], to: Agent[Int], amount: Int): Boolean = {
  atomic { txn
    if (from.get < amount) false
    else {
      from send (_ - amount)
      to send (_ + amount)
      true
    }
  }
}
```

When looking at this example. It seems that it is never possible to overdraw the from balance. But as we have learned here, this isn't the case, because the actual withdrawal is done when the transaction is committed. And another thread is able to send an update action during the time period of the balance check and the final commit of the transaction. Actually it is possible that multiple update actions are already waiting to be processed when calling the get method. So this isn't the way to protect the Agent from overdraft. Actually the STM transaction with an Agent isn't able to solve this problem, because the actual update is done in another thread. For this we need to coordinate multiple atomic blocks within different threads. This is what Coordinated transactions and transactors can do and it is described in the next section.

12.3 Actors within transactions

In the previous section we saw that we can use Agents within a STM transaction. In this section, we show how we can integrate Actors with the transactions. In Akka there are two approaches that accomplish this. The first is to distribute the transaction over multiple actors, using a 'coordinated' transaction. The other approach is to use transactors. These are Actors implementing a general pattern for coordinating transactions, which can be used in most cases. We start this section explaining the Coordinated transactions using the example of a transfer of money between accounts. In the second part, we show the same example, but using transactor's instead of a coordinated transaction.

12.3.1 Coordinated transactions

As we explained in the previous section, we can't use Agents and transactions to solve the problem of transferring money. The problem is that the two actors are completely unrelated, but we need them to coordinate their actions. Let's look at Figure 12.5

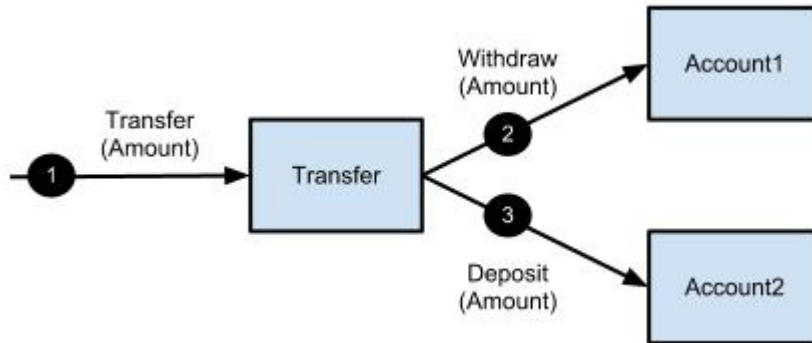


Figure 12.5 Transfer an amount from one account to another.

When transferring money, we need to withdraw from one account and deposit that amount into the other. But if one action fails, for whatever reason, we need all steps to be canceled; we want the transaction to be committed only when all steps have succeeded. To be able to do that with Akka we need "Coordinated Transactions." The idea is to create an Atomic transaction and distribute it over multiple actors. All actors will start and finish the transaction at the same time, because it is like one big atomic block.

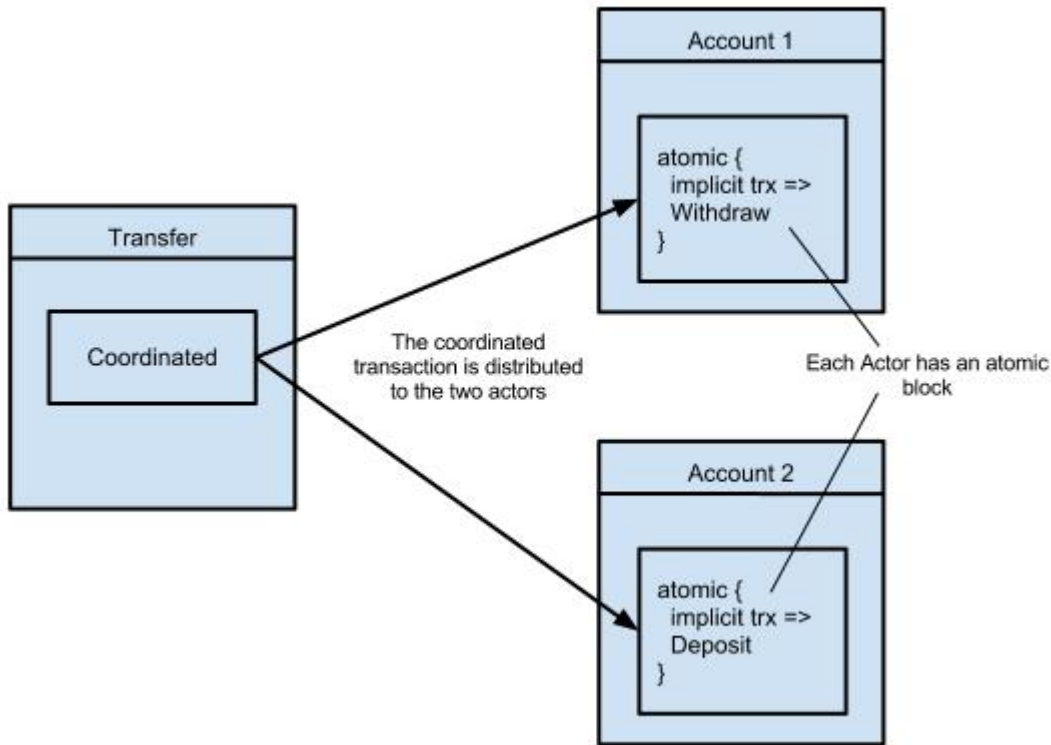


Figure 12.6 Coordinated transaction

In Figure 12.6 the transfer actor creates the transaction and distributes it to the two account actors. The actual update action of the account is done within this distributed transaction. Each atomic block will only commit its changes when all other atomic blocks also succeeded, and if one fails, the whole transaction fails. This is what is needed to implement the transfer example: once we take money from one party, we must be certain it went to the other party. To enable this required coordination, we need to create a Coordinated transaction, which also requires a timeout. This can be defined implicitly when creating the Coordinated class.

```
import akka.transactor.Coordinated
import scala.concurrent.duration._
import akka.util.Timeout

implicit val timeout = Timeout(5 seconds)
val transaction = Coordinated()
```

To distribute the transaction we need to send it to the Actors with our message. By sending the coordinated transaction to an Actor, the actor is automatically included in the transaction.

USING COORDINATED TRANSACTION IN THE ACCOUNT ACTOR

When we implement the actor that represents an account, we need to handle the Coordinated message just like other normal messages. But when processing a deposit or withdrawal, we need to do these requests within a transaction. In listing 12.7 we have created an actor which keeps the balance of an account.

Listing 12.7 Account Actor using coordinated transactions

```

case class Withdraw(amount:Int)
case class Deposit(amount:Int)
object GetBalance

class InsufficientFunds(msg:String) extends Exception(msg)

class Account() extends Actor {
  val balance = Ref(0)
  def receive = {
    case coordinated @ Coordinated(Withdraw(amount)) {
      coordinated atomic { implicit t
        val currentBalance = balance()
        if ( currentBalance < amount) {
          throw new InsufficientFunds(
            "Balance is too low: "+ currentBalance)
        }
        balance() = currentBalance - amount
      }
    }
    case coordinated @ Coordinated(Deposit(amount)) {
      coordinated atomic { implicit t
        balance() = balance() + amount
      }
    }
    case GetBalance => sender ! balance.single.get
  }

  override def preRestart(reason: Throwable, message: Option[Any]) {
    self ! Coordinated(
      Deposit(balance.single.get))(Timeout(5 seconds))
    super.preRestart(reason, message)
  }
}

```

- ❶ Balance must be wrapped by the STM Ref
- ❷ Match the withdraw message with a Coordinated transaction

- 3 Start the distributed atomic block
- 4 Update the balance within the atomic block
- 5 Match the deposit message with a Coordinated transaction
- 6 Get the balance which isn't in a distributed transaction

The balance is an STM reference, because we want the read and writes part of the transaction. And we have learned in section 12.1.1 when we reference variables within an atomic block, we need to wrap it with an STM reference. In the receive function, we match the expected messages, which is a Coordinated class containing our messages. Processing the messages we need to start the transaction. This is done with the following lines

```
coordinated atomic { implicit t
  ...
}
```

Within the transaction we check if the balance is sufficient to withdraw the requested amount. When there isn't enough money on the account we throw an `InsufficientFunds` exception. This causes the transaction to fail and it also restarts the actor. We don't want to lose the current balance after a restart so we send the balance to self when restarting.

Our account is finished and can be used within a coordinated transaction. In Listing 12.8 we start with a deposit on our first account.

Listing 12.8 Sending Coordinated transaction to an Actor

```
val account1 = system.actorOf(Props[Account])
implicit val timeout = new Timeout(1 second)

val transaction = Coordinated()
transaction atomic { implicit t =>
  account1 ! transaction(Deposit(amount = 100))
}

val probe = TestProbe()
probe.send(account1, GetBalance)
probe.expectMsg(100)
```

In this example we do a deposit within a transaction. But this isn't much of a transaction sending one message. This is because sending one message is already atomic. We want to start a transaction by sending the Coordinated message, but without becoming a participant in the transaction. This can also be done differently

and with less code. Normally we use the following to send a coordinated message without participating in the created transaction (this makes the code more readable).

```
val account1 = system.actorOf(Props[Account])
implicit val timeout = new Timeout(1 second)

account1 ! Coordinated(Deposit(amount = 100))

val probe = TestProbe()
probe.send(account1, GetBalance)
probe.expectMsg(100)
```

Another difference is that because we are not part of the transaction our thread can proceed without waiting for the transaction to complete, so it also improves the performance of the calling thread or actor.

CREATING THE COORDINATED TRANSACTION IN THE TRANSFER ACTOR

Now that we can create Coordinated messages, we are able to distribute the Coordinated transaction. Let's make a Transfer Actor shown in Figure 12.7.

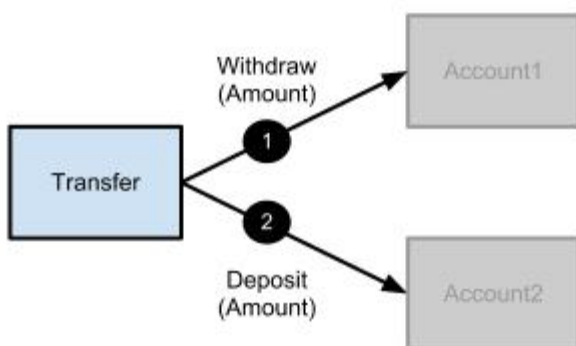


Figure 12.7 Transfer an amount from one account to another.

This transfer actor, shown in listing 12.9, will receive the requests to transfer an amount from one account to another. And at the end of each transfer it returns if the request has succeeded or failed.

Listing 12.9 Implementation of the Transfer actor creating the Coordinated transaction

```

case class TransferTransaction(amount: Int,
                               from: ActorRef,
                               to: ActorRef)

class Transfer() extends Actor {
  implicit val timeout = new Timeout(1 second)

  override def preRestart(reason: Throwable, message: Option[Any]) {
    message.foreach(_ => sender ! "Failed")
    super.preRestart(reason, message)
  }

  def receive = {
    case TransferTransaction(amount, from, to) => {
      val transaction = Coordinated()
      transaction atomic { implicit t
        from ! transaction(Withdraw(amount))
        to ! transaction(Deposit(amount))
      }
      sender ! "done"
    }
  }
}

```

1 Send failure message

2 Create a transaction

3 Send Success message

This transfer Actor is just a normal actor which creates a coordinated transaction to be able to transfer money in a consistent way. When there is a sufficient amount in account1, we get the expected result

```

transfer ! TransferTransaction(amount = 50,
                               from = account1,
                               to = account2))

expectMsg("done")

```

And when the balance of account1 is insufficient (25), we get the expected failure message and both accounts are unchanged, just as we expect.

```

transfer ! TransferTransaction(amount = 50,
                               from = account1,
                               to = account2))

expectMsg("Failed")

```

```
probe.send(account1, GetBalance)
probe.expectMsg(25)
probe.send(account2, GetBalance)
probe.expectMsg(0)
```

As you can see, we need to deal with the Coordinated transaction in different places in the code. And most of the time, we create a similar structure dealing with Coordinated transactions. To improve upon this DRY (don't repeat yourself) violation, Transactors were created.

12.3.2 Creating transactors

Transactors are actors that are capable of dealing with messages that comprise Coordinated transactions. A transactor has several methods which we can use to implement the wanted functionality, without exposing the Coordinated class. This way we can deal with the functionality of the transactions and not worry about Coordinated transactions. To show how we can use transactors, we are going to implement the same transfer example as in the previous section using Coordinated transactors. All the functional code will be seen again in the example, only the Coordinated part is removed, because the transactor will hide it from our code.

We start by transforming the Account Actor into a Transactor, by extending the Transactor instead of Actor:

```
import akka.transactor.Transactor

class AccountTransactor() extends Transactor {
  val balance = Ref(0)

  ...
}
```

In the transactor, we don't need to implement the receive method, but there is a method named `atomically` which is the transaction part of the transactor. In this method, we need to implement our withdrawal and deposit functionality.

```
def atomically = implicit txn => {
  case Withdraw(amount) => {
    val currentBalance = balance()
    if ( currentBalance < amount) {
      throw new InsufficientFunds(
        "Balance is too low: "+ currentBalance)
    }
    balance() = currentBalance - amount
  }
}
```

```

case Deposit(amount) => {
  balance() = balance() + amount
}
}

```

All this code is executed in the Coordinated transaction, but all the coordination boilerplate code is hidden. Our previous example had the message `GetBalance` which wasn't part of a transaction. This can be implemented by overriding the `normally` method.

```

override def normally = {
  case GetBalance => sender ! balance.single.get
}

```

This is also a partial function just as `receive`. All messages which are implemented in the `normally` function, will not be passed to the `atomically` function. In this method you can implement normal actor behavior, or use the normal STM `atomic` for local transactions. Because a `transactor` is an `Actor` we can also override the `preRestart` method, so we can reuse this part. And our first `transactor` is done and the complete listing is shown in Figure 12.8.

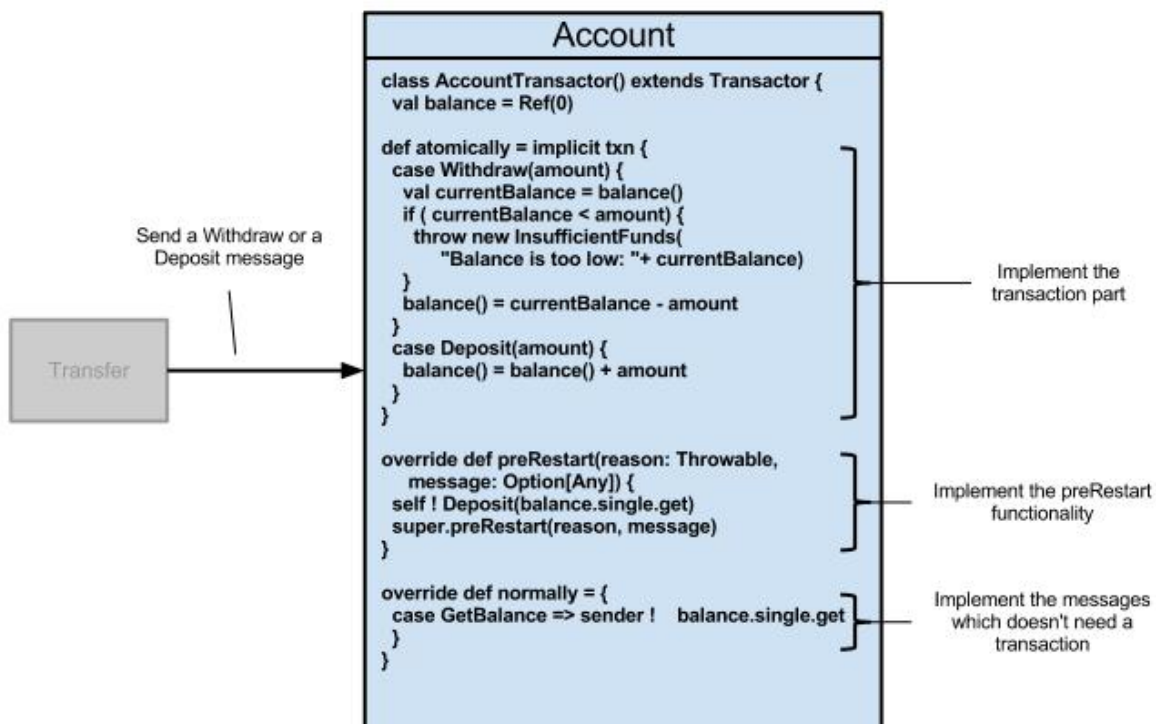


Figure 12.8 Account Actor rewritten as a transactor

The `transactor` now has the same functionality as the `Account Actor 12.7` and its

behavior is the same. This transactor can now be used in a transaction just like a coordinated Actor.

```
val account1 = system.actorOf(Props[AccountTransactor])
val account2 = system.actorOf(Props[AccountTransactor])

val transaction = Coordinated()
transaction atomic { implicit t
  account1 ! transaction(Withdraw(amount = 50))
  account2 ! transaction(Deposit(amount = 50))
}
```

Because it is a transactor, it is also possible to send just the transfer message. When a transactor receives a message without a coordinated transaction, it makes a new transaction and processes the message. For example, when we deposit some money in an account it doesn't need to be done in a Coordinated transaction. We already saw that we can send a coordinated message without joining the transaction, but when using a transactor, we can also send just the message.

```
account1 ! Coordinated(Deposit(amount = 100))

account1 ! Deposit(amount = 100)
```

These two lines of code are equivalent when using a transactor. Next, we are going to implement the Transfer Actor as an transactor, because this actor needs more functionality than the account example showed. The AccountTransactor only acts within a transaction, but it doesn't include other actors within its transaction. When the transfer Actor starts a coordinated transaction we need to include both accounts in the transaction. For this, a transaction has the coordinated method. Here we define which actors need to be in the transaction and which messages need to be sent. This is done with the sendTo method.

```
override def coordinate = {
  case TransferTransaction(amount, from, to) =>
    sendTo(from -> Withdraw(amount),
           to -> Deposit(amount))
}
```

In our case we need to send two messages to two actors. The Withdraw message to the "from" actor and the deposit to the "to" actor. This is all we need to

do to include the two actors into the transactor. When you want to send the received message to the other actors, you can also use the include method:

```
override def coordinate = {
  case msg:Message => include(actor1, actor2, actor3)
}
```

The above code sends the received Message to the three actors. We can include other actors within our transactor now, but we don't have the same functionality yet. We need to send a message when we are done. For these kind of actions, a transactor has two methods which can be overridden, the before and after method. These methods are called just before and after the atomically method and are also partial functions. For our example, we don't need the before method, but using the after to be able to send the "done" message when the transaction has successfully ended.

```
override def after = {
  case TransferTransaction(amount, from, to) => sender ! "done"
}
```

Now we can put all the parts together and create our TransferTransactor as shown in Figure 12.9

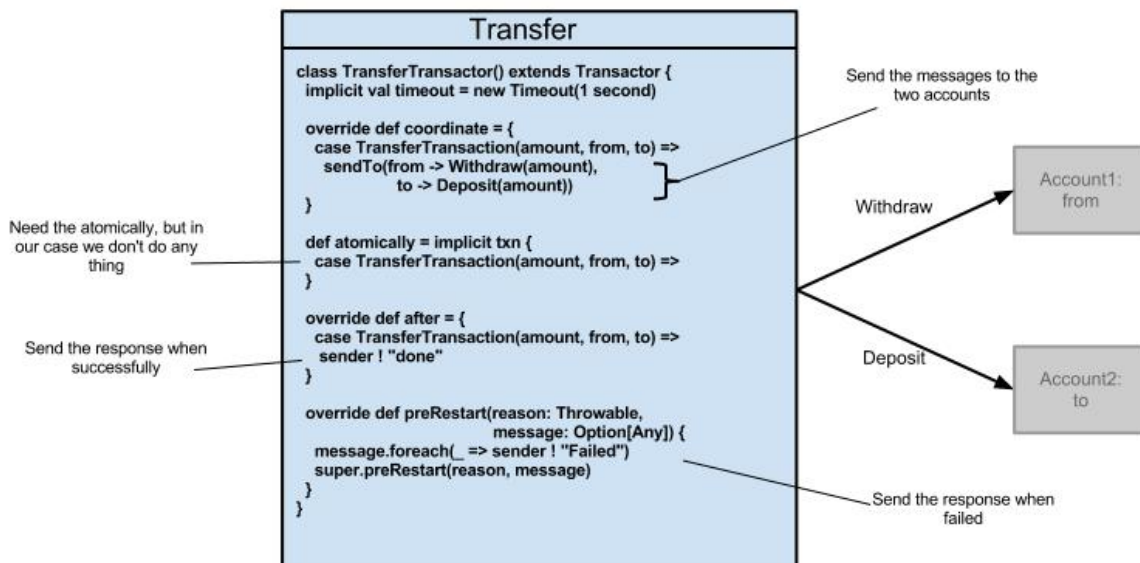


Figure 12.9 Transfer Actor rewritten as a transactor

Again we have rewritten the actor into a transactor and didn't need to deal with the Coordinated transaction. This transactor behaves almost the same as our Coordinated transaction version. There is one difference. This transactor can also join an existing transaction and the Coordinated version always creates its own transaction and isn't able to join an existing transaction. It is possible but in that case, we need to implement it.

In this section, we have seen how we can create and use transactions that involve multiple actors. We started with coordinated transactions, which can define transactions. But Akka has also transactors which are a general structure for using actors with transactions. Using these transactors will hide the use of coordinated transactions and reduces the amount of code. The transactors have support for adding other actors in the transaction and it is possible to implement normal behavior in combination with the transaction, by overriding the before and after methods or use the normally method to skip the transaction completely.

12.4 Summary

For a lot of professional programmers, one of the hardest things to imagine about leaving behind traditional state-based frameworks in favor of stateless, is the idea of going without transactions. This chapter has shown that Akka's statelessness does not mean no transactions, but rather transactions are simply implemented differently, and the guarantees they offer are available, but they are simply obtained by other means.

The implementation exercises also illustrated the fact that transactions in Akka do involve a few new elements, but by and large, they are accomplished using all the programming techniques we have been learning throughout the book.

The takeaways, as we go forward into making complete solutions, are:

- Using STM, we can achieve the familiar transaction behaviors of commits and rollbacks
- While optimistic locking takes a different approach than the more common pessimistic variant, we can still guarantee atomicity with it
- We can implement Transactions that involve multiple parties by doing Coordinated Transactions where each participant is drawn in by a coordinator and the results are orchestrated by the coordinator
- Finally, Coordinated Transactions can be unburdened of the coordination boilerplate (and the resulting code made more readable) by using Transactors

In the next chapter, we are going to take the many approaches we have shown throughout the book and use them to create a case study. This will be the 'pulling it all together' part of the book.

13

Integration

In this chapter

- Camel
- Endpoints
- Rest
- Spray
- Play-mini
- Consumer/producer

In this chapter, we are going to look at some examples of actors being used to integrate to other external systems. Applications today are more and more complex, requiring connections to different information services and applications. It is almost impossible to create a system that doesn't either rely on or supply information to other systems. To be able to communicate with other systems, the two sides have to go through an agreed upon interface. We start with some Enterprise Integration patterns. Next we describe how the Camel extension (Camel is an Apache Project for doing integration) can help in integrating a system with other external systems. We finish with some REST examples, and detail the different approaches to implementing integration with Akka.

13.1 Message endpoints

In the previous chapter, we showed how to build systems using various enterprise patterns. In this section, we describe the patterns that apply when different systems need to exchange information. Consider a system that needs customer data from a customer relations application, yet you don't want to manage this data in multiple applications. The implementation of an interface between two systems isn't always easy, because the interface contains two areas: the transport layer and the data which is sent over this transport layer. Both areas have to be addressed to integrate the systems. There are also patterns to help us to design the integration between multiple systems. For example, we are creating an order system for use in a book stockroom, that processes orders from all kinds of customers. These customers can order the books by visiting the store. The bookstore already uses an application to sell and order books. So the new system needs to exchange data with this existing application. This can only be done if both systems agree on which messages are sent and how they are sent. Because you probably can't change the external application, you have to create a component that can send and/or receive messages from the existing application. This component is called an endpoint. Endpoints are part of your system and are the glue between the external system and the rest of your system, which is shown in Figure 13.1.

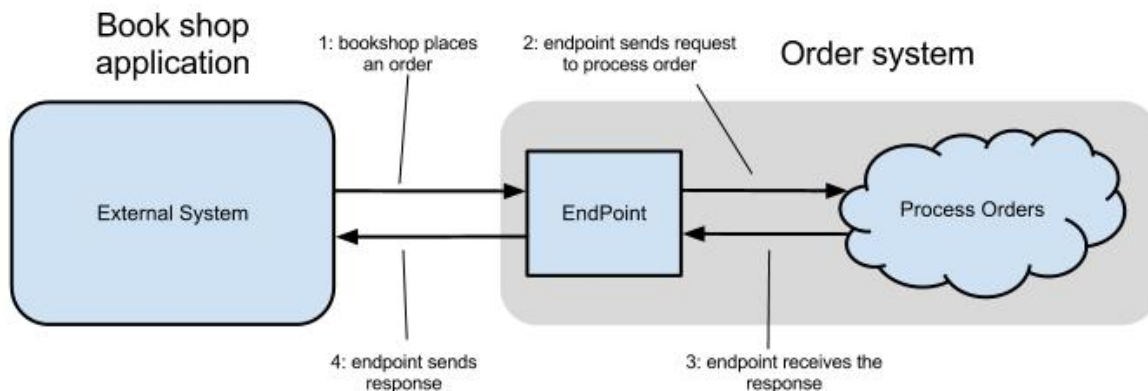


Figure 13.1 Endpoint as glue between order system and book shop application

The endpoint has the responsibility of encapsulating the interface between the two systems in such a way that the application itself doesn't need to know how the request is received. This is done by making the transport pluggable and using a canonical data format. There are a lot of different transport protocols to potentially support: REST/HTTP, TCP, MQueues, or simple files. And after receiving the message, the endpoint has to translate the message into a message format that is

supported by our order system. By translating the message, the rest of the system doesn't know that the order was received from an external system. In this example, the endpoint receives a request from the external system and sends a response back. This is called a consumer endpoint because it consumes the request. It is also possible that our system needs some data from another system for example the customer details, which are kept in the customer relation application.

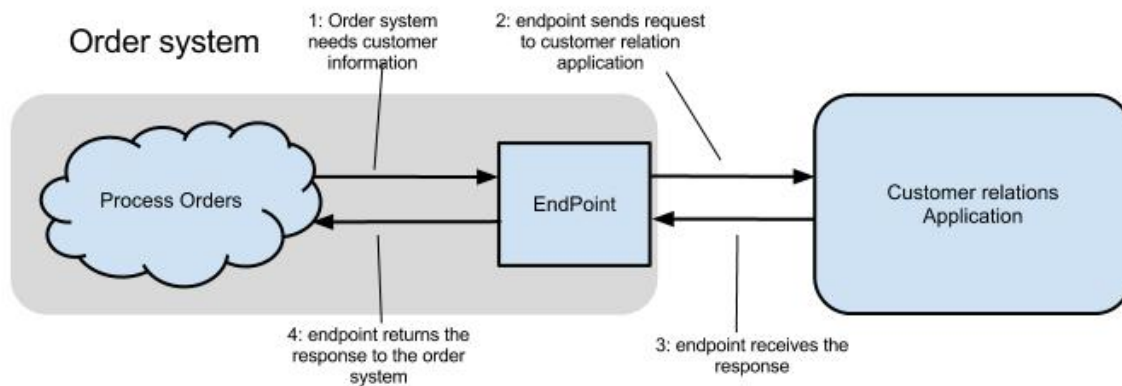


Figure 13.2 Endpoint as glue between order system and customer relation application

In Figure 13.2 the order system is initiating the communication between systems, and because the endpoint produced a message, which is sent to the external system, this is called a producer endpoint. Both usages of the endpoints are hiding the details of the communication from the rest of the system and when the interface between the two systems change, only the endpoint needs to be changed. There are a few patterns in the Enterprise Integration Pattern catalog that apply for such endpoints. The first pattern we are going to describe is the Normalizer pattern.

13.1.1 Normalizer

We have seen that our order system receives the orders from the bookshop application, but it is possible that our system also receives orders from a web shop, or by customers sending email. We can use the Normalizer pattern to make these different sources all feed into a single interface on the application side. The pattern translates the different external messages to a common, canonical message. This way all the message processing can be reused, without the knowledge that different systems are sending these messages.

We create three different endpoints to consume the different messages, but translate them into the same message, which is sent to the rest of the system. In Figure 13.3 we have the three endpoints, which handle the details on how to get the

needed information and translate it into the common message format the order system expects.

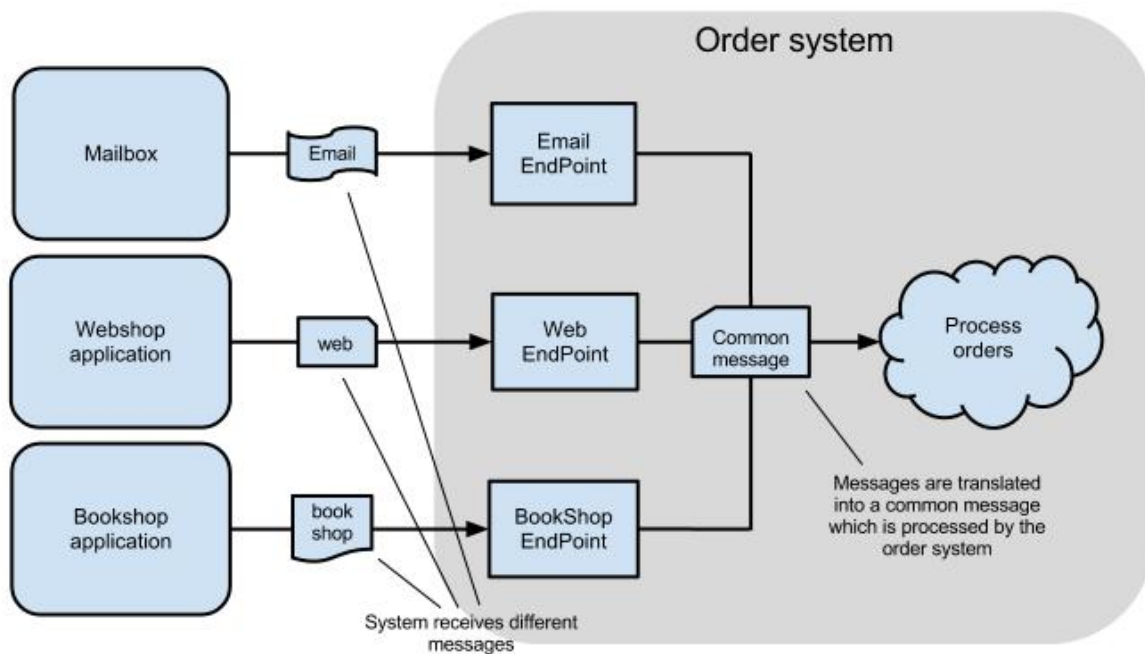


Figure 13.3 Multiple Endpoint example

Translating the different messages into a common message is called the Normalizer pattern. This pattern combines router and translator patterns into an endpoint. This implementation of the Normalizer pattern is the most common one. But when connecting to multiple systems using different transport protocols and different messages, it is desirable to reuse the translators of the messages; this makes the pattern implementation a little bit more complex. Let us assume that there is another bookshop that is connecting to this system using the same messages but using MQueue to send those message. In cases of more complex implementations such as this, the Normalizer pattern can be divided into three parts. Figure 13.4 shows the three part. The first is the implementation of the protocol, next a router decides which translator has to be used. And finally the actual translation takes place.

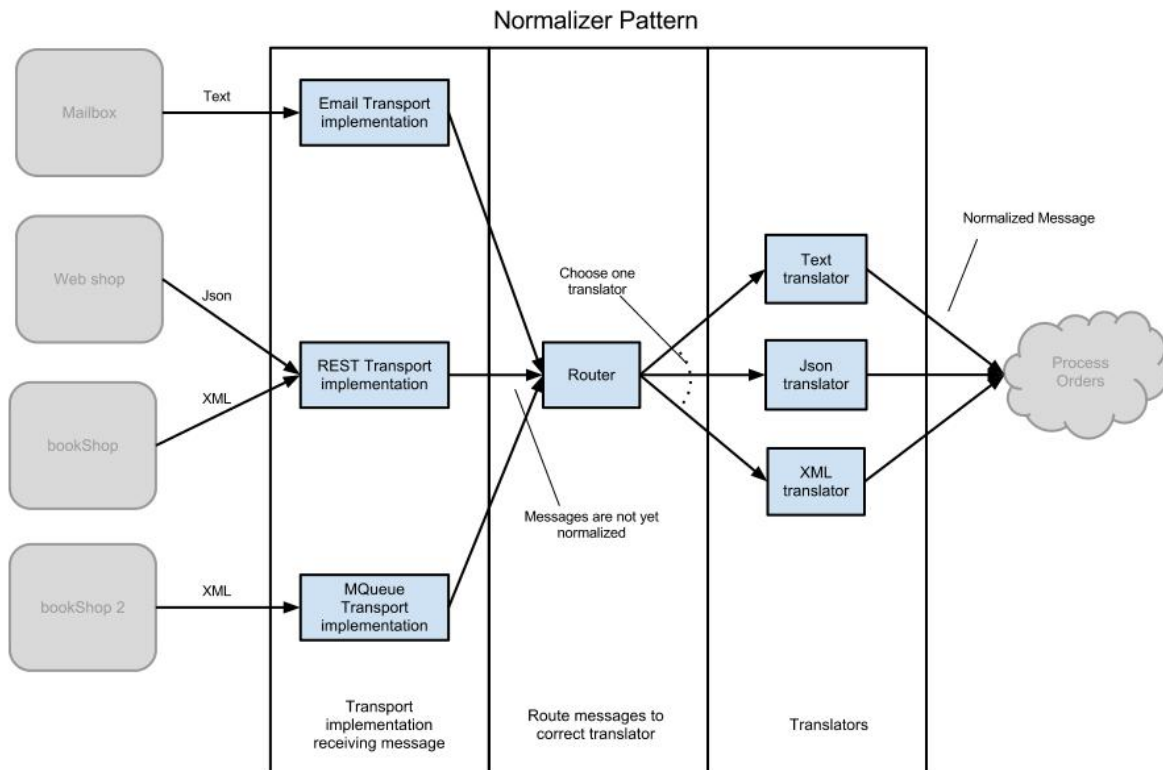


Figure 13.4 Normalizer pattern divided into three parts

To be able to route the message to the correct translator requires the ability to detect the type of the incoming message. How this has to be done differs greatly and depends on the external systems and types of messages. In our example, we support three types of messages: plain text, JSON and XML, which can be received from any of three transport layer types: Email, REST and MQueue. In most cases, the simplest implementation would make the most sense: the endpoint and translation (and no router) implemented as a single component. In our example, it is possible to skip the router for the Email and MQueue protocol and go directly to the correct translator, because we are receiving only one type of message. This is a trade-off between flexibility and complexity: when using the router, it is possible to receive all types of messages on all protocols without any extra effort, but we have more components. Only the router needs to know how to distinguish between all the message types. Tracing the messages can be more difficult, which can make this solution more complex, and most of the time, you don't need this flexibility, because only one type of system is being integrated (supporting only one message type).

13.1.2 Canonical Data Model

The Normalizer pattern works well when connecting one system to another external system. But when the connectivity requirements between the systems increases we need more and more endpoints. Let's go back to our example. We have two back office systems, the order system and the customer relations system. In the previous examples, the shops were only connected to the order system, but when they also needed to communicate with the customer relations system, the implementation becomes more complex, as shown in figure 13.5.

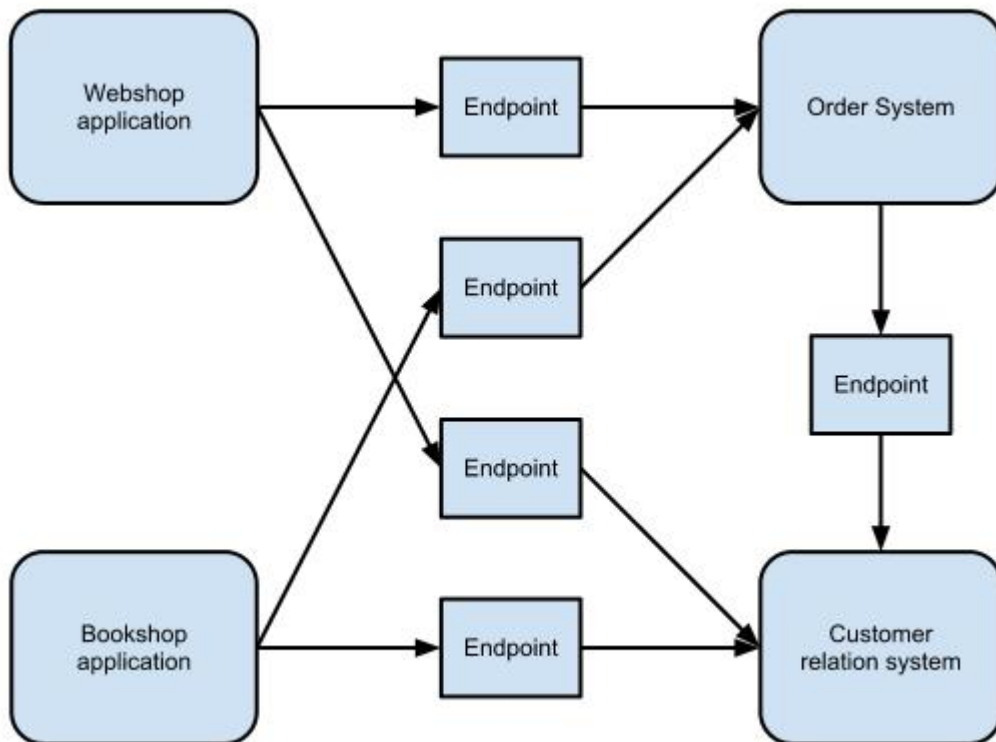


Figure 13.5 Connectivity diagram between systems

At this point, it isn't important which system is implementing the endpoints, the problem is that when it is necessary to add a new system to integrate to, we need to add more and more Endpoints: one for the shop applications, and two endpoints to integrate the existing back office systems. Over time, increasing exponentially, the number of endpoints will explode.

To solve this problem, we can use the Canonical Data Model. This pattern connects multiple applications using interface(s) that are independent of any specific system. Then each system we wish to integrate with will have to have incoming and outgoing messages converted to the canonical form for the given endpoint.

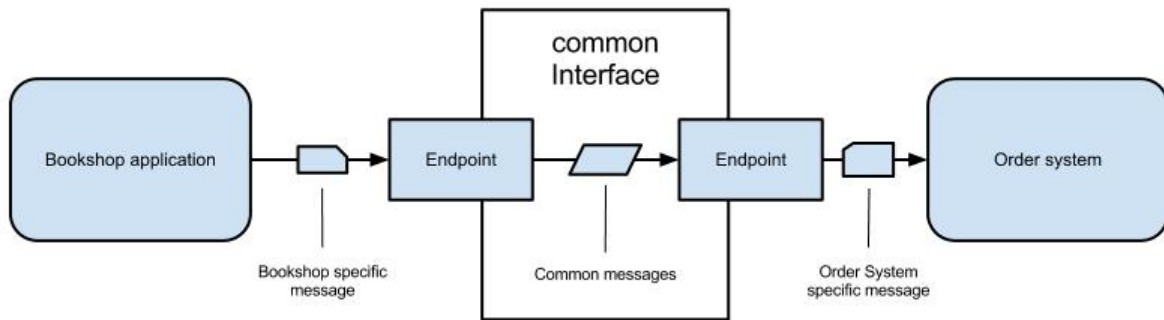


Figure 13.6 Use a common interface between systems

This way, every system has an endpoint which implements a common interface and uses common messages. Figure 13.6 shows when the Bookshop application wants to send a message to the order system, the message is first translated to the canonical format and then it is sent using the common transport layer. The endpoint of the Order system receives the Common message which translates it to an Order System message. This looks like an unnecessary translation, but when applying this to a number of systems the benefit is clear, see Figure 13.7

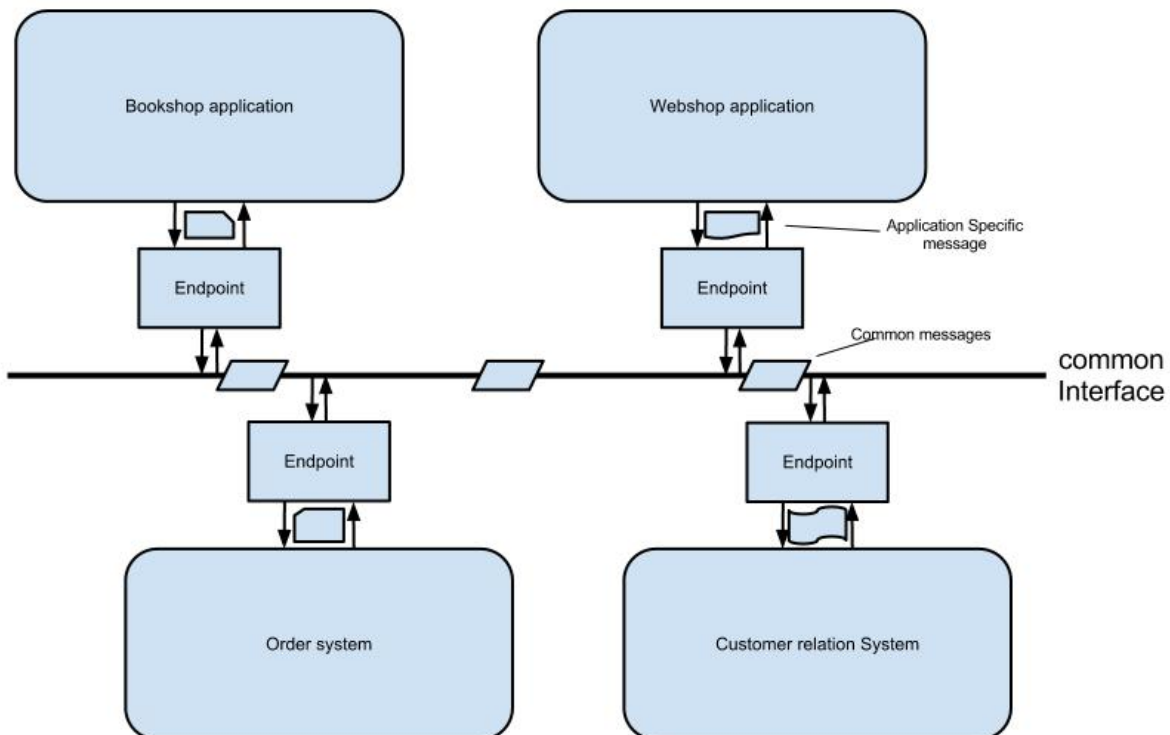


Figure 13.7 Canonical pattern using to connect multiple systems

As you can see every System or application has one endpoint. And when the Web shop needs to send a message to the Order System it uses the same endpoint

as when sending it to the customer relations system. When we add a new system to be integrated, we need only one endpoint instead of the four shown in Figure 13.5. This reduces the number of endpoints greatly when there are a large number of integrated systems.

The Normalizer pattern and the Canonical Data Model are quite helpful when integrating a system with other external systems or applications. The Normalizer pattern is used to connect several similar clients to another system. But when the number of integrated systems increases, we need the Canonical Data Model, which looks like the Normalizer Pattern, because it also uses normalized messages. The difference is that the Canonical Data Model provides an additional level of indirection between the application's individual data formats and those used by the remote systems. While the Normalizer is only within one application. And the benefit of this additional level of indirection is that, when adding a new application to the system, only the translator into these common messages has to be created, no changes to the existing system are required.

Now that we know how we can use endpoints, the next step is to implement them. When implementing an endpoint we need to address the transport layer and the message. Implementing the transport layer can be hard, but most of the time the implementation is application independent. Wouldn't it be nice if someone already implemented the transport layers? In fact, this is what the Camel Framework provides. Let's see how Camel can help us with implementing an endpoint.

13.2 Implementing endpoints using the Camel Framework

Camel is an apache framework whose goal is to make integration easier and more accessible. It makes it possible to implement the standard enterprise integration patterns in a few lines of code. This is achieved by addressing three areas:

1. Concrete implementations of the widely used Enterprise Integration Patterns
2. Connectivity to a great variety of transports and APIs
3. Easy to use Domain Specific Languages (DSLs) to wire EIPs and transports together

The support of a great variety of transport layers is the reason why we would want to use Camel with Akka. Because this will enable us to implement different transport layers without much effort. In this section, we explain some of what Camel is and how to send and receive messages using the Camel Consumer and Producer.

The Akka Camel module will allow you to use Camel within Akka. And it

enables you to use all the transport protocols and API's implemented in Camel. A few examples of protocols supported are HTTP, SOAP, TCP, FTP, SMTP or JMS. At the moment, approximately 80 protocols and APIs are supported.

To use the module extension is every easy. Just add akka-Camel to the project dependencies and you can use the Camel Consumer and/or Producer classes to create an Endpoint. Using these classes will hide the implementation of the transport layer. The only functionality you have to implement is the translations between your system messages and the interface messages.

Because the transport layer implementations are completely hidden, it is possible to decide which protocol to use at runtime. This is the next great strength of using the Camel extensions. As long as the message structure is the same, no code changes have to be made. So when testing we could write all the messages to the file system, because we don't have the correct external system available in the test environment, and as soon as the system is in the acceptance environment, we can change the used Camel protocol into a REST interface for example, with only one configuration setting.

The Camel module works internally with Camel classes. Important Camel classes are the camel context and the ProducerTemplate. The CamelContext represents a single Camel routing rule base, and the ProducerTemplate is needed when producing messages. But for more details, look at the Camel documentation. The Camel module hides the uses of these camel classes, but some times one needs them when more control of how messages are received or produced is required. The Camel Module creates a Camel extension for each Akka system. Because several underwater actors are created and need to be started in the correct ActorSystem. To get a systems CamelExtension, one can use the CamelExtension object.

```
val camelExtension = CamelExtension(system)
```

When a specific Camel class is needed, like the context or the ProducerTemplate, this extension can be used. We will see in the next sections some examples. We start with a simple consumer example that reads files and changes them using other protocols like TCP connections and ActiveMQ. We end this section by creating a Producer that can send messages to the created Consumer. So let us begin by using Camel in creating a consumer.

13.2.1 Implement a consumer endpoint receiving messages from an external System

The example we are going to implement is an Order System receiving messages from a bookshop. Of course, this order system must be able to receive messages from different books stores. Let's say the received messages are XML files in a directory. The transport layer is in this case the file system. The endpoint of the order system needs to track new files and when there is a new file it has to parse the XML content and create a message the system can process. Before we start implementing our endpoint consumer we need to have our messages shown in Figure 13.8.

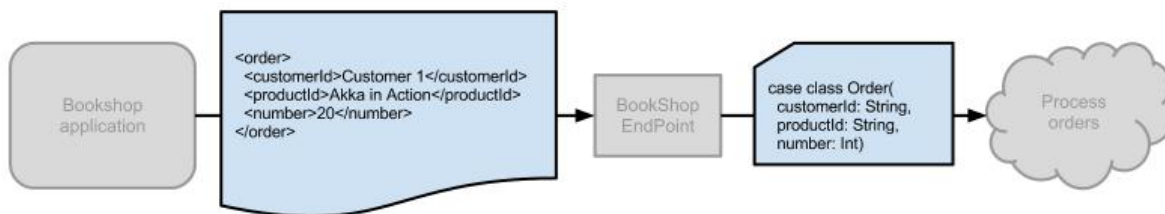


Figure 13.8 Messages received and sent by our endpoint

The first message to look at is the XML sent by the Bookshop application to our endpoint indicating that "customer 1" wants 20 copies of "Akka in Action". The second message is the class definition of the message the order system can process.

IMPLEMENTING A CAMEL CONSUMER

Now that we have our messages we can start implementing our consumer endpoint. We start by extending our Actor class with the camel Consumer trait instead of the normal Akka Actor class.

```
class OrderConsumerXml() extends akka.camel.Consumer { }
```

The next step is to set the transport protocol, this is done by overriding the endpoint Uri. This Uri is used by the Camel framework to define the transport protocol and its properties. In our case we want to be able to change this URI, so we are going to add the URI to our constructor. And of course we need to implement the receive method, because it is also an Akka actor. Figure 13.9 shows the implementation of the Consumer.

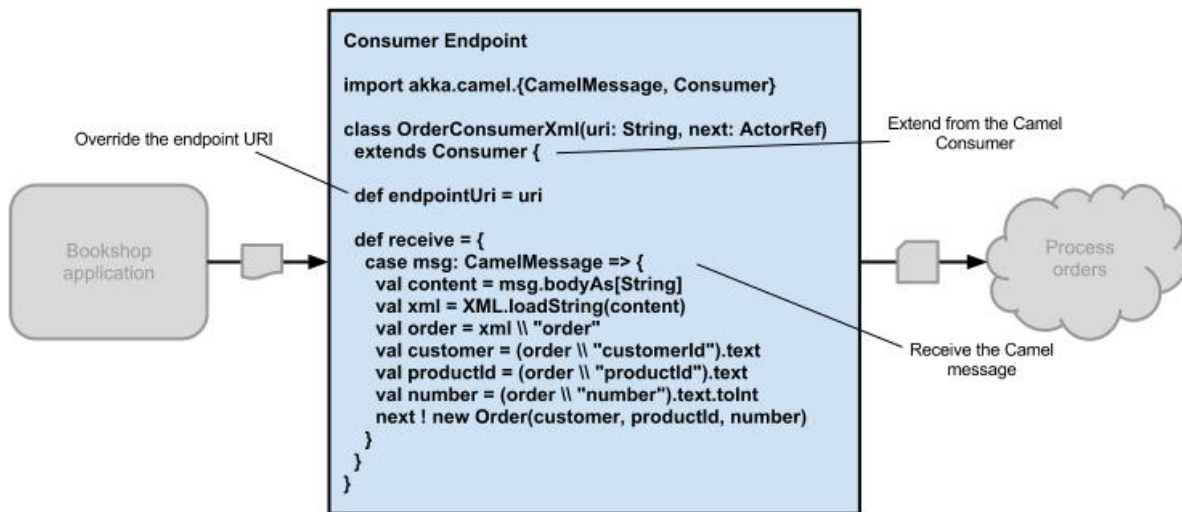


Figure 13.9 OrderConsumerXml, the implementation of the consumer endpoint

When a new message is received, it comes to the Actor through its usual method, as a `CamelMessage`. A `CamelMessage` contains a body, which is the actual message received, and a map of headers. The content of these headers depends on the protocol used. In the examples in this section, we don't use these headers, but we use them in the Camel REST example later in section 13.3.1.

When a `CamelMessage` is received we convert the body to a string and parse the XML into an `Order` message and send it to the next actor that is available to process the `Order`.

We have implemented the translation from the XML to the `Order` object, but how do we pickup these files? This is all done by the Camel framework. All we have to do is to set the Uri. To tell Camel we want it to pickup files, we use the following Uri

```
val camelUri = "file:messages"
```

This Uri starts with the Camel component. In this case we want the file component. The second part depends on the component chosen. When using the file component the second part is the directory where the message files are placed. So we are expecting our files in the directory "messages". All the possible components can be found at <http://camel.apache.org/components.html>. The possible options are also described at this site.

So let's start creating the consumer so we can see how that works.

```
val probe = TestProbe()
```

```
val camelUri = "file:messages"
val consumer = system.actorOf(
  Props(new OrderConsumerXml(camelUri, probe.ref)))
```

Because we use the Camel Consumer trait, a lot of components are started and we have to wait for these components before we can proceed with our test. To be able detect that Camel's startup has finished, we need to use the CamelExtension.

```
val camelExtention = CamelExtension(system) ❶
val activated = camelExtention.activationFutureFor( ❷
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.result(activated, 5 seconds) ❸
```

- ❶ Get the CamelExtension for this Akka system
- ❷ Get the activation future
- ❸ wait for Camel to finish starting up

This Extension contains the `activationFutureFor` method, which returns a Future. The Future triggers when the Camel route is done starting up. So after that, we can proceed with our test.

Listing 13.1 Test the Order Consumer

```
val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
  <customerId>{ msg.customerId }</customerId>
  <productId>{ msg.productId }</productId>
  <number>{ msg.number }</number>
</order> ❶
val msgFile = new File(dir, "msg1.xml")
FileUtils.write(msgFile, xml.toString()) ❷
probe.expectMsg(msg) ❸
system.stop(consumer)
```

- ❶ Create the XML content
- ❷ Write the file in the message directory
- ❸ Expect the Order message to be sent by the Consumer

As you can see, we receive an Order message when a file containing XML is placed in the messages directory. Note, we are not required to provide any code

dealing with checking for files and reading them in; all this functionality is provided by Camel Consumer.

CHANGING THE TRANSPORT LAYER OF OUR CONSUMER

This is nice, but it's just the starting point of Camel's real benefit. Let's say that we also get these XML messages though a TCP connection. How should we implement this? Actually, we already have. To support the TCP connection, all we have to do is to change the used URI and add some libraries to the runtime.

Listing 13.2 TCP test using Order Consumer

```
val probe = TestProbe()
val camelUri =
    "mina:tcp://localhost:8888?textline=true&sync=false" ❶
val consumer = system.actorOf(
    Props(new OrderConsumerXml(camelUri, probe.ref)))
val activated = CamelExtension(system).activationFutureFor(
    consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.result(activated, 5 seconds)
val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
    <customerId>{ msg.customerId }</customerId>
    <productId>{ msg.productId }</productId>
    <number>{ msg.number }</number>
</order>

val xmlStr = xml.toString().replace("n", "") ❷
val sock = new Socket("localhost", 8888)
val ouputWriter = new PrintWriter(sock.getOutputStream, true)

ouputWriter.println(xmlStr) ❸
ouputWriter.flush()
probe.expectMsg(msg)
ouputWriter.close()
system.stop(consumer)
```

- ❶ Use another Uri
- ❷ Due to the textline option newlines indicate end of message, so we need to remove them
- ❸ Send XML message using TCP

In this Example we use the Mina component to deal with the TCP connection. The second part of the URI looks completely different, but is needed to configure the connection. We start with the protocol we need (TCP) and then we indicate on which interface and port we want to listen. After this we include two options (as parameters).

- `textline=true`
This indicates that we are expecting plain text over this connection and that each message is ended with a newline
- `sync=false`
This indicates that we don't create a response

As you can see, without any code changes to the consumer, we can change the transport protocol. Can we change to any protocol without changes? The answer is no, some protocols do require code changes. For example, what about a protocol that needs a confirmation? Let's see how we can do that. Let's assume that our TCP connection needs an XML response. To be able to do this, we need to change our consumer. But it's not that hard. We just send the response to the sender and the Camel Consumer will take care of the rest.

Listing 13.3 Confirm Order Consumer

```
class OrderConfirmConsumerXml(uri: String, next: ActorRef)
  extends Consumer {

  def endpointUri = uri

  def receive = {
    case msg: CamelMessage => {
      try {
        val content = msg.bodyAs[String]
        val xml = XML.loadString(content)
        val order = xml \ "order"
        val customer = (order \ "customerId").text
        val productId = (order \ "productId").text
        val number = (order \ "number").text.toInt
        next ! new Order(customer, productId, number)

        sender ! "<confirm>OK</confirm>" ❶
      } catch {
        case ex: Exception =>
          sender ! "<confirm>%s</confirm>".format(ex.getMessage)
      }
    }
  }
}
```

❶ Send the reply to the sender

That is all and when we change the Uri we can test our new consumer. But before we do that we see that we also have to catch a possible exception and didn't

we say in chapter 3 that we let our actors crash when there are any problems? And that the supervisor should correct these problems? We are now implementing an endpoint which is the separation between a synchronous interface and a messages passing system, which is an asynchronous interface. On these boundaries between the synchronous and asynchronous interfaces, the rules are a little different because the synchronous interface always expects a result, even when it fails. When we try to use supervision, we are missing the sender details to correctly service the request. And we can't use the restart hook either, because the supervisor can decide to resume after an exception, which doesn't result in calling the restart hooks. Therefore, we are catching the exception and are able to return the expected response. Having said this, let's test our Consumer.

Listing 13.4 TCP test using Order Confirm Consumer

```

val probe = TestProbe()
val camelUri =
  "mina:tcp://localhost:8888?textline=true" ❶
val consumer = system.actorOf(
  Props(new OrderConfirmConsumerXml(camelUri, probe.ref)))
val activated = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.result(activated, 5 seconds)
val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
  <customerId>{ msg.customerId }</customerId>
  <productId>{ msg.productId }</productId>
  <number>{ msg.number }</number>
</order>
val xmlStr = xml.toString().replace("\n", "")
val sock = new Socket("localhost", 8888)
val ouputWriter = new PrintWriter(sock.getOutputStream, true)
ouputWriter.println(xmlStr)
ouputWriter.flush()
val responseReader = new BufferedReader(
  new InputStreamReader(sock.getInputStream))
var response = responseReader.readLine() ❷
response must be("<confirm>OK</confirm>")
probe.expectMsg(msg) ❸
responseReader.close()
ouputWriter.close()
system.stop(consumer)

```

- ❶ Remove the sync parameter, default is true
- ❷ Receive the confirmation message
- ❸ And the Order is still received

We hardly changed the consumer and were able to generate responses over TCP. Most of the functionality is done by the Camel module (which uses the camel components).

USING THE CAMEL CONTEXT

There is one other example we want to show. Sometimes a Camel component needs more configuration than only a URI.

For example when we want to use the ActiveMQ component. To be able to use this we need to add the component to the Camel context and define the MQ broker. This requires the camel context.

Listing 13.5 Add broker configuration to CamelContext

```
val camelContext = CamelExtension(system).context
camelContext.addComponent("activemq", ❶
  ActiveMQComponent.activeMQComponent(
    "vm:(broker:(tcp://localhost:8899)?persistent=false)"))
```

❶ Component name should be used in the Uri

First we get the CamelExtension for the used system and then we add the ActiveMQ component to the CamelContext. In this case we create a broker that listens on port 8899 (and don't use persistence queues).

Now we can do the test. For this example, we use the first Consumer without a response.

Listing 13.6 Test when using ActiveMQ

```

val camelUri = "activemq:queue:xmlTest" ❶
val consumer = system.actorOf(
  Props(new OrderConsumerXml(camelUri, probe.ref)))
val activated = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
Await.result(activated, 5 seconds)
val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
  <customerId>{ msg.customerId }</customerId>
  <productId>{ msg.productId }</productId>
  <number>{ msg.number }</number>
</order>
sendMQMessage(xml.toString())
probe.expectMsg(msg)
system.stop(consumer)

```

- ❶ The ActiveMQ URI starting with the same as the component name when adding the ActiveMQ component.

The test isn't any different from the other consumer test, other than how the message is delivered.

Because a broker is started, we also need to stop them when we are ready. This can be done using the `BrokerRegistry` of ActiveMQ

```

val brokers = BrokerRegistry.getInstance().getBrokers
brokers.foreach { case (name, broker) => broker.stop() }

```

Using the `BrokerRegistry`, we can close all the brokers. Note `getBrokers` returns a `java.util.Map`. We are using the `collection.JavaConversions` to convert this map into a Scala Map.

As you can see, it is very simple to implement a Camel Consumer. And because Camel has a lot of components, this gives us the ability to support many transport protocols without any effort.

13.2.2 Implement a producer endpoint sending messages to an external System

In the previous section, we created an endpoint that receives messages. In this section, we are going to implement the functionality to send messages using Camel. To show the producer functionality, we move to the other side of our example: with the consumer we were working on a endpoint at the Order System, but for these examples we are going to implement an endpoint in the Bookshop application, see Figure 13.10.

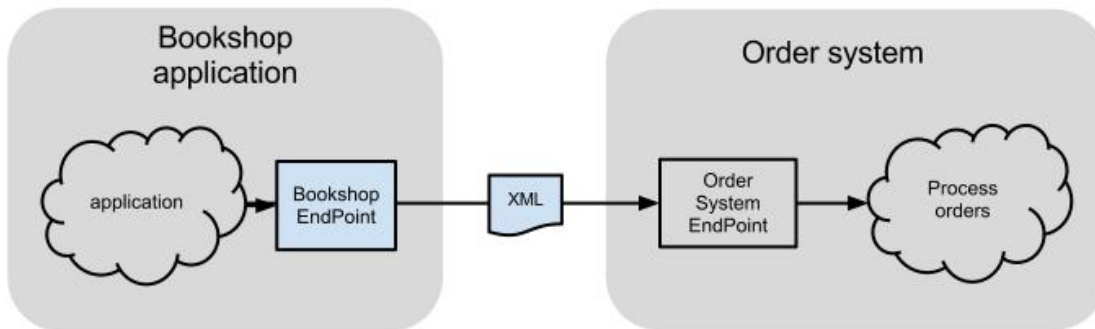


Figure 13.10 Producer endpoint sending messages

To implement a producer, the Camel module has another trait we will extend, named `Producer`. The producer is also an `Actor`, but the `receive` method is already implemented. The simplest implementation is just to extend the `Producer` trait and set the `Uri`, as shown in Figure 13.11.

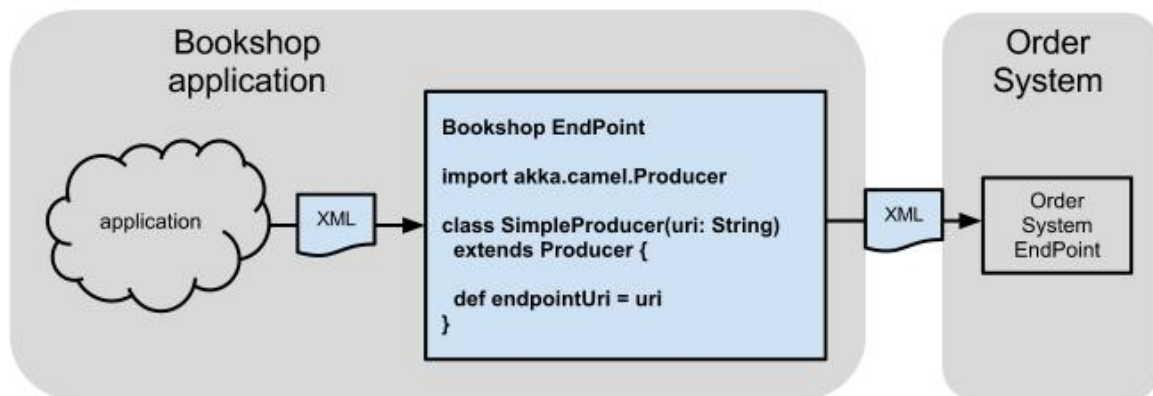


Figure 13.11 Implementation of a simple Producer endpoint

This `Producer` sends all received messages to the Camel component defined by

the Uri. So when we create an XML string and send it to the Producer, it can be sent using a TCP connection. In this example we use our consumer from the previous section to receive the message. And because we now have two Camel actors, we can't start the test until both Actors are ready. To wait for both we use the `Future.sequence` method. This uses a list of Futures we want to wait for (and need), and an implicit `ExecutionContext`.

Listing 13.7 Test simple producer

```
implicit val ExecutionContext = system.dispatcher
val probe = TestProbe()
val camelUri =
  "mina:tcp://localhost:8888?textline=true&sync=false"
val consumer = system.actorOf(
  Props(new OrderConsumerXml(camelUri, probe.ref)))
val producer = system.actorOf(
  Props(new SimpleProducer(camelUri))) ❶
val activatedCons = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
val activatedProd = CamelExtension(system).activationFutureFor(
  producer)(timeout = 10 seconds, executor = system.dispatcher)
val camel = Future.sequence(List(activatedCons, activatedProd)) ❷
Await.result(camel, 5 seconds)
```

- ❶ Create the simple producer
- ❷ Create a Future to wait for both actors to finish starting up.

So every message will be sent to the defined URI. But most of the time you need to translate the message to another format. In our shop system, we use the `Order` object when sending messages between the system actors. To solve this we can override the `transformOutgoingMessage`. This method is called before sending the message. Here we can do the translation of our message to the expected XML

Listing 13.8 Translate message in Producer

```
class OrderProducerXml(uri: String) extends Producer {
  def endpointUri = uri
  override def oneway: Boolean = true ❶

  override protected def transformOutgoingMessage(message: Any): Any = ❷
  {
    message match {
      case msg: Order => {
        val xml = <order>
          <customerId>{ msg.customerId }</customerId>
          <productId>{ msg.productId }</productId>
          <number>{ msg.number }</number>
        </order>

        xml.toString().replace("\n", "") ❸
      }
      case other => message
    }
  }
}
```

- ❶ indicate that we don't expect a response
- ❷ Implementing the transformOutgoingMessage
- ❸ create message ended with a newline

In the transformOutgoingMessage we create an XML string and, just as in the consumer test, we need a message on a single line ended with a new line. Because our Consumer does not send a response, we need to signal the underlying framework that it doesn't need to wait for one. Otherwise, it will be consuming resources for no reason. And it is possible that we could consume all the threads, which will stop your system. So it is important to override the oneway attribute when there are no responses.

Now we are able to send an Order object to the Producer endpoint and the producer translates this into an XML. But what happens when we do have responses? For example when we use the OrderConfirmConsumerXML. Figure 13.12 shows the default behavior of a producer that sends the received CamelMessage, which contains the XML response to the original sender.

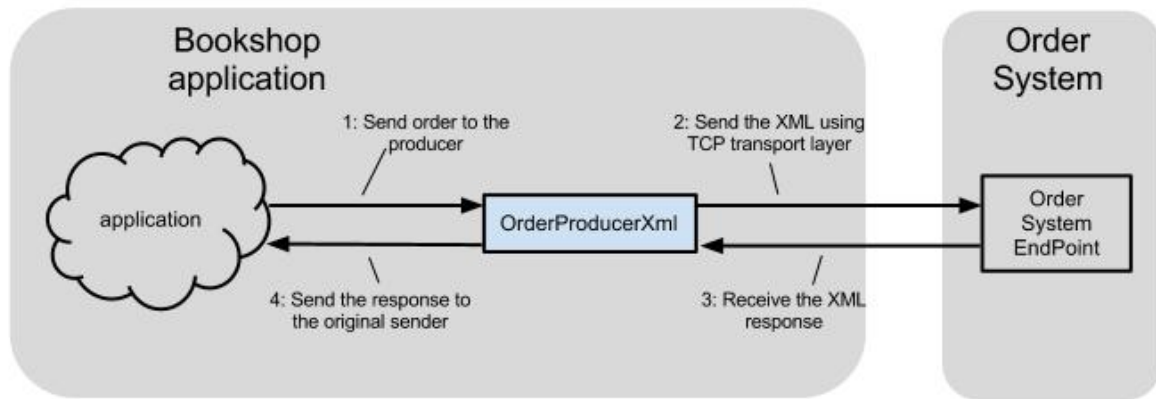


Figure 13.12 Using responses with the Camel Producer

But just as we need to translate our message when sending it, we need also a translation of the response. We don't want to expose the `CamelMessage` to the rest of our system. To support this we can use the `transformResponse` method. This method is used to convert the received message into a system-supported message, and the producer will send this response.

Listing 13.9 Translate responses and message in Producer

```

class OrderConfirmProducerXml(uri: String) extends Producer {
  def endpointUri = uri
  override def oneway: Boolean = false

  override def transformOutgoingMessage(message: Any): Any = {
    message match {
      case msg: Order => {
        val xml = <order>
          <customerId>{ msg.customerId }</customerId>
          <productId>{ msg.productId }</productId>
          <number>{ msg.number }</number>
        </order>
        xml.toString().replace("\n", "") + "\n"
      }
      case other => message
    }
  }

  override def transformResponse(message: Any): Any = { ❶
    message match {
      case msg: CamelMessage => {
        try {
          val content = msg.bodyAs[String]
          val xml = XML.loadString(content)
          (xml \ "confirm").text
        } catch {
          case ex: Exception =>
            "TransformException: %s".format(ex.getMessage)
        }
      }
      case other => message
    }
  }
}

```

- ❶ Transform the CamelMessage into a String containing the result.

The `transformResponse` is called when a response is received, before it is sent to the sender of the initial request. In this example, we parse the received XML and select the value of the `confirm` tag. Let's see how this works in a test.

Listing 13.10 Test the Producer with responses

```

implicit val ExecutionContext = system.dispatcher
val probe = TestProbe()
val camelUri =
  "mina:tcp://localhost:8889?textline=true"
val consumer = system.actorOf(
  Props(new OrderConfirmConsumerXml(camelUri, probe.ref)))
val camelProducerUri = "mina:tcp://localhost:8889?textline=true"
val producer = system.actorOf(
  Props(new OrderConfirmProducerXml(camelProducerUri)))
val activatedCons = CamelExtension(system).activationFutureFor(
  consumer)(timeout = 10 seconds, executor = system.dispatcher)
val activatedProd = CamelExtension(system).activationFutureFor(
  producer)(timeout = 10 seconds, executor = system.dispatcher)
val camel = Future.sequence(List(activatedCons, activatedProd))
Await.result(camel, 5 seconds)
val msg = new Order("me", "Akka in Action", 10)
producer ! msg
probe.expectMsg(msg) ❶
expectMsg("OK") ❷
system.stop(producer)
system.stop(consumer)

```

- ❶ Message is received by Consumer
- ❷ Confirmation is received by our test class

This is nice, but I don't want to send the confirmation to the original sender of the request, but to another actor. Is this possible? Yes, there is a method called `routeResponse`. This method is responsible for sending the received response to the original sender. This can be overridden and here you can implement the functionality to send the message to another actor. But be careful when you are also using the `transformResponse` method: you have to call it in this overridden method because the default implementation is calling the `transformResponse` before sending the response message to the original sender.

As you can see, creating producers is as easy as creating consumers. Camel provides a lot of functionality when creating an endpoint, and support for a lot of transport protocols. This is the greatest benefit of using the Camel module for Akka: to get support for a lot of protocols without additional effort.

In the next section, we are going to look at two examples of consumer endpoints that contain the actual connection to the Order system for creating a response.

13.3 Example of implementing a REST interface

In the previous sections we have seen how Camel can help us to implement endpoints. But because in the course of implementing a real system, all kinds of little problems arise; Camel isn't the only framework that can help us. Actually Akka is designed in such way that it can be integrated with any framework. So you will be able to use your preferred framework or the one you know best. Next, we are going to show an example of how to use Akka in a different environment: using a REST interface. REST is a standard protocol to expose intuitive interfaces to systems. We are still creating an endpoint for our system. In section 4.3.2 we have already implemented a REST interface using play-mini. This framework is based on the play framework, targeting services that don't have a UI. As we already mentioned, we are going to show two other possible implementations. The first is using Camel, which will be a minimal implementation of a REST interface, but when you need more functionality and are designing a REST interface with a lot of messages, Camel may be a little too minimalist. In these cases, Spray can help. Spray is an open-source toolkit for REST/HTTP and low-level network IO on top of Scala and Akka. The examples we show will be simple, but it addresses the issues of general integration techniques. We start with the example description and then we show you the two implementations. Starting with Camel and then Spray

13.3.1 The REST example

We are going to implement our Order System example again. But this time we are also implementing a Mockup of our processing Order system. So we can see how the endpoint forwards the request to the system and waits for the response before returning its response. We are going to do this by implementing one endpoint which uses the REST transport protocol. The overview of this example is shown in Figure 13.13.

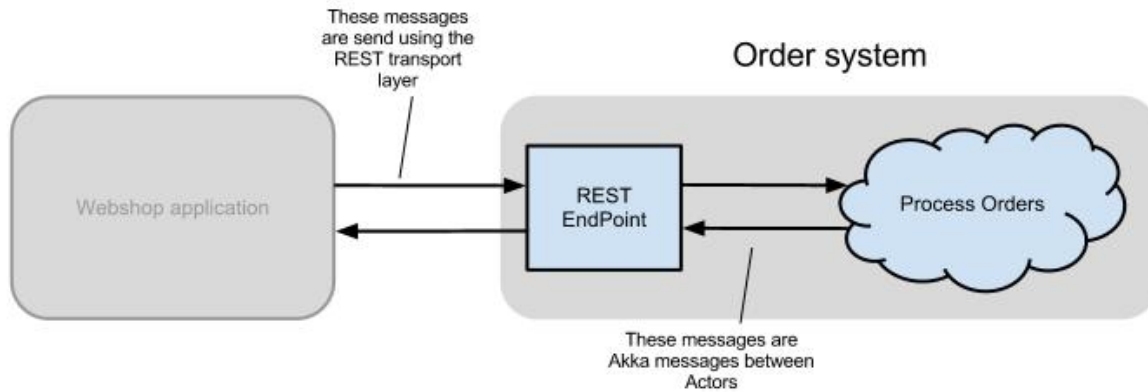


Figure 13.13 REST Example overview

The example has two interfaces: one between the Web shop and the endpoint and one between the endpoint and the order processor. We start by defining the messages for both interfaces. The Order system will support two functions. The first function is to add a new order and the second function is to get the status of an order. The REST interface we are going to implement supports a POST and a GET. With the POST, we add a new order to our system and with the GET we retrieve the status of that order. Lets start with adding an order. Figure 13.14 shows the messages and the flow.

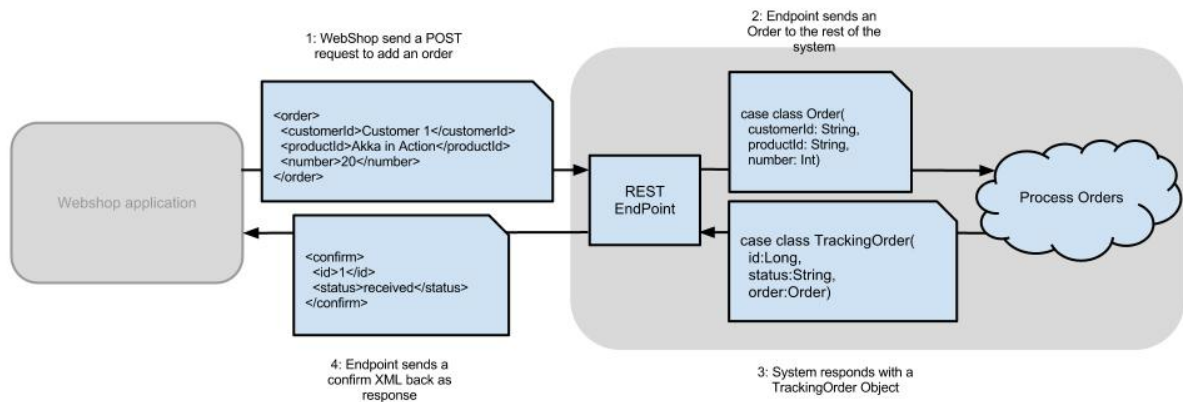


Figure 13.14 Message flow when adding an order

The Web shop sends a POST request to the endpoint containing the XML already used in the Camel examples in section 13.2.1. And the endpoint translates it to the Order message and sends it to the rest of the system (Process Orders). When done, the response is a TrackingOrder object, which contains the order, a unique id, and the current status. The Endpoint translates this to a confirm XML containing the Id and status and sends it back to the Web shop. In this example, the new order got the id 1 and the status 'received.'

The next Figure 13.15 shows the messages when getting the status of an order already in the order system.

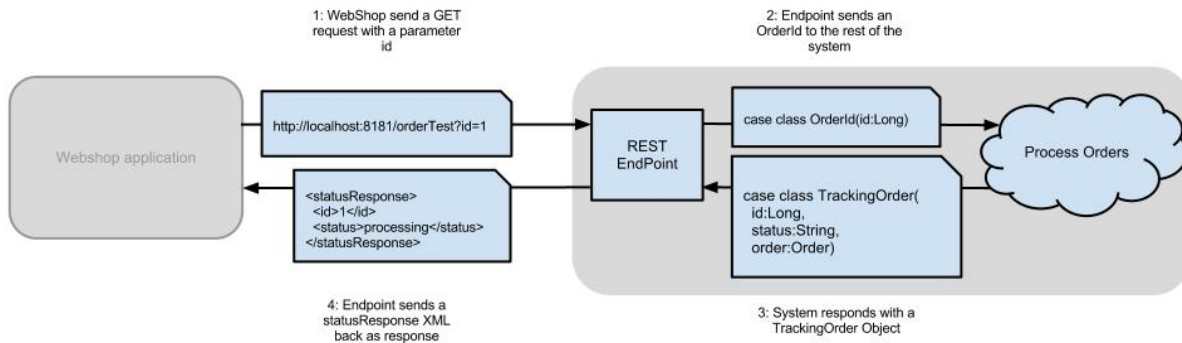


Figure 13.15 Message flow when getting the status of an order

To get the status, the Web shop will send a GET request with a parameter id, which contains the requested order id, in this case 1. This is translated to an OrderId. The response of the system is again a TrackingOrder Message when the order is found. The endpoint translates this response into a statusResponse XML. When the Order isn't found, the system will respond with a NoSuchOrder object, shown in Figure 13.16.

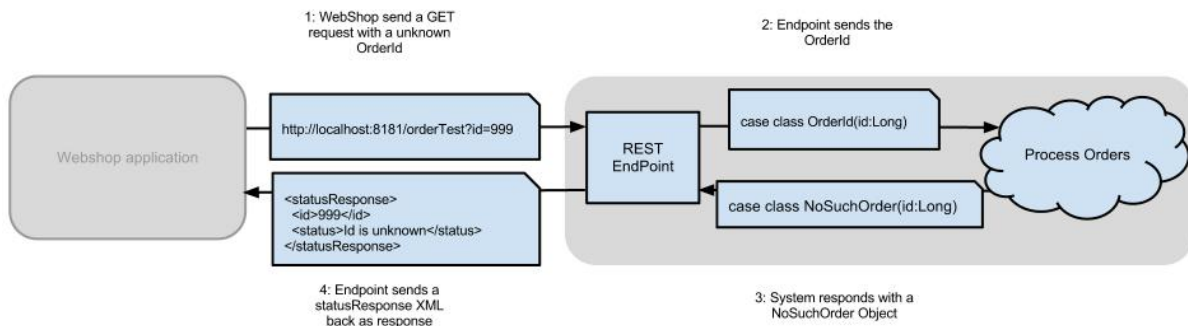


Figure 13.16 Message flow when trying to get the status of an unknown order

When the endpoint receives the NoSuchOrder message, the status of the XML response will be "id is unknown" and the id is filled with the unknown OrderId. Now that we have defined the messages sent through the system, we are ready to implement the Process Orders component. Figure 13.17 shows the implementation of the interface we just defined.

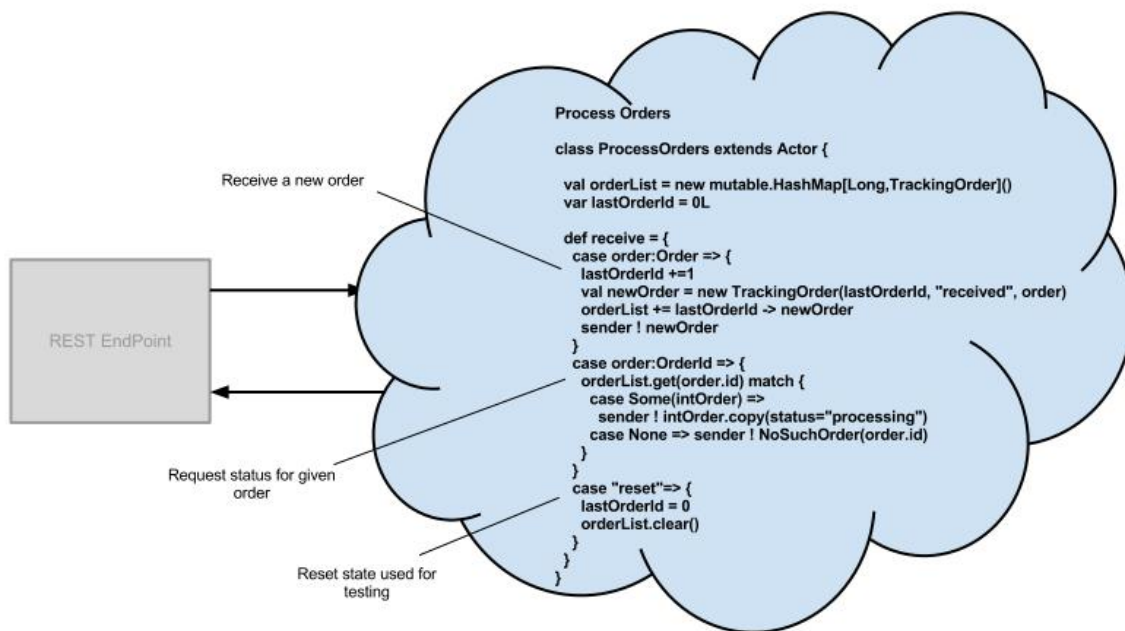


Figure 13.17 Implementation of the process orders

This is a simple representation of a complete system that implements two possible requests. We added also a reset function which can be used while testing the complete system.

Now we are ready to implement the REST endpoint. As we mentioned earlier we show two examples of how to do that using different frameworks. We start by using the Camel framework.

13.3.2 Implementing a Rest endpoint with Spray

We have implemented a REST interface with Camel in the previous section. Another framework which is often used in combination with Akka is the Spray toolkit. This is also a lightweight and modular toolkit. So just as is the case with Akka, you only have to include the parts of Spray you actually use. Spray also has its own test kit and is able to test your code without building a complete application. When you need REST/HTTP support, Spray is a great way to connect your Akka applications to other Systems.

To give you a feeling of how Spray can help you in implementing a REST interface, we are going to implement the same example REST endpoint using it. But keep in mind, this is only a small part of Spray, there is much more.

As we mentioned earlier, the Spray toolkit uses Actors to implement its functionality. Therefore we start by creating an Actor which extends the spray `HttpService`. A good practice is to separate the route definitions and the actual actor, because this enables you to test the routes without starting the actor. For this,

Spray has its own test kit which enables you to test only the route. But in our example we want to use the complete actor environment. Figure 13.18 shows both the Actor and the trait containing the route.

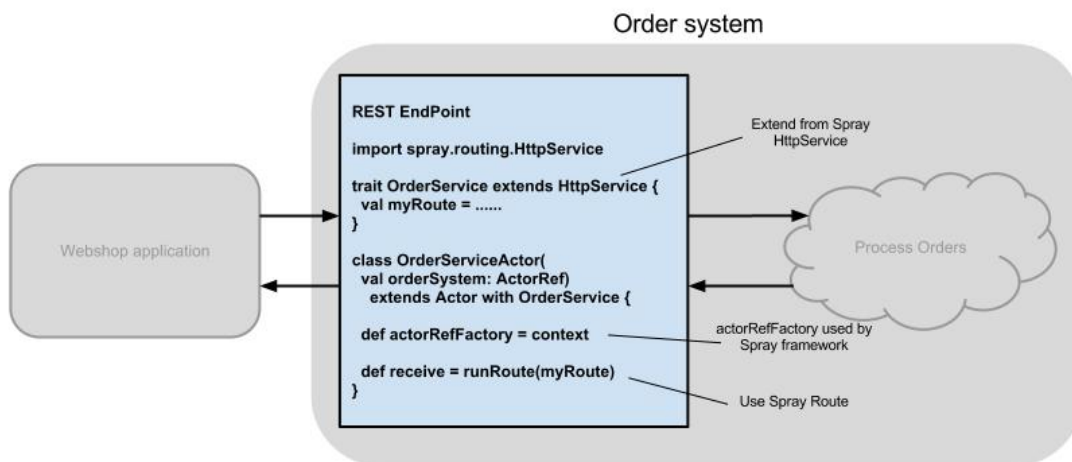


Figure 13.18 Endpoint implementation using Spray

The trait `OrderService` contains the route, which defines the REST interface, The `OrderServiceActor` is the placeholder and parent Actor of the processing of the Spray components. To make it work we need to set the `actorRefFactory` of the `HttpService`. This is the actor context, which is used when creating the necessary spray components. And we need to implement the `receive` method. Here we define which route we are going to use. This is all we need to do in the `ServiceActor`. Our `OrderService` extends the `HttpService` and contains all the functionality of our REST implementation.

In the trait we define a route. This is the REST interface and defines which messages we accept. Defining the route is done by using Directives. You can see our Directive: as a rule the received messages should match. A directive has one or more of the following functions

- Transform the request
- Filter the request
- Complete the request

Directives are small building blocks out of which you can construct arbitrarily complex route and handling structures. The generic form is:

```
name(arguments) { extractions => ... // inner route }
```

Spray has a lot of predefined Directives and of course, you can create custom directives. We are using some of the most basic and common directives in this example. We start with the path directive. This defines the Path of the request we are going to implement.

```
path("orderTest") {
  ...
}
```

Just as in the Camel example, we are going to implement the request path /orderTest. And we are going to support the POST and the GET requests. This is done by the next two directives the get and post directive. These directives are placed in the inner route of the path.

```
path("orderTest") {
  get {
    ...
  } ~
  post {
    ...
  }
}
```

1 The ~ appends multiple directives

The get and post take no arguments so the braces can be left out. And because we append two directives we need to add the ~. When we forget this ~ the compiler will not complain, but every Directive after this position will not be included in the route and therefore, when sending a POST request, it will not be processed as intended; every request that doesn't conform to this definition will get the "File not found" response. When a request with another path like "order" or method DELETE the framework will respond with the "file not found" error.

At this point we can implement our request. We start with the implementation of the get method. This request needs the parameter id. This can also be retrieved by using a directive.

```
path("orderTest") {
  get {
    parameter('id.as[Long]) { id =>
      val orderId = new OrderId(id)
      ...
    }
  }
}
```

The parameter directive retrieves the parameter id from the request and converts it to a long. When the GET request doesn't contain the id parameter, the selection fails and therefore the "file not found" error is sent back. When we have the id, we can create our business Object OrderId, which we will proceed to send on to our system. But before we describe this next step, we are going to show an even easier way to create our OrderId. When using a case class as business object, Spray is able to create the OrderId for us. The parameters must have the same name as the arguments of our case class. When we do this, we can create our business object using the following

```
path("orderTest") {  
  get {  
    parameters('id.as[Long]).as(OrderId) { orderId =>  
      ...  
    }  
  }  
}
```

This code is doing the same as the previous example. Now that we have our OrderId we can send the message to our system and create the response when the reply is received. This is done by using the complete directive.

Listing 13.11 Implementation GET status request

```

path("orderTest") {
  get {
    parameters('id.as[Long]).as(OrderId) { orderId =>
      complete {
        val askFuture = orderSystem ? orderId ❶
        askFuture.map {
          case result:TrackingOrder => { ❷
            <statusResponse>
              <id>{result.id}</id>
              <status>{result.status}</status>
            </statusResponse>
          }
          case result:NoSuchOrder => { ❸
            <statusResponse>
              <id>{result.id}</id>
              <status>ID is unknown</status>
            </statusResponse>
          }
        }
      }
    }
  }
}

```

- ❶ Send OrderId to process orders System
- ❷ Translate TrackingOrder response to XML
- ❸ Translate NoSuchOrder response to XML

The complete directive returns the response from the request. In the simplest implementation, the result is returned directly. But in our case, we need to wait for the reply from our system before we can create the response. Therefore, we return a Future which will contain the response of the request. We send the OrderId to our order system and use the map method on the received Future to create the response. Remember, the code block of the map method is executed when the Future finishes, which isn't in the current thread, so be careful what references you use. By returning XML, Spray sets the content type of the response automatically to text/xml. This is all there is to implementing the GET method.

Next, we start to implement the POST request. This is almost the same as the GET implementation. The only difference is that we don't need a parameter of the query, but need the body of the post. To do this, we use the entity directive

```

post {

```

```
entity(as[String]) { body =>
  val order = XMLConverter.createOrder(body.toString)
  ...
}
```

Now that we have our order object we can implement the response of the POST request. When we finish our created trait we get the following

Listing 13.12 Implementation OrderService

```

trait OrderService extends HttpService {
  val orderSystem: ActorRef

  implicit val timeout: Timeout = 1 second

  val myRoute = path("orderTest") { ❶
    get { ❷
      parameters('id.as[Long]).as(OrderId) { orderId => ❸
        //get status
        complete { ❹
          val askFuture = orderSystem ? orderId
          askFuture.map { ❺
            case result: TrackingOrder => {
              <statusResponse>
                <id>{ result.id }</id>
                <status>{ result.status }</status>
              </statusResponse>
            }
            case result: NoSuchOrder => {
              <statusResponse>
                <id>{ result.id }</id>
                <status>ID is unknown</status>
              </statusResponse>
            }
          }
        }
      }
    } ~ ❿
  }

  post { ❻
    //add order
    entity(as[String]) { body => ❼
      val order = XMLConverter.createOrder(body.toString)
      complete { ❽
        val askFuture = orderSystem ? order
        askFuture.map { ❾
          case result: TrackingOrder => {
            <confirm>
              <id>{ result.id }</id>
              <status>{ result.status }</status>
            </confirm>.toString()
          }
          case result: Any => {
            <confirm>
              <status>
                Response is unknown{ result.toString() }
              </status>
            </confirm>.toString()
          }
        }
      }
    }
  }
}

```


- Bind our service with the Http server

When we want to run it in a web server, we have to extend a similar class `WebBoot`, which will be created by the `spray.servlet.Initializer`. This and the servlet `spray.servlet.Servlet30ConnectorServlet` have to be added to the `web.xml` file. But for more details, look at the Spray web site.

But for our test we are using the `SprayCanHttpServerApp` trait.

Listing 13.14 Implementation of test boot class

```
class OrderHttpServer(host: String, portNr: Int, orderSystem: ActorRef)
  extends SprayCanHttpServerApp {

  //create and start our service actor
  val service = system.actorOf(Props( ❶
    new OrderServiceActor(orderSystem)), "my-service")

  //create a new HttpServer using our handler tell it where to bind to
  val httpServer = newHttpServer(service) ❷
  httpServer ! Bind(interface = host, port = portNr) ❸

  def stop() { ❹
    system.stop(httpServer)
    system.shutdown()
  }
}
```

- ❶ Start our service
- ❷ Create Http server
- ❸ Bind our service to the Http server
- ❹ Add Stop method to be able to stop the server while testing

In this class we added a stop method to be able to gracefully shutdown.

Now we have our REST implementation and can start a server, we are able to do our tests. To build up our test environment we have to do the following

```
val orderSystem = system.actorOf(Props[OrderSystem])
val orderHttp = new OrderHttpServer("localhost", 8181, orderSystem)
```

And at this point we can do the same tests as we did with the Camel implementation.

Listing 13.15 Test post and get request

```

orderSystem ! "reset"
val url = "http://localhost:8181/orderTest"
val msg = new Order("me", "Akka in Action", 10)
val xml = <order>
    <customerId>{ msg.customerId }</customerId>
    <productId>{ msg.productId }</productId>
    <number>{ msg.number }</number>
</order>
val urlConnection = new URL(url)
val conn = urlConnection.openConnection()
conn.setDoOutput(true)
conn.setRequestProperty("Content-type",
    "text/xml; charset=UTF-8")
val writer = new OutputStreamWriter(conn.getOutputStream) ①
writer.write(xml.toString())
writer.flush()
//check result
val reader = new BufferedReader(
    new InputStreamReader((conn.getInputStream)))
val response = new StringBuffer()
var line = reader.readLine()
while (line != null) {
    response.append(line)
    line = reader.readLine()
}
writer.close()
reader.close()
conn.getHeaderField(null) must be("HTTP/1.1 200 OK") ②
val responseXml = XML.loadString(response.toString) ③
val confirm = responseXml \ "confirm"
(confirm \ "id").text must be("1")
(confirm \ "status").text must be("received")
val url2 = "http://localhost:8181/orderTest?id=1"
val urlConnection2 = new URL(url2) ④
val conn2 = urlConnection2.openConnection()
//Get response
val reader2 = new BufferedReader(
    new InputStreamReader((conn2.getInputStream)))
val response2 = new StringBuffer()
line = reader2.readLine()
while (line != null) {
    response2.append(line)
    line = reader2.readLine()
}
reader2.close()
//check response
conn2.getHeaderField(null) must be("HTTP/1.1 200 OK") ⑤
val responseXml2 = XML.loadString(response2.toString)
val status = responseXml2 \ "statusResponse" ⑥

```

```
(status \ "id").text must be("1")
(status \ "status").text must be("processing")
```

- ① Send a post Request
- ② Check the status code of the response
- ③ Check the content of the response
- ④ Send a get status Request
- ⑤ Check the status code of the response
- ⑥ Check the content of the response

And our Spray implementation is working just like our Camel implementation. As we mentioned before this is only a small part of the capabilities of Spray. When you need to create a REST/HTTP interface, you have to take a look at Spray.

13.4 Summary

System Integrations tend to require many of the things that Akka offers out of the box:

- Asynchronous, message-based tasks
- Easy ability to provide data conversion
- Service production/consumption

We pulled in Spray and Camel to make the REST easy, which allowed us to focus on implementing many of the typical integration patterns using just Akka and not writing a lot of code that was tied to the chosen transports or component layers.

Akka brings a lot to the party on the System Integration front. In addition to the topics covered here: the blocking and tackling of consuming services, getting data, converting it, and the providing it to other consumers, the core aspects of the Actor Model, concurrency and fault tolerance, represent critical contributors to making the integrated system reliable and scalable. It's easy to imagine expanding any of our pattern examples here to include some of the replaceability we saw in the fault tolerance chapter, and the scaling abilities from chapter 6. Quite often, this is the most onerous aspect of Integration: dealing with the pressure of real flows going in and out, against performance constraints and reliability requirements.

14

Clustering

In this chapter

- Cluster Membership
- Cluster Aware Routers
- Cluster Patterns

In chapter 6 you learned how to build your first distributed application with a fixed number of nodes. The approach we took, using static membership, is simple but provides no out-of-the-box support for load balancing or fail over. A cluster makes it possible to dynamically grow and shrink the number of nodes used by a distributed application, and removes the fear of a single point of failure.

Many distributed applications run in environments that are not completely under your control, like cloud computing platforms or data centers located across the world. The larger the cluster, the greater the chance of failure. Of course, despite this, there are complete means of monitoring and controlling the lifecycle of the cluster. In the first section of this chapter, we'll look at how a node becomes a member of the cluster, how you can listen to membership events, and how you can detect that nodes have crashed in the cluster.

First we're going to build a clustered actor system to make a word count table from a piece of text. Within the context of this example, you will learn how routers can be used to communicate with actors in the cluster, how you can build a resilient coordinated process consisting of many actors in the cluster, as well as how to test a clustered actor system.

14.1 Why use Clustering?

A Cluster is a dynamic group of nodes. On each node is an actor system that listens on the network (like we saw in chapter 6). Cluster builds on top of the akka-remote module. Clustering takes location transparency to the next level. The actor might exist locally or remotely and could reside anywhere in the cluster, your code does not have to concern itself with this. Figure 1.1 shows a cluster of 4 nodes:

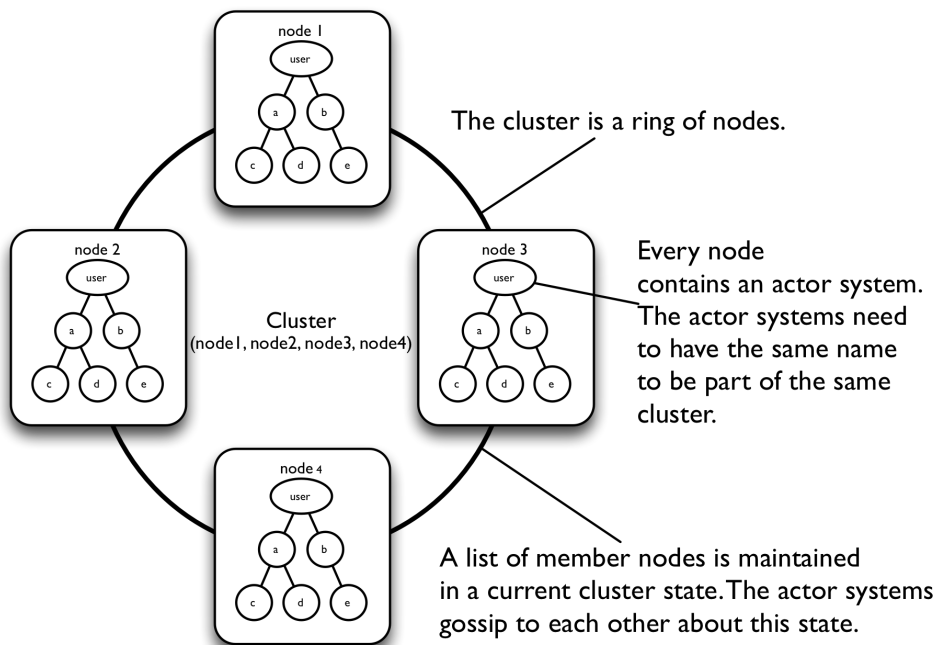


Figure 14.1 A 4 node clustered actor system

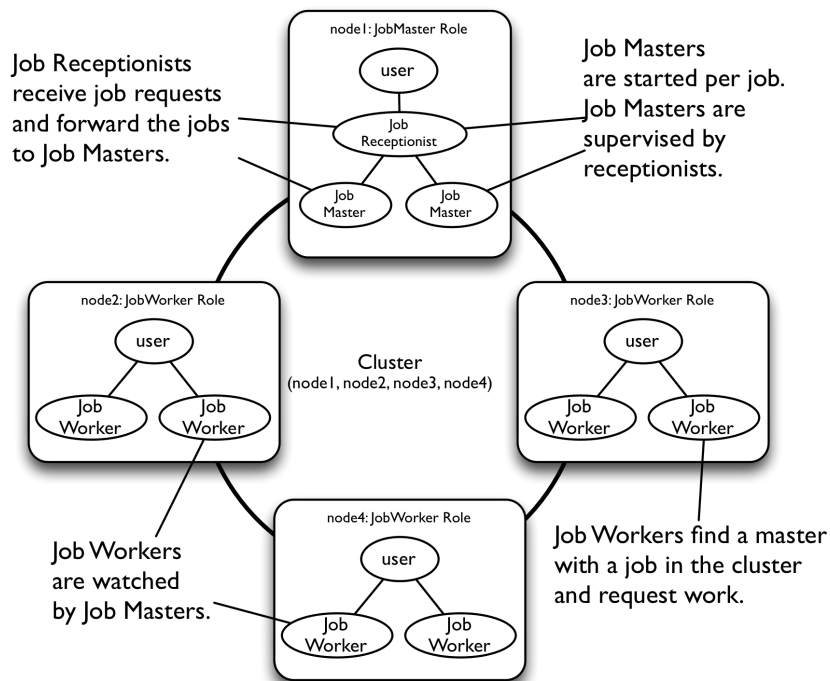
The ultimate goal for the Cluster module is to provide fully automated features for actor distribution, load balancing and fail over. Right now the cluster module supports the following features:

- Cluster membership - Fault tolerant membership for *Actor* systems.
- Load balancing - routing messages to actors in the cluster based on a routing algorithm.
- Node partitioning - A node can be given a specific *Role* in the cluster. Routers can be configured to only send messages to nodes with a specific role.
- Partition points - An actor system can be partitioned in actor sub-trees that are located on different nodes. Right now only top level *partition points* are supported. This means that you can only access top level actors on nodes in the cluster using Routers.

We will dive into the details of these features in this chapter.

Although these features do not provide everything we need for a fully location transparent cluster (like failover, re-balancing, re-partitioning and replication of state) they do provide the means to scale applications dynamically.

A single purpose data processing application is the best candidate for using cluster right now, for instance data processing tasks like image recognition or real-time analysis of social media. Nodes can be added or removed when more or less processing power is required. Processing Jobs are supervised: if an actor fails, the job is restarted and retried on the cluster until it succeeds. We will look at a simple example of this type of application in this chapter. Figure 1.2 shows an overview for this type of application, don't worry about the details here, because we will introduce the terms you may not be familiar with later in this chapter:



Let's get on to writing the code to compile our clustered word count application. In the next section, we'll dig into the details of cluster membership so that the Job Masters and Workers can find each other to work together.

14.2 Cluster Membership

We will start with the creation of the cluster. The processing cluster will consist of job master and worker nodes. Figure 1.3 shows the cluster that we are going to build:

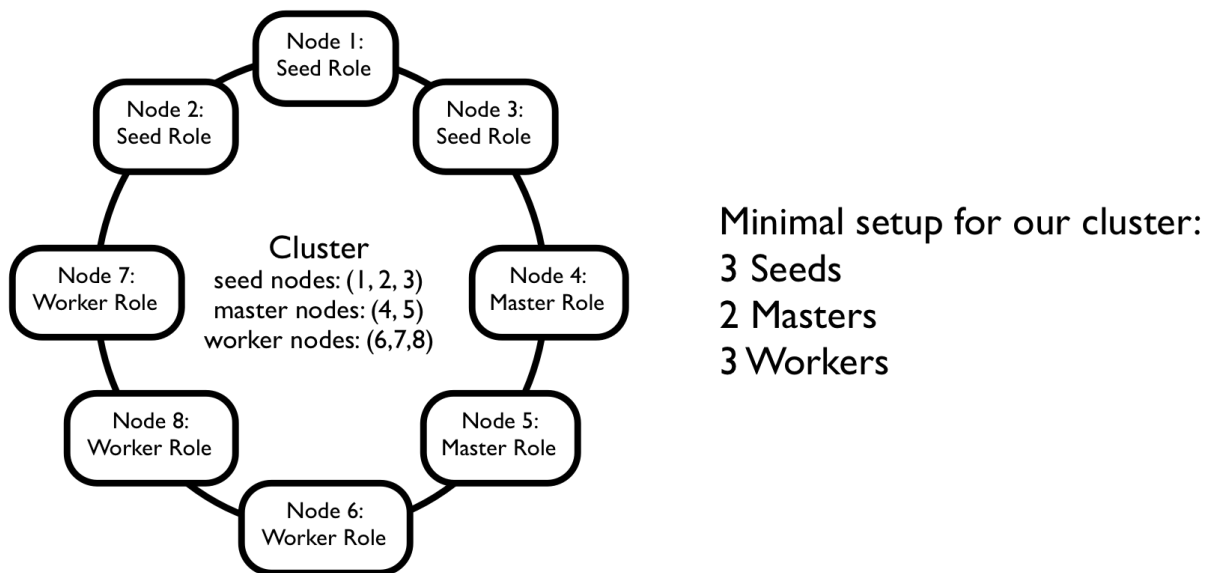


Figure 14.2 Words counting Cluster

The job master nodes control and supervise the completion of word counting jobs. The job workers request work from a job master and process parts of the text and return the partial results to the master. The job master reports the result once all word counting has been done. A job is repeated if any master or worker node fails during the process.

Figure 1.3 also shows another type of node that will be required in the cluster, namely *seed nodes*. The seed nodes are essential for starting the cluster. In the next section we will look at how nodes become seed nodes and how they can join and leave the cluster. We will look at the details of how a cluster is formed and experiment with joining and leaving a very simple cluster using the REPL console. You will learn about the different states that a member node can go through and how you can subscribe to notifications of these state changes.

14.2.1 Joining the cluster

Like with any kind of group you need a couple of 'founders' to start off the process. Akka provides a *seed node* feature for this purpose. Seed nodes are both the starting point for the cluster and they serve as the first point of contact for other nodes. Nodes join the cluster by sending a *Join* message which contains the unique address of the node that joins. The Cluster module takes care of sending this message to one of the registered seed nodes. It is not required for a node to contain any actors so it is possible to use pure seed nodes. Figure 1.4 shows how a first seed node initializes a cluster and how other nodes join the cluster:

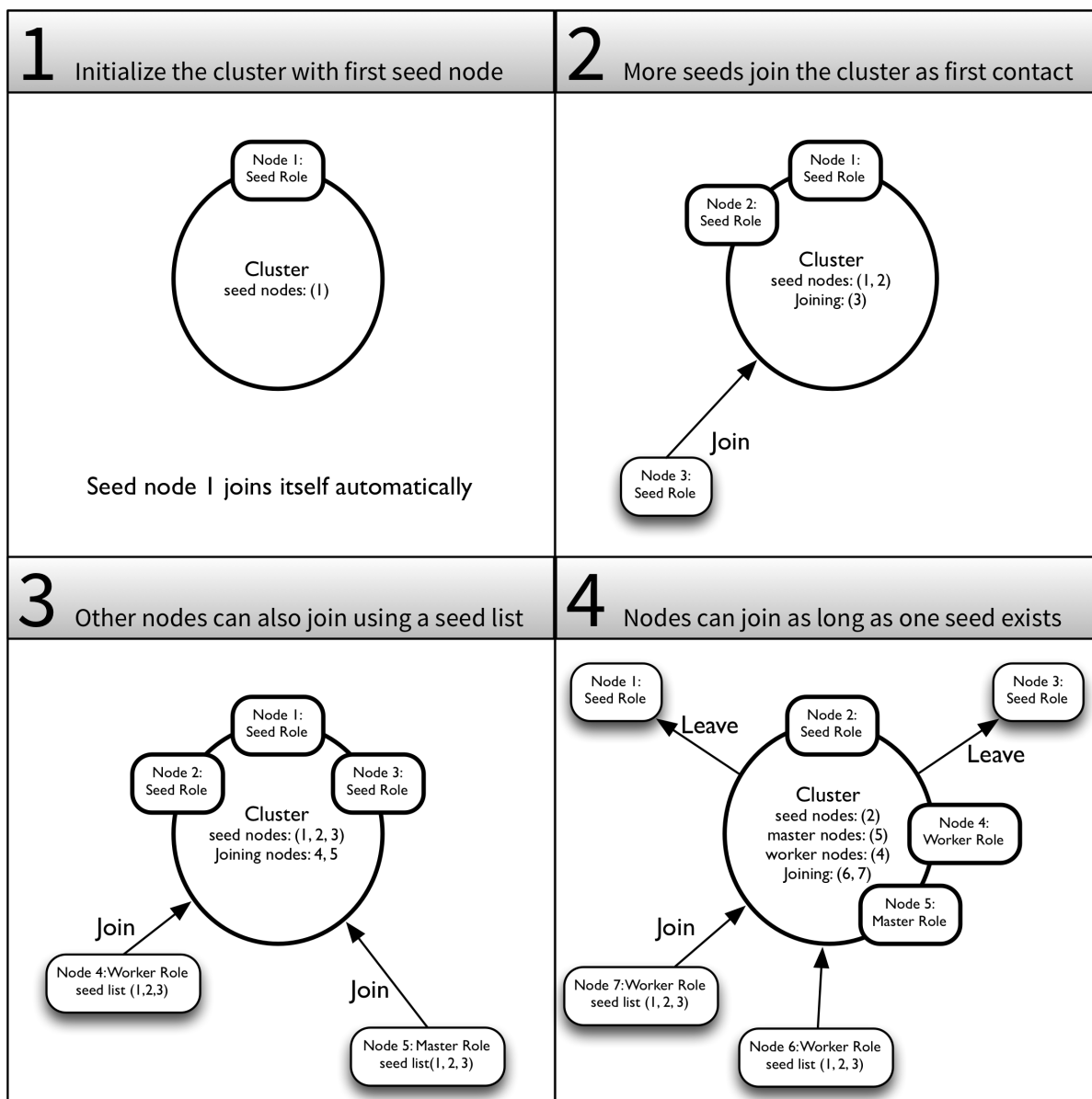


Figure 14.3 Initializing a cluster with seed nodes

Cluster does not (yet) support a zero-config discovery protocol like TCP-Multicast or DNS service discovery. You have to specify a list of seed nodes. The first seed node in the list has a special role in initially forming the cluster. Subsequent seed nodes are dependent on the first seed node in the list. The first node in the seed list starts up and automatically joins itself and forms the cluster. The first seed node needs to be up before next seed nodes can join the cluster. This constraint has been put in place to prevent separate clusters from forming while seed nodes are starting up.

NOTE**Manually joining cluster nodes**

The seed nodes feature is not required: you can create a cluster manually by starting a node that joins itself. Subsequent nodes will then have to join that node to join the cluster.

Which means that they will have to know the address of the first node so it makes more sense to use the seed functionality. There are cases where you cannot know IP addresses or DNS names of servers in a network beforehand. In that case, there are two choices that seem plausible:

- Use a list of known pure seed nodes with well known IP addresses or DNS names, outside of the network where host name addresses cannot be predetermined. These seed nodes do not run any application specific code and purely function as a first point of contact for the rest of the cluster.
- Get your hands dirty building your own cluster discovery protocol that fits your network environment. This is a non-trivial task.

The seed nodes can all boot independently as long as the first seed node in the list is started at some point. The subsequent seed nodes will wait for the first node to come up. Once this first node is started other nodes join the cluster. The first seed node can safely leave the cluster once the cluster has two or more members. Figure 1.5 shows an overview of how we're going to build a cluster of masters and workers after at least the first seed node has started:

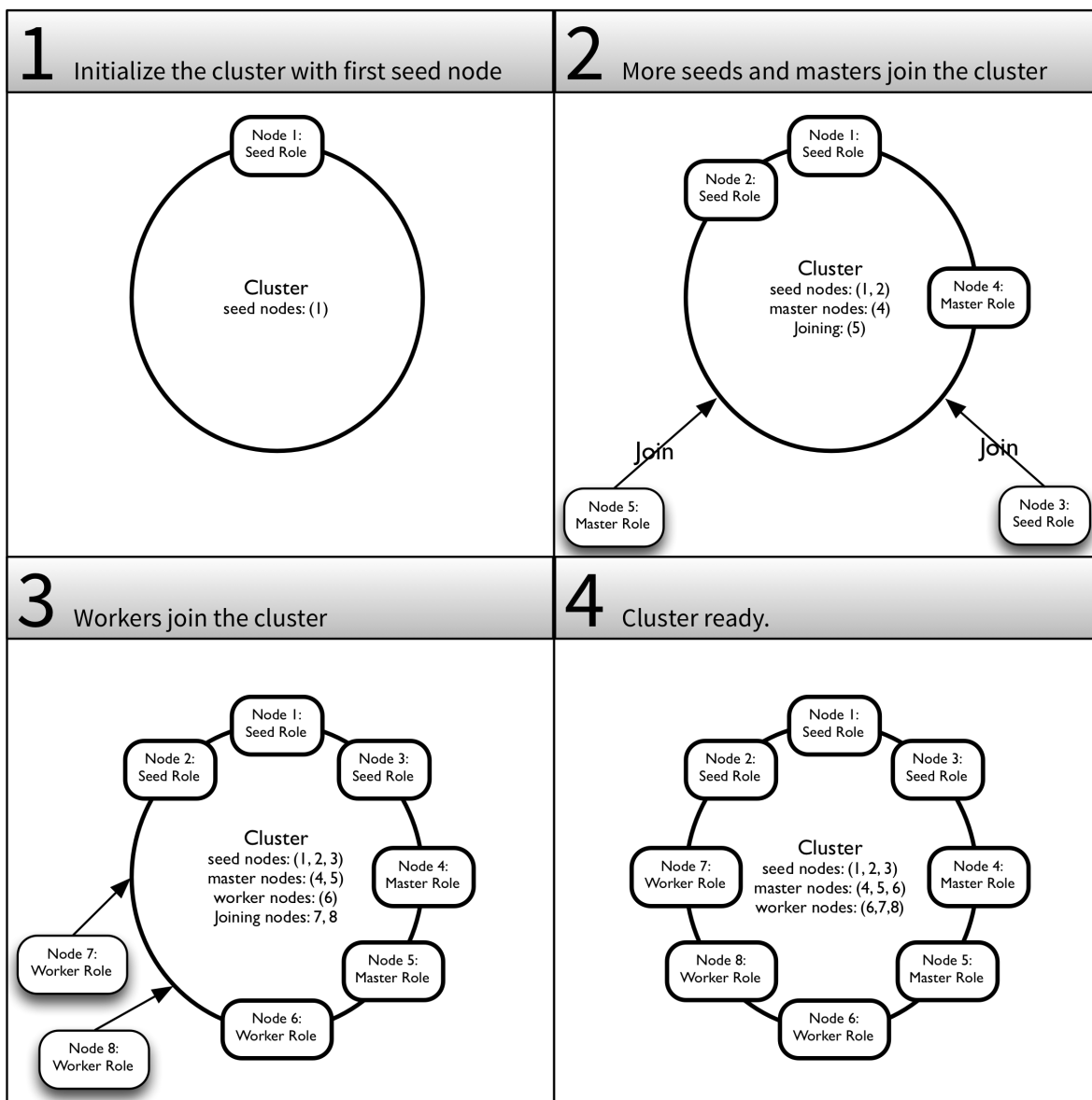


Figure 14.4 A job processing cluster

Let's start by creating the seed nodes using the REPL console. You can find the project for this example on the github repository under the *chapter-cluster* directory.

A node first needs to be configured to use the cluster module. The *akka-cluster* dependency needs to be added to the build file as shown below:

```
"com.typesafe.akka" %% "akka-cluster" % akkaVersion 1
```

1 The build file defines a val for the version of Akka

The *akka.cluster.ClusterActorRefProvider* needs to be configured in much the same way the remote module needed a *akka.remote.RemoteActorRefProvider*. The Cluster API is provided as an Akka Extension. The *ClusterActorRefProvider* initializes the Cluster extension when the actor system is created.

Listing 1.2 shows a minimal configuration for the seed nodes.

Listing 14.1 Configuring the seed nodes

```
akka {
  loglevel = INFO
  stdout-loglevel = INFO
  event-handlers = ["akka.event.Logging$DefaultLogger"]

  actor {
    provider = "akka.cluster.ClusterActorRefProvider" ❶
  }

  remote { ❷
    enabled-transport = ["akka.remote.netty.tcp"]
    log-remote-lifecycle-events = off
    netty.tcp {
      hostname = ""
      host = ${HOST}
      port = ${PORT}
    }
  }

  cluster {
    seed-nodes = ["akka.tcp://words@127.0.0.1:2551",
                 "akka.tcp://words@127.0.0.1:2552",
                 "akka.tcp://words@127.0.0.1:2553"] ❸
    roles = ["seed"] ❹
  }
}
```

- ❶ Initializes the cluster module.
- ❷ Remote configuration for this seed node.
- ❸ The seed nodes of the cluster.
- ❹ The seed node is given a seed role to differentiate from workers and masters.

We'll start all the nodes locally throughout these examples. If you want to test this on a network, just replace the `-DHOST` and `-DPORT` with the appropriate hostname and port respectively. Start `sbt` in three terminals using different ports. The first seed node in the list is started as shown below:

```
sbt -DPORT=2551 -DHOST=127.0.0.1
```

Do the same for the other two terminals, replacing the `-DPORT` to 2552 and 2553. Every node in the same cluster needs to have the same actor system name ("words" in the above example). Switch to the first terminal in which we'll start the first seed node.

The first node in the seed nodes must automatically start and form the cluster. Let's verify that in a REPL session, start the console in sbt in the first terminal started with port 2551 and follow along with listing 1.4. Figure 1.6 shows the result.

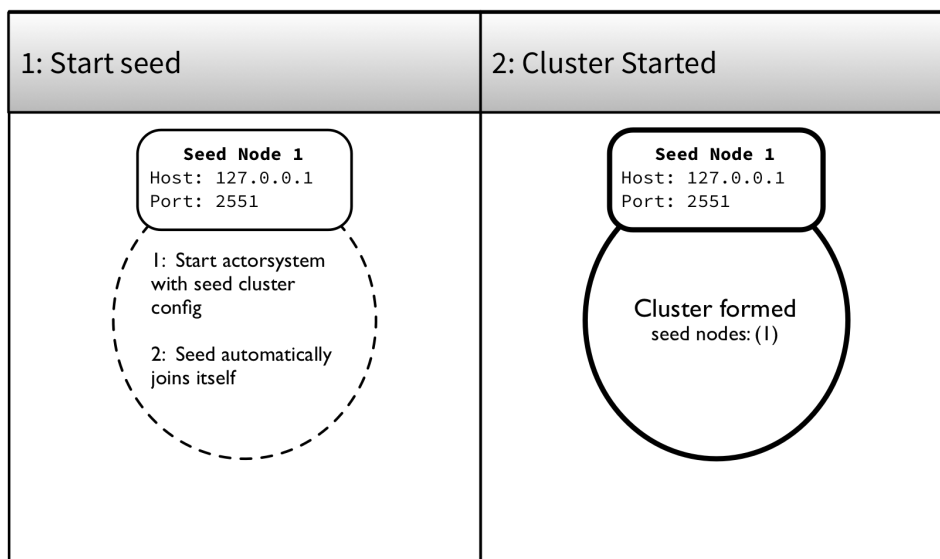


Figure 14.5 Startup the first seed node

Listing 14.2 Starting up a seed node

```

...
scala> :paste
// Entering paste mode (ctrl-D to finish)

import akka.actor._

import akka.cluster._

import com.typesafe.config._

val seedConfig = ConfigFactory.load("seed") ❶
val seedSystem = ActorSystem("words", seedConfig) ❷

// Exiting paste mode, now interpreting.

[Remoting] Starting remoting ❸
[Remoting] listening on addresses :
[akka.tcp://words@127.0.0.1:2551]
...
[Cluster(akka://words)] ❹
Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Started up successfully ❺
Node [akka.tcp://words@127.0.0.1:2551] is JOINING, roles [seed]
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Leader is moving node [akka.tcp://words@127.0.0.1:2551] to [Up] ❻

```

- ❶ Load the configuration for the seed node. It contains the config shown in Listing 1.2. This config file is located in `src/main/resources/seed.conf`
- ❷ Start the "words" actor system as seed node.
- ❸ Remote and cluster modules are automatically started. The console output is simplified to show the most relevant messages.
- ❹ The cluster name is the same as the name of the actor system.
- ❺ The "words" cluster seed node is started.
- ❻ The "words" cluster seed node has automatically joined the cluster.

Start the console on the other two terminals and paste in the same code as in Listing 1.4 to start seed node 2 and 3. The seeds will listen on the port that we provided as `-DPORT` when we started `sbt`. Figure 1.7 shows the result of the REPL commands for seed node 2 and 3.

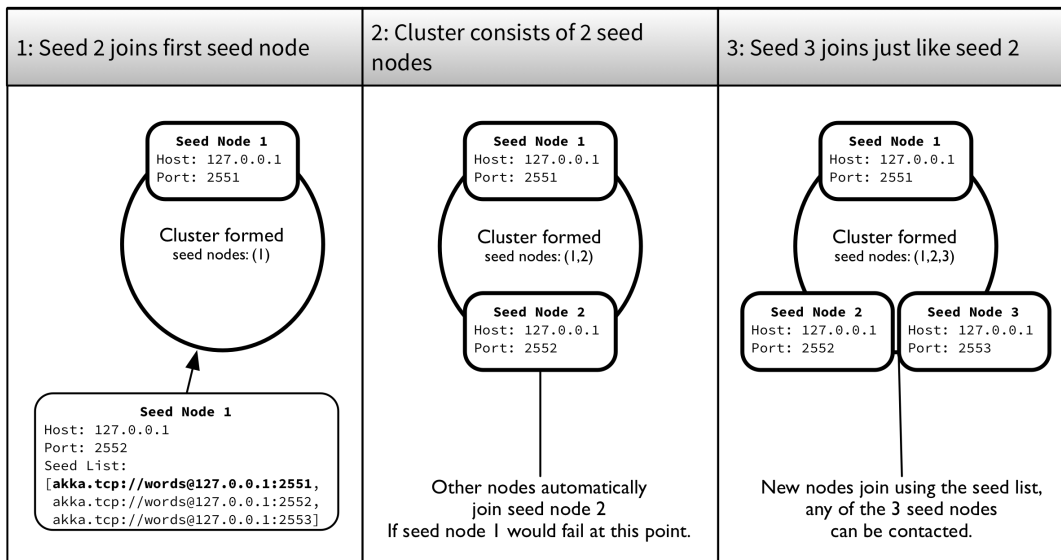


Figure 14.6 Startup the second seed node

You should see something similar to Listing 1.5 in the other two terminals, confirming that the nodes joined the cluster.

Listing 14.3 Seed node 3 confirming joining the cluster

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2553]
- Welcome from [akka.tcp://words@127.0.0.1:2551] ❶
```

❶ Output formatted for readability, will show as one line in the terminal

Listing 1.6 shows the output of the first seed node. The output shows that the first seed node has determined that the two other nodes want to join.

Listing 14.4 Terminal output of seed node 1

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551] ❶
- Node [akka.tcp://words@127.0.0.1:2551] is JOINING, roles [seed] ❷
- Leader is moving node [akka.tcp://words@127.0.0.1:2551] to [Up]
- Node [akka.tcp://words@127.0.0.1:2552] is JOINING, roles [seed] ❸
- Leader is moving node [akka.tcp://words@127.0.0.1:2552] to [Up]
- Node [akka.tcp://words@127.0.0.1:2553] is JOINING, roles [seed] ❹
- Leader is moving node [akka.tcp://words@127.0.0.1:2553] to [Up]
```

❶

- 1 Output abbreviated and formatted for readability.
- 2 The first seed node joins itself and becomes the leader.
- 3 Seed node 2 is joining.
- 4 Seed node 3 is joining.

One of the nodes in the cluster takes on special responsibilities; to be the *Leader* of the cluster. The leader decides if a member node is up or down. In this case the first seed node is the leader.

Only one node can be leader at any point in time. Any node of the cluster can become the leader. Seed node 2 and 3 both request to join the cluster, which puts them in the *Joining* state. The leader moves the nodes to the *Up* state, making them part of the cluster. All three seed nodes have now successfully joined the cluster.

14.2.2 Leaving the cluster

Let's see what happens if we let the first seed node leave the cluster. Listing 1.7 shows seed node 1 leaving the cluster:

Listing 14.5 Seed 1 leaving the cluster

```
scala> val address = Cluster(seedSystem).selfAddress 1
address: akka.actor.Address = akka.tcp://words@127.0.0.1:2551

scala> Cluster(seedSystem).leave(address) 2

[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Marked address [akka.tcp://words@127.0.0.1:2551] as [Leaving] 3
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Leader is moving node [akka.tcp://words@127.0.0.1:2551] to [Exiting] 4
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Shutting down...
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2551]
- Successfully shut down
```

- 1 Get the address for this node.
- 2 let seed node 1 leave the cluster
- 3 Marked as Leaving
- 4 Marked as Exiting

Listing 1.7 shows that seed node 1 marks itself as *Leaving*, then as *Exiting* while it is still the leader. These state changes are communicated to all nodes in the cluster. After that the cluster node is shutdown. The actor system itself (the *seedSystem*) is not shut down automatically on the node. What happens with the

cluster? The leader node just shut down. Figure 1.8 shows how the first seed node leaves the cluster and how leadership is transferred.

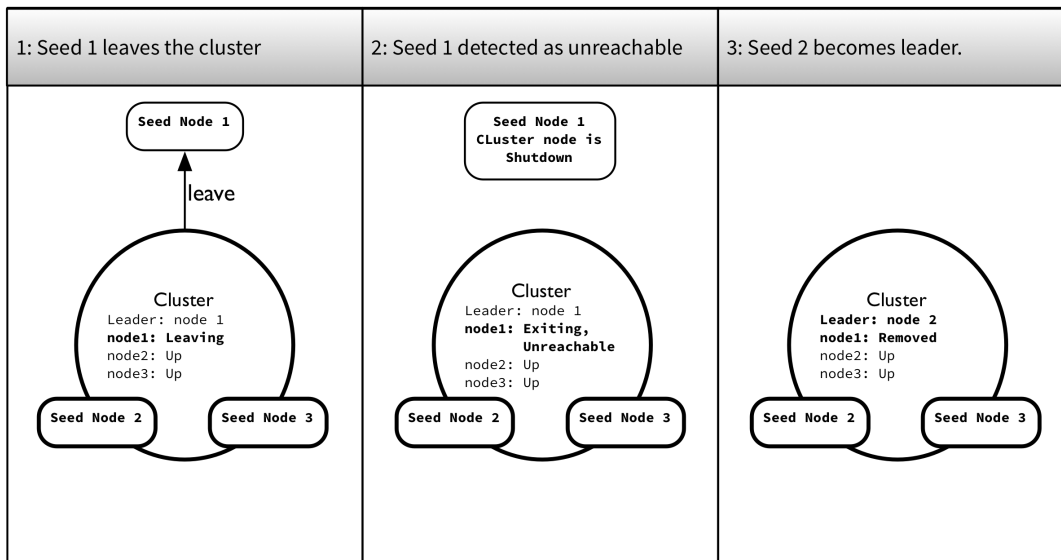


Figure 14.7 First seed node leaves the cluster

Let's look at the other terminals. One of the two remaining terminals should show output similar to Listing 1.8.

Listing 14.6 Seed 2 becomes leader and removes seed 1 from the cluster

```
[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Marking exiting node(s) as UNREACHABLE
[Member(address = akka.tcp://words@127.0.0.1:2551, status = Exiting)].
This is expected and they will be removed. ❶

[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Leader is removing exiting node [akka.tcp://words@127.0.0.1:2551]
```

- ❶ The exiting seed node has the *Exiting* state.
- ❷ The leader removes the exiting node.

Both remaining seed nodes detect that the first seed node has been flagged as *Unreachable*. Both seed nodes are also aware that the first seed node has requested to leave the cluster. The second seed node automatically becomes the leader when the first seed node is in an *Exiting* state. The leaving node is moved from an *Exiting* state to a *Removed* state. The cluster now consists of 2 seed nodes.

NOTE**Gossip Protocol**

You might have wondered how the seed nodes in the example knew about the fact that the first seed node was leaving, then exiting and was finally removed. Akka uses a *Gossip* protocol to communicate the state of the cluster to all member nodes of the cluster.

Every node gossips about its own state and the states that it has seen to other nodes (the gossip). The protocol makes it possible for all nodes in the cluster to eventually agree about the state of every node. This agreement is called *Convergence* which occurs over time while the nodes are gossiping to each other.

A leader for the cluster can be determined after convergence. The first node, in sort order, that is Up or Leaving automatically becomes the leader.

The actor system on the first seed node cannot join the cluster again by simply using `Cluster(seedSystem).join(selfAddress)`. The actor system is removed and can only join the cluster again if it is restarted. Listing 1.9 shows how the first seed node can 're-join':

Listing 14.7 Seed 2 becomes leader and removes seed 1 from the cluster

```
scala> seedSystem.shutdown ❶
scala> val seedSystem = ActorSystem("words", seedConfig) ❷
```

- ❶ Shutdown the actor system.
- ❷ Start a new actor system with the same configuration. The actor system automatically joins the cluster.

An actor system can only ever join a cluster once. A new actor system can be started with the same configuration, using the same host and port which is what is done in Listing 1.9.

So far we've looked at gracefully joining and leaving the cluster. Figure 1.9 shows a state diagram of the member states that we have seen so far. The leader performs a leader action at specific member states, moving a member from Joining to Up and from Exiting to Removed.

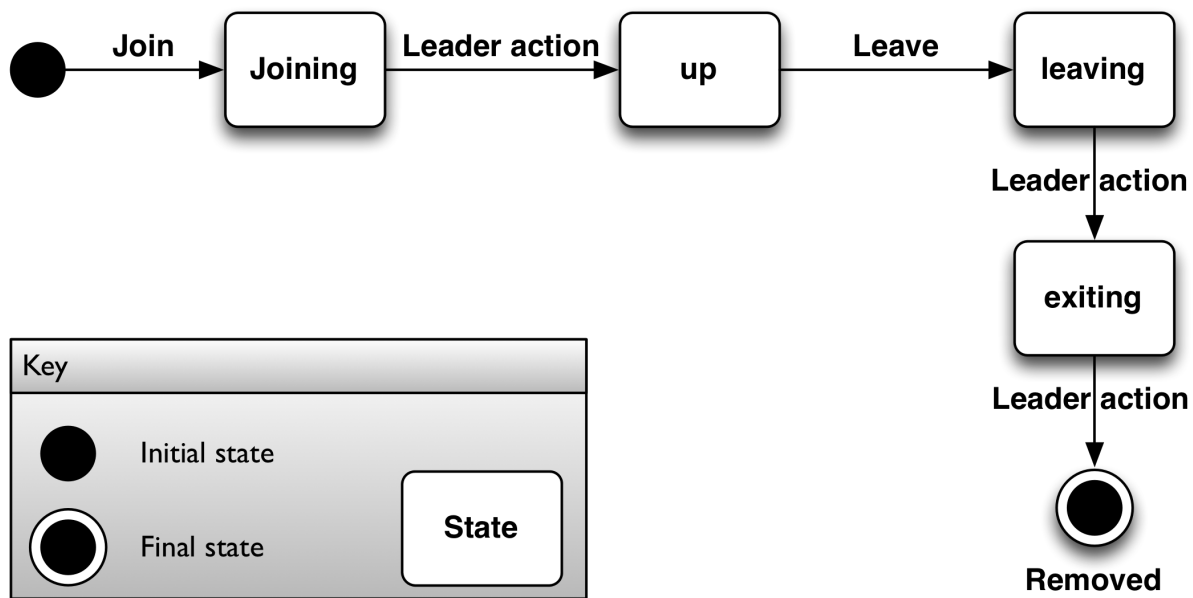


Figure 14.8 Graceful state transitions of a node joining and leaving the cluster

This is not the complete picture yet, Let's look at what happens if one of the seed nodes crashes. We can simply kill the terminal that runs seed node 1 and look at the output of the other terminals. Listing 1.10 shows the output of the terminal running seed node 2 when seed node 1 has been killed abruptly:

Listing 14.8 Seed 1 crashes

```

Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Marking node(s) as UNREACHABLE
  [Member(address = akka.tcp://words@127.0.0.1:2551, status = Up)] ❶
  
```

❶ Seed node 1 becomes unreachable.

Seed node 1 has been flagged as unreachable. The Cluster uses a failure detector to detect unreachable nodes. The seed node was in an Up state when it crashed. A node can crash in any of the states we've seen before. The leader can't execute any leader actions as long as any of the nodes are unreachable, which means that no node can leave or join. The unreachable node will first have to be taken down. You can take a node down from any node in the cluster using the `down` method. Listing 1.11 shows how the first seed node is downed from the REPL:

Listing 14.9 Taking down Seed 1 manually

```
scala> val address = Address("akka.tcp", "words", "127.0.0.1",2551)
scala> Cluster(seedSystem).down(address) ❶

[Cluster(akka://words)] Cluster Node [akka.tcp://words@127.0.0.1:2552]
- Marking unreachable node [akka.tcp://words@127.0.0.1:2551] as [Down]
- Leader is removing unreachable node [akka.tcp://words@127.0.0.1:2551]

[Remoting] Association to [akka.tcp://words@127.0.0.1:2551]
having UID [1735879100]
is irrecoverably failed. UID is now quarantined and
all messages to this UID
will be delivered to dead letters.
Remote actorsystem must be restarted to recover from this situation.
```

- ❶ Seed node 1 is down.
- ❷ Seed node 1 is quarantined and removed.

The output also shows that if the seed node 1 actor system would want to re-join it will have to restart. An unreachable node can also be taken down automatically. This is configured with the `akka.cluster.auto-down-unreachable-after` setting. The leader will automatically take unreachable nodes down after the set duration in this setting. Figure 1.10 shows all possible state transitions for a node in the cluster:

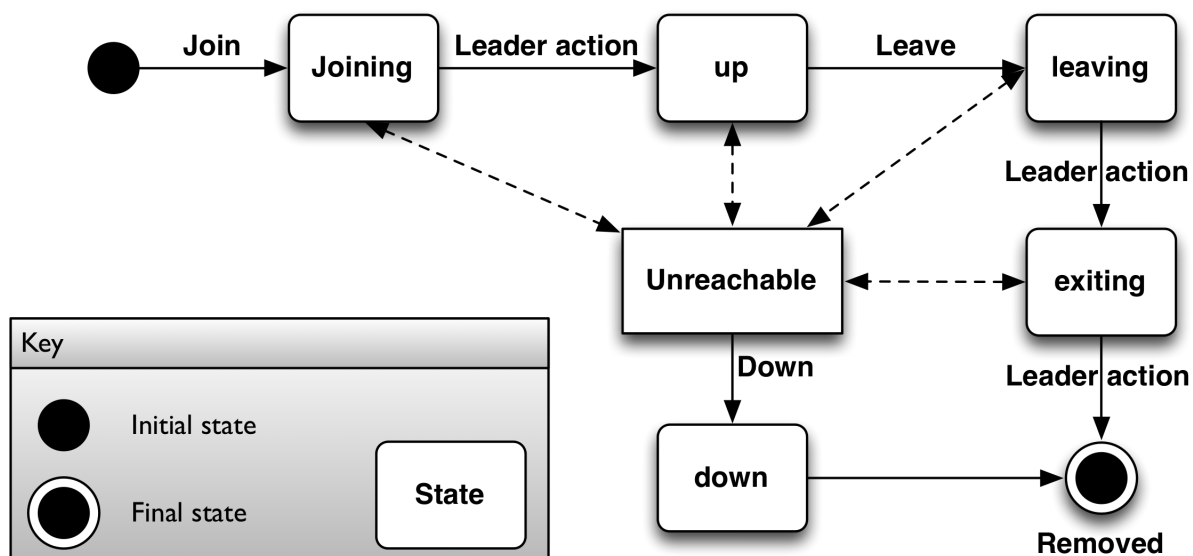


Figure 14.9 All States and transitions of a node

NOTE**Failure Detector**

The cluster module uses an implementation of a *Accrual Phi Failure Detector* to detect unreachable nodes. Detecting failures is a fundamental issue for fault-tolerance in distributed systems.

The Accrual Phi Failure Detector calculates a phi value on a continuous scale instead of determining a boolean value indicating failure (if the node is reachable or not). This phi value is used as an indicator for suspecting that something is wrong (a suspicion level) instead of determining a hard and fast yes or no result.

The suspicion level concept makes the failure detector tunable and allows for a decoupling between application requirements and monitoring of the environment. The cluster module provides settings for the failure detector which you can tune for your specific network environment in the *akka.cluster.failure-detector* section, amongst which a threshold for the phi value at which a node is deemed to be unreachable.

Another reason that occurs often is when a node is unreachable for a longer period of time because it is in a *GC-Pause* state, which means that it is taking far too long to finish garbage collection and cannot do anything else until GC has completed.

We would definitely want to be notified if any of the nodes in the cluster fails. You can subscribe an actor to cluster events using the `subscribe` method on the `Cluster` extension. Listing 1.12 shows an actor that subscribes to cluster domain events:

Listing 14.10 Subscribe to Cluster Domain Events

```

...
import akka.cluster.{MemberStatus, Cluster}
import akka.cluster.ClusterEvent._

class ClusterDomainEventListener extends Actor with ActorLogging {
  Cluster(context.system).subscribe(self, classOf[ClusterDomainEvent]) ❶

  def receive = {
    case MemberUp(member) => log.info(s"$member UP.")
    case MemberExited(member) => log.info(s"$member EXITED.")
    case MemberRemoved(m, previousState) =>
      if(previousState == MemberStatus.Exiting) {
        log.info(s"Member $m gracefully exited, REMOVED.")
      } else {
        log.info(s"$m downed after unreachable, REMOVED.")
      }
    case UnreachableMember(m) => log.info(s"$m UNREACHABLE")
    case ReachableMember(m) => log.info(s"$m REACHABLE")
    case s: CurrentClusterState => log.info(s"cluster state: $s")
  }

  override def postStop(): Unit = {
    Cluster(context.system).unsubscribe(self) ❷
    super.postStop()
  }
}

```

- ❶ Subscribes to the cluster domain events on actor creation
- ❷ Listen for cluster domain events
- ❸ Unsubscribe after the actor is stopped

The example `ClusterDomainEventListener` simply logs what has happened in the cluster.

The Cluster domain events tell you something about the cluster members but in many cases it suffices to know if an actor in the cluster is still there. We can simply use `DeathWatch` using the `watch` method to watch actors in the cluster as you'll see in the next section.

14.3 Clustered Job Processing

It's time to process some jobs with a cluster. We're going to focus first on how the actors in the cluster communicate with each other to complete a task. The cluster receives a text whose words we wish to count. The text is divided into pieces and delivered to several worker nodes. Every worker node processes its part by counting the occurrences of every word in the text. The worker nodes process the text in parallel, which should result in faster processing. Eventually the result of the counting is sent back to the user of the cluster. The fact that we're going to count the occurrences of words is of course not the focus; you can process many jobs in the way that is shown in this section.

The example can be found in the same chapter-cluster directory as used before for the examples on joining and leaving the cluster. Figure 1.11 shows the structure of the application:

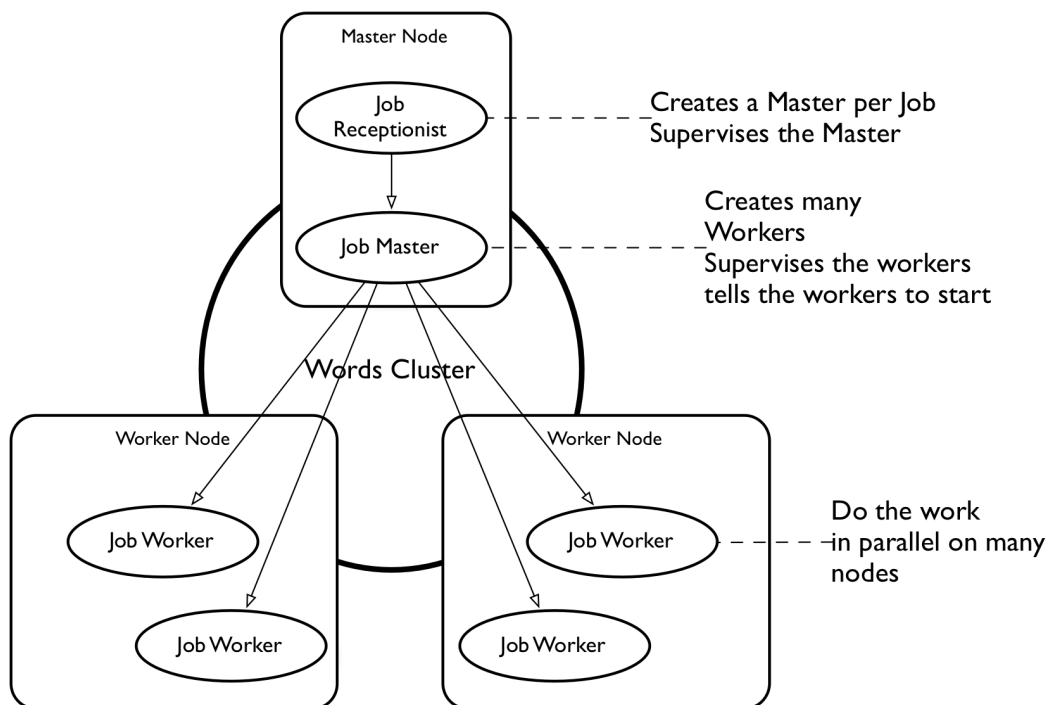


Figure 14.10 Words cluster actors

The JobReceptionist and JobMaster actors will run on a Master Role Node. The JobWorkers will run on a Worker Role node. Both JobMasters and JobWorker actors are created dynamically, on demand. Whenever a JobReceptionist receives a JobRequest it spawns a JobMaster for the Job and tells it to start work on the job.

The JobMaster creates JobWorkers remotely on the Worker Role nodes. Figure 1.12 shows an overview of the process. We will address each step in detail in the rest of this chapter.

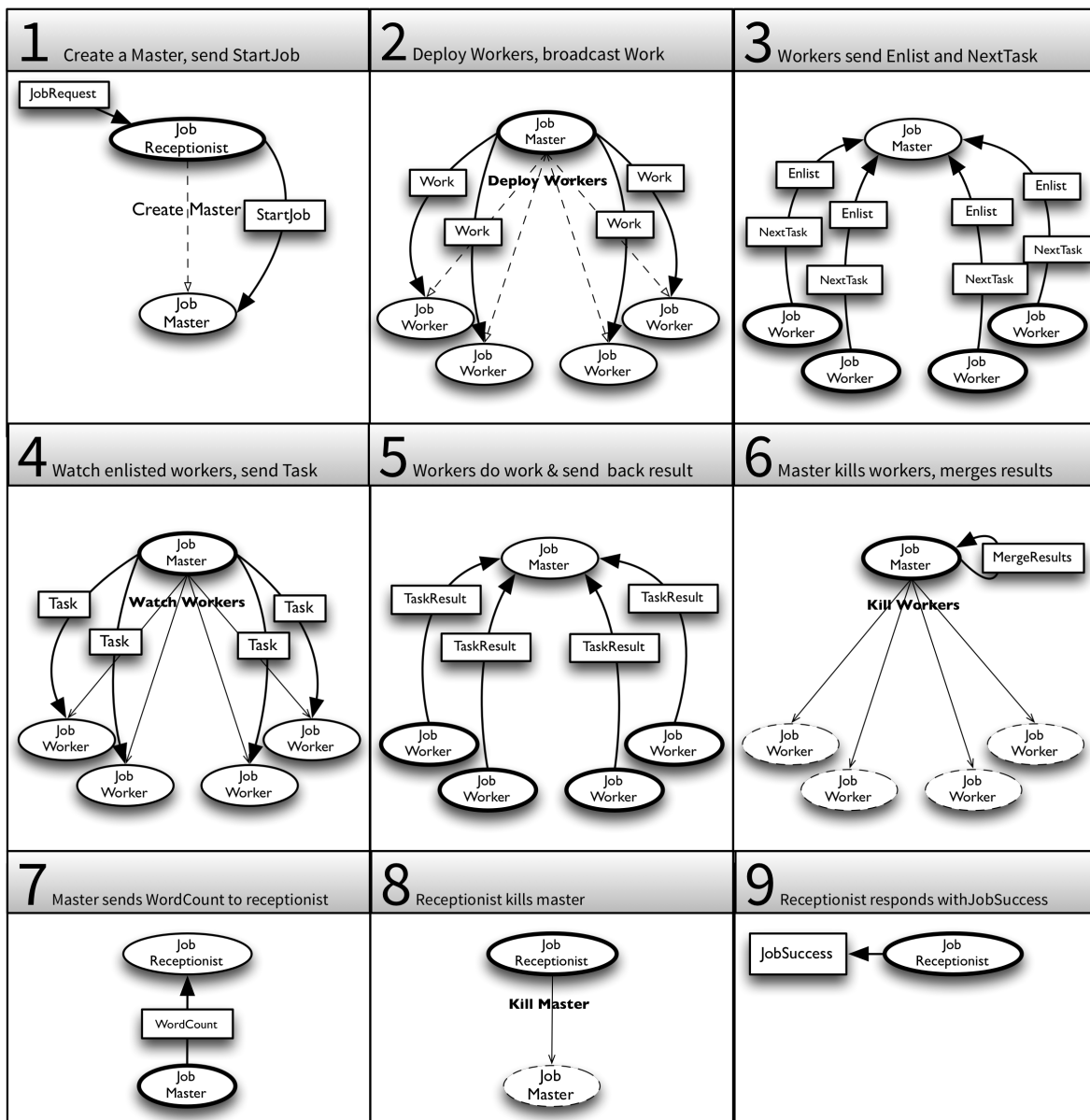


Figure 14.11 Job processing

Every `JobWorker` receives a `Task` message which contains a portion of the text. The `JobWorker` splits the text in words and counts the occurrence of every word, returning a `TaskResult` which contains a `Map` of word counts. The `JobMaster` receives the `TaskResults` and merges all the maps, adding up the counts for every word, which is basically the reduce step. The `WordCount` result is eventually sent back to the `Job Receptionist`.

In the next sections, we will address all the steps in the overview. First we will start the cluster, then we will distribute the work that has to be done between master and workers. After that we're going to look at how we can make the job processing resilient including restarting the job when a node crashes. Finally we'll address how to test the cluster.

NOTE

Some caveats for the example

This example is kept simple on purpose. The `JobMaster` keeps intermediate results in memory and all the data that is processed is sent between the actors.

If you have to deal with batch processing of extremely large amounts of data you need to put effort into getting the data close to the process before processing it, stored on the same server that the process is running on, and you can't simply collect data in memory. In, for instance Hadoop based systems, this means pushing all the data onto HDFS (Hadoop Distributed File System) before processing it and writing all the results back to HDFS as well. In our example, we will simply send the workload around in the cluster. The reduce step that adds up the results of all the workers is done by the master to simplify things a bit instead of having parallel reducers that start as soon as the first task has been completed.

It is possible to achieve all of this, but it's more than we can cover here in this chapter. Our example will show how to do resilient job processing and can be a starting point for more realistic cases.

14.3.1 Starting the Cluster

You can build the example in the `chapter-cluster` directory using `sbt assembly`. This creates a `words-node.jar` file in the target directory. The jar file contains three different configuration files, one for the master, one for the worker and one for the seed. Listing 1.13 shows how to run 1 seed node, one master and 2 workers locally on different ports:

Listing 14.11 Run nodes

```

java -DPORT=2551 \
    -Dconfig.resource=/seed.conf \
    -jar target/words-node.jar
java -DPORT=2554 \
    -Dconfig.resource=/master.conf \
    -jar target/words-node.jar
java -DPORT=2555 \
    -Dconfig.resource=/worker.conf \
    -jar target/words-node.jar
java -DPORT=2556 \
    -Dconfig.resource=/worker.conf \
    -jar target/words-node.jar

```

The `master.conf` and `worker.conf` file define a list of local seed nodes running on 127.0.0.1 and ports 2551, 2552 and 2553. The seed node list can also be configured using a system property.

NOTE**Overriding the seed node list from the command line**

You can override the seed nodes with `-Dakka.cluster.seed-nodes.[n]=[seednode]` where `[n]` needs to be replaced with the position in the seed list starting with 0 and `[seednode]` with the seed node value.

The master can't do anything without the workers so it would make sense to have the `JobRecipient` on the master start up only when the cluster has some minimum number of worker nodes running. You can specify the minimum number of members with a certain role in the cluster configuration. Listing 1.14 shows part of the `master.conf` file for this purpose:

Listing 14.12 Configure minimum number of worker nodes for MemberUp event.

```

role {
    worker.min-nr-of-members = 2
}

```

The configuration of the master specifies that there should be at least 2 worker nodes in the cluster. The cluster module provides a `registerOnMemberUp`

method to register a function that is executed when the member node is up, in this case the master node, which takes the minimum number of worker nodes into account. The function is called when the master node has successfully joined the cluster and when there are 2 or more worker nodes running in the cluster. Listing 1.15 shows the Main class that is used to start all types of nodes in the words cluster:

Listing 14.13 Configure minimum number of worker nodes for MemberUp event.

```
object Main extends App {
  val config = ConfigFactory.load()
  val system = ActorSystem("words", config)

  println(s"Starting node with roles: ${Cluster(system).selfRoles}")

  val roles = system.settings
    .config
    .getStringList("akka.cluster.roles")
  if(roles.contains("master")) { ❶
    Cluster(system).registerOnMemberUp { ❷
      val receptionist = system.actorOf(Props[JobReceptionist],
        "receptionist" ) ❸
      println("Master node is ready.")
    }
  }
}
```

- ❶ Only if this node has a master role.
- ❷ Register a code block to be executed when the member is up
- ❸ The JobReceptionist is only created when the cluster is up with at least 2 worker role nodes present.

The worker node does not need to start any actors, the JobWorkers are going to be started on demand as you will see in the next section. We will use a Router to deploy and communicate with the JobWorkers.

14.3.2 Work Distribution using Routers

The JobMaster needs to first create the JobWorkers and then broadcast the Work message to them. Using routers in the cluster is exactly the same as using routers locally. We just need to change how we create the routers. We'll use a Router with a BroadcastPool RouterConfig to communicate with the JobWorkers. A Pool is a RouterConfig that creates Actors where a Group is a RouterConfig that is used to route to already existing Actors as explained in the Routing chapter. In this case we want to dynamically create the JobWorkers and kill them after the job is done so a Pool is the best option. The JobMaster Actor uses a separate trait to create the Router. The separate trait will come in handy during testing as you will see later. The trait is shown in Listing 1.15, which creates the worker router:

Listing 14.14 Create a Clustered BroadcastPool Router

```

trait CreateWorkerRouter { this: Actor => ❶
  def createWorkerRouter: ActorRef = {
    context.actorOf(
      ClusterRouterPool(BroadcastPool(10), ❷
        ClusterRouterPoolSettings(
          totalInstances = 1000, ❸
          maxInstancesPerNode = 20, ❹
          allowLocalRoutees = false, ❺
          useRole = None
        )
      ).props(Props[JobWorker]), ❺
      name = "worker-router")
    }
  }
}

```

- ❶ Needs to mixin with an actor.
- ❷ The ClusterRouterPool takes a Pool.
- ❸ Total maximum number of workers in the cluster.
- ❹ Max number of workers per node.
- ❺ Do not create local routees. We only want Workers on the other nodes.
- ❻ Nodes with this role will be routed to!.
- ❼ Create JobWorkers with standard Props.

NOTE Router configuration

In this case the JobMaster is created dynamically for every Job so it needs to create a new router every time, which is why it is done in code. It is also possible to configure router deployment using the configuration as described in the Router chapter. You can specify a cluster section in the deployment configuration to enable the router for clustering and set the ClusterRouter Pool or Group settings, like *use-role* and *allow-local-routees*.

The `CreateWorkerRouter` trait only does one thing, create the router to the workers. Creating the clustered router is very similar to creating a normal router. All you need to do is pass in a `ClusterRouterPool` which can use any of the existing Pools, like `BroadcastPool`, `RoundRobinPool` and `ConsistentHashingPool` and the like. The `ClusterRouterPoolSettings` controls how instances of the JobWorkers are created. JobWorkers will be added to joining worker nodes as long as the `totalInstances` has not been reached yet. In the above configuration 50 nodes could join the cluster before the router stops deploying new JobWorkers. The JobMaster creates the router when it is created as shown in Listing 1.16 and uses it to send out messages to the workers, also shown in Figure 1.13:

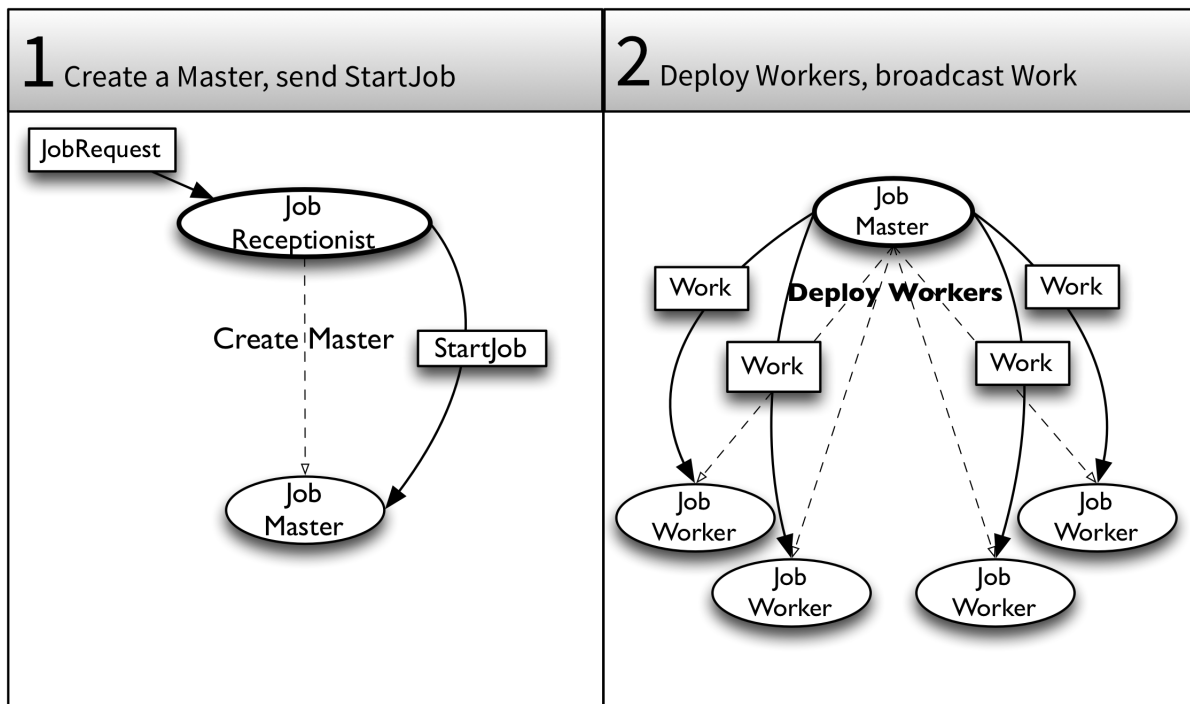


Figure 14.12 Deploy JobWorkers and broadcast Work messages

Listing 14.15 The JobMaster uses the router to broadcast Work messages to JobWorkers

```

class JobMaster extends Actor
    with ActorLogging
    with CreateWorkerRouter { ❶
  // inside the body of the JobMaster actor..
  val router = createWorkerRouter ❷

  def receive = idle

  def idle: Receive = {
    case StartJob(jobName, text) =>
      textParts = text.grouped(10).toVector
      val cancel = system.scheduler.schedule(0 millis,
                                             1000 millis,
                                             router, ❸
                                             Work(jobName, self))
      become(working(jobName, sender, cancel))
  }
  // more code

```

- ❶ Mix in the CreateWorkerRouter trait.
- ❷ Create the router.
- ❸ Schedule a message to the router.

The code snippet in Listing 1.14 also shows something else. The JobMaster actor is a state machine and uses `become` to go from one state to the next. It starts in the `idle` state until the Job Receptionist sends it a `StartJob` message. Once the JobMaster receives this message it splits up the text in parts of 10 lines and schedules the `Work` messages without delay to the workers. It then transitions to the *working* state to start handling responses from the workers. The `Work` message is scheduled in case other Worker nodes join the cluster after the Job has been started. State machines make a distributed coordinated task more comprehensible. In fact, both the JobMaster and JobWorker actors are state machines.

There is also a `ClusterRouterGroup` which has a `ClusterRouterGroupSettings` similar to how the `ClusterRouterPool` is setup. The actors that are routed to need to be running before a Group Router can send messages to them. The words cluster can have many master role nodes. Every master role starts up with a `JobReceptionist` actor. In the case that you would want to send messages to every `JobReceptionist` you could use a `ClusterRouterGroup`, for instance sending a

message to the `JobReceptionists` to cancel all currently running jobs in the cluster. Listing 1.17 shows how you can create a Router which looks up Job Receptionists on master role nodes in the cluster:

Listing 14.16 Send messages to all JobReceptionists in the cluster

```
val receptionistRouter = context.actorOf(
  ClusterRouterGroup( ❶
    BroadcastGroup( Nil ), ❷
    ClusterRouterGroupSettings(
      totalInstances = 100,
      routeesPaths = List( "/user/receptionist" ), ❸
      allowLocalRoutees = true,
      useRole = Some( "master" ) ❹
    )
  ).props(),
  name = "receptionist-router")
```

- ❶ The `ClusterRouterGroup`.
- ❷ The number of instances is overridden by the cluster group settings.
- ❸ The path for looking up the (top level) receptionist actor.
- ❹ Route to master nodes only.

So far we've looked at how the `JobMaster` distributes the `Work` message to the `JobWorkers`. In the next section we'll look at how the `JobWorkers` request more work from the `JobMaster` until the work is done and how the cluster recovers from failure during job processing.

14.3.3 Resilient Jobs

The `JobWorker` receives the `Work` message and sends a message back to the `JobMaster` that it wants to enlist itself for work. It also immediately sends the `NextTask` message to ask for the first task to process. The figure shows the flow of messages. Listing 1.18 shows how the `JobWorker` transitions from the idle state to the enlisted state:

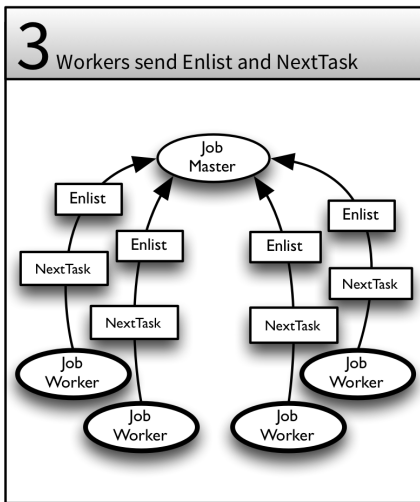


Figure 14.13 JobWorker Enlists itself and requests NextTask

The JobWorker indicates to the JobMaster that it wants to take part in the job by sending an Enlist message. The Enlist message contains the JobWorker's ActorRef so that the JobMaster can use it later. The JobMaster watches all the JobWorkers that enlist, in case one or more of them crashes and stops all the JobWorkers once the job is finished.

Listing 14.17 JobWorker transitions from idle to enlisted state.

```

def receive = idle ❶

def idle: Receive = {
  case Work(jobName, master) => ❷
    become(enlisted(jobName, master)) ❸

    log.info(s"Enlisted, will start working for job '${jobName}'.")
    master ! Enlist(self) ❹
    master ! NextTask ❺

    watch(master)
    setReceiveTimeout(30 seconds)

def enlisted(jobName:String, master:ActorRef): Receive = {
  case ReceiveTimeout =>
    master ! NextTask
  case Terminated(master) =>
    setReceiveTimeout(Duration.Undefined)
    log.error(s"Master terminated for ${jobName}, stopping self.")
    stop(self)
  ...
}

```

- ❶ Start as idle.
- ❷ receives the Work message.
- ❸ Becomes enlisted.
- ❹ Send Enlist message to master.
- ❺ Send NextTask to master.

The JobWorker switches to the enlisted state and expects to receive a Task message from the Master to process. The JobWorker watches the JobMaster and sets a ReceiveTimeout. If the JobWorker receives no messages within the ReceiveTimeout it will ask the JobMaster again for a NextTask as shown in the enlisted Receive function. The JobWorker stops itself if the JobMaster dies. As you can see there is nothing special about the watch and Terminated messages, DeathWatch works just like in non-clustered actor systems. The JobMaster is in the working state in the mean time, shown in listing 1.19:

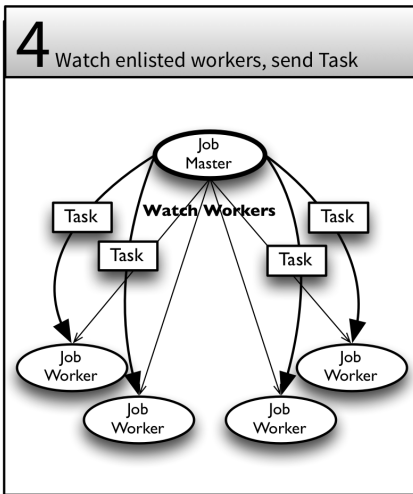


Figure 14.14 JobMaster sends Tasks to JobWorkers and watches them

Listing 14.18 JobMaster enlists worker and sends Tasks to JobWorkers.

```

// inside the JobMaster..

import SupervisorStrategy._

override def supervisorStrategy: SupervisorStrategy = stoppingStrategy ❶

def working(jobName:String,
            receptionist:ActorRef,
            cancellable:Cancellable): Receive = {

  case Enlist(worker) => ❷
    watch(worker)
    workers = workers + worker

  case NextTask => ❸
    if(textParts.isEmpty) {
      sender ! WorkLoadDepleted
    } else {
      sender ! Task(textParts.head, self)
      workGiven = workGiven + 1
      textParts = textParts.tail
    }

  case ReceiveTimeout => ❹
    if(workers.isEmpty) {
      log.info(s"No workers responded in time. Cancelling $jobName.")
      stop(self)
    } else setReceiveTimeout(Duration.Undefined)

  case Terminated(worker) => ❺
    log.info(s"Worker $worker got terminated. Cancelling $jobName.")
    stop(self)

//more code to follow..

```

- ❶ Use a StoppingStrategy
- ❷ Watches the worker that enlisted and keeps track of the workers in a list
- ❸ receives the NextTask request from the worker and sends back a Task message.
- ❹ JobMaster stops if no workers have enlisted within a ReceiveTimeout.
- ❺ JobMaster stops if any of the JobWorkers fail.

The Listing shows that the JobMaster registers and watches the workers that want to take part in the work. The JobMaster sends back a `WorkLoadDepleted` to the JobWorker if there is no more work to be done.

The JobMaster also uses a `ReceiveTimeout` (which is set when the Job is started) just in case no JobWorkers ever report to enlist. The JobMaster stops itself

if the `ReceiveTimeout` occurs. It also stops itself if any `JobWorker` is stopped. The `JobMaster` is the supervisor of all the `JobWorkers` it deployed (the router automatically escalates problems). Using a `StoppingStrategy` makes sure that a failing `JobWorker` is automatically stopped which triggers the `Terminated` message that the `JobMaster` is watching out for.

The `JobWorker` receives `Tasks`, processes the `Task`, sends back a `TaskResult` and asks for the `NextTask`. Listing 1.21 shows the enlisted state of the `JobWorker`:

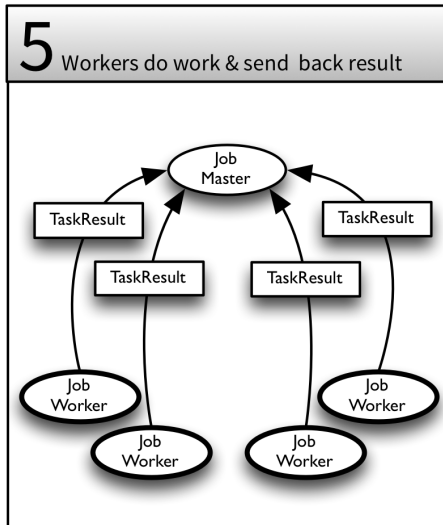


Figure 14.15 `JobWorker` processes `Tasks` and sends back `TaskResult`.

Listing 14.19 JobWorker processes Tasks and sends back TaskResult.

```

def enlisted(jobName:String, master:ActorRef): Receive = {
  case ReceiveTimeout =>
    master ! NextTask

  case Task(textPart, master) =>
    val countMap = processTask(textPart) ❶
    processed = processed + 1
    master ! TaskResult(countMap) ❷
    master ! NextTask ❸

  case WorkLoadDepleted =>
    log.info(s"Work load ${jobName} is depleted, retiring...")
    setReceiveTimeout(Duration.Undefined) ❹
    become(retired(jobName))

  case Terminated(master) =>
    setReceiveTimeout(Duration.Undefined)
    log.error(s"Master terminated for ${jobName}, stopping self.")
    stop(self)
}

def retired(jobName: String): Receive = { ❺
  case Terminated(master) =>
    log.error(s"Master terminated for ${jobName}, stopping self.")
    stop(self)
  case _ => log.error("I'm retired.")
} // definition of processTask follows in the code...

```

- ❶ Process the task
- ❷ Send the result to the JobMaster
- ❸ Ask for the next task
- ❹ Switch off ReceiveTimeout and retire, the job is done.
- ❺ The retired state.

There are some benefits to requesting the work from the JobWorker as is done in this example. The main one is that the workload is automatically balanced between the JobWorkers because the JobWorkers request the work. A JobWorker that has more resources available to do the work simply requests tasks more often than a JobWorker which is under higher load. (This is how thread pools work.) If the JobMaster were instead forced to send tasks to all JobWorkers in round-robin fashion, it would be possible that one or more of the JobWorkers is overloaded while others sit idle.

Listing 14.20 JobMaster stores and merges intermediate results, completes the Job.

```

def working(jobName:String,
            receptionist:ActorRef,
            cancellable:Cancellable): Receive = {
  ...

  case TaskResult(countMap) =>
    intermediateResult = intermediateResult :+ countMap ❶
    workReceived = workReceived + 1

    if(textParts.isEmpty && workGiven == workReceived) {
      cancellable.cancel() ❷
      become(finishing(jobName, receptionist, workers)) ❸
      setReceiveTimeout(Duration.Undefined)
      self ! MergeResults ❹
    }
  }
  ...
def finishing(jobName: String,
             receptionist: ActorRef,
             workers: Set[ActorRef]): Receive = {
  case MergeResults => ❺
    val mergedMap = merge() ❻
    workers.foreach(stop(_)) ❼
    receptionist ! WordCount(jobName, mergedMap) ❽

  case Terminated(worker) =>
    log.info(s"Job $jobName is finishing, stopping.")
  }
  ...

```

- ❶ Store the intermediate results coming from the JobWorkers
- ❷ Remember the scheduled task that sends out Work messages? it's now time to cancel it.
- ❸ Transition to finishing state
- ❹ Send a MergeResults to self so that the results are merged in the finishing state.
- ❺ Receiving the MergeResults message the JobMaster sent to itself
- ❻ Merging all results.
- ❼ Kill all the workers, the job is done.
- ❽ Send the final result to the JobReceptionist.

The JobReceptionist finally receives the WordCount and kills the JobMaster, which completes the process. The JobWorker crashes when it encounters a text

with the word FAIL in it to simulate failures by throwing an Exception. The JobReceptionist watches the JobMasters it creates. It also uses a StoppingStrategy in case the JobMaster crashes. Let's look at the Supervision hierarchy for this actor system and how Death Watch is used to detect failure in Figure 1.18:

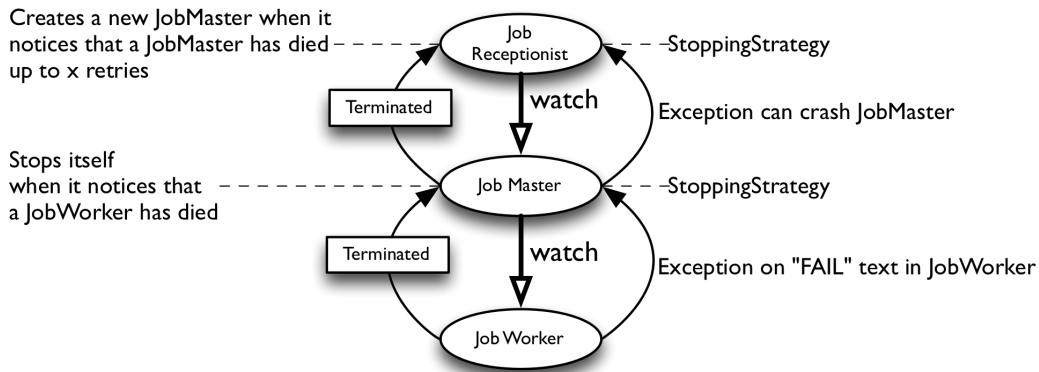


Figure 14.17 Supervision Hierarchy for the words actor system.

Of course we use ReceiveTimeout to detect that the actors are not receiving messages in time so that we can take action. The JobReceptionist keeps track of the jobs it has sent out. When it receives a Terminated message, it checks if the job has been completed. If not, it sends itself the original JobRequest which results in the process starting over again. The JobReceptionist simulates the resolution of the failure simulated with the "FAIL" text by removing the "FAIL" text from the job after a number of retries, as shown in listing 1.23:

Listing 14.21 JobReceptionist retries JobRequest on JobMaster failure.

```

case Terminated(jobMaster) =>
  jobs.find(_.jobMaster == jobMaster).foreach { failedJob =>
    log.error(s"$jobMaster terminated before finishing job.")

    val name = failedJob.name
    log.error(s"Job ${name} failed.")
    val nrOfRetries = retries.getOrElse(name, 0)

    if(maxRetries > nrOfRetries) {
      if(nrOfRetries == maxRetries -1) {
        // Simulating that the Job worker
        // will work just before max retries

        val text = failedJob.text.filterNot(_.contains("FAIL"))
        self.tell(JobRequest(name, text), failedJob.respondTo) ❶
      } else self.tell(JobRequest(name, failedJob.text),
                       failedJob.respondTo) ❷

        updateRetries
      }
    }
  }
}

```

- ❶ Send the JobRequest without the simulated failure.
- ❷ Send the JobRequest again.

We're going to use this simulation of failures in the next section where we're going to test the words cluster.

14.3.4 Testing the Cluster

You can use the `sbt-multi-jvm` plugin and the `multi-node-testkit` module just like with the `Remote` module. It's also still convenient to test the actors locally, which is easily done if we isolate the creation of actors and routers into traits. Listing 1.22 shows how test versions of the `Receptionist` and the `JobMaster` are created for the test. Traits are used to override the creation of the worker routers and job masters.

Listing 14.22 JobReceptionist retries JobRequest on JobMaster failure.

```

trait CreateLocalWorkerRouter extends CreateWorkerRouter {
  def context: ActorContext ❶

  override def createWorkerRouter: ActorRef = {
    context.actorOf(BroadcastPool(5).props(Props[JobWorker]),
                    "worker-router") ❷
  }
}

class TestJobMaster extends JobMaster
  with CreateLocalWorkerRouter ❸

class TestReceptionist extends JobReceptionist ❹
  with CreateMaster {
  override def createMaster(name: String): ActorRef = {
    context.actorOf(Props[TestJobMaster], name) ❺
  }
}

```

- ❶ This trait requires that the 'mixee' define a context, which is the JobMaster class in our case.
- ❷ Create a non-clustered router.
- ❸ Create a test version of the JobMaster, overriding how the router is created.
- ❹ Create a test version of the JobReceptionist, overriding how the JobMaster is created.
- ❺ Create a test version of the JobMaster.

The local test is shown in Listing 1.23. As you can see the test is business as usual, JobRequests are sent to the JobReceptionist. The Response is verified using expectMsg (the ImplicitSender automatically makes the testActor the sender of all messages, as described in the TDD chapter.)

Listing 14.23 Local Words Spec.

```

class LocalWordsSpec extends TestKit(ActorSystem("test"))
    with WordSpec
    with MustMatchers
    with StopSystemAfterAll
    with ImplicitSender {

  val receptionist = system.actorOf(Props[TestReceptionist], 1
    JobReceptionist.name)

  val words = List("this is a test ",
    "this is a test",
    "this is",
    "this")

  "The words system" must {
    "count the occurrence of words in a text" in {
      receptionist ! JobRequest("test2", words)
      expectMsg(JobSuccess("test2", Map("this" -> 4,
        "is" -> 3,
        "a" -> 2,
        "test" -> 2)))

      expectNoMsg
    }
    ...
    "continue to process a job with intermittent failures" in { 2
      val wordsWithFail = List("this", "is", "a", "test", "FAIL!")
      receptionist ! JobRequest("test4", wordsWithFail)
      expectMsg(JobSuccess("test4", Map("this" -> 1,
        "is" -> 1,
        "a" -> 1,
        "test" -> 1)))

      expectNoMsg
    }
  }
}

```

- ❶ Create the test version of the Job Receptionist.
- ❷ The failure is simulated by a job worker throwing an exception on finding the word FAIL in the text.

The multi-node test does not modify the creation of the Actors and the Router. To test the cluster we first have to create a `MultiNodeConfig`, which is shown in Listing 1.24.

Listing 14.24 The MultiNode configuration, defines roles

```
import akka.remote.testkit.MultiNodeConfig
import com.typesafe.config.ConfigFactory

object WordsClusterSpecConfig extends MultiNodeConfig {
  val seed = role("seed")           ❶
  val master = role("master")
  val worker1 = role("worker-1")
  val worker2 = role("worker-2")

  commonConfig(ConfigFactory.parseString(" " ❷
    akka.actor.provider="akka.cluster.ClusterActorRefProvider"
    " " ))
}
```

- ❶ Define the roles in the test.
- ❷ Provide a test configuration. The `ClusterActorRefProvider` makes sure cluster is initialized. You can add more common configuration here for all nodes in the test.

The `MultiNodeConfig` is used in the `MultiNodeSpec` as you might recall from chapter 6. The `WordsClusterSpecConfig` is used in the `WordsClusterSpec`, which is shown in Listing 1.25.

Listing 14.25 Words Cluster Spec

```

class WordsClusterSpecMultiJvmNode1 extends WordsClusterSpec ❶
class WordsClusterSpecMultiJvmNode2 extends WordsClusterSpec
class WordsClusterSpecMultiJvmNode3 extends WordsClusterSpec
class WordsClusterSpecMultiJvmNode4 extends WordsClusterSpec

class WordsClusterSpec extends MultiNodeSpec(WordsClusterSpecConfig)
with STMultiNodeSpec with ImplicitSender {

  import WordsClusterSpecConfig._

  def initialParticipants = roles.size

  val seedAddress = node(seed).address ❷
  val masterAddress = node(master).address
  val worker1Address = node(worker1).address
  val worker2Address = node(worker2).address

  muteDeadLetters(classOf[Any])(system)
  "A Words cluster" must {

    "form the cluster" in within(10 seconds) {

      Cluster(system).subscribe(testActor, classOf[MemberUp]) ❸
      expectMsgClass(classOf[CurrentClusterState])

      Cluster(system).join(seedAddress) ❹

      receiveN(4).map { case MemberUp(m) => m.address }.toSet must be(
        Set(seedAddress, masterAddress, worker1Address, worker2Address)) ❺

      Cluster(system).unsubscribe(testActor)

      enterBarrier("cluster-up")
    }

    "execute a words job" in within(10 seconds) {
      runOn(master) { ❻
        val receptionist = system.actorOf(Props[JobReceptionist],
          "receptionist")
        val text = List("some", "some very long text", "some long text")
        receptionist ! JobRequest("job-1", text)
        expectMsg(JobSuccess("job-1", Map("some" -> 3,
          "very" -> 1,
          "long" -> 2,
          "text" -> 2)))
      }
      enterBarrier("job-done")
    }
  }
}

```

```

    ...
  }
}

```

- ① One for every node in the test
- ② Get the address for every node
- ③ Subscribe the testActor so it is possible to expect the cluster member events
- ④ Join the seed node. The config is not using a seed list so we manually start the seed role node.
- ⑤ Verify that all nodes have joined
- ⑥ run a job on the master and verify the results. The other nodes only call enterBarrier.

The actual test is almost exactly the same as the local version as you can see. The clustered one only makes sure that the cluster is up before the test is run on the master. The test that recovers from failure is not shown here but is exactly the same as the test in Listing 1.25 with a "FAIL" text added to the text to trigger the failure, just like in the local version.

NOTE**Cluster Client**

The test sends the JobRequest from the master node. You might wonder how you can talk to a cluster from the outside, for instance in this case how you can send a JobRequest to one of the nodes in the cluster from outside the cluster.

The akka-contrib module contains a couple of cluster patterns, one of them is the *ClusterClient*. A ClusterClient is an Actor which is initialized with a list of initial contacts (for instance the seed nodes) that forwards messages to actors in the cluster using *ClusterRecipient* Actors on every node. We will use the ClusterClient in the next chapter on Actor Persistence, as well as some of the other patterns in akka-contrib.

That concludes how actors can be tested in a cluster. We've just shown a few test cases here; in real life you would obviously test far more scenarios. Testing locally has the benefit of simply testing the logic of how the actors communicate, where the multi-node-testkit can help you find issues in cluster startup or other cluster specific issues. We're hoping to have demonstrated that testing a clustered actor system is not very different from testing local actors and does not necessarily have to be hard. multi-node tests are great for high level integration tests where you can verify sequentially how the cluster initializes or what happens if a node crashes.

14.4 Summary

Dynamically growing and shrinking a simple application ended up being rather simple with the cluster extension. Joining and leaving the cluster was easily done and we could once again just test the functionality in a REPL console, a tool which allows you to experiment and verify how things work. If you have followed along with the REPL sessions it should have been immediately apparent how solid this extension is; crashes in the cluster are properly detected and death watch works just as you would expect.

Clustering has been a notoriously painful chasm to cross, usually requiring a lot of admin and programming changes. In this chapter we saw that Akka makes it much easier, and doesn't require rewriting code. In the process, we also learned:

- how easy it is to form a cluster
- the node lifecycle state machine
- how to put it all together into a working app
- how to test the cluster logic

Of course the example was not about counting words, but about Akka's generic way of processing jobs in parallel in a cluster. We made use of clustered routers and some simple messages for actors to work together and deal with failure.

Finally we were able to test everything, another big advantage you quickly get used to when using Akka. Being able to unit test a cluster is quite a unique feature and makes it possible to find problems in your application before you get to large scale production deployment. The words cluster actors used some temporary state about the job, spread amongst the various actors. We could reproduce the input from the JobRequest that was stored at the JobReceptionist when a failure occurred in the masters or the workers. One situation this solution cannot recover from is when the JobReceptionist crashes, because the JobRequest data will be lost, making it impossible to resend it to a master. In the next chapter, we're going to look at how we can even restore state for actors from a persistent store using the *akka-persistence* module so that we can even recover from a JobReceptionist crash.