Table of Contents

The Actor Model

The truth is that our reality is bounded by the speed of light. The laws of physics, and the speed of light, put an upper bound on causality. Unless two things occupy the same space (which, of course, is impossible), for one event to have an effect on another, there must be a passage of time between the two. Two observers watching the same event from different distances will experience that event at different times. The person closer to the event will experience it first, while the person further away will experience it slightly later. Yet, despite the difference in time, both experiences are real. Both are equally valid. It is this idea, rooted in physics, on which the Actor Model is based.

Let's go back to our coffee example. When your brain decides that you want a sip of coffee, it must first pause time, at least in a localized fashion. You need to halt the world around you so that you can ensure that nothing changes between the moment you decide to have that coffee and the moment when the coffee reaches your lips. Nothing can interfere with that. If you reach out to take the coffee, only to discover someone else got there first, you have chaos. Instead, you freeze everything around you, locking the state in place, so that you can guarantee that won't happen. Anyone else who might have been reaching for that same cup will now be stopped, frozen until you have completed your action. This means not just pausing another person, but also pausing the air, the light, everything around that cup. Everything between the cup and you must stop, or else your state might become invalid. This sounds very complicated, much more complicated than if we had just taken the time to model the system correctly in the first place.

This is one of the fundamentals of the Actor Model — with it, we can build software that reflects how reality actually works instead of assuming a frozen-in-time world that doesn't actually exist.

Deconstructing the Actor Model

In 1973, Carl Hewitt, along with Peter Bishop and Richard Steiger, set out to help solve some of these problems in a paper called "Universal Modular Actor Formalism for Artificial Intelligence." Although many programming paradigms are based around mathematical models, the Actor Model was, according to Hewitt, inspired by physics. It has evolved over the years, but the fundamental concepts have stayed the same.

It is important to understand that when working with Akka, you can write code without using the Actor Model. The fact that you are using actors does not imply that you are using the Actor Model. There are many developers who have been using Akka for years and are unaware of the Actor Model.

The difference between using the Actor Model and simply using actors is the way that you treat those actors. If you use those actors as the top-level building blocks, such that all code in your system resides within a system of actors, you are using the Actor Model. On the other hand, if you build your system such that you have actors residing within nonactors, then you are not using the Actor Model.

It is also important to understand that programming in the Actor Model is not about any one tool or technology. Entire languages have been written around the idea of the Actor Model. In this way, it is more like a programming paradigm than a set of tools. Learning it is like learning object-oriented programming (OOP) or functional programming. And like those two methodologies, the Actor Model predates Akka, and there are many other implementations of it.

The Pony language, for instance, is based on actors and can be easily used to implement the Actor Model. Erlang's processes are equivalent to Akka actors; they are a fundamental feature of Erlang. In Ada, the idea of the Task exists, along with messages called "entries" — which are queued by the asynchronous Tasks — meaning the Actor Model can be implemented in this language, as well.

To implement the Actor Model, there are a few fundamental rules to follow:

- All computation is performed within an actor
- Actors can communicate only through messages
- In response to a message, an actor can:

  Change its state or behavior

  Send messages to other actors

  Create a finite number of child actors

All Computation Is Performed Within an Actor

Carl Hewitt called the actor the fundamental unit of computation.[1] What does it mean to be the fundamental unit of computation? It means that when you build a system using the Actor Model, everything is an actor. Whether you are computing a Fibonacci sequence or maintaining the state of a user in your system, you do so within an actor, or multiple actors.

This idea that everything is an actor is not without difficulties, though. If every computation needs to happen within an actor, this implies that every function and every state variable could be its own actor. And even though this is technically possible, it's not always pragmatic. Often, we have groups of related functions and it is usually more convenient to wrap all those functions within a single actor. Doing so doesn't violate the Actor Model. But how do we decide where to draw the line? We will discuss this in more detail in Chapter 3 when we learn about domain-driven design (DDD). The building blocks we introduce for DDD are excellent candidates to turn into actors and we can use them as guiding principles for when to create a new actor.

Actors in the Actor Model embody not only state but also behavior. This might sound suspiciously like the definition of OOP — in fact the two are very closely related. Alan Kay, who coined the term "object-oriented programming" and was one of the original creators of the Smalltalk language, was largely influenced by the Actor Model. And while Smalltalk and OOP eventually evolved away from their Actor Model roots, many of the basic principles of the Actor Model remain in our modern interpretation of OOP. In truth, the original focus of OOP was not the objects themselves but rather the messages flowing between them.

> **NOTE**
>
> OOP models the fundamental unit of computation as an object (an instance of a class), and is familiar to developers coming from Java and similar languages. Functional programming, on the other hand, models computation around functions and their applications, and is seen in languages such as Lisp and Haskell.

Another aspect of the model that is useful for highly concurrent applications is the idea that an actor's state is isolated. It is never directly exposed to the outside world. We don't allow actors to look at or modify the state of another actor, except indirectly through messages. This isolation applies equally to the actor's behavior. The internal methods or mechanisms of the actor are never exposed to other actors. In fact, state and behavior can largely be treated as the same thing within an actor (more on this later).

Actors in the model can take many different forms. They can exist as highly technical constructs like a database access layer, or they can be domain-specific constructs like a person or schedule. They can even be used to perform simple mathematical operations. Anything in the system that needs to perform any amount of computation is an actor.

The actor is the fundamental building block in the Actor Model as well as Akka-based applications, as we will see.

Actors Can Communicate Only Through Messages

We have created our actors in an isolated fashion so that they never expose their state or their behavior. This is an important element of the Actor Model. Because we have isolated them in this way, we need to find other ways by which we can communicate with them and learn about the state of the system.

Messages are the backbone of all communication in the Actor Model. They are what enables communication *between* actors.

Every actor, when created, is given an address. This address is the entry point for communication with that actor. You cannot use the address to access the actor directly, but you can use it to send messages to that actor.

> **NOTE**
>
> Akka separates the concept of an address from a reference. Most actor communication is done using references. Akka actors also have an address that can be used in some circumstances; however, they are generally avoided. Nonetheless, whether you are using a reference or an address, the basic principle is the same: you have a means to locate the mailbox for the actor so that you can deliver a message to it.

Messages that are sent to an actor are pieces of immutable data. These messages are sent to a mailbox that exists at the address provided for the destination actor. What happens after the message reaches the mailbox is beyond the control of the sender. Messages can be delivered out of order, or they might not even be delivered at all. The Actor Model provides an *At Most Once* guarantee of delivery. This means that failure is an option, and if we need to guarantee delivery, we will need other tools to enable that.

> **NOTE**
>
> At Most Once delivery is the default guarantee provided by the Actor Model and by Akka. However, At Least Once delivery can be built on top of At Most Once delivery, and there are mechanisms within Akka to provide that.
>
> Akka further provides a slightly stronger guarantee of ordering. In Akka, message order is guaranteed between any pair of actors. That is, if one actor sends multiple messages to another, we can guarantee that the order of those messages will be preserved.

After the messages are in the mailbox, the actor can receive those messages. However, messages are received one at a time. No single actor can process two messages simultaneously. This is a critical guarantee because it allows us to operate within the actor in a *single-threaded illusion*. Actors are free to modify their internal state with no worries about whether another thread might also be operating on that state. As long as we maintain the single-threaded illusion, our state is protected against any of the normal concerns of

concurrency.

The exact nature of the messages is dependent on the actor's functionality. For domain actors, it would be common to see the messages as commands or events presented by using domain terms like "AddUser" or "CreateProject" in our scheduling example. Of course, if the actor is more technical in nature, the messages can be more technical, like "Save" or "Compute."

Because actors communicate only through messages, it allows for some interesting options. As long as the actor on the other end of the mailbox is able to process the message, it can take any form necessary to get the job done (Figure 1-1). This means that the receiving actor can process the message directly or it can act as a proxy for the message, simply forwarding it on to another actor to do the necessary work. It might also break the message into smaller chunks, sending those chunks on to other actors (or possibly itself) to be processed. In this way, the details of how the message is processed can be as simple or as complex as required, and the details of this implementation are transparent to the sender of the message.

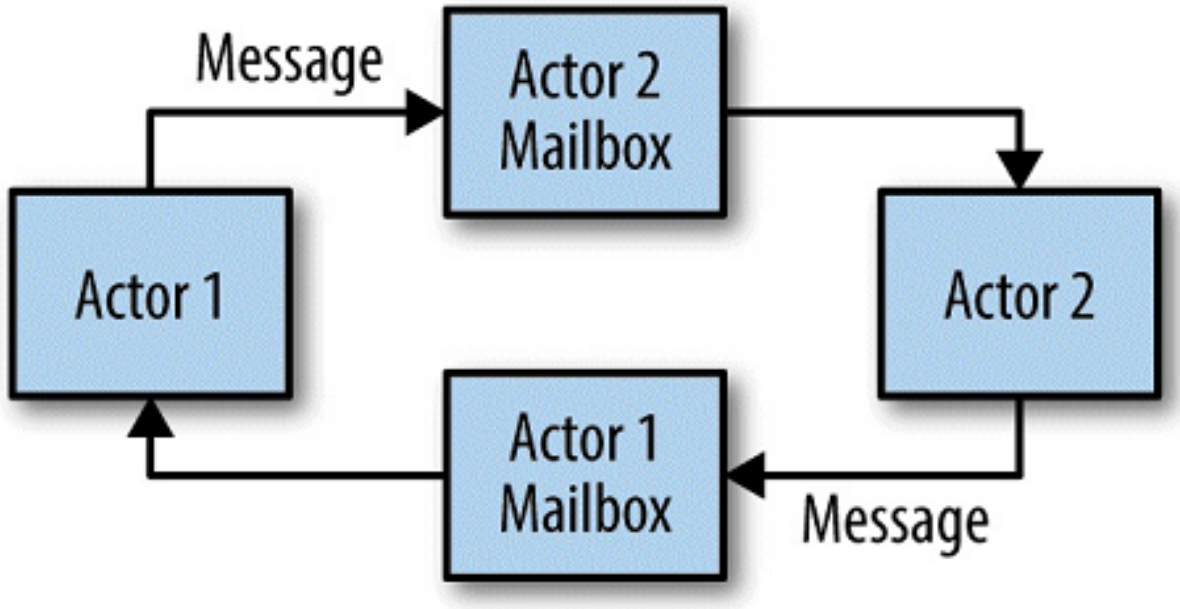

Figure 1-1. Actors communicating with one another

NOTE

It is worth mentioning that Akka treats its mailboxes slightly differently. The default implementation in Akka provides an ordered mailbox so that you can guarantee that messages will be processed by the actor in the order in which they are placed in the mailbox.

Actors Can Create Child Actors

In the Actor Model, everything is an actor, and actors can communicate only through messages. But actors also need to know that other actors exist.

One of the actions that is available to an actor when it receives a message is that it can create a finite number of child actors. The parent will then know about the child actor(s) and will have access to the child's address. This means that a parent actor can send messages to a child actor.

In addition to knowing about actors by creating them as children, an actor can pass the address to another actor through a message. In this way, a parent could inform a child about any other actor the parent is aware of, including itself. Thus, a child actor can know the address of its parent or siblings with very little difficulty. With a little more work, the child can know about other actors that exist in the hierarchy, as well. In addition, if the address used for the actor follows a set pattern, it is possible to synthesize addresses for other actors, though this can create undesirable complexity, and possibly security concerns, if not used carefully.

This hierarchy of actors means that with the exception of the root, all actors will have a parent and any actor can have one or more children. The collection of these actors, starting from the root and working down through the tree, is called an *actor system*.

Each actor in the actor system can always be uniquely identified by its address. It is not critical that it follow any particular pattern, so long as the address can be guaranteed to uniquely identify an actor. Akka uses a hierarchical pattern of naming, much like a directory structure (similar to that shown in Figure 1-2), or you could use randomly generated unique keys.

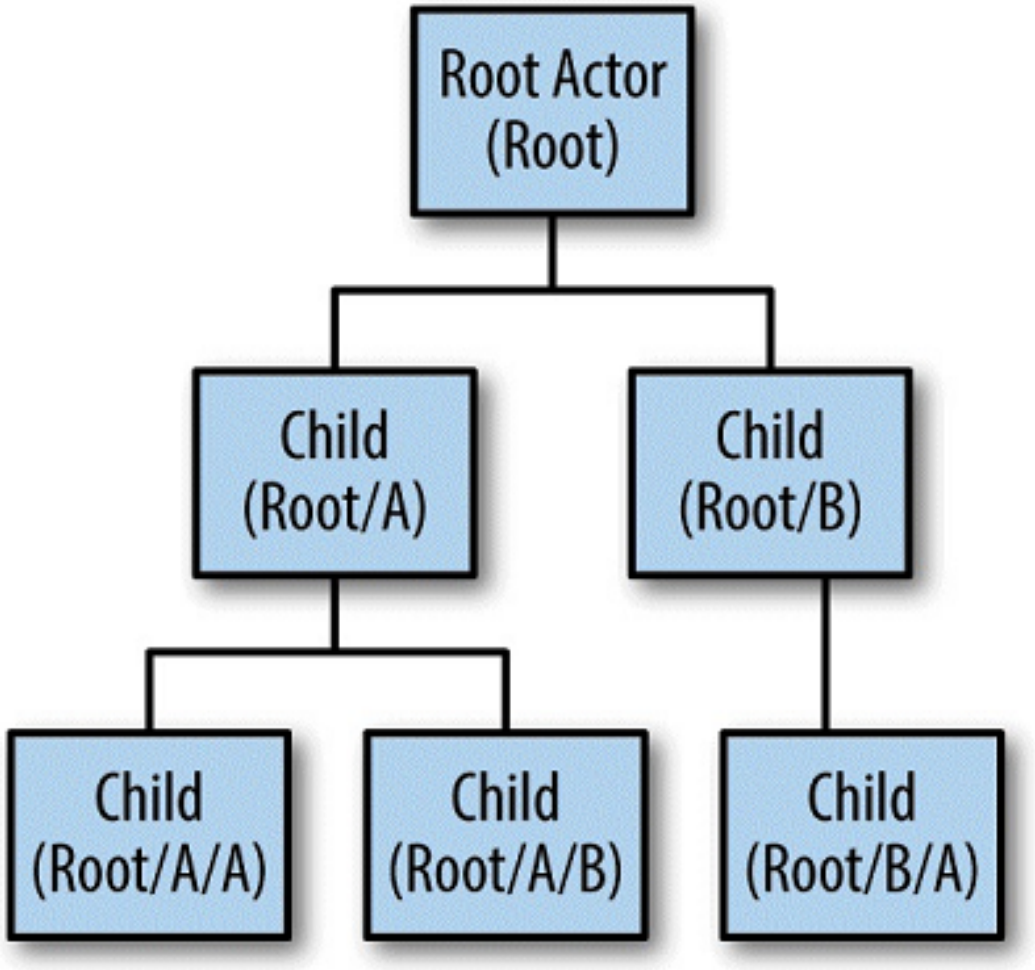

Figure 1-2. A hierarchical system of actors

In Figure 1-2, the *root* actor is the top-level actor under which all others will be created. The root actor then has two children A and B. The address for A is Root/A and the address for B is Root/B. Beneath A are two more actors named A and B. Even though these share the name of other actors, their address is still unique (Root/A/A and Root/A/B). Child B also has a child of its own with the path Root/B/A. Again, the address is unique, even if the name is not. This is just one example of how addresses can be created.

Child actors are one of the most valuable integration techniques in the Actor Model, and also form the basis of actor supervision in Akka, as we will see.

Actors Can Change Their State or Behavior

In addition to sending messages and creating child actors, actors can also change how they will react to the next message. We often use the term "behavior" when talking about how an actor will change, but this is slightly misleading. The change in behavior, or the change in how an actor will react, can also manifest itself as a change of state.

Let's consider the scheduling example discussed in the introduction. If we have an actor that represents a person's availability, the initial state might indicate that this person is available for a project. When a message comes in to assign that person to a project, the actor can alter itself such that when a new message comes in, it will show as unavailable. Even though this seems to be a change in state, we still consider it to be a change to the actor's behavior because the actor is now behaving like it has a value of unavailable when the next message comes in.

We can also have an actor that changes the computation that it will perform when it receives the next message. The actor representing a person in this system might be able to exist in different states. While that person is employed and available to be on a project, they can exist in an Active state. However, certain conditions can result in that person moving to an Inactive state (termination, extended leave, etc.). While in an Active state, project requests can be processed as normal. However, if a message comes into the actor that puts it into an Inactive state, the system might begin rejecting project requests. In this case, the actor alters the computation that it performs when the next message comes in.

Because actors can alter their behavior and state using this method, it makes them an excellent tool for modeling finite-state machines. Each behavior in the system can represent a state, and the actor can move from one state to the next when it receives a message.

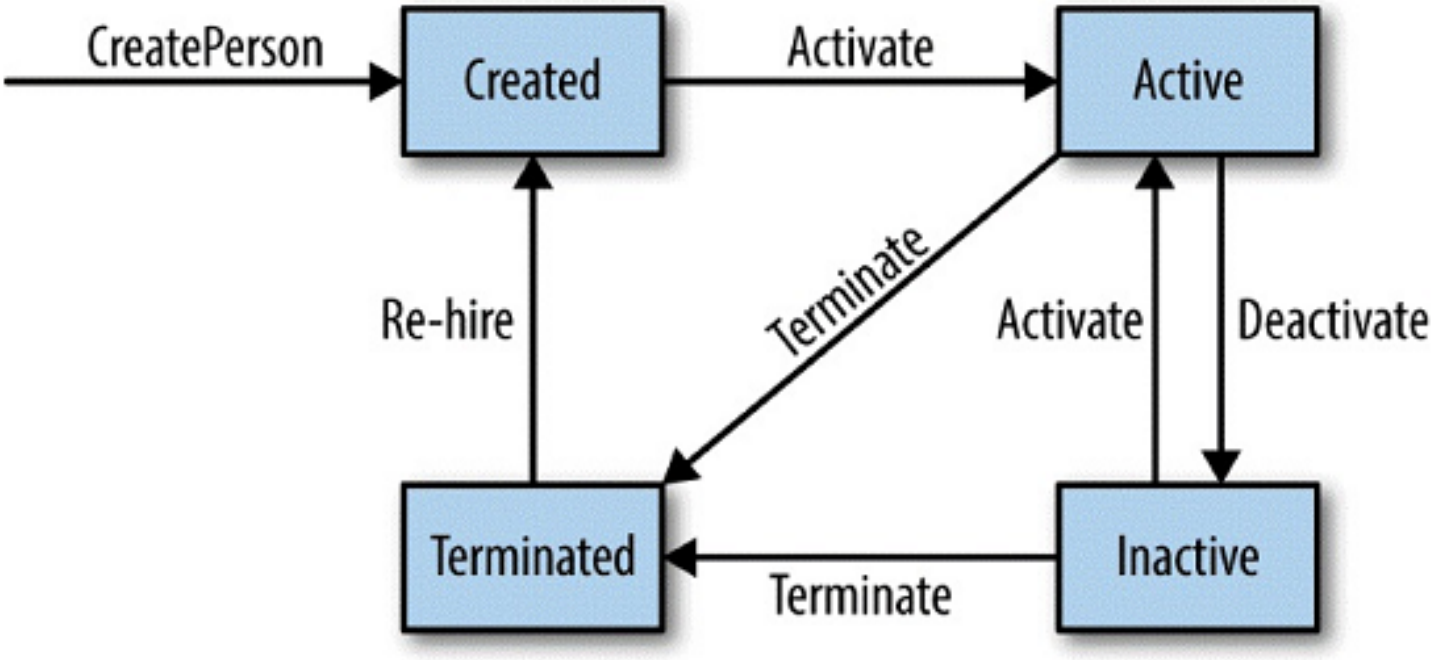

Figure 1-3. A finite-state machine using actors

> **NOTE**
>
> Actors are an excellent way to model a finite-state machine and are commonly used in this way. Akka provides specific tools that make the creation of finite-state machines easier in the form of the Akka FSM.

We obviously can create far more complex scenarios, but this gives you an idea of the types of things that you can accomplish by altering the behavior of an actor. Whether it is altering the messages that are accepted, altering the way those messages are handled, or altering the state of the actor, this all falls under the heading of behavior. Akka provides several sophisticated techniques for altering actor behavior, as we will discuss in detail.

Everything Is an Actor

Now that you understand the building blocks of an actor and of a system of actors, let's return to the idea that everything is an actor and see what that means, and what it looks like.

Let's consider our scheduling domain again. A person in our scheduling domain will have a variety of associated information. Each person might have a schedule indicating availability. Of course, that schedule would break down further into more discrete time periods.

In a more traditional object-oriented architecture, you might have a class to represent the person. That class would then have a schedule class associated with it. And that schedule might break down further into individual dates. When a request comes into the system, it would call functions on the person class, which would in turn call functions on the schedule class and the individual dates.

The Actor Model version of this isn't much different, except now you have an actor representing the person. But in the Actor Model, the schedule can also be an actor because it potentially makes a computation. Each individual date can also be an actor because they might also need to make a computation. In fact, it is possible that the request itself can have an actor associated with it. In some cases, the request might need to aggregate information as it is computed by the other pieces of the model. In this case, we might want to create a `RequestWorker` that will handle the aggregation.

As Figure 1-4 shows, in this model, instead of calling functions that will alter the state of the objects, you pass messages between them. Messages like `CheckAvailability` flow from the `Person` actor into the `ScheduleActor` actor. In this example, the `Person` represents the topmost actor. The `Schedule` is a child of `Person`, and the individual `Dates` are children of the `Schedule`. And because we are using the Actor Model, all messages are handled concurrently. This way, we can compute multiple dates in the schedule at the same time; we don't need to wait for one to complete before moving on to the next. In fact, we don't even need to complete the previous request before starting on the next one. The schedule can be working on two simultaneous requests, each looking at overlapping dates. Due to the single-threaded illusion, if two requests try to schedule the same date, only the first request will succeed. This happens because the same actor is used to process both requests for that date, and the actor can process only one message at a time. On the other hand, if there is no overlap in dates, both requests can be processed and completed simultaneously, allowing us to make better use of our resources.

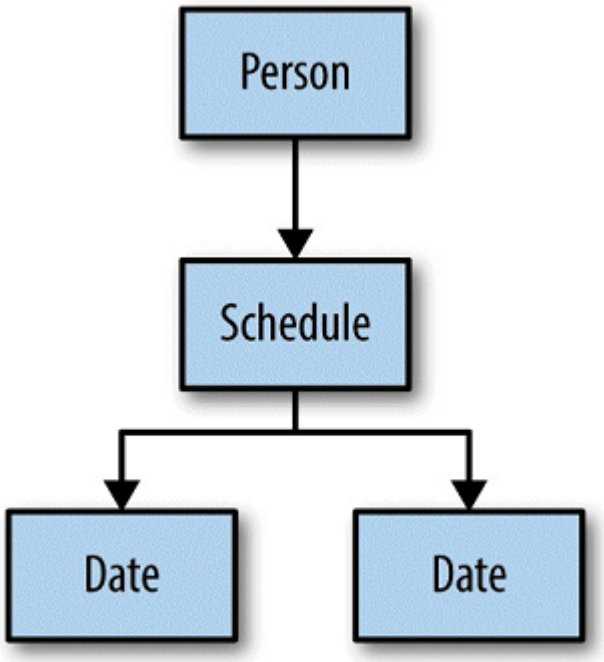

Figure 1-4. Representing a person's schedule using actors

Uses of the Actor Model

The Actor Model is a powerful tool, and when applied carefully, can be effective at providing highly scalable, highly concurrent applications. But like any tool, it is not perfect. If you apply it incorrectly, the Actor Model can be just as likely to create complex code that is difficult to follow and even more difficult to debug. After you learn the tools, you might decide that you want to apply the Actor Model at all levels of your application, creating a true Actor System. But early on, it is sometimes safer to just get your feet wet and ease into it. In later chapters, we will explore these ideas further and provide additional guidelines about what an individual actor should be modeling.

One thing to realize about the Actor Model is that, like other techniques, there are times when you might want to apply it, but there are other times when you might not. You might find that in certain cases an alternative model is more suitable for the task at hand. And in those cases, you should not be afraid to use those tools or techniques. The Actor Model is not an all-or-nothing proposition. The key is to figure out where you need it and where you don't, and then create a clear and distinct line separating the two.

Defining Clear Boundaries

Many successful systems are modeled with actors representing top-level entities in the system, and then using a more functional approach within those actors. There is nothing wrong with this approach. In addition, it is common to see a small group of actors wrapped within an a interface such that clients of that interface aren't aware that they are communicating with actors. Again, this is a good approach. Both create clear boundaries. In the first case, you have actors only at the top level. All of the implementation is done by using functional programming. In the second case, the interface creates a boundary between what are actors and what are not.

If you have decided to use the Actor Model for a portion of your application, you should stick to the model throughout that section. You should only break from the model on clearly defined boundaries. These boundaries could be the boundary between your frontend and backend services. Or, it could be the boundary between two different microservices in your application. There are many ways in which to partition your application and many ways to apply the Actor Model. You should, however, avoid breaking the Actor Model mid context. One of the things that contributes to messy, complex code is when the code transitions between styles with no clear pattern or explanation as to why the transition has occurred. You don't want to find yourself in a situation in which you jump into and out of the Actor Model multiple times within a single context. Instead, look to break from the model along consistency boundaries or domain boundaries.

Let's consider a concrete example. In our scheduling domain, we might want to build a library to handle the scheduling. Within that library, we would prefer to use the Actor Model, but outside the boundaries of the library we want to use other techniques. This is a good example of a system boundary where it is OK to transition from one model of computation to another.

One way to handle this would be to expose everything as actors. The external client code would be fully aware that it is interfacing with actors. Thus, we might have a message protocol that looks something like this:

```
object ProjectProtocol {
  case class ScheduleProject(project: Project)
  case class ProjectScheduled(project: Project)
}
```

Using this code within the client would mean sending a message along the lines of this:

```
def scheduleProject(details: ProjectDetails): Future[Project] = {

  val project = createProject(details)

  val result = (projects ? ProjectProtocol.ScheduleProject(project))
    .mapTo[ProjectProtocol.ProjectScheduled]
    .map(_.project)

  result
}
```

This is OK, but it's not great. The problem here is that we have actor code interspersed with other code. This other code might have different concurrency mechanisms. There can actually be several steps in this process. You might first need to create a project, and then schedule the project and potentially perform other operations, as well. Each of these can involve going to the scheduling system, which means interfacing with actors. Yet there might be other parts of the code that communicate with other contexts that don't use actors, so there can be function calls and futures and other mechanisms here, as well. This constant jumping in and out of the Actor Model can become difficult to follow when it's not well isolated. A better approach in this case is to create a wrapper API over the top of your actors, as demonstrated here:

```
class ProjectApi(projects: ActorRef) {
    def scheduleProject(project: Project): Future[Project] = {
        val result = (projects ? ProjectProtocol.ScheduleProject(project))
            .mapTo[ProjectProtocol.ProjectScheduled]
            .map(_.project)
    }
}
```

This thin wrapper around the actors provides the insulation layer. It gives us a clear transitional boundary between what are actors and what are not. Now, when we want to use this API, it looks like the following:

```
def scheduleProject(details: ProjectDetails): Future[Project] = {

  val project = createProject(details)

  val result = projectApi.scheduleProject(project)

  result
}
```

The code is shorter, but really we have just moved any complexity within a separate function. So is that really better? The advantage of this approach is that the client code doesn't need to be aware that actors are being used internally. Actors in this case have become an implementation detail rather than an intrinsic part of the API. Now when we use this API, we are taking advantage of it using function calls, which means when we nest them alongside other function calls, everything looks more consistent and more clean. As the code base grows, and the actors become more complex, this kind of isolation can be critical to keeping the system maintainable.

Should we do this everywhere? Should all actors have an API wrapper around them so that the application never needs to know that it is dealing with actors? The simple answer is "no." Within our scheduling library, which is built using the Actor Model, this kind of wrapper interface is not only unnecessary, it creates unwanted complexity. It actually makes it more difficult to work with the actors within the actor system. When we're using the Actor Model we should be dealing with actors, and we should not be trying to hide that fact. It's only on the boundaries of the system — where we want to transition to a different model — that we should be creating this kind of isolation.

When Is the Actor Model Appropriate?

We have decided that there are times to use actors by themselves and other times for which we might use the full Actor Model. And we know that when we use them, we need to be careful to do so within a clearly defined context. But when is the right time to use them? When should we use standalone actors and when should we build a full-fledged actor system? Of course, it is impossible to say you should always use one or the other in a certain situation. Each situation is unique. Still, there are some guidelines that you might take into account when making the decision.

The first obvious candidate for using actors is when you have a high degree of concurrency and you need to maintain some sort of state. Maintaining concurrent state is the bread and butter of the Actor Model.

Another obvious case for actors and the Actor Model is when you're modeling finite-state machines, as an actor is a very natural construct for this task. Again, if you are limited to only a single finite-state machine, a single actor might do the job, but if you are going to have multiple machines, possibly interacting with one another, the Actor Model should be considered.

A more subtle case that might suit actors is when you need a high degree of concurrency, but you need to be careful in how that concurrency is managed. For example, you might need to ensure that a specific set of operations can run concurrently with other operations within your system, but also that they cannot operate concurrently with one another. In our scheduling example, this might mean that you want multiple users to be able to concurrently modify projects in the system, but you don't want those users to be able to modify the same project at the same time. This is an opportunity to use the single-threaded illusion provided by actors to your advantage. The users can all operate independently, but you can funnel any changes to a particular project through an actor associated with that project.

What Is Akka?

Akka is described on its home page as "an open source toolkit and runtime simplifying the construction of concurrent and distributed applications on the Java Virtual Machine (JVM)."

It goes on to say that Akka supports multiple programming models but emphasizes actor-based concurrency, with inspiration drawn from Erlang.

This is a good high-level description, but Akka is much more. Let's look a bit deeper.

Akka Is Open Source

Akka is an open source project released under the Apache 2 License, a recognized open source license, making it free to use and extend in both other open source efforts and in commercial libraries and applications.

Although it is fully usable from Java, Akka is itself written in Scala and gains the benefit of the attributes of this language as a result, including strong type safety, high performance, low memory footprint, and compatibility with all other JVM libraries and languages. One of the key features of the Scala language that is used heavily in Akka is the focus on immutable data structures. Akka makes heavy use of them in its messaging protocols. In fact, it is not uncommon to see Java users write their messages in Scala in order to save time.

There are even adapter toolkits to use Akka from Clojure, an implementation of Lisp on the JVM.

Akka Is Distributed by Design

Akka, like many other implementations of the Actor Model, doesn't only target multicore, single computer systems — you can also use it with clusters of systems. It is, therefore, designed to present the same programming model for both local and distributed development — virtually nothing changes in the way you develop your code when deploying across many systems, making it easy to develop and test locally and then deploy distributed.

With Akka, it is natural to model interactions as messages, as opposed to systems such as Remote Procedure Call (RPC), for which it is more natural to model interactions as procedure calls. This is an important distinction, as we will see in detail later.

Although it is possible to write code that is aware of the distributed nature of the cluster (e.g., by listening to events as nodes join and leave the cluster), the actual logic within the actors themselves does not change. Whether dealing with local actors or remote, the communication mechanisms, delivery guarantees, failure handling, and other concepts remain unchanged.

Akka provides a key element in the construction of so-called *Reactive* applications; that is, applications built to support the fundamental attributes discussed in the Reactive Manifesto (which you can see at *http://www.reactivemanifesto.org/)*. The manifesto describes applications that are *responsive*, *resilient*, *elastic*, and *message driven*. The message-driven part is where Akka comes in, and through its capabilities, supports the rest of the attributes.

In an Akka system, an actor can be *local* or *remote*. It is local with respect to any other actor if the sending and receiving actors are in the same JVM.

If an actor is determined to be local, Akka can make certain optimizations in delivery (no serialization or network call is required), but the code is no different than if the receiving actor were remote.

That's what we mean by "distributed by design": no code changes between local operation and distributed operation.

Akka Actor

The primary component of the Akka library is the implementation of actors themselves, which are the fundamental building block on which Akka is built. All of the attributes of the Actor Model are supplied here, but only on a single JVM — distributed and clustering support are optional components.

Akka actors implement the Actor Model with no shared state and asynchronous message passing. They also support a sophisticated error-handling hierarchy, which we'll dig into later.

The Akka API intentionally limits the access to an actor. Communication between actors can happen only through message passing. There is no way to call methods in the actor in a synchronous fashion, so actors remain decoupled from one another both in API and in time. It does this through the use of an `ActorRef`. When an an instance of an actor is created, only an `ActorRef` is returned. The `ActorRef` is a wrapper around the actual actor. It isolates that actor from the rest of the code. All communication with the actor must go through the `ActorRef`, and the `ActorRef` does not provide any access to the actor that it wraps.

This means that unlike regular object-oriented method calls, whereby the caller blocks, passes control to the called object, and then resumes when the called object returns, the messages to an actor are sent via the `ActorRef`, and the caller continues immediately. The receiving actor then processes the messages that were sent to it one at a time, likely on a completely different thread from the caller. The fact that messages are processed one at a time is what enables the single-threaded illusion. While inside of an Akka actor, we can be confident that only a single thread will be modifying the state of that actor, as long as we don't actively break the single-threaded illusion. Later we will discuss ways by which we can break the single-threaded illusion, and why that should be avoided.

The message-driven nature of the Actor Model is implemented in Akka through the `ActorRef`. An `ActorRef` provides a number of methods that allow us to send messages to the underlying actor. Those messages typically take the form of immutable case classes. Even though there is nothing preventing us from sending mutable messages to an actor, it is considered a bad practice because it is one way that we can break the single-threaded illusion.

Changes in the behavior of an actor in Akka are implemented through the use of `become`. Any actor can call `context.become` to provide a new behavior for the next message. You also can use this technique to change an actor's state. However, actors can also maintain and change state by using mutable fields and class parameters. We will discuss techniques for changing state and behavior in more detail later.

Child Actors

Akka actors also can create child actors. There are various factory methods available within an actor that allows it to create those children. Any actor in the system will have access to its parent and children and can freely send messages to those other actors. This is also what enables the supervision mechanism in Akka because parent actors will automatically supervise their children.

The supervision mechanism in Akka means that this message-passing structure can focus on the "happy path," because errors have an entirely separate path through which to propagate.

When an actor encounters an error, the caller is not aware of it — after all, the caller could be long gone or on another machine entirely, so sending it the error isn't necessarily helpful. Instead, Akka passes errors from an actor to its *supervisor* actor; that is, the actor that started it in the first place. Those supervisor actors then use a special method to handle those exceptions, and to inform the actor that threw the exception as to what to do next — to ignore the error or to restart (with various options).

This is the origin of the "let it crash" saying that is associated with Akka; in other words, the idea is that it is acceptable for the actor that had the problem to simply "crash." We don't try to prevent the failure of the actor. Instead, we handle it appropriately when it happens. If an actor crashes, we can take certain actions to recover that actor. This can include stopping the actor, ignoring the failure and continuing with message processing, or it can even mean restarting the actor. When an actor is restarted, we transparently replace it with a new copy of that actor. Because communication is handled through the `ActorRef`, the clients of the actor need not be aware that the actor failed. From their perspective, nothing has changed. They continue to point to the same `ActorRef`, but the actor that supports it has been replaced. This keeps the overall system resilient because the error is isolated to the failing actor. It need not propagate any further.

Figure 2-1 shows three actors in a single actor system within a single JVM. Any actor can send messages to any other, in both directions (although we only show two possible routes).

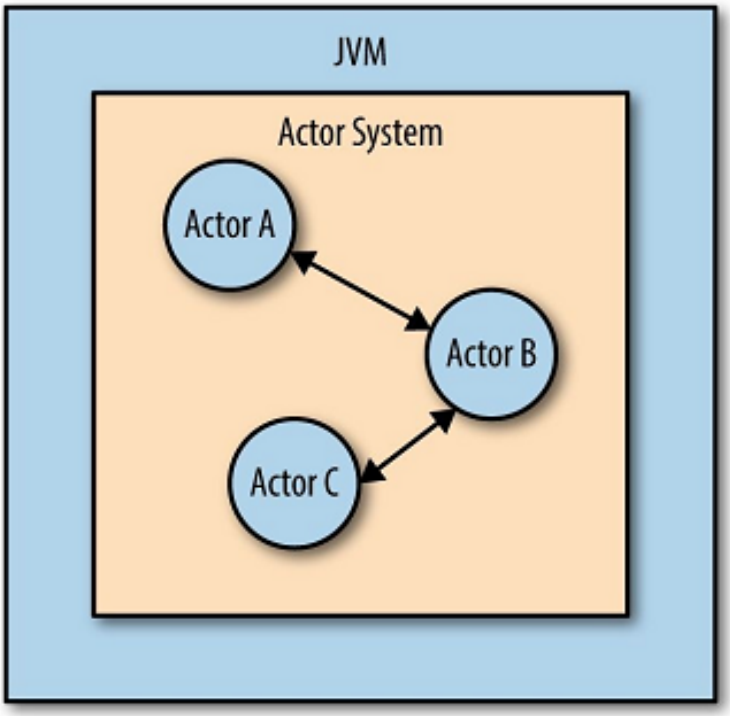

Figure 2-1. Actors in a single JVM

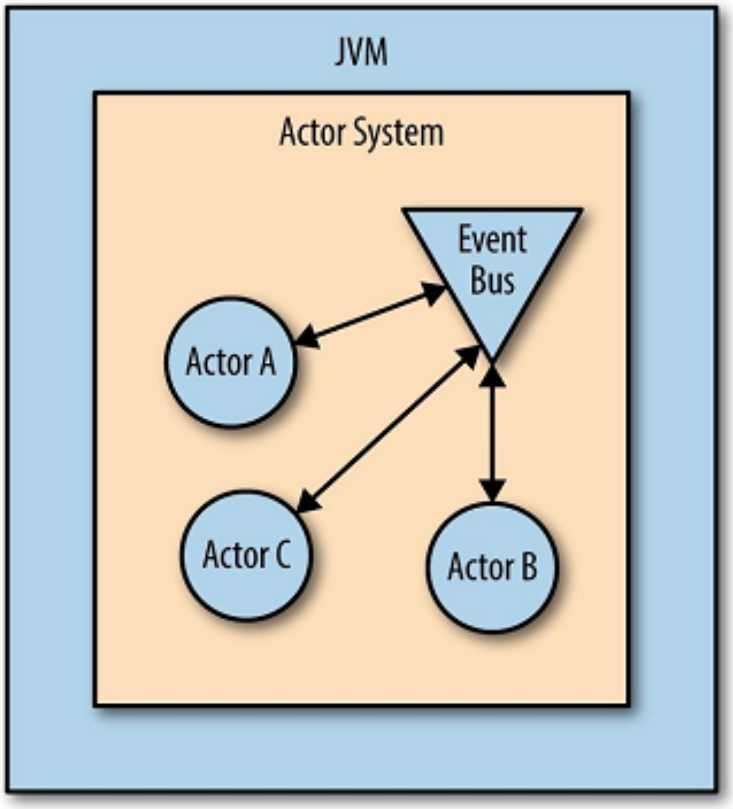

Figure 2-2. Actors and event bus

In addition to the usual message-passing mechanism of actors, which is point-to-point, Akka also provides a one-to-many messaging mechanism, called the *event bus*. The event bus allows a single actor to *publish* a message, and other actors to *subscribe* to that message by its type. The sending actor then remains completely decoupled from the receiver, which is very valuable in some situations.

Note that the event bus is optimized only for local messaging; when using Akka in a distributed environment, the publish/subscribe module provides equivalent functionality optimized for multiple nodes.

Figure 2-2 shows the three actors again, but this time, instead of communicating directly with one another, they communicate bidirectionally with the event bus, improving decoupling.

Remoting: Actors on Different JVMs

Akka provides location transparency for its actors; that is, when you have an actor reference to which you can send messages, you don't actually need to know if that actor is on the local system or on another machine altogether.

Remoting provides the ability for the actor system to communicate over a network, and to serialize and deserialize messages so that actors on different JVMs can pass messages to one another.

This is enabled in part through the actor's unique address. An actor's address can include not only the unique path to the actor in the hierarchy, but also a network address for that actor. This network address means that we can uniquely identify any actor in any actor system, even when that actor resides in a different JVM running on a different machine. Because we have access to that address, we can use it to send messages to the remote actor as though it were a local actor. The Akka platform will take care of the delivery mechanics for us, including serializing and sending the message.

The serialization mechanism is pluggable, as is the communication protocol, so a number of choices are available. When starting out, the default mechanisms are often sufficient, but as your application grows, you might want to consider swapping in different serializers or communication protocols.

Remoting in Akka can be added by configuration only: no code changes are necessary, although it is also possible to write code to explicitly perform remoting.

With remoting, each node must be *aware* of the other nodes and must have their network address because the messaging is point-to-point and explicit. Akka Remoting does not include any discovery mechanisms. Those come as part of Akka clustering.

Clustering: Automatic Management of Membership

For larger groups of servers, remoting becomes awkward: adding a node to a group means that every other node in the group must be informed, and message routing can grow pretty complex.

Akka provides the *clustering* module to help simplify this process.

Akka clustering offers additional capabilities on top of Remoting that make it easier to provide for location-independence. Clustering provides the ability for actor systems running in multiple JVMs to interact with one another and behave as though they are part of the same actor system. These systems form a single, cohesive clustered actor system.

Akka clustering makes the location-transparency of actors much more accessible. Before clustering, you needed at least some part of your code to be aware of the location of remote actors, and you needed to build addresses for those actors to send messages to them. Failover was more difficult to manage: if the node you were communicating with for your remote actors became unavailable or failed, you needed to handle starting up another node and creating connections to the new actors on that node yourself.

With Akka clustering, instead of one node needing to be aware of other nodes on the network directly, it needs to be aware of only one or more *seed nodes*. These are specially designated nodes that can be used to connect to the cluster by new nodes that are looking to join.

You can (but probably shouldn't) have a single seed node in your cluster, or you can designate every node as a possible seed. The idea is that as long as one of the seed nodes is available, new nodes can join the cluster.

No special code is required: configuration informs your actor system as to where the seed nodes are, and it contacts one of them automatically on startup.

The new node then goes through a series of lifecycle steps (as seen in Figure 2-3) until it is a full member of the cluster, at which point it is not only capable of sending messages to other actors in the cluster, it can also start new actors, either in its own actor system or on another node of the cluster. In essence, the cluster becomes a single virtual actor system.

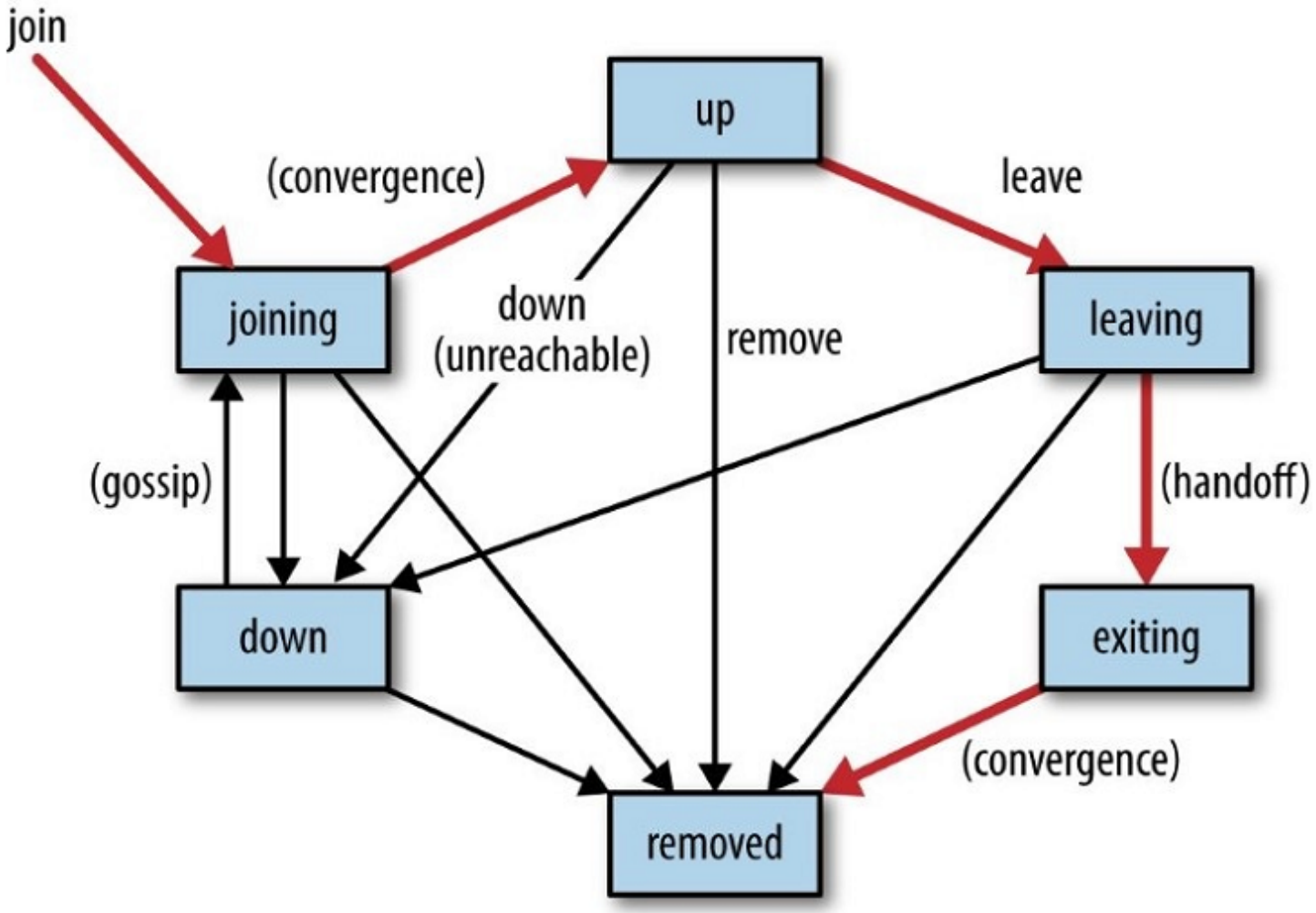

Figure 2-3. Cluster states

Although the actors within the cluster can send messages to one another, the cluster management itself is handled separately. A variant of the *gossip* protocol is used by the actor system itself to manage cluster membership. Random peers communicate what information they have about cluster membership to another node — that node passes on what it knows to another, and so on. After a time, *convergence* occurs. Convergence is when all nodes get a cohesive picture of the membership of the cluster (and all have the same version of the cluster information structure), yet there is no one central node that contains this information. It is a robust and resilient system, used in many other distributed applications such as Amazon's Dynamo and Basho's Riak. *Vector clocks*, which do not actually track wall-clock time, but are a way of determining a partial ordering of events in a distributed system, are used to reconcile messages received via the gossip protocol into an accurate state of the cluster.

Special event messages are sent automatically as cluster lifecycle events occur — for instance, a new node joining, or an existing node leaving the cluster. You do not *need* to listen to these events to use the cluster, but if you want more fine-grained control or monitoring, you can.

For nodes determined to be missing from the cluster, a Phi Accrual failure detector is used to

determine when a node has become *unreachable*, and eventually, that the node is *down*. The cluster supports a heartbeat to support this detection. Of course, a node can leave the cluster intentionally, as well — for instance, during a controlled downscaling or shutdown.

For a description of the Phi Accrual failure detection method, check out the paper "The ϕ Accrual Failure Detector".

Cluster leader

Each cluster has a *leader*. Every node can determine algorithmically which node should be the current leader based on the information supplied by gossip; it is not necessary that the leader be elected. The leader, however, is not a point of failure, because a new leader can be established by any node as necessity arises. It is also not especially relevant to developers, because the leader is used only internally by Akka to make certain cluster-related decisions.

The leader does two things for the cluster: it adds new members to the cluster when they become eligible to be added, and it takes them out when appropriate. It also schedules the *rebalancing* of the cluster as necessary; that is, the movement of actors between members of the cluster. Because the leader has limited duties, the cluster can continue to function even in the absence of a leader. In this case, all traffic among cluster nodes can continue as before; it is only the changes to the cluster that will be delayed until a new leader is established. You will not be able to add or remove nodes from the cluster, but it can still perform all of its other duties despite the absence.

A potential issue with distributed systems is the issue of *partitioning*. In a cluster, if a node terminates unexpectedly, other nodes simply take over its function, and all is well. However, if a group of nodes is suddenly disconnected from the cluster — say, due to a network failure — but does *not* terminate, an issue can arise. The surviving nodes carry on, but who are the survivors? Each group of nodes, now isolated from the others, can potentially form a new cluster, leading to a situation called *split-brain*, in which there should only be one cluster but there are in fact two. In this situation, guarantees such as the uniqueness of a cluster singleton can be compromised.

There are several techniques to deal with this situation. The simplest is to disallow autodowning. When autodowning is enabled, if a node becomes unreachable, the system will automatically remove it from the cluster after a period of time. This seems like a good thing on paper, but in practice this is what leads to difficulties. It leads to the split-brain. One or more downed nodes can form their own cluster, giving rise to the problem. By disabling autodowning, this is no longer possible. Now, when nodes become unreachable, human intervention is required in order to remove the node from the cluster. In this way, new clusters can't form, because it is impossible to reach the critical convergence required. Generally, autodowning should not be enabled in production.

However, if autodowning is used, there are still other options. It is possible to restrict the minimum size of a cluster (this is something you can configure). In this case, if a single node

is disconnected, it cannot form a cluster, because it doesn't have enough members, so it terminates. Picking the right minimum value is situational, though: there's no one right answer.

Lightbend also provides a smart *split-brain resolver* product as a commercial add-on that helps in this situation. It uses more sophisticated techniques to determine if a cluster partition has occurred and then takes appropriate action.

Cluster sharding

The idea of *sharding* was initially applied to databases, for which a single set of data could grow to be too large to fit in a single node. To spread the data more or less evenly across multiple nodes, the idea of a *shard key* was defined: some value that was part of the data that had a good, wide distribution. This shard key could be used to chop up the data set into a number of smaller parts.

Akka cluster sharding takes this further by applying the principles of sharding to live actors, allowing them to be distributed across a cluster.

An example of a shard key might be the first letter of the last name of a customer: you could then break up your data into a maximum of 26 *shards*, wherein shard 1 would have the data for all customers with last names beginning with "A," for instance.

In practice, this is a terrible shard key, but it's a simple illustration.

As you begin to add more nodes to your Akka cluster, the number of actors that are available grows substantially, in many cases, and you can apply cluster sharding in much the same manner as you would apply it to a database.

Akka cluster sharding takes the concept of sharding and extends it so that you can shard live actors across a cluster.

Each node can be assigned to a shard *region*, and there can be (and probably should be) more than one node assigned to each region.

Messages that are intended for use in a cluster-sharded environment are then wrapped in a special envelope that includes a value that can be used to determine the shard region to which that message should be delivered. All messages that resolve to that shard region are delivered only to that region, allowing for state contained in those actors to be stored only on those nodes.

In the previous, simple example, if you have a shard region for all customers whose last names begin with "A," the value of the customer's last name can be used to resolve to the shard region, and the message will be delivered to the correct node(s).

Distributed domains with cluster sharding

One of the significant advantages of cluster sharding is that persistence for the state of the actors within a region can be restricted to the shard region nodes — it need not be replicated

or shared across the entire cluster. For instance, suppose that you're keeping an account balance for customers. You can persist this balance by storing every message that changes the balance and then save this journal to disk. In this way, when the actor that encapsulates this state is restarted, it can just read the journal and be back at the correct balance.

With the capabilities of Akka clustering and cluster sharding, it is possible to build a system that uses what has been called a *distributed domain*. This is the technique of having a single actor holding state for an instance of a domain (say, a customer) somewhere in a cluster-sharded actor system. We will explore this pattern in some depth later in this book.

If the actor in question is for a customer whose last name begins with "A," its journal need be kept only on the nodes in that shard region — because we can guarantee that the actor for that customer starts up *only* on one of those nodes — as opposed to the usual cluster situation, in which an actor can be started anywhere in the cluster. You need to take care when applying this technique, however, because it restricts the flexibility of the cluster significantly.

Cluster singleton

Another refinement in clustering is the *cluster singleton* — that is, a specific actor that must always have one, and *exactly* one, instance in the cluster. It doesn't matter where it is in the cluster, but it's important that there is just the one.

In this case, Akka provides a special means to ensure this: each node is capable of starting the singleton, but only one node can do so at a time. If that actor for some reason fails, the same or another node will start it back up again, ensuring that it remains available as much as possible. Every node that needs to send messages to the singleton has a *proxy*, which routes messages to the singleton no matter where it is in the cluster.

Of course, like any singleton, there are disadvantages, but the failover mechanism takes away at least one of them. You can still end up with a performance bottleneck, however, so you must use cluster singletons with care.

Akka HTTP

Inspired by the Spray HTTP library, Akka HTTP replaces it while providing a deeper integration with Akka and Akka Streams. It provides a way to build HTTP APIs on top of Akka and is the recommended approach for interfacing Akka with other HTTP-based systems.

Akka HTTP provides the recommended way to expose actors as REST endpoints, and build RESTful web services.

Akka Streams

Akka Streams provides a higher-level API for interacting with actors while at the same time providing automatic handling of “back pressure” (which we will discuss in detail later on).

Streams provides a way of building complex structures called streams and graphs that are based on actors. You can use these structures to consume and transform data via a convenient and familiar domain-specific language (DSL).

These streams follow the Reactive Streams initiative.

Message Passing

The Actor Model specifies that message passing should be the only way for actors to communicate, and that all processing should occur in response to this message passing.

In Akka, messages are the only means for actors to interact with one another and the outside world. Messages are passed to other actors via a nonblocking queue called the mailbox. The object reference of the actor itself is not used directly — only an intermediary called the `ActorRef`, (for Actor Reference). This queue is normally first-in/first-out, but can take other forms if desired.

There are three message-passing mechanisms in Akka, to be precise. Let's take a look at each one.

Tell

The preferred mechanism for sending messages is the *tell*, sometimes called "bang," based on the name of the shorthand for the method, "!".

You specify a "tell" by using the "!" operator, like this:

```
projectsActor ! List

Which is the equivalent of

projectsActor.tell(List)
```

(Let's assume that List is a case object here.)

The tell method is the classic "fire and forget" message in the Actor Model: it neither blocks nor waits for any response.

Ask

You need to use caution when using the ask method because it is easy to write blocking code that will significantly reduce the advantages of an asynchronous system with *ask*.

The shorthand for ask is "?", as demonstrated here:

```
projectsActor ? List
```

An ask indicates that a response is expected. The response is captured in a future because the actor to which the ask is sent might not respond immediately.

Publish/subscribe

The final means of message sending in Akka is via the event bus, described previously. The sender of a message uses a reference to the event bus to *publish* a message.

A receiver of the message must independently *subscribe* to the type of the message, and will therefore receive every message of that type that another actor publishes. The two actors are

not directly aware of each other, and no *sender* reference is available for a message received this way.

Actor Systems

A single actor is no actor at all, according to the Actor Model. Akka provides the concept of actor *systems*; that is, groups of actors that are able to exchange messages readily, whether they are running in a single JVM or across many.

Although it is possible for actors in different actor systems to communicate, it is not common.

An actor system also serves as the facility to create or locate actors, like so:

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

val system = ActorSystem("demo")

val actor = system.actorOf(Props[DemoActor])
```

Creating new actors

Actors in Akka can be created both from outside the system, as well as by other actors, satisfying Hewitt's requirement that new actors can create "child" actors.

Changing behavior

Actors in Akka have the ability to swap out the behavior that they use to respond to messages with another behavior, which can handle perhaps a completely different set of messages. Akka does this with the `become` and `unbecome` operations with untyped actors. Akka Typed, a future version of actors that we will explore in more detail in just a moment, returns the behavior for handling the next message after every message is handled.

Akka goes a step further and supplies a convenient DSL for creating finite-state machine actors specifically, a common use case for changing behavior.

DDD Overview

DDD is a set of guiding principles for software architecture. The principles of DDD are not revolutionary. In fact, a common reaction to people hearing them for the first time is “well, of course. That’s obvious.” It is not the principles themselves that are so powerful; rather, it’s how they are applied and how they are combined. Applying them consistently throughout a project can transform that project from something that is cumbersome and awkward into one that is elegant and considerably more useful.

The most important concept in DDD is the focus on the Domain Model. If you are not familiar with the term “domain,” it is the set of requirements, constraints, and concepts that make up the business or field that you are trying to model. In DDD, we focus on that business domain and we model our software around it. We are trying to model our software in a way that reflects the real world and how it operates. We want our model to capture the truth of the domain that we are trying to model. This involves conversations with domain experts. These experts are people who are knowledgeable about the domain but might not be technically savvy. They can include lawyers, marketing staff, support staff, business managers, or anyone else with a knowledge of the domain. This means that it is important to use language that makes sense to those experts, not just in our conversations, but also in our code. This language that we develop is shared between the developers and the domain experts and is called the *ubiquitous language*.

When we establish a ubiquitous language, it eases our ability to communicate about the software, not just among developers but with the domain experts, as well. It allows us to have a conversation about the software, referring to actions and objects within the application in a way that is still intelligible to nondevelopers. This in turn helps the domain experts feel involved in the software in a way that would otherwise be impossible. When you begin explaining the model using their language, they are able to point out flaws in the way that you are using that language. Those flaws will often be reflected as deficiencies in the developer’s understanding of the domain that will have crept into the software itself. Being able to speak in this common language is an invaluable tool in the development process.

This common language can change meaning from one area of the domain to another. The individual concepts, what actions are available, and how they interact within the domain might not always be the same. Each concept is bound within a particular context. Within that context, the language has a certain meaning. When we leave that context and move to another area of the domain, the meaning can change.

A key part of this is recognizing that the domain is not fixed. It is a fluid entity that can change over time. Sometimes, the rules of the business change. Sometimes, our understanding of those rules evolves. We need to be prepared for those changes and we need to be prepared to adapt the Domain Model to accommodate the changes. If we build a model expecting it to handle all cases and to never change, we are doomed to failure.

DDD focuses on building models that are equipped to evolve. We don't build a model the first time and expect it to last. We build it with the understanding that it will fail at some point and we will need to adapt. Accordingly, we need to build it in a way that allows it to adapt. The cost of change must remain small. DDD gives us a set of tools to help keep that cost of change low.

The Benefits of DDD

One of the worst ways that you can cripple your software and prevent it from evolving is by creating too much coupling between areas of the system that are conceptually different. In particular, when you create coupling between portions of an application that are considered part of the domain and portions that are considered part of the infrastructure, you reduce your ability to adapt. DDD helps to distinguish what is domain and what is infrastructure, but it also helps to create the right abstractions around them so that you don't violate those boundaries.

Domain Value Objects

Value Objects, on the other hand, are different from Entities. A Value Object has no identity beyond the attributes that it contains. Two Value Objects that contain the same data are considered to be the equal; you don't bother trying to distinguish between them. Value Objects, unlike Entities, are immutable. They must be immutable because if their properties change, they become different Value Objects — they are no longer equal.

In Akka, the messages passed between actors are natural Value Objects. Those messages are immutable if we are following best practices and are usually not identifiable. They are just data containers. They might contain references to other Entities, but the message itself is not usually an Entity. We can also use Value Objects as the containers that hold the state of our actors. We can swap in different states as required, but the state itself doesn't have any identity. It is only when it is present inside of the actor that it becomes identifiable. If we were to create two unique actors that both had the same state, the actors would be considered Entities, whereas the state would be considered Value Objects.

Our User actor might have a series of messages that need to be passed to that actor in order to alter its state. For example, if you need to change the name of the user, you might do that by using a message like `SetName`:

```
object User {
    case class SetName(firstName: String, lastName: String)
}
```

In this case, the `SetName` message is a Value Object. If you have two `SetName` objects for which the value of `firstName` and `lastName` is the same, those two messages can be considered equivalent. Whether you send one or the other doesn't matter, the effect on the user is the same. Conversely, if you were to change the value of `firstName` in one of the messages, it would be a different message. Sending it to the user would have a completely different effect. There is no unique identifier for the `SetName` message. There is no way to distinguish one message from the other except by the contents of the message itself. This is what makes it a Value Object.

The messages that you use to pass between actors are called the *message protocol*. A good practice is to embed messages for a particular type of actor in a common protocol object. This can be either in the companion object for the individual actor or it can be a separate protocol object (e.g., `UserProtocol`). The latter case is particularly useful if you want multiple types of actors to handle the same set of messages.

Aggregates and Aggregate Roots

Aggregates are collections of objects within an application. An aggregate creates a logical grouping of many different elements of a system. Every aggregate is bound to an *aggregate root*. An aggregate root is a special entity within the aggregate that has been given responsibility for the other members of that aggregate. One of the properties of an aggregate is that other aggregates are forbidden from holding a reference to anything inside the aggregate. If you want to gain access to some element inside the aggregate, you must go through the aggregate root; you do not access the inner element directly. For instance, if your person aggregate root has an address entity, you don't directly access the address, but instead access the appropriate person, and then reference the contained address.

Aggregates, and their associated roots, are a tricky concept. It can be difficult to determine what is an aggregate in your system, or more importantly, what is the right aggregate root. Generally aggregate roots are the top-level pieces of a system. All of your interactions with the system will in one way or another interface with an aggregate root (with a few exceptions). So how do you determine what your aggregate roots are?

One simple rule is to consider deletion. If you pick a specific Entity in the system and delete it, does that delete other Entities in the system? If your system consists of people who have addresses, and you delete an address, does it delete other parts of the system? In this case, probably not. On the other hand, if you delete a person from the system, there is a good chance that you don't need to keep that person's address around anymore, so in this case, a person might aggregate an address. Keep in mind, however, that although people are often aggregate roots in a system, it is not always the case. Take a bowling score-keeping system as an example. In this system, you might have the concept of a game and a player. The player might seem like a natural candidate for an aggregate root. They have scores, and if you delete a player the scores associated with that player are deleted, too. But if you step up another layer, what happens if you delete a game? Well, you could argue that deleting the game does not delete the player, but that isn't quite accurate. Deleting the game does not delete the person, but if a person is not involved in any games, is that person actually still considered a player? In this case, it might make more sense to say that the game is the aggregate root.

Remember, though, that you might get it wrong. The point isn't to pick the right aggregate root directly from the start. The important thing is to keep the cost of changing your mind later as low as possible.

In Akka, aggregate roots are often represented by parent actors. When you delete/stop those parents, all of their children go with them. But they don't need to be top-level actors. Sometimes, it is beneficial to have a layer or two between the top level and the actual aggregate roots. For example, if your users are the aggregate roots, you might want a layer above the user that will be the parent of all users. In this case, your user is still the aggregate root, but you have another component in your system that is responsible for managing those users. This is especially true when you begin introducing concepts like cluster sharding (see

Back Pressure

An alternative to work-pulling is to provide a means for the consumer to provide *back pressure*. Much like the analogy of a pipe with water flowing in it, the data "fills the pipe," and the system exerts pressure back on the sender to slow down the flow, as illustrated in Figure 5-5.

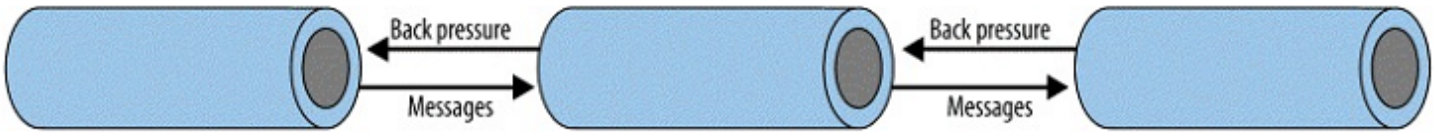

Figure 5-5. Back pressure

This can take the form of a special message that the consumer emits when it is reaching a maximum rate of processing, or it can be supplied by the transport mechanism, as we will see soon.

In either case, the message has the same effect: notify the producer to slow down its rate of production so as not to overwhelm the queue and the consumer.

There are several approaches to back pressure.

Acks

A simple means of producing back pressure is to have the consumer send back to the producer an acknowledgment, usually abbreviated as it is in networking to an "ack." This ack could be identified; for example, it could specify *which* message was received, or it could simply say "OK, I've processed one, you can send one more." Of course, the same technique can be applied for a size greater than one — the consumer, for instance, could confirm processing every hundred or every thousand messages — the principle is the same.

High-Water Marks

Instead of acknowledging messages, singly or in groups, another option is to do the reverse: send a message when the producer should slow down or stop, in response to some value exceeding a so-called high-water mark.

Instead of saying, "I've done X messages, send some more," the high-water mark technique says the reverse: "Things are getting hot, slow down," is the meaning of the message in this case.

You should take care to use a different channel to send the control messages, of course, because a backlog on *that* channel can defeat the operation of the control message, leading to obvious issues.

Queue-Size Monitoring

Tracking the size of the queue between the producer and the consumer is another option; in fact, this can be the high-water mark that is used in the previous technique.

Caution is warranted, however, because it's possible to place additional load on the system by monitoring the queue, given that this is a performance-critical area, especially if the producer and consumer are not singular components — there could be a great many producers and consumers, and they might be separated via a network.

Rate Monitoring

If you have a previously benchmarked consumer, you might be able to find the approximate rate at which messages can be consumed and control the flow entirely from the producer end by using this information.

This technique says, "If the consumer can handle *X* messages in *Y* time, the producer must send no more than *X* messages every *Y* period." Assuming that all messages take close to the same amount of time to process, this can be a simple technique that requires no direct communication between the producer and consumer.

Like all time-based operations, however, it is subject to variance over time, depending on the load on the overall system, so it is not reliable when conditions change. Suppose that you begin running some new process on the same node as your consumer; suddenly you're not consuming at the same rate as you were before, so the producer will begin sending faster than can be handled.

Of course, it's possible to build a self-tuning system wherein the consumer actually sends a message periodically to the producer, updating its rate information from time to time. This is a bit more complex, but very flexible, and still requires comparatively little communication between the consumer and producer.

The problem with all of these techniques — whether it's work-pulling, back pressure, rate monitoring, or something else — is that they all require extra effort on your part. None of them are built in to the system. When building pure actor-based systems, you need these techniques in order to prevent memory problems, but it would be nice if there were something built in to help you.

Akka Streams

You have seen how streams can improve message throughput and how routers can improve both throughput and latency depending on how they are used. You have also seen how mailboxes can overflow and how you can fix that by using techniques like back pressure. The evolution of these ideas takes the form of Akka *Streams*. Akka Streams are Akka's implementation of Reactive Streams. They take the concepts of streams, routers, back pressure, and more and roll it all into a single consistent domain-specific language (DSL) that allows you to write type-safe streams, including junction points, that support back pressure all the way through. They make it possible to build complex flows of data without having fear of overrunning the actor's mailbox. At the same time, they provide all the same advantages that we got from creating actor streams with routers as branch points.

Akka Streams are built on a few basic concepts. These include sources, sinks, flows, and junctions. Each of these has a role to play when building up a stream of data. We will explore them each in detail so that you can fully understand how they address the problems we face.

To help visualize the role that each of these pieces play, it is helpful to imagine an Akka Stream as a flow of water through a set of pipes. As we explore each concept, we will see how this works.

Source

A source in Akka Streams represents a source of data. This is the origin point of your stream, whether it is a file, a database, or some other input to the system. When back pressure needs to be applied and you need to slow down the system, this is where it will need to occur. There are different ways in which you can slow down the flow depending on what the source of the data is. It might mean slowing down the rate you pull from a file, it might mean buffering data to disk, or it might mean dropping data. It all depends on the nature of the source and what level of control you have over its speed.

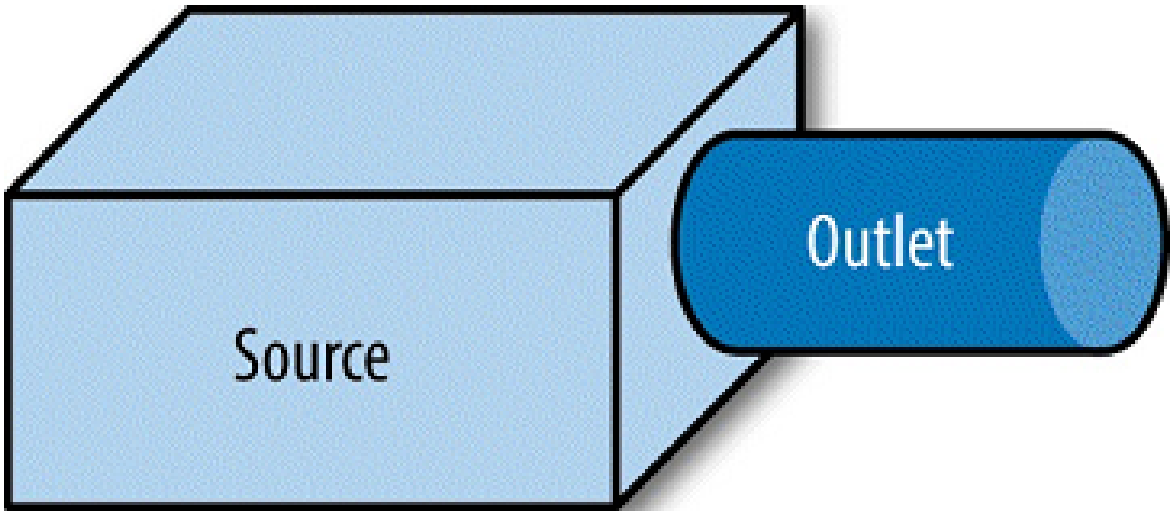

Figure 5-7. Plumbing

From here, you apply `GraphStageLogic`. The `GraphStageLogic` has an `OutHandler` with a corresponding `onPull` method. The `onPull` method will determine what action to take when data is requested from the stream. In this case, you have implemented something very simple (a random number), but this can become as complex as you need it to be. It can read from a file, draw from a buffer, or whatever you need it to do.

One key to implementing this is that any mutable state must be contained within the `GraphStageLogic`. The `GraphStage` is just a template for the logic. It is reusable and can

Sink

A sink in Akka Streams represents the endpoint of the stream. At some point all streams need to end. Without an endpoint, streams don't really have a purpose. Whether we are writing to a database, producing a file, or displaying something in a user interface, the goal of the stream is to produce this output. In fact, for a stream to be complete it needs only two pieces: the source and the sink.

As with a source, there are multiple ways to create a sink. There are very simple examples, like the following:

```
val printString = Sink.foreach[Any](println)
```

This creates a sink that will simply call `println` on any items coming into it. You also can use

a `fold` method:

```
val sumIntegers = Sink.fold[Int, Int](0) { case (sum, value) => sum + value }
```

The `fold` method is going to take each element and *fold* it into some result — in this case, a `sum`. The result of the fold will be a `Future[Int]`, which will resolve to the `sum`, assuming that the stream eventually terminates.

RunnableGraph

A `RunnableGraph` is the result of connecting a source and a sink together. These two things give you the minimal complete system. After you have a `RunnableGraph`, the data can begin moving. Until you have both, you can't do anything. Either you lack a source for the data or you lack a destination. One way or the other, your system is incomplete.

For a graph to be complete, each outlet must be connected to exactly one inlet, as demonstrated in Figure 5-10. If any inlets or outlets remain unconnected, the graph is incomplete and cannot function.

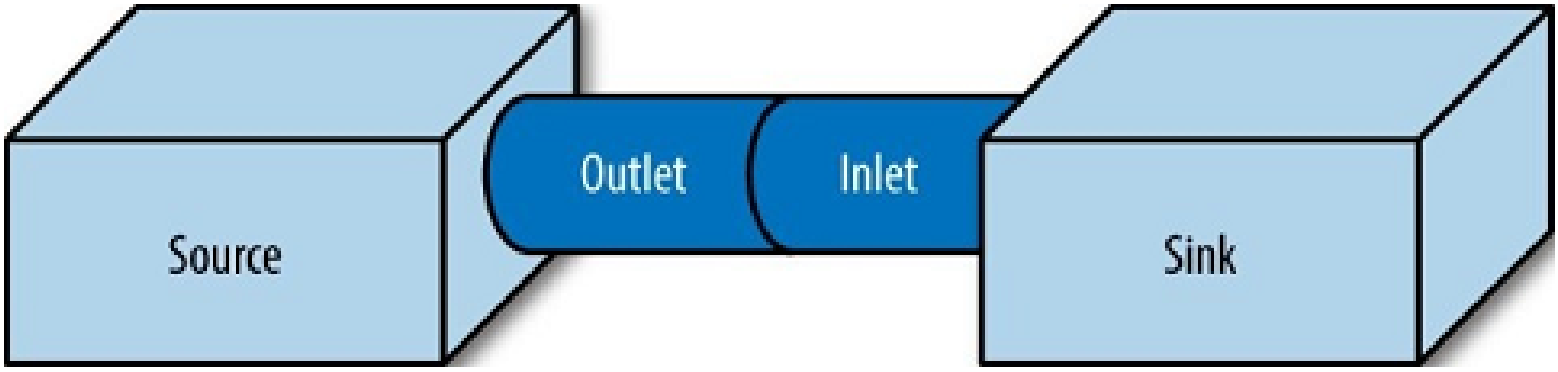

Figure 5-10. RunnableGraph

From the plumbing perspective, a `RunnableGraph` occurs when you connect your water source to your sink (Figure 5-11). Now the water can flow freely from one place to the other. Until you make that connection, you are either pumping water and spraying it out the end of the pump with no real destination, or you have a sink and no water to put in it.

Flow

As already mentioned, simply connecting your source and your sink is enough to create a runnable flow but it's not really that useful. You are basically just taking the data from one place and putting it into another place without any modification or change. Sometimes, this is desirable, but more commonly you are going to want to perform an operation on that data. You'll want to transform it in some way. This is the purpose of a flow. A flow is a "connector" in which you can transform the data (Figure 5-12). There are many different kinds of flows. Some might modify the data coming from the Source. Others might reduce the data by filtering it or by only taking a portion of it. Flows are connected between the source and the sink. They take your basic `RunnableFlow` and enhance it, giving it more functionality.

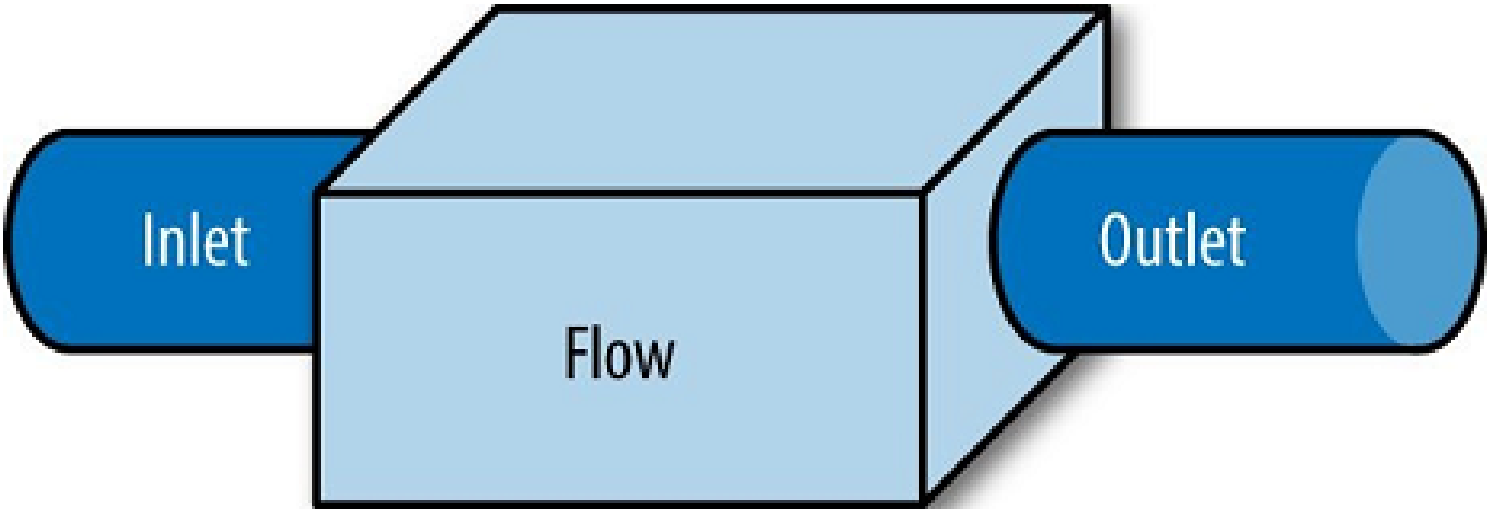

Figure 5-12. Flow

Referring back to our plumbing analogy, a flow is your pipes. You have a source and you have a sink, but it is unrealistic to think that you can simply attach the two with a single length of pipe. Typically, you need to transport the water over some distance. As Figure 5-13 demonstrates, you probably need to change directions a few times. The pipes might become narrower or wider in order to change the pressure. All of these things happen using pipes of various shapes and sizes. Each of these pipes represents a flow.

Junctions

As your data moves through the system, you might find it necessary to branch out and send portions of it down different paths. Or, it might be necessary to take data from multiple sources and combine it in some way. This is accomplished by using a junction. A junction is basically a branch point. It can be either a fan-in or a fan-out. You can either take a single flow and branch it into many, or combine many flows into one, as shown in Figure 5-14. When you begin using junctions, you are no longer dealing with a simple stream: you are dealing with a *graph*.

Junctions are created as graph elements with multiple inlets or multiple outlets.

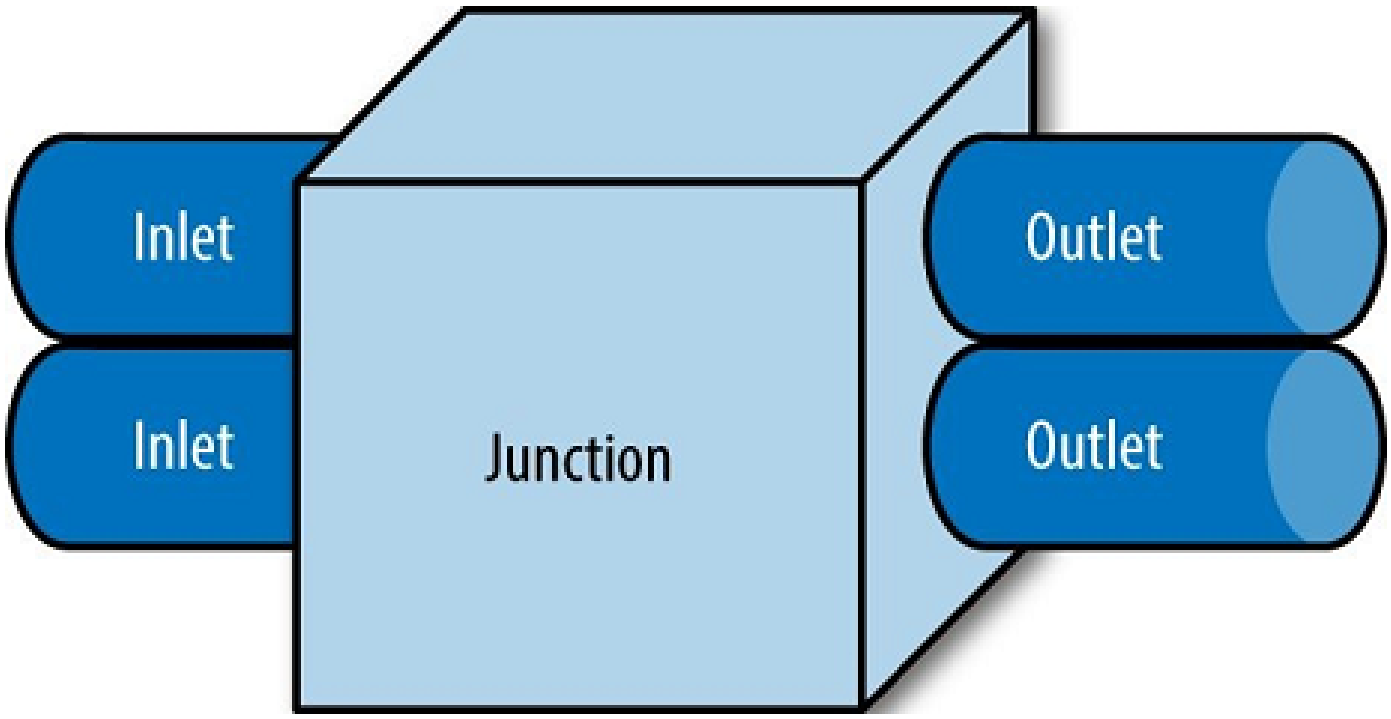

Figure 5-14. A junction

In the plumbing analogy, junctions are represented by things like T-joints and manifolds. They take the water flow coming from the source and branch it to go to multiple different houses, each with its own sink. Or, they might take hot water and cold water and combine them to give us warm water.

Akka Streams give us the means to use combinations of sources, sinks, flows, and junctions to create complex systems called graphs, such as the one depicted in Figure 5-15.

Back Pressure in Akka Streams

Akka Streams is built on the concept of back pressure. In this case, the back pressure is implemented by using a push–pull mechanism. When a source and a sink are connected to form a runnable flow, a message is sent from the sink back through the various flows to the source. The message is basically saying that the sink can accept more data. When the source receives this message, it knows that it can send more data to the sink. When data becomes available, the source will pass it through the flows into the sink.

The source is allowed to send only as much data as the sink has requested. If the source receives data and the sink has not requested any, the source must decide what to do with it. It will need to either stash it somewhere or discard it. This might mean that the source simply doesn't draw any more data from the database or file, or it may mean that it must stash or drop a request. This sounds a bit like we have simply moved our overflow problem, not solved it; instead of the overflow happening at the end of the pipe, we have just pushed it up to the beginning of the pipe.

Initially that sounds bad, but when you think about it, that's the best case. At the beginning of the pipe, we are likely far more able to control the data flow and recover from any issues. In most cases, if your system is overloaded, it is likely going to be preferable to block a single user, rather than accepting all requests and then crashing the entire system when it runs out of memory. In the former case, a single user is affected in a very controllable fashion. In the latter case, all users might be affected, and how they are affected could be very nondeterministic.

At the end of the day, with a limited amount of hardware, it is impossible to continue to accept all data no matter how fast it is pushed into the system. At some point, you need to put pressure back on the input to slow down. This is what Akka Streams enables. It forces that pressure back to the entry point into the system. It provides the necessary mechanisms so that when the system experiences excessive load, you can communicate the need to slow down back to the source, which is the only place where we are truly equipped to deal with it.

Chapter 6. Consistency and Scalability

Consistency is a complex attribute of a system, and the need for it varies depending on the system in question. It is very frequently misunderstood, so let's first look at what we mean by consistency before we examine ways to control it.

A system can be said to be *consistent* if different pieces of related data throughout the system are in agreement with one another about the state of the world.

For a simple example, if I calculate the total amount of deposits I've made to a bank account as well as the total number of withdrawals, the system would be *consistent* if it reports a bank balance that agrees with the delta between these two numbers.

On a classic single-processor, single-memory-space von Neumann system, this kind of consistency is not difficult to achieve, in general. Of course, this is assuming that our system doesn't have any bugs.

As soon as a system has any aspect of parallelism, however, consistency becomes a bit more difficult. If we introduce multiple cores, we have some parallelism. If we make the system distributed, the situation becomes even more complex.

As we explained in Chapter 1, the real world doesn't actually have a globally consistent state, so perhaps it's acceptable if our software systems don't either.

In this chapter, we talk about what a transaction means in the context of a distributed system, what the tradeoffs are on consistency, and why global consistency is not the friend of a scalable system. We'll talk about message delivery guarantees and how to avoid the enemies of scalability.

Strong Versus Eventual Consistency

Data updated via a transaction, in a single atomic instant, is considered to be immediately consistent; that is, the overall system goes from one point at which it is fully consistent to another state in which it is also fully consistent (but with the change). There is no period of time when different parts of the system might have a differing view on the state of the world.

Eventual consistency, on the other hand, implies an update that happens over a span of time, no matter how short that span might be, whereby two different parts of the system might, for a moment, have different opinions on the state of the world. They must come into agreement at some point in the future, but that's the "eventually" part. Consistency is attained, just not instantaneously.

Strangely enough, eventual consistency is usually much easier to achieve than immediate consistency, especially in a distributed environment.

Concurrency Versus Parallelism

Concurrency means that two processes start at some points in time, and their progress overlaps at other points during their processing. This could mean that we start one process, then stop, then start the other, work for a while, stop and start the first again, and so forth. It does not necessarily require the processes to be happening at the same time, just to overlap.

Parallelism, however, introduces the idea of simultaneous progress between two processes; in other words, at least a part of the two (or more) processes are happening in exactly the same time interval. Parallelism implies concurrency, but not the reverse: you can be concurrent without being parallel.

A distributed system is, by definition, parallel, whereas even a single-core, single-processor system can be concurrent by switching which task it's working on over time until both are complete.

Why Globally Consistent Distributed State Doesn't Scale

Aside from the violation of the law of causality in the real world, globally consistent distributed state is very difficult to even approach, much less achieve.

Setting aside whether it is even desirable, let's assume that we want globally consistent distributed state. When we update that state, we need all nodes that are influenced by it (e.g., they have a copy of the data or some data derived from it, such as our bank balance) to have exactly the same opinion as to the state of the world at the moment the update is done.

This effectively means that we need a "stop the world" situation when state is updated, during which some relevant portion of each system is examined and potentially updated to the correct state, and then we can resume processing on all the nodes.

What if one node becomes unavailable during this process? How do we even agree on "now" between nodes (and, no, wall-clock time won't do it). These are difficult problems, that in and of themselves could take a separate book to explore.

The mechanisms to do this in a distributed environment begin to consume a significant part of the resources of the system as a whole, and this gets worse as the size of the distributed group grows. At the same time, during the periods when we are updating state, each node must avoid computing with that state until they all agree on its new value, severely limiting our ability to do processing in parallel, and thus our scalability.

Location Transparency

Location transparency is the feature of a system by which we can perform computations without being concerned with the location (e.g., the node) at which the computation occurs.

In Akka, the flow of messages between actors should be, to the developer, location transparent; that is, we should not care when we send a message whether the recipient is on the same node as the sender. The system itself moves messages across the network as needed to the correct location, without our direct involvement.

Delivery-Guarantees

At Most Once

At Most Once delivery is the simplest guarantee to achieve. It requires no storage, either in memory or on disk, of the messages on either end of the pipeline. This delivery guarantee means that when you send a message, you have no idea whether it will arrive, but that's OK. With this delivery guarantee you must accept the possibility that the message might be lost and thus design around that.

At Most Once delivery simply means that you send the message and then move on. There is no waiting for an acknowledgment. This is the default delivery guarantee in Akka. When a message is sent to an actor using Akka, there is no guarantee that the message will be received. If the actor is local, you can probably assume that it will be delivered, as long as the system doesn't fail, but if the system does fail before the actor can remove the message from the mailbox, that message will be lost. Remember, message processing is asynchronous, so just because the message has been sent does not mean it has been delivered.

Things become even more complicated if the actor is a remote or clustered actor. In this case, you need to deal with the possibility of the remote actor system crashing and losing the message, but you must also consider the possibility that the message might be lost due to a network problem. Regardless of whether the actor is local or remote, the message might not be delivered. If the delivery of the message is critical, you will need to try to work toward an alternative delivery guarantee.

At Least Once

At Least Once delivery is more difficult to achieve in any system. It requires storage of the message on the sender as well as an acknowledgment from the receiver. This storage might be in memory or it might be on disk depending on how important that message delivery is. If the delivery absolutely must succeed, you need to ensure that it is stored somewhere reliable; otherwise you could lose it if the sender fails.

Whether you are storing in memory or on disk, the basic process is to store the message, and then send it and wait for an acknowledgment. If no acknowledgment is received, you resend the message, and continue to do so until you successfully receive an acknowledgment.

With an At Least Once delivery guarantee, there is the possibility of receiving the message twice. If the acknowledgment is lost, when you send the message again, you will have a duplicate delivery. However, because you continue to send until you receive the acknowledgment, you are always guaranteed to get the message at least once, eventually. This delivery mechanism is reliable.

In Akka, you can implement At Least Once delivery a few ways. The first way is to do it manually. In this case, you simply store the message in a queue somewhere, send it, and then expect a response. When the response comes back, you remove the message from the queue. You also need a recovery mechanism so that if the response is never received, you can resend the message.

You can implement this easily in memory by using the Ask pattern. In this case, it might look something like the following:

```
class MySender(receiver: ActorRef) extends Actor {
  import context.dispatcher
  implicit val askTimeout = Timeout(5.seconds)

  sendMessage(Message("Hello"))

  private def sendMessage(message: Message):Future[Ack] = {
    (receiver ? message).mapTo[Ack].recoverWith {
    case ex: AskTimeoutException => sendMessage(message)
    }
  }

  override def receive: Receive = Actor.emptyBehavior
}
```

This is a very trivial example — you send a message, and then in the event of an `AskTimeoutException`, you try resending it. Of course, this type of delivery is only reliable as long as the sender doesn't crash. If it does, this is not going give you the At Least Once delivery guarantee.

You could adapt the solution that we just described, introducing some database or reliable disk storage. However, it turns out that Akka Persistence has all of this logic built in and provides an `AtLeastOnceDelivery` trait that you can use for this exact purpose, as shown in the

code that follows:

```
case class SendMessage(message: String)
case class MessageSent(message: String)

case class AcknowledgeDelivery(deliveryId: Long, message: String)
case class DeliveryAcknowledged(deliveryId: Long)

object MySender {
  def props(receiver: ActorRef) = Props(new MySender(receiver))
}

class MySender(receiver: ActorRef) extends PersistentActor
  with AtLeastOnceDelivery {
  override def persistenceId: String = "PersistenceId"

  override def receiveRecover: Receive = {
    case MessageSent(message) =>
      deliver(receiver.path)(deliveryId =>
        AcknowledgeDelivery(deliveryId, message))
    case DeliveryAcknowledged(deliveryId) =>
      confirmDelivery(deliveryId)
  }

  override def receiveCommand: Receive = {
    case SendMessage(message) => persist(MessageSent(message)) { request =>
      deliver(receiver.path)(deliveryId =>
        AcknowledgeDelivery(deliveryId, message))
    }
    case ack: DeliveryAcknowledged => persist(ack) { ack =>
      confirmDelivery(ack.deliveryId)
    }
  }
}
```

This is a simple implementation of an actor that uses the `AtLeastOnceDelivery` mechanism. Note that it extends `PersistentActor with AtLeastOnceDelivery`. This actor will receive a command in the form of `SendMessage`. When the actor receives this command, it sends the message to another actor, but here you want to guarantee delivery. Delivery is guaranteed by persisting the message prior to sending it, and then using the `deliver` method rather than the standard `tell`. When you use the `deliver` method, a delivery ID is generated, which you need to include in the outgoing message (in this case, `AcknowledgeDelivery`).

When the receiving actor gets the resulting message, including the delivery ID, it responds to the sender with another message that contains that same delivery ID (in this case, `MessageAcknowledged`):

```
sender() ! MessageAcknowledged(deliveryId)
```

The sender receives and persists the acknowledgment, and simultaneously confirms delivery, as shown in Figure 6-1.

In Figure 6-1, you can see how the sender stores outbound messages and then checks them against `DeliveryAcknowledged`, at which point the outbound message is confirmed to be delivered and no longer needs to be stored.

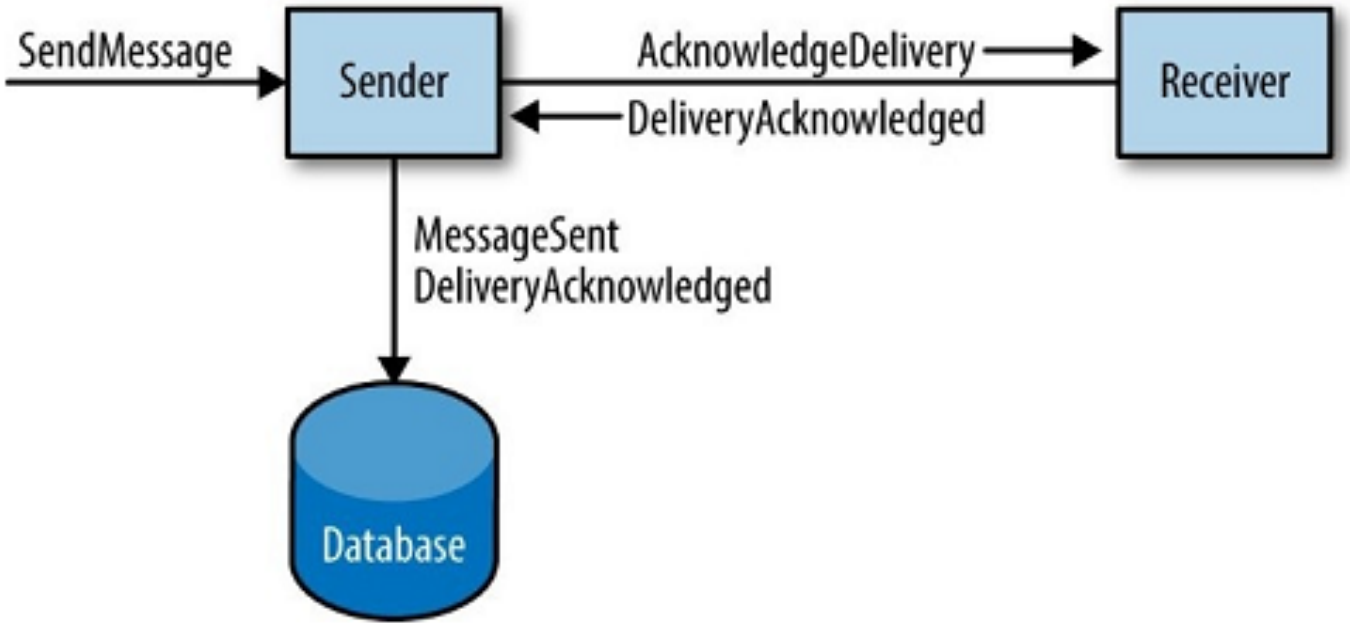

Figure 6-1. Message flow for At Least Once delivery

Under the hood, the actor has a configurable timer. If it does not receive the acknowledgment within the allotted time, it will automatically attempt redelivery.

So, what happens if the sender fails? In the event of a failure, the sender will replay the persisted messages and their corresponding acknowledgments when it is restarted. However, unlike with a standard persistent actor, there is a buffering mechanism in place. During the replay, the messages are not resent immediately. Instead, they are buffered. After all messages have been replayed, the sender can then go through the buffer and compare the delivery requests against the confirmations. It will cross off all the messages that have a corresponding acknowledgment. Then, the sender will resend any messages that are left.

This means that if the receiver fails, or the acknowledgment is lost, the timeout will ensure that the message is redelivered. If the sender fails before it can receive the acknowledgment, when it is restored it will see that no acknowledgment was received and will resend the message. This guarantees that the message will be delivered despite any failures.

Exactly Once Doesn't Exist (But Can Be Approximated)

In many cases, the delivery guarantee that you want is an Exactly Once guarantee. That is, you want each message to be delivered to the destination exactly once, no more, no less. Unfortunately, this type of delivery is impossible to achieve in any system. Instead, you need to strike a balance that uses the delivery guarantees that you can achieve.

Let's consider a very simple example. In our domain, there are multiple bounded contexts. There's a bounded context responsible for updating information on the people in the system, and there's the scheduling bounded context that is responsible for managing the allocation of those people. If you need to delete or deactivate a person (perhaps they have moved to another company), you will need to update both systems. Of course, you want both systems to agree on whether the person is in the system and what the person's allocation looks like. The simple solution seems to be to update one system and have it send a message to the other to perform the appropriate action.

So, the people information system is updated to remove a specific person. That system then sends a message to the scheduling system to remove that person. It is important that this message is delivered, so you wait for an acknowledgment. But what if the acknowledgment never comes? What if there is a network partition while the people information system was trying to send the message? How do you recover? You could assume the message never made it through and send it again. But if the message did get through and it was the acknowledgment that was lost, resending the message would result in a duplicate delivery. On the other hand, if you assume that the message has been received and don't resend it, you run the risk of never delivering the message. Any way you look at it, there is no way to guarantee that the message was delivered exactly once.

But Exactly Once delivery is often what you want. You need to find a way to simulate it.

How Do You Approximate Exactly Once Delivery?

The guarantee that many developers and designers *think* they want is Exactly Once; that is, a message is delivered reliably from the sender to the receiver exactly one time, with no duplications.

In practice, Exactly Once delivery is still impossible, but it is possible to achieve something close to Exactly Once processing, but only by means of an underlying mechanism that abstracts away the problem for us. Under the covers, that solution will be using a mechanism that provides At Least Once delivery to make it appear at the high level that you have Exactly Once processing, at least from the programmer's perspective.

The developer can then send a message via this Exactly Once abstraction and rely on the underlying mechanism to transmit the message, receive an acknowledgment, retransmit if needed, and so forth until the message has been confirmed as received, deduplicated for any repetition on the receiving side, and finally, delivered to the recipient. Of course, this process can affect performance, especially if many retransmissions must occur before the confirmation can be made. It also definitely requires persistence, at least on the sender's side, and often on both the sender and receiver's side.

It is often desirable to have more visibility to the underlying mechanism, so it is fairly uncommon in Akka systems to see an abstraction to simulate Exactly Once.

Cluster Singleton

Sometimes, there are areas within a system that by necessity must be unique. Creating multiple copies is unacceptable because these areas of the system might have state that must be very tightly controlled. An example of this would be a unique ID generator. A unique ID generator might need to keep track of previously used IDs. Or, it might use a monotonically increasing number value. In either case, it might be critical that you ensure that only a single call to generate an ID happens at any given time. Trying to generate IDs in parallel could perhaps result in duplicates.

An ID generator such as in this example is a bottleneck in the system and should be avoided whenever possible. But sometimes there are situations for which it is necessary. In this case, Akka provides a facility for ensuring that only a single instance of an actor is available throughout a clustered system. This is done through an Akka *cluster singleton*.

A cluster singleton works by having the singleton always run on the oldest available node in the cluster. Determining which node is the oldest is done via the actor system's gossip mechanism. After the cluster has established the oldest node, the singleton instance of the actor can be instantiated on that node. All of this is handled behind the scenes through the use of a cluster singleton manager. This manager will take care of the details of ensuring the singleton is running.

There is a wrapper actor around the singleton called the *cluster singleton proxy*. This proxy is used to communicate with the singleton. All messages are passed through the proxy, and it is the job of the proxy to determine to which node to send the message. Again, where that singleton lives is transparent to clients. They need know only about an instance of the proxy, and the details are handled under the hood.

But what happens if the node hosting the singleton is lost? In this case, there will be a period of time during which that singleton is unavailable. In the meantime, the cluster will need to reestablish which node is the oldest and reinitialize the singleton on that node. During this period of unavailability, any messages sent to the cluster singleton proxy will be buffered. As soon as the singleton becomes available again, the messages will be delivered. If those messages had a timeout associated with them, it is possible that they will timeout before the singleton is reestablished.

One of the big disadvantages of the singleton pattern is that you cannot guarantee availability. There is simply no way to ensure that there is always a node available. There will always be a transition period from when the singleton is lost to when it is reestablished. Even though that period might be short in many cases, it still represents a potential failure in the system.

Another disadvantage to the singleton pattern is that it is a bottleneck. Because there can be only one instance of the actor, it can process only one message at a time. This means that the system could be stuck waiting for long periods while it works its way through a backlog.

Another potential issue is that when the singleton is migrated from one node to the next, any

state stored in the singleton is lost. It will then be necessary to re-create that state when the singleton is reestablished. One way to do this would be to use Akka Persistence to maintain that state. In that case, it will be automatically restored when the singleton is re-created.

Because of the disadvantages of the cluster singleton pattern, it is recommended that you avoid it, except when absolutely necessary. And when it is necessary, you should keep it as small as possible. For example, if you were using the singleton to generate unique user IDs, it would be best if that was all it did. Although it might be tempting to have the singleton create the users in the database and initialize their state, each additional operation the singleton needs to perform creates a bigger bottleneck. If you can limit it to a single operation, ideally in memory, you can ensure that you are keeping it as available as possible.

Here is an example of what a very simple ID generator might look like if it were implemented using Akka Persistence:

```
object IdProvider {
  case object GenerateId
  case class IdGenerated(id: Int)

  def props() = Props(new IdProvider)
}

class IdProvider extends PersistentActor {
  import IdProvider._

  override val persistenceId: String = "IdProvider"

  private var currentId = 0

  override def receiveRecover: Receive = {
    case IdGenerated(id) =>
      currentId = id
  }

  override def receiveCommand: Receive = {
    case GenerateId =>
      persist(IdGenerated(currentId + 1)) { evt =>
        currentId = evt.id
        sender() ! evt
      }
  }
}
```

The `IdProvider` class is very simple. It takes a single message, `GenerateId`, and returns a result `IdGenerated`. The actor itself, upon receiving a `GenerateId` command, will create an `IdGenerated` event, persist that event, update the current ID, and then send the event back to the sender.

When the `IdProvider` is re-created, in the event of a failure, it will playback all the previous messages using the `receiveRecover` behavior, restoring its state in the process, so nothing will have been lost. It will then continue to generate IDs on the recovered node as though there were no interruption.

This `IdProvider` is in no way related to Akka Cluster. This code could be used in any actor

system, whether it is clustered or not. This is an example of how location transparency can be a benefit. You could build your system under the assumption that this will be a local actor, and later, when you decide to make it a clustered actor, this code doesn't need to change.

After you have decided that you want to cluster this actor and in fact make it into a singleton, you can do that with very little actual code. To cluster this actor, use the following:

```
val singletonManager = system.actorOf(ClusterSingletonManager.props(
    singletonProps = IdProvider.props(),
    terminationMessage = PoisonPill,
    settings = ClusterSingletonManagerSettings(system)),
  name = "IdProvider"
)
```

This is creating the instance of the `ClusterSingletonManager`. This manager must be created on any nodes on which you might want to host the singleton. It will then ensure that the singleton is created on one of those nodes.

You don't send messages through the manager; to send messages to the actor, you need to create a `ClusterSingletonProxy`. To create a proxy, use the following:

```
val idProvider = system.actorOf(ClusterSingletonProxy.props(
    singletonManagerPath = "/user/IdProvider",
    settings = ClusterSingletonProxySettings(system)),
  name = "IdProvider")
```

This proxy takes a path to the actor. What you get back is an `ActorRef`, but that `ActorRef` will now proxy to the singleton, wherever it might be. You can use this `ActorRef` just like any other `ActorRef` in the system. Again, this demonstrates the power of location transparency. Any actors that previously were sending messages to a local `ActorRef` can now be given this proxy instead. They don't need to know that the proxy is now a cluster singleton actor. They can continue to operate as though the actor were local, and the bulk of the code remains unchanged. The only place in the code where you do need to make changes is in the construction of the actor. Rather than constructing a local actor, you are now constructing the cluster singleton proxy. That's it.

Scalability

Scalability refers to the ability of a system to handle higher load without failure or sharp degradation in performance. It is only tangentially related to performance, and optimizing performance is no guarantee of scalability (and vice versa).

Increasing performance means that your system can respond *faster* to the *same* load, whereas scalability is more concerned with the system's reaction to *higher* load, whether that is more simultaneous requests, a larger dataset, or simply a higher rate of requests.

Figure 6-2 graphs the response time under increasing load for a system. The graph shows that a system that fails to scale will tend to have an inflection point where it either fails (wholly or partially) and some requests are not handled, or where it begins to rapidly decline in performance; that is, the response time for each request goes up rapidly with increasing load. The latter will usually also result in failure at some point — the client will often time out when the response time is too long.

Controlling Failures by Using Circuit Breakers

Suppose that the system encountered a failure that caused it to no longer meet your SLA. Calls to a particular piece of the system are timing out. You used bulkheading in some manner to isolate that failure so that the rest of the system can continue to operate. Perhaps the reason for the failure is heavy load on that particular area of the system.

Let's consider what happens if another call comes in to the failed area of the system. That call will be added to the queue for the failing system, and the failing system will need to process it when it can. But that system is already struggling to keep up, which means that the added request is just going to make things worse. As more requests continue to be sent to the failing system, the problem intensifies. Requests take longer and longer without giving the system any time to catch up. Meanwhile, all those requests have an SLA that isn't being met, which means they are timing out. The system might even retry the request, hoping that a second attempt will be successful, but that in turn further compounds the existing problem. This can end up being a big mess.

Let's look at how Akka helps to improve this scenario. Akka provides a tool called a Circuit Breaker that is similar in concept to an electrical circuit breaker in your home. The purpose of an electrical circuit breaker is to detect a fault in the electrical system and shut off the flow of electricity, preventing the problem from becoming worse. An Akka Circuit Breaker serves the same purpose. It is there to detect a fault and shut off the flow of messages in order to prevent the problem from escalating.

An Akka Circuit Breaker is used to wrap what might be considered a "dangerous" call. The dangerous call can be either a synchronous call that returns a value or throws an exception, or it could be an asynchronous call that returns a future. During normal operation the calls will be handled as expected. Each successful execution resets a failure counter to zero. In this case, the breaker is in the "closed" state.

When the dangerous call wrapped by the Circuit Breaker begins to fail, the behavior of the breaker changes. The failure counter begins to increment with each failure until it reaches a configurable maximum. When this maximum value is reached, the breaker moves into the "open" state. In the open state, the breaker automatically fails all calls. This means that it won't even attempt the calls. Instead, it just automatically generates a failure; in this case, the failure is a `CircuitBreakerOpenException`. These failures happen quickly because there is no need to even attempt to do the call. The Circuit Breaker will continue to operate in the open state for a configurable time period.

After the configured time period has elapsed, the Circuit Breaker will move into a "half-open" state. While in the half-open state, the first call coming into the Circuit Breaker will be allowed to proceed as though it were in the closed state. If this call succeeds, the breaker will move into the closed state and normal operation can resume. If the first call fails, though, the breaker will move back into the open state and it will continue to automatically fail the calls. If

it does move back to the open state, it will again stay in that state for the configured time period (Figure 7-4).

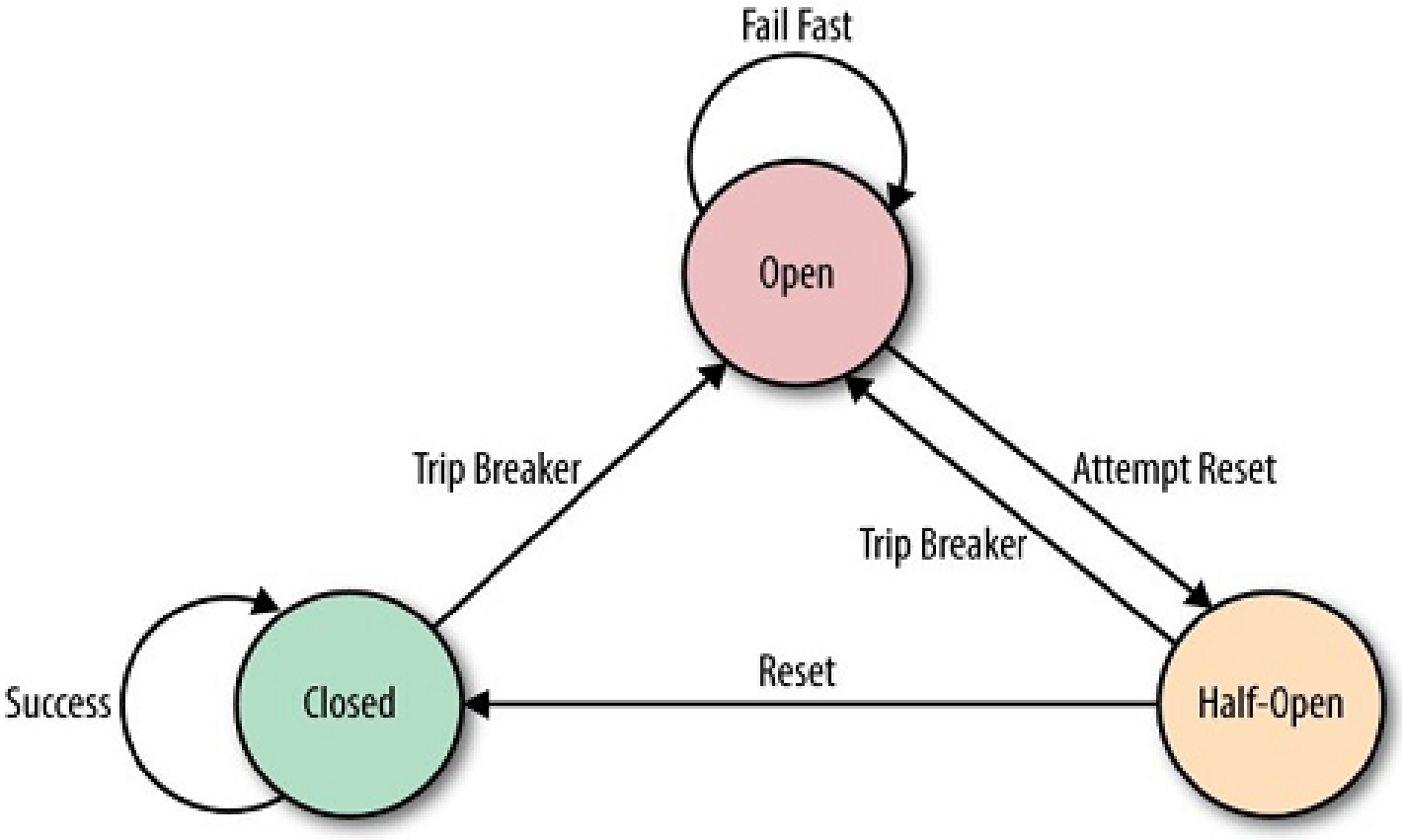

Figure 7-4. Circuit Breaker states

The benefit to using the Circuit Breaker is that it provides a way for you to give the failing system time to recover. Rather than continuing to call it, potentially making the problem worse, you instead automatically fail the calls without even trying the failed system. You should do this for a period of time, during which the failed system has an opportunity to recover. If it recovers and you make another call, it will succeed and normal operation will continue. If the system does not recover by the end of the time period, you will let through only a single call before automatically failing. In this way, you have created only a small amount of load on that failed system and avoided overstressing it, which will hopefully give the system time to recover.

Creating an Akka Circuit Breaker is fairly simple:

```
val circuitBreaker = new CircuitBreaker(
  context.system.scheduler,
  maxFailures = 5,
  callTimeout = 5.seconds,
  resetTimeout = 30.seconds)
```

This creates a Circuit Breaker that will allow for a maximum of five failures. It also includes a `callTimeout` so that if the call doesn't fail but simply takes too long to complete, that too

can be counted as a failure. Finally, there is a `resetTimeout`. This sets the length of time that the system must wait after the breaker is in the open state before it can transition to the half-open state.

You can use the Circuit Breaker to wrap both synchronous and asynchronous calls. For an asynchronous call, you can do something like the following:

```
val result = circuitBreaker.withCircuitBreaker {
  (people ? CreatePerson(id)).mapTo[PersonCreated]
}
```

The call to `withCircuitBreaker` wraps the future passed into it such that the result will still be a future of the same type containing the result if the call succeeds, or the resulting exception if the call fails. But if the Circuit Breaker is in an open state, that future will instead fail with a `CircuitBreakerOpenException`.

To wrap a synchronous call, do the following:

```
val result = circuitBreaker.withSyncCircuitBreaker {
  // dangerous code
}
```

This returns the result of the call directly without wrapping it in a future. It is a blocking call. If the Circuit Breaker is open, the call to `withSyncCircuitBreaker` will throw the `CircuitBreakerOpenException`.

Dealing with Failures

Although isolating and controlling failures is important for building a strong system, it isn't enough by itself. You can use techniques like graceful degradation and circuit breakers to keep the problem from spreading, but what happens if you need to recover from the problem? Or, can you simply prevent the problem from even occurring in the first place?

It's the same with Akka. We do our best to prevent failures, but we recognize that they are a reality. When they happen we simply "let it crash." It's how we recover from that crash that makes our system resilient. Understanding the "let it crash" mentality is critical to building reactive systems. All systems, no matter the level of complexity, have the potential for failures. If we don't plan for those failures and react accordingly, the system will fail.

The techniques we use to deal with a failure can vary depending on the nature of the failure. We deal with an exception differently than we deal with a hardware failure. And yet, at the core, we can apply the mentality of "let it crash" no matter at what level the failure occurs.

A `supervisorStrategy` also includes a `Decider`. A `Decider` is a `PartialFunction[Throwable, Directive]`. The `Decider` has the job of determining what course of action to take depending on what type of Exception has occurred. A `Decider` basically maps from the exception type to one of the following directives:

`Resume`

> The `Resume` directive tells the supervisor that the failing actor should continue processing messages as though no failure occurred. No change in state occurs. This is useful if the actor is stateless or if the actor's state has not been invalidated by the failure.

`Restart`

> The `Restart` directive tells the supervisor that the failing actor should be restarted. This means that the existing actor will be stopped and a new actor will be put in its place. This will result in any state for that actor being reset to the initial state. This is useful if the actor's state has been invalidated by the failure.

`Stop`

> The `Stop` directive tells the supervisor that the failing actor should be stopped and not replaced. This directive is useful if the failure results in an actor that is no longer needed or that can't be recovered in any way. For example, if it was a single-use actor, created only to accomplish the current task, and that task fails, the actor is no longer needed and we can just stop it.

`Escalate`

> If the supervisor is unable to provide an appropriate action to take for the given failure, we can escalate that failure. This essentially results in the supervisor failing, and it will be up to its supervisor to handle the failure, instead. Think of this like rethrowing an exception.

We can now create a `supervisorStrategy` for one of the People actors. The People actor has the responsibility of supervising the Person actors. A very simple strategy for this People actor might look like this:

```
override val supervisorStrategy =
  OneForOneStrategy() {
    case _: AskTimeoutException => Resume
    case _: IllegalArgumentException => Stop
  }
```

This is a very trivial example. In this case, if you get an `AskTimeoutException`, you are assuming that the state of the actor is not corrupted. Maybe you can try the `Ask` again or move onto the next message, so we will simply resume when that occurs. For the `IllegalArgumentException`, let's assume that this occurs due to a problem with the data being passed into the constructor of the actor. If that data is illegal in some way, resuming or restarting the actor isn't going to help. Instead, you should stop the actor.

But simply providing a trivial action like this usually isn't enough. Presumably, you want this actor to do something. So, when it fails, you probably don't want to just ignore the failure. You probably also want to take some additional recovery actions. You might want to resend the failed message to try again. Perhaps you want to redirect that failed message to another actor to see if that actor can complete the task instead. Or, you might want to perform any number of other complex activities in order to properly recover from the failure.

In our restaurant, it isn't sufficient for the manager to tell the cook to just keep going about his business. The lost order needs to be remade — or if it can't be, something else needs to be done to recover from the situation. The server might need to go back to the customer to explain the situation. These actions all need to be initiated, probably by the manager.

A supervisor strategy doesn't need to be a simple mapping from an exception to a directive. You can put additional logic in, as well. For example, when you receive a particular exception, you could send a message to an actor to deal with that exception. You also could change the state of the supervisor if necessary. These options open up interesting possibilities for recovering from a failure. As long as you include enough information in the exception, it becomes possible for the supervisor to perform any steps necessary to recover from the error.

Let's take a very simple example. If a message fails to be processed for some reason, you might include that message within the exception:

```
case class MessageException(msg: Message, cause: Throwable) extends
  Exception(s"The Message Failed to Process: $msg", cause)
```

When that message is available, you can make use of it within the `supervisorStrategy`:

```
override val supervisorStrategy =
  OneForOneStrategy() {
    case MessageException(msg) =>
      sender ! msg
      Resume
  }
```

This is a very simple example of a self-healing system. In this case, when the exception occurs, you wrap it inside another exception and include the message in that exception. This gives the supervisor access to that message. The supervisor receives the exception, extracts the message, resends the message to the actor, and then directs the actor to resume processing. In this way, you essentially get to retry the failed message.

In truth, this isn't a great way to solve this particular problem. We use it here as a simple demonstration that you can do more than simply return a directive. A much better solution would be to leverage the `AtLeastOnceDelivery` mechanism covered in Chapter 6. This is far more customizable and allows the retry to be persistent as well.

Fatal errors in the JVM

Sometimes, no matter how careful you are, an error occurs that you can't handle. In these cases, whether it is an out-of-memory error, or a stack overflow, or some other error, the system is going to terminate. In these cases, there is little that you can do to heal the error within the JVM. The system crashes, and you are done.

Does this mean that you should accept these types of crashes as a fact of life and require manual intervention to recover? Or are there tools that you can use to recover even in the event of a catastrophic failure of this kind? Even for failures that are going to bring down the system, you can take steps when designing the system that will help it to recover from even the worst kinds of failures.

The first thing you need in order to provide self healing in the case of an absolute failure of the system is some way for the failed application to be restarted. There is nothing specific to Akka by which you can do this, but there are tools available that can help. Lightbend offers its own solution for this: ConductR. ConductR is a tool that you can use to create a cluster and specify how many of each application you want running. In the event that an application fails, ConductR will ensure that a new instance of that application is launched somewhere in the cluster to compensate. Other orchestration tools such as Mesos and Docker Swarm also

provide solutions for ensuring that instances of an application remain running.

Whether you use ConductR or an alternative application, the basic idea is the same. You need to have an application that will monitor your process in some way. If that process fails, the monitoring application will automatically restart it. This is a bit like your Akka supervisors, only in this case the directive is always `Restart`, and rather than detecting exceptions, you are determining failure by using other means.

Monitoring your applications and automatically restarting them can be a lifesaver from an operations perspective. We have seen cases in which an application has a severe bug that causes it to restart every couple of minutes, yet despite this, the application has been able to serve clients all night simply because every time it failed the monitoring application restarted it. And due to the asynchronous reactive design of the system, clients were unaware that a problem had occurred. This type of setup can save you from the dreaded 3 A.M. phone call.

But it's not enough just to restart the application. This is a step in the right direction, but much like with our nonfatal errors, we also need to have some kind of self-healing mechanism in place to recover from the failure. So how can you implement self healing in a system like this?

The key to healing these systems is to have a fallback point in the application. You need a point in the system at which you can draw a line in the sand and say that when the system reaches that point, your data is safe. Prior to that point, any failures will result in the loss of that data; after that point, any failures will be recoverable using the data that was provided up to that point. The ideal circumstance is to push that point as close to the beginning of the process as possible. You want to reach your safe point as quickly as possible. What does that look like in a real system?

Let's look at our scheduling domain example. Upon making a request to the scheduling domain to perform the scheduling actions for a particular project, when do we consider that command to be complete? Do we consider it complete when the project has been fully scheduled and all the results are known? That is the end goal, but what if that takes hours? We don't really want to have any processes that take hours, so how can we simplify that?

A better way to handle this is to consider the command to be done as soon as it is accepted and persisted. In other words, when we make a request to schedule a project, we take that request, push it on to a persistent queue of some kind, and then at that point, we can send an acknowledgment back to the client saying that its command has been accepted and will be processed asynchronously. A failure before the command is persisted will result in an error to the client. A failure after the command has been persisted will result in a retry or engaging some kind of self-healing logic.

There are many ways that you might implement this kind of logic, but they all rely on one simple concept: all of them require At Least Once delivery. You need to be able to guarantee that a message will be delivered, even if a failure occurs. One way to do this is to use an

external persistent message bus. You want a message bus that will deliver to a client and allow the client to acknowledge receipt of the message. If you have this, you can use it to provide us with automatic retries. When a command enters the system, it performs any validation required on that command and then converts it to another message. That message is then sent to the message bus. After the message is sent to the message bus, the command is considered complete. After that point, any further processing happens asynchronously and failure is no longer an option. You might delay processing due to an error, but that processing will eventually be recovered. If the message is unable to be delivered or is not acknowledged, the message bus will retry at a later point.

Within the context of Akka, you can use tools like Akka Persistence to achieve a similar effect. Again there are multiple ways by which you can do this, but let's look at one possibility. Akka Persistence introduces the concept of At Least Once delivery. You can take advantage of this to provide a message delivery guarantee. The Command processor can receive the command, validate it, and then send the resulting message into the system by using `AtLeastOnceDelivery`. This means that if the system fails to receive and acknowledge the message, the Command processor will resend it. It will continue to resend until you have confirmation that the message has been completed. This gives you a form of reliable delivery, but it also provides the retry mechanism you were looking for. You can now guarantee that the message will eventually be handled. Even if the Command processor were to fail, because it is a persistent actor, when it is re-created it will continue to try to send that message.

Here's your fallback point in the event of an error. If an error occurs in the system and you are unable to finish processing the message, the message will not be acknowledged. This will then force the system to retry the message at a later date. Depending on the complexity of the message, it might be necessary to create multiple fallback points within the message processing pipeline. Throughout the pipeline, you can use `PersistentActors` and `AtLeastOnceDelivery` where required to ensure that if the system experiences a catastrophic failure, it can re-create the necessary state for it to continue.

This only works in a system that has embraced an asynchronous design. If your system is trapped by the need to make everything synchronous, you lose this capability. In a synchronous system, you can't allow failure. Everything needs to complete successfully before the system acknowledges that the command has been completed. If a piece of the system fails, you need to fail the command. You can't wait 15 minutes for that piece of the system to be reestablished. On the other hand, if the system is mostly asynchronous, this isn't a problem. The system can accept the command and persist it. If it can't be completed now, perhaps due to a failure in the system, the system can try to complete it later, after it comes back online. No one needs to even know that the failure occurred.

Availability

Microservices Versus Monoliths

Today, there are two often-discussed approaches to building applications. Applications can either be *monolithic* or they can be built using *microservices*. Both approaches exist on either end of a spectrum on which most software projects fall somewhere in between. Very few applications are completely monolithic or perfect microservices.

Monolithic applications are built such that all of the components of the application are deployed as a single unit. Complexity in monolithic applications is isolated by creating libraries that are then knitted together into the larger whole.

Microservices, on the other hand, are built by dividing the larger application into smaller, specialized services. These microservices perform very small tasks in an isolated fashion. Complexity is isolated to the individual microservices and then put together using other microservices.

Availability is not really a question of monoliths or microservices. You can create highly available applications using either approach. However, the manner in which you do that, and the effects on the scalability of your application, are dependent on which route you choose.

In a monolithic application there is only a single deployable unit. In this case, the way we provide availability — and scalability — is to simply deploy more units. If any one unit is lost, others will pick up the slack. Akka doesn't really help us here. The deployed units don't need to communicate with one another, so there is no need to introduce features like Akka Cluster. Within the monolith, we can use Akka to help us make better use of resources. We can also use it to help isolate faults by using bulkheading and other techniques. Akka does have a place within a monolithic system, but when we are speaking of availability using Akka, we are more interested in techniques that don't fit well in a monolithic application.

Providing availability with microservices is similar to monoliths in that the way to provide it is by deploying multiple instances of each microservice. However, in this case, because each microservice is separate, it can be deployed independently, scaled independently, and our availability needs can be tuned independently. Some areas of a system might be critical and need to be highly available. Other areas might be less important, perhaps only needing to run for a few hours a day or even less. In these cases, availability might be much less of a concern.

When we have multiple microservices that need to communicate with one another, Akka provides facilities that can help. But, before we go into detail about what facilities Akka provides, let's first take a moment to talk about how we divide up an application into microservices.

Bounded Contexts as Microservices

Domain-driven design (DDD) gives us a very natural way to divide an application into smaller pieces. Bounded contexts are excellent boundaries on which to divide. By definition, bounded contexts isolate areas of the domain from one another. Therefore, following this idea, we can isolate areas of our application along the same dividing lines. This means separating those areas out into separate microservices.

Sometimes, those microservices might have some shared elements, and you might need some common libraries in certain cases to share code. In DDD, bounded contexts can share domain elements through what is known as a *shared kernel*. This shared kernel is basically a set of common domain elements that can be used by multiple bounded contexts. In Akka, the message protocol that is passed between clustered applications is commonly found in the shared kernel.

This message protocol, as defined in the shared kernel, represents the API for the bounded context or microservice. It defines the inputs and outputs for that microservice. Depending on the mechanism of communication being used between the microservices, it might consist of a series of simple case classes that are directly sent to other actors, or it might represent case classes that are first serialized to JSON or some other format and then sent via an HTTP call to an Akka HTTP endpoint.

In our scheduling domain example, we have three bounded contexts defined: the User Management context, the Scheduling context, and the Skills context. We can separate each of these into microservices. We can then define either an HTTP endpoint to interact with those services, or we can directly send messages to the actors using Akka Cluster or Akka Remoting.

After we have successfully partitioned our application into microservices, we can then begin thinking about how we want to scale and deploy the application. Certain microservices might need many copies in order to provide scalability, whereas others might, by necessity, be limited to a single copy. Depending on the criteria, this can have an effect on how we make our application available.

Distributed Data

Within an eventually consistent, distributed system, such as the one we are building, we sometimes encounter situations in which there is transient data. This data is not something that needs to be saved in a database. It exists only for the life of the application. If the application is terminated, it is safe to terminate that data, as well. This could include things like user session information. When a user logs in, you might need to store information about that user. When did she log in? What security token did she use? What was the last activity she performed? Information like that is interesting, but keeping it isn't always valuable. After the user has logged out, that information becomes irrelevant and potentially needs to be cleaned up.

At the same time, it might be important that the transient information be available on all nodes. If a user is experiencing network problems and she becomes disconnected and then reconnected, she might end up reconnected to a different node. If that information is unavailable, the system loses its value. What this situation calls for is a way to maintain this data across several nodes in a cluster without saving it to a database.

In fact, there is a way. If you can represent that data by using data structures that follow specific criteria, you can replicate it reliably in an eventually consistent fashion. This replication can happen entirely in memory; no database needs to be involved. This gives you a distributed, eventually consistent method of storing and retrieving data.

These eventually consistent data types are called Conflict-Free Replicated Data Types, or CRDTs. CRDTs are an emerging concept. The idea first appeared in a paper in 2011 by Marc Shapiro et al. At this point, CRDTs are not widely used, but interest in them is growing, particularly for use cases in which a system handles very high volumes of data traffic but still needs to be performant.

Akka has its own implementation of CRDTs called *Distributed Data*. This is a new module in Akka, and it is still experiencing some flux. The API is changing as developers iron out the details of how it should work.

The basic idea behind CRDTs in Akka is that you have certain data types. These include Counters, Sets, Maps, and Registers. To be considered CRDTs, these data types must include a conflict-free merge function. This merge function has the job of taking two different states of the data (coming from two different locations in the cluster) and merging them together to create a final result. If this merge can be done without conflict, you can use this data structure to replicate across nodes.

Here's how it works. As each node receives updates to its data, it broadcasts its current state to other nodes. Other nodes receive the updated state, merge it with their own state, and then store the end result.

CRDTs typically work by storing extra information along with the state. Additive operations are often safe, but removal operations become more complex. For example, what happens if you try to remove an element from a Set that has not yet received the update to add that

Graceful Degradation

One of the benefits of breaking your application into microservices is that it enables graceful degradation. Within an application, we enable graceful degradation by setting up failure zones in the application. By creating actors in hierarchies that allow a section of the application to fail without bringing down the entire system, we allow our application to degrade in pieces rather than failing all at once. Microservices enable this same behavior but spread across multiple Java Virtual Machines (JVMs) and potentially multiple machines.

We would like to avoid the situation implied in the old quote "If at first you don't succeed, then perhaps skydiving is not for you." We want our applications to continue even in the face of failure.

In our sample scheduling domain, we can separate out a few services. We have the scheduling engine, the Project Management service, the Person Management service, and the Skills service. This means that if our scheduling engine fails, it doesn't prevent us from adding or removing projects or people. It only prevents us from scheduling people on a project. You can do this within a single monolithic application, by using actors to create failure zones, or you can do it by using microservices.

Graceful degradation means that while portions of an application might fail, the application as a whole can continue to operate, keeping it available even in the face of failure. It also means that noncritical portions of the application can be taken down, perhaps for maintenance or other reasons, without necessarily affecting your users in an adverse way.

Let's take a look at that. The scheduling engine in our system doesn't necessarily need to respond in a rapid fashion. It might be reasonable for a new project to be created but for it to take a period of time, minutes, or maybe even hours, before it returns results. There is no expectation that the moment a project is created it should be scheduled, as well. This means that if that section of the application were in need of some maintenance, perhaps a database upgrade, you could take the entire application down and perform that upgrade. In the meantime, users can still add new people and they can still add new projects; they just won't get the schedule for that project until after the maintenance is completed. And that's OK.

You can use the Circuit Breaker pattern discussed in Chapter 7 to provide graceful degradation. Typically, detection of the failure of an external system is a time-consuming operation. The system might need to wait for a connection to time out. If it did this for every subsequent request until the resource became available again, the system as a whole would take on an additional burden. By using the Circuit Breaker, the system is able to detect the first time a problem occurs and then quickly fail any additional requests until the timeout has elapsed. This helps keep subsequent failing requests from taking longer than they need to, and it also reduces the load on the system so that it can recover. This improves availability because rather than presenting timeout alerts, which can be time consuming, the system instead can quickly inform you about the error and which service is unavailable. Even though

a portion of the system is unavailable, it is still responsive, even if all it's doing is notifying you of the error.