Reactive Applications with Akka.NET

Anthony Brown

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Reactive Applications with Akka.net
Version 12

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

welcome

Thank you for purchasing the MEAP for *Reactive Applications with Akka.NET*, I hope that this book lays the solid foundations needed to ensure that you're able to make the most of the reactive manifesto to create applications and services which are truly capable of standing the trials and tribulations of a wide audience.

As software developers, we're at an interesting time thanks to the significant growth in popularity of computers of all shapes and sizes, whether they're in traditional devices such as laptops and desktops, smart entertainment devices such as TVs or in the booming Internet of Things market. This increase in demand is leading to the requirement for new and innovative solutions to be able to handle such a demand.

As you work through the book, you'll see how the Reactive Manifesto and reactive concepts fit into this new era of software development. In part 1 of the book we'll see an overview of reactive and why it's needed over the coming years as well as a more in depth look at how we're able to design new systems with the reactive traits in mind. From there, we'll introduce Akka.Net, an actor model implementation in .Net which allows us to write applications in the reactive style. Following these introductions, we'll start to build up an understanding of the fundamentals of writing applications using Akka.Net before we look at how we're able to apply these principles in the applications we write, all thanks to Akka.Net and the Akka.Net ecosystem.

Reactive Applications with Akka.NET has been written with a focus on those with little to no experience with any of Akka.Net, the actor model or reactive systems but experience of the difficulties you might encounter when you try and make applications which are resilient and scalable. I hope the book helps you on the journey to a thorough understanding of reactive applications and how using Akka.Net you might be able to alleviate some of these difficulties you've experienced in the past.

Since the title is available through MEAP, I implore you to leave comments such that I'm able to ensure that you and others have a solid understanding of what reactive is all about.

Once again, thank you,
—Anthony Brown

brief contents

1

Why reactive?

1.1 What does it mean to be reactive?

Over the past several decades, computers and the internet have moved from a position of relative obscurity into being a core component of many aspects of modern life. You now rely on the internet for all manner of day-to-day tasks, including shopping and keeping in contact with friends and family. This proliferation of computers and devices capable of accessing the internet has increased pressure on software developers to create applications that are able to withstand this near-exponential growth, meaning that you need to develop applications that are able to meet the demands of a modern populous who have grown to become dependent upon technology. Demands range in scope from a desire to provide information to users as soon as it's available, to the need to ensure that the services you provide are resilient to any issues they might encounter due to increased usage or an increased likelihood of failure, which may be caused by factors entirely outside of your control. When this is twinned with the demands of a rapidly evolving company, trying to beat the competition to find new gaps in an ever-changing marketplace, you're left needing to build applications that are not only able to stand satisfy demands imposed by users, but are also designed to be sufficiently malleable that they can be rapidly adapted and modified to fill potential gaps in the market.

In response to this, technology companies working across a broad range of different domains began to notice common patterns that were able to fulfill these new requirements. Common trends began to emerge, which were clearly visible to companies building the next generation of modern applications with a strong focus on huge datasets, up to the petabyte scale in certain instances, which needed to be analyzed and understood in record time, with results being delivered to users at near-instantaneous speeds. After following these principles, these systems were seen to be more robust, more resilient and more open to change. These principles were collected together and form the outcomes one can expect when you develop applications by implementing the Reactive Manifesto: a set of common shared traits that

exemplify a system design capable of standing up to the challenges we are exposed to when building applications to fulfill users' demands.

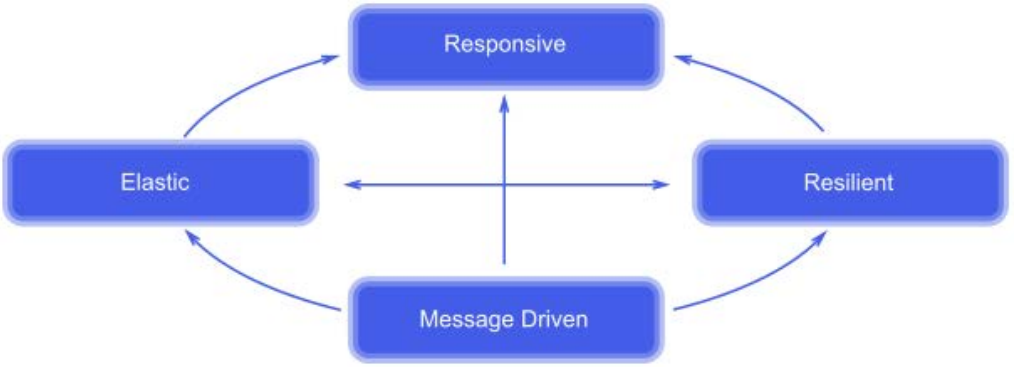
1.1.1 The heart of the Reactive Manifesto

At the core of the Reactive Manifesto is a common understanding that applications designed according to these traits should be able to respond to changes in their environment as quickly as is physically possible. A change in the environment could include any number of variables, whether it's a response to a change in the data of another component in the system, an increase in the error rate when attempting to process data or communicate with an external system, or an increase in the amount of data flowing through the system across component boundaries.

The implication of this is that the most important property of a modern application is that it should be responsive, and should respond to requests from users as quickly as possible. For example, in the context of a web application, the user should expect to see changes as soon as possible, whether this is by the application pushing data changes directly down to the user's web browser, or ensuring that it is able to be retrieved as quickly as possible when the user next requests the change. The term "responsive" is quite broad, and the definition of responsive in one domain may be vastly different to responsive in another context, so some consideration should be applied to what responsive means when applied to your applications. Many of the examples in this book apply to either web applications or real-time data solutions. These two cases alone include a number of potential interpretations of what responsive means. But you can exemplify these two cases as such: a web application should be responsive by responding to an HTTP request in the shortest possible time, whereas a data streaming solution should ensure data flows at a constant rate in an effort to prevent a stalled stream, which may have knock-on effects for other components earlier in the stream.

In order to achieve this level of responsiveness, you need to ensure that the systems you design are able to handle greater scale. Let's consider the example of a web application again. If this receives more web requests than the server is capable of handling, then it is inevitable that the incoming requests will start to be queued up, until the resources are available to service the requests. This queuing then leads to an increase in the response time for users, ultimately making the application less responsive. Similarly, in the case of a streaming data solution, you need to ensure that if more events start to flow through the stream, then you should have the ability to process them within a fixed amount of time; otherwise, it may cause the results for subsequent events to be delayed. But it's not enough to constantly provide more computing power; while computing power has dropped in price significantly in recent years, it's still far from cheap. As such, you should be able to respond to periods of inactivity or reduced throughput by negatively adjusting your provisioned compute resources so that you don't have to maintain or pay for these unnecessary resources. This can ultimately be categorized as the need for systems to be designed with elasticity in mind, that is, have the ability to expand when needed, but shrink back down to a bare minimum set of resources for the system to continue to operate when not.

In parallel to elasticity, it's important that systems are equally resilient, that is, they're able to react to a failure, whether it's a failure that originated from within the system, which you have some degree of control over, or it originates from other systems external to yours and over which you have no control. Within a streaming data solution, this might translate into the ability for your stream processing system to handle a situation in which you start to receive bad or invalid data from an incoming data source. For example, if this was an Internet of Things device sending sensor data, then your stream processing solution should be able to handle incoming data that may contain invalid sensor readings caused by a faulty sensor. If your application started to fail, then this would likely cause knock-on failures in other components within your system. Therefore, it's important that for an application to be resilient it should focus on the containment of errors in the smallest possible area of the application. Following this containment, you should then seek to recover from these failures automatically, without the need for manual intervention. This notion of resilience then ensures that the client does not end up being burdened with the responsibility for handling any failures that may occur within the system.



Finally, in order to drive the concepts we've seen thus far, message-driven systems are the core component that links everything together. By using messaging as the basis of communication between components within the system, you're able to perform work asynchronously and in a completely non-blocking manner. This ensures that you're able to perform more work in parallel, leading to an increase in overall responsiveness. By using message passing as the basis of communication, you're also able to redirect and divert messages at runtime as appropriate, allowing you to reroute a message from a failing component to one that is able to service the request. For example, if you had two servers, each of which was capable of servicing a request, then by using message passing you're able to change which server ultimately receives the request if one server becomes unavailable to service it. Similarly, if you notice one server has become a bottleneck, then you're able to divert a message to another server which is able to service the request. This means that you're able to dynamically add or remove new instances and automatically redirect the messages to the target instance.

You can see how these concepts work together, with messaging being the core building block that powers the resilience, elasticity, and responsiveness of the application. You can also see that elasticity and resilience are a shared concern; once you have the underlying infrastructure in place for resilience, this also provides the necessary logic for elasticity. Once all of these concepts are linked together, you're left with applications that are ultimately responsive.

1.1.2 Reactive systems vs reactive programming

The concepts surrounding the Reactive Manifesto are far from new, and ultimately stretch back over several decades. The manifesto is itself a formalization of a significant amount of domain knowledge from varying organizations. Due to the relatively broad concepts covered in the Reactive Manifesto, there is some overlap between two somewhat related programming concepts: reactive programming and reactive systems.

Reactive programming, like the programming model offered by Reactive Extensions, offers a smaller-scale overview of reactive programming, tailored to how data flows in a single application. Typical applications are driven by a threaded execution model, in which operations are performed sequentially in an order that you've defined, leaving you to deal with many of the underlying flow control primitives needed to synchronize data. In contrast, reactive programming is driven by the execution of code only when new data is available; typically, this is in the form of events arising from a data source. One such example of this would be a timer that ticks once every 5 minutes. Using typical programming patterns, you'd have to set up a loop that continuously polls until the minimum time period has elapsed before you progress through your application flow. But, when dealing with reactive programming, you create handlers that receive an event and are executed whenever a new event is received.

Reactive systems, however, focus on applying the same concepts on a much larger scale that deals with the integration of multiple distinct components. Many of the applications you build today are no longer basic programs, taking in an input and producing an output; instead, they are complex systems made up of an array of components, where each component could itself be an entire system. This level of interconnectedness brings with it complexities. Systems might not be running on the same physical hardware, and in fact may not even be collocated, with one system existing thousands of miles from another. This means that you now need to consider what happens in the event of failures, or how other system components will respond in the event of sudden floods of information passing through the system. We saw when discussing the Reactive Manifesto that these are the core requirements for an overall system to remain responsive, and we saw the way to achieve these aims was through the use of a higher-level message passing-based API.

This is the core difference between reactive programming and a reactive system. Reactive programming involves the notion of events: data that is broadcast to everybody who is listening to that event. This can be compared to reactive systems. These are message-driven, which brings with it individually addressable components to support targeted messages. Akka.NET is one example of a tool that simplifies the building of larger-scale reactive systems,

which we'll be seeing throughout this book, whereas Reactive Extensions is an example of reactive programming, which you won't be considering in this book. The two concepts can, however, be combined, with reactive programming being built on top of a reactive system, or reactive programming existing within a single component of a reactive system. But the combination of these concepts won't be addressed in this book.

1.2 Applying Akka.NET

Akka.NET is positioned as a platform upon which reactive systems can be built. This opens the door to using it across multiple distinct domains. It has seen similar use in Internet of Things-based applications, e-commerce, finance, and many other domains. But it is the internal requirements of these applications that drive whether Akka.NET is an ideal fit. One of the common concerns across these types of application is the requirement to update components based on the results of operations from previous components. Akka.NET starts to become a powerful tool once you need immediate responses from multiple components all integrated together.

1.2.1 Where to use Akka.NET

One example of where Akka.NET would be an ideal fit is in the world of commercial air travel. Here, multiple distinct components all produce data at an incredible rate, which needs to be processed and delivered to the user as soon as possible. For example, while a passenger sits in the terminal waiting to board their flight, they wait to see which gate their flight will depart from. This is particularly important in large airports where it might take 20-30 minutes to walk between gates. But there are a vast number of systems that are all integrated together and dictate where a flight ultimately travels and ends up. There is also national air traffic control, which reroutes flights in the event of an emergency and to prevent in-air collisions between planes in a congested airspace. There is also the airport's air traffic control, responsible for directing planes towards the correct runway; in the case of a large airport, landing on a difference runway could force the plane to a different gate. There are also airport operations that may be forced to divert a flight to a different gate due to a scheduling issue with another airline, which prevents the plane from arriving at that gate. Similarly, there's also data from the airline's internal systems, which might force a gate change due to internal scheduling problems. A vast array of data sources all publish data that needs to be processed as quickly as possibly, so that passengers are immediately aware of any changes that might occur as part of the effort to ensure that aircraft are able to turn around and take off as soon as possible after landing.

Although not all systems are as complex, or rely on as many distinct data sources, as an airline, there is a clear pattern of integrating multiple components together into a larger system, while also catering for any difficulties that might be encountered in the process. An airline, for instance, needs to immediately respond to changes when they are published, in an effort to protect the safety and security of all passengers and staff involved.

1.2.2 Where not to use Akka.NET

Although Akka.NET makes it easier to build larger reactive systems, it brings with it some difficulties. We've already seen how complex systems can be, which ultimately forces you to consider these complexities. For example, you need to think about partial failures of system components that might impact other components, you need to consider data consistency and how that should be handled in the case of partial failures, and plenty of other issues. Akka.NET brings these difficulties and complexities to the surface as first-class principles, which ultimately means that you have to deal with them. In dealing with them, you also bring to the surface a number of other complexities, notably harder debugging and the requirement to think about concurrency. Therefore, for fairly simple web applications that have basic requirements, Akka.NET is unlikely to provide any significant benefits. This includes relatively basic CRUD (Create, Read, Update, Delete) applications that are backed by a basic database model.

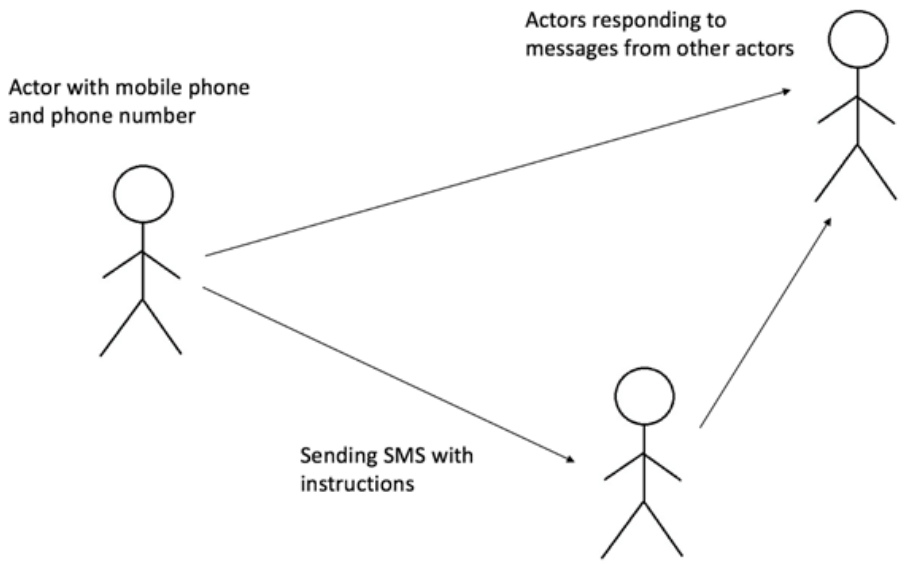
At its core, Akka.NET provides a concurrency model that is designed to allow multiple components to operate simultaneously. This means that when developing systems with Akka.NET, you need to think carefully about the data in your system. Although Akka.NET removes the possibility of concurrent access to shared data, there is still the opportunity for data races to occur, as well as the potential for deadlocks. For a system that doesn't need to operate concurrently, Akka.NET is likely to complicate matters and force more complications into your application, rather than simplifying it further.

1.3 How does Akka.NET work?

Although the concept of Akka.NET and how it works might be new to many developers, the underlying principles have been in development for decades, in the form of the actor model. As part of the actor model, independent entities, known as actors, are responsible for performing work. You can have multiple different types of actor within a system, and each of these types can have multiple instances running within the system. Every actor runs independently of every other actor within the system, meaning that two running actors are not capable of directly interfering or interacting with one another. Instead, each actor is supplied with a mailbox, which receives messages, and an address, which can be used to receive messages from other actors within the system. Actors will sit idle and not do anything until a new message is received in the actor's mailbox; at this point, the actor is able to process the message using its internal behavior. Its behavior is the brain of the actor and defines how it should respond to each message it receives. If an actor receives more than one message, the messages are queued up in the order in which they were received, and the actor processes each message sequentially. Each actor will only process a single message at any one time, although multiple actors can process their respective message at the same time. This allows you to create highly concurrent applications, without having to concern yourself with the underlying multithreading infrastructure and code that is typically required when developing concurrent applications. It's important to also note that the actors are completely isolated, meaning that any internal information or data owned by an actor is not accessible by anything other than that actor.

You can think of actors as being similar to people with a mobile phone. Each person has an address through which they can be contacted; in this case, the address is the phone number of the user you are calling. You also have access to a unique address for the person initiating the phone call; once again this is a phone number. Once you've got that address, you're able to communicate with the person you want to talk to. You can do this by sending an SMS with some data in it. As humans, the data you might include in an SMS is typically a question if you want to know an answer, or a statement if you want to inform the other person of something. The SMS you send ends up in the other person's mailbox, where they're able to asynchronously deal with it when they have the resources and bandwidth available. Like actors, every person is an independent, isolated entity with no ability to directly access other people's information. If you want to find out what plans a friend has for the weekend, you don't have direct access to that information; instead, what you do is send them an SMS asking for the information. This is the same pattern you use when sending data between actors: rather than directly accessing an actor's data, you send the actor a message asking for it and await the response.

Similar to humans, actors are able to perform a number of operations upon receiving a new message. The simplest approach is to ignore a message; if it's particularly important, the other party will resend the message and attempt to retrieve a response a second time. Alternatively, it could choose to send a message elsewhere in response to receiving a message. The actor might not have all the information available to create a complete response, but it's able to contact other actors within the system, who might have the information available, after which the actor is able to act on the message. For a particularly intense or long-running task, an actor can also spawn another actor that is solely responsible for performing that task. This is similar to how people delegate work to other people if they lack the time needed to perform the task, or if they have other pressing matters to attend to. You can also choose how you respond to the next message you receive by modifying the internal state of the actor. This is analogous to hearing some new information from one person that then influences answers to questions you receive from other people.



The core takeaway when considering the actor model is that its core design principle is to form an abstraction over the top of low-level multithreading concepts to simplify the process of developing concurrent applications. Understanding this, combined with the isolated nature of individual actors, ensures that the systems you build on top of Akka.NET are able to line up with the traits of the Reactive Manifesto.

1.4 What will you learn in this book?

This book focuses on how Akka.NET provides abstractions, which you as a developer can use to reduce the complexity of reactive applications. This chapter has shown the core traits that make up the Reactive Manifesto; in later chapters, we'll see how you can apply concepts from Akka.NET to closely align your systems with the aims of the Reactive Manifesto. In each chapter, we'll also look at a short case study where we will see how the Akka.NET feature can be used in a real-world application to help model a solution, using concepts covered in that chapter. In addition to these shorter case studies, two larger case studies will also be presented, showing the application of the Reactive Manifesto to a small component of an e-commerce application, as well as the use of Akka.NET to model a reactive solution to an Internet of Things solution. We'll also see the inherent complexities involved in building larger systems, where the actor model of Akka.NET helps to bring these issues to the forefront so that you are forced to consider the ramifications of failure within your system or supporting systems.

In order to get the most out of this book, you should be a software developer with some experience of the .NET framework, and specifically the C# language. All examples presented in this book will use the C# Akka.NET API, which is equally usable by other .NET languages, but

more idiomatic APIs are available and while the concepts remain the same, there may be significant differences between APIs. An understanding of the tradeoffs that must be made when developing larger systems in .NET is also beneficial, as well as a grasp of the difficulties typically encountered when developing asynchronous applications.

1.5 Conclusion

In this chapter, you learned:

- The core reasoning behind the move to reactive systems
- The underlying programming model behind Akka.NET

2

Reactive application design

This chapter covers

- **Overall design patterns typically used when designing reactive systems with Akka.NET**
- **Design concepts that can be utilized in e-commerce applications, healthcare systems, enterprise integration solutions, IoT products and many other types of application**
- **Understanding the tradeoffs which need to be made when designing a reactive system**

In chapter 1, we saw many of the reasons why you might want to design an application using the traits laid out in the Reactive Manifesto. This has been primarily driven by the changing face of technology over the past several decades. Whereas computers were once an obscurity, used primarily by researchers or organizations with sufficient funds, they have since been transformed into a ubiquity, with the vast majority of households now having at least one computer, smartphone, or tablet. This number is set to grow with the introduction of the Internet of Things, which is transforming many of the mundane tasks we perform on a daily basis by harnessing the power of an interconnected network of smart devices. This transformation is likely to replicate many of the changes we have already seen over the past few decades, where we've seen a number of industries adapt to provide their services in the internet age. One such example of this is the world of e-commerce, where we've seen more and more retailers providing their products and services through online stores. Online shopping has grown, with both more retailers and more consumers opting to use the internet as the basis for the majority of their shopping habits. This has led to a situation whereby all online retailers are in direct competition with each other. Although this level of competition directly benefits consumers by opening up readily available, competitive pricing to all, it puts a huge amount of pressure on retailers to ensure their overall experience is as close to perfect as possible; otherwise, customers will be able to easily transfer over to using competitors. These

are all areas that have been extensively researched, with findings showing that the likes of errors encountered and page loading times can all lead to customers moving to competitors.

2.1 Basic reactive system design

Given that the overall aim of the Reactive Manifesto, as we saw in chapter 1, is to provide a responsive experience to the end user, it becomes apparent that the principles of reactive application design could have significant benefits for the world of e-commerce. We saw the four tenets of a reactive application in chapter 1; it is responsive, fault tolerant, elastically scalable, and message-driven. Of these four, three are directly relevant to the end user's experience of an e-commerce website. If you want to ensure that customers are likely to stay on the website, then you need to ensure that pages load quickly and other actions are performed promptly. If a customer wants to spend money with a given e-commerce website, that website should do everything possible to ensure the customer's experience is as fluid as possible; otherwise, that website runs the risk of alienating the user and sending them to a competitor. You also want to ensure that an e-commerce website is elastically scalable, especially given the prevelance of sharp spikes of large numbers of visitors during peak periods. When we consider common shopping habits, we frequently see a large number of users all trying to access the website at a given period, whether that is driven by gift giving holidays such as Christmas, or around key shopping discount events such as Black Friday. In these cases, you want to ensure that you're able to service as many of those requests as possible. If you have a spike of an order of magnitude of more users wanting to make a purchase, then you should try and accommodate them as best as you can possibly manage; otherwise, you once again run the risk of customers flocking en masse to competing websites. Similarly, when designing for failure, you want to ensure that even if a non-essential component of an e-commerce website fails, you still want to accept the user's payment. For example, if a customer navigates to the checkout page to complete their purchase, they don't want to be faced with errors caused by non-essential features, such as recommendation services or advertising features intended to sell additional products or services. If either of these services fail, then you should still be able to complete the basic purchase.

This combination of basic requirements suggests that designing an e-commerce application using the traits specified in the Reactive Manifesto may provide significant benefits. But, effectively designing an application using the concepts of the Reactive Manifesto can require significant changes, both in terms of thinking and in the application architecture. In order to better understand reactive application design, in this chapter we'll look at how you're able to design a traditionally CRUD (create, read, update, delete)-based application using actors with Akka.NET and designed using the traits from the Reactive Manifesto. You'll consider some of the challenges and design decisions you're likely to encounter as you design applications using the Reactive Manifesto, as well as how you can effectively design an application using many of the features Akka.NET makes available to you.

2.2 Reactive e-commerce application design with actors

As we've already considered, the way we use computers has rapidly changed over the past several decades and they are now seen as a commodity that exists in the majority of households, many of whom also have an internet connection. But, users have also become more demanding, requiring more features to enhance their overall shopping experience. This includes the likes of recommendation engines to better suggest alternative products, trend calculations to predict which products are due to be the most popular and integrations with external third-party services that provide additional features and benefits. But there is also the requirement from within the business itself to add other sources of information in order to ensure that the company is able to gain insights into how customers are shopping and better position the business to respond to these demands. This produces high demands on the scalability of both the traffic and the application architecture.

Let's now consider how you're able to effectively design a reactive system, using an actor system-based approach, whereby you represent the entities of your system using actors. You'll put this into context by using a system many of us are familiar with, the e-commerce application. If you haven't had the experience of writing e-commerce websites, you've likely used one to make purchases in the past. We saw in the previous section why the world of e-commerce is a strong candidate for reactive application design, given that the final aim is to create applications that are responsive.

Given that an average e-commerce website is quite large, you won't focus on every component within the system; instead, you'll focus on one key aspect of the application: the final purchasing experience. The final purchasing experience is the part of the website the user will ultimately navigate to once they have finished browsing the website and are ready to confirm to the e-commerce site that they want to purchase a number of items they've seen. As such, this component will have a number of requirements, such as providing a shopping cart where users are able to store items as they browse the site, a checkout where users enter their shipping address and a payment gateway where users will enter any card details.

In chapter 1, we addressed what an actor is and how actors allow you to design applications with concurrency handled transparently for you. We also addressed the requirements of a reactive application and the principles you should try and adhere to if you want to be successful when building reactive applications. In the rest of this chapter, we'll start to consider how you're able to incorporate these ways of thinking into a real-life application. We'll also consider how these components would fit into the context of an application designed using Akka.NET, by linking these design ideas to the functionality and features provided as core components of the Akka.NET distribution.

2.2.1 A reactive shopping cart

The first component that a potential customer is likely to encounter is the shopping cart. This will be analagous to the shopping cart a consumer would encounter in a physical store, which they can use when browsing the store to maintain a collection of all of the items they've picked up and intend to purchase. As you may have many thousands of users browsing the site

at once, this will require you to support the simultaneous use of thousands of shopping carts in the application. To design this, you'll link a shopping cart, which is nothing more than a list of items and the quantity of items. Each of the shopping carts will be accessed through a unique identifier, which will be stored in the user's session. We saw in chapter 1 how one of the core principles of actors is the ability to store state. The state you can store in an actor is able to vary per type of actor you have within your application. In this case, however, you could choose to store a dictionary of the user's session identifier, along with a list of items and quantities. This would be how you might model the component if you were to use a database. But actors serialize all incoming messages; only one message is processed at any one time, in order. This means that if lots of users are trying to access their shopping carts at the same time, then they would have to sit through long queues. This then defeats the aim of providing a responsive experience to users.

But, as actors are capable of performing work in parallel, and are an incredibly cheap abstraction in terms of memory and computation, you can instead choose to create many actors to work concurrently. In terms of your shopping cart example, you could create a single actor per shopping cart. In this case, the actors are responsible for simply storing a list of items and associated quantities, where the actors are then addressed by the shopping cart identifier. You can see an example of this in the following figure; each actor is effectively a shopping cart, which is similar to how this might be modeled in the physical world, where you have one physical entity per customer.

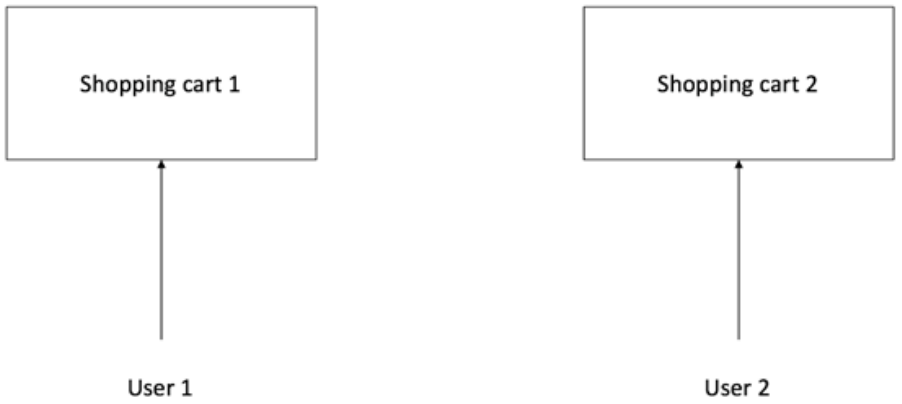


Figure 2.1 Creating one actor per shopping cart allows each shopping cart to be accessed independently of the others. This means there's no contention for resources between User 1 and User 2.

This leads us to the first design pattern you're likely to encounter in an actor-based reactive system. Where possible, you should partition work into actors based on concurrency boundaries; in the case of a shopping cart, a concurrency boundary is an individual shopping cart. Designing systems like this allows for the greatest throughput, ensuring your application

remains responsive. It also allows for easier scalability, as it allows you to fine-tune the deployment of the actor to better handle any increased throughput it might experience. We'll look at how to effectively design actors in the context of Akka.NET in the next chapter; we'll also see how you can handle an increased throughput by scaling actors out across multiple servers in chapter 12.

2.2.2 Changing states of actors

While a customer is browsing the store, they'll hopefully be adding to their cart items that interest them. But, there are a number of states a shopping cart can exist in. For example, the two key states are browsing, where the customer continues to browse through the store, and a state where the user is completing a purchase. When a user is browsing the store, they should still be able to add more items to their cart, but once they start the process of completing a purchase, you typically don't allow the cart to be modified. This is for a number of reasons, such as the need to reserve items while the purchase is being completed, and computing the overall cost of the shopping cart. You therefore want to prevent new items from being added to the cart while it is in the purchase completion state. We saw in the previous section how the shopping cart actor can be represented as an actor and as part of the actor model. Actors can change their state on demand to respond differently to subsequent messages. Given how this is such a fundamental component of designing actors within actor systems, Akka.NET provides the functionality to switch between states to invoke different behavior and simplify the process of responding to messages. In the following figure, you can see how you're able to switch between multiple different states within your shopping cart actor, which we discussed before.

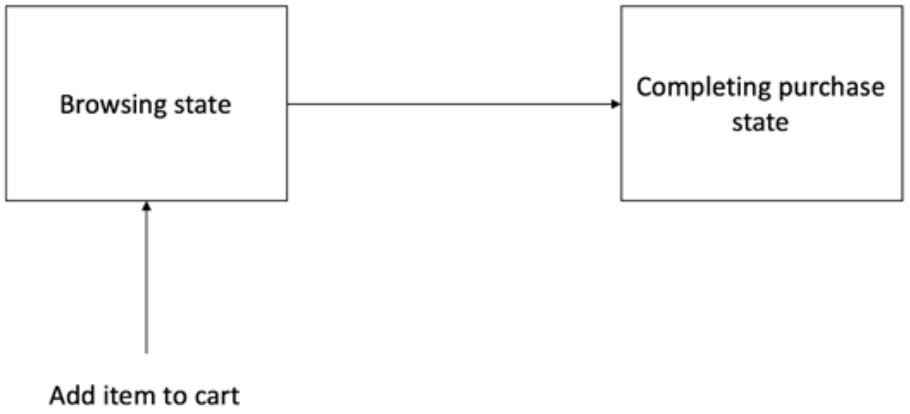


Figure 2.2 The use of state machines with actors in Akka.NET allows you to simplify the process of developing actors with complex internal logic

This shows another pattern you'll typically encounter with applications written using Akka.NET, whereby you switch between multiple different states as required. Specifying multiple different states allows you to focus on one individual state and the messages it is able to respond to, without needing to consider alternative scenarios simultaneously. This allows you to easily understand the logic embedded within actors at a later stage. We'll see how this can be applied to actors in chapter 4, where we consider the importance of state machines in Akka.NET applications and how you can represent them.

2.2.3 Making the purchase

Once a customer has finally decided on everything they want to purchase, you then need them to proceed through to the purchasing stage where you process the user's choice of payment. In an e-commerce website, you'll typically allow for purchases to be made with a credit or debit card. In order to simplify the development of complex systems involving credit card details, a simple solution is to integrate with an external payment gateway. The usual approach to handling this is to send a request to an external third-party service with a token representing the user's credit card details and the total value of the purchase. Given that you're integrating with external services, there's a high probability of failures happening, so you need the system to be able to cater for this. As we saw in chapter 1, for a system to be truly responsive, it needs to continue to work even in situations where individual components aren't working. Due to the high potential for failure when designing complex systems, the Akka.NET mindset is to embrace failure and provide the ability to recover from it quickly.

Within Akka.NET, you're able to supervise other actors that are spawned as children of the current actor. When designing actors in this way, you're able to perform a number of different operations once an actor is discovered to have failed; for example, you may want to restart the actor and attempt to retry whatever work it had been performing, or you might want to stop the actor's operations altogether. When designing applications using actors, you should try and push as much work down to child actors as possible. This allows us to isolate any potential issues you might encounter, as well as also allowing for a broader restart than simply retrying the operation, which hopefully will be sufficient to resolve the issue. In figure 2.3, you can see how an actor has to perform two tasks: upload some data to a web API responsible for processing the payment on a user's credit card, and then send a message to another actor on completion, informing it that the purchase has been completed. You push the potentially dangerous work, in this case the web API call, down to a child actor who is then responsible for performing the operation.

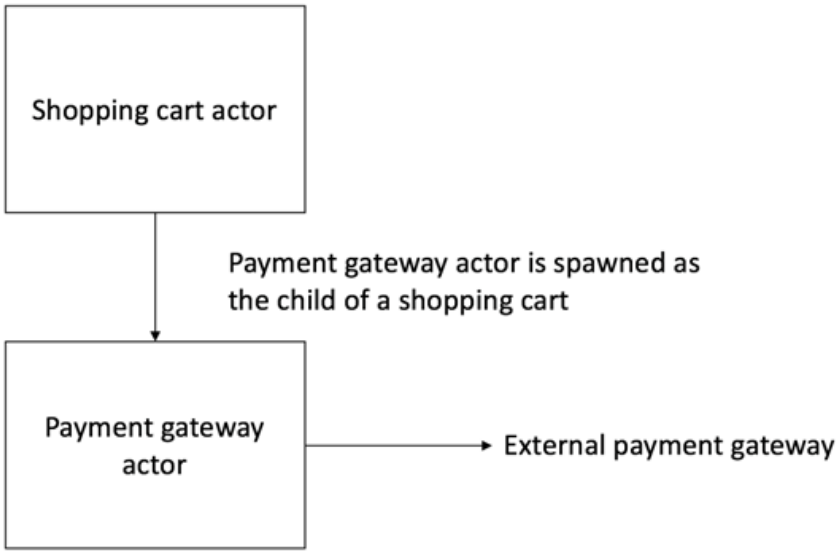


Figure 2.3 Creating child actors allows you to perform dangerous work by isolating it from the rest of the application

This is another common design pattern within Akka.NET. Because actors are completely isolated from other actors, this ensures that should one actor encounter difficulties, it won't directly endanger the other actors executing as part of an application. We'll see the concept of passing work down to children in chapter 3, when we look at how you're able to spawn new actors to perform work. We'll then see how you're able to use actor-based supervision as a means of creating more fault tolerant applications later on in the book, in chapter 7.

2.2.4 Data transfer between services

Underpinning the methodology behind designing applications with Akka.NET is the actor model, a concurrency model that solves the problems of coordination and synchronisation of operations across multiple threads by preferring isolation instead. As part of the actor model, every actor is independent of every other actor and they must communicate by sending messages to each other, in much the same way as humans communicate with each other. This means that when you start to design applications using Akka.NET, you need to think about how actors are able to talk to each other. Let's consider the example of the payment actor you saw in the previous section. The payment actor is solely responsible for interacting with the external payment service, but there are many components that make up the payment flow of an e-commerce website. For example, after a customer has made the purchase, what should the next step be? The logical next step is to attempt to fulfill the order as soon as possible. If your e-commerce application is selling digital media, then you may have another actor responsible

for assigning the privileges to the user's account, which allows them to view the content they have just purchased. Alternatively, if you're selling physical products, then you need to retrieve the products from the warehouse and prepare printed invoices and shipping labels, which will be sent with the purchase.

Because you want to prioritize fine-grained operations within actors, primarily for reasons of scalability and fault tolerance which we'll address in depth later in the book, you'll have multiple actors which are responsible for selected operations within the checkout flow. For these independent components, you need to be able to share results between them. Each actor has an address associated with it, through which other actors are able to communicate with the target actor; this allows you to simply pass messages between fixed addresses for actors, rather than needing to have direct references to actors themselves. For example, after the customer has completed their payment and you've verified its authenticity, you need to pass the message on to the component responsible for fulfilling the order. In the following figure, you can see how the payment actor has automatically sent a message to the addresses for multiple subsequent actors in response to payment completion.

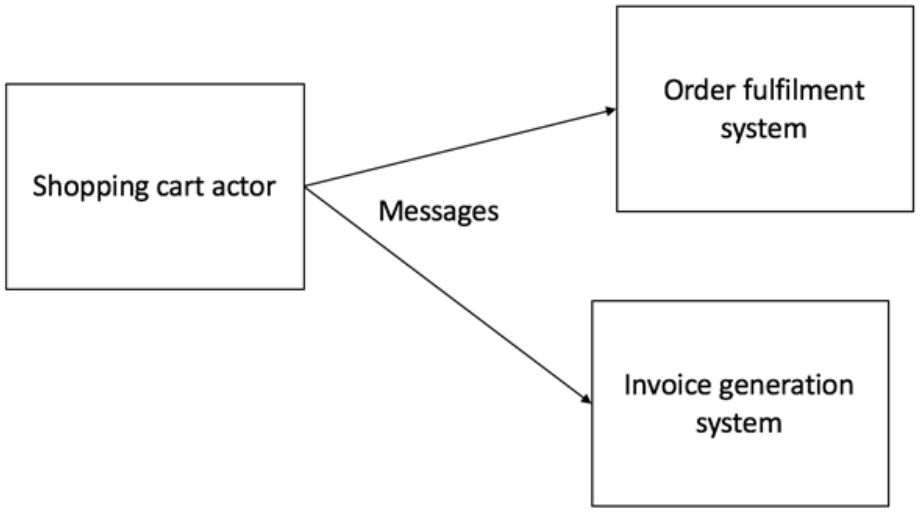


Figure 2.4 Sending messages between actors is a fundamental component of reactive application design

Messaging between actors is a fundamental aspect of reactive applications designed with Akka.NET, as it plays such a crucial role in the Reactive Manifesto by providing the foundation upon which you are able to provide fault tolerance and elastic scalability. As such, you'll see messaging appear in every chapter throughout the rest of the book, but we'll introduce messages in more detail in chapter 3 when you create your first actors.

2.2.5 Scaling work with routers

In your e-commerce application, there are many situations where you have a single actor that is responsible for a certain operation. One such example is when dealing with searching for products. You would have a single actor responsible for a single operation, but due to the concurrency guarantee offered by Akka.NET, which specifies that only one message should be processed at once, there may be a large queue of messages waiting to be processed by an actor. When applied to searching for products, you maintain a common index of the words used across all products so that you can quickly look up products that contain the search term used. A search actor stores this as its internal state and then receives a message to look through its word index to find which products contain the search term supplied as part of the message. But if you have several hundred users all searching for products simultaneously, you may encounter queues as the actor performs each search individually. Although this actor is stateful, in that it stores the product index internally, it is stateless between requests, which ensures that you can easily parallelize the search operation. You can see in the following figure how you no longer have a single actor processing each message, but instead have multiple independent actors, which are treated as a single target.

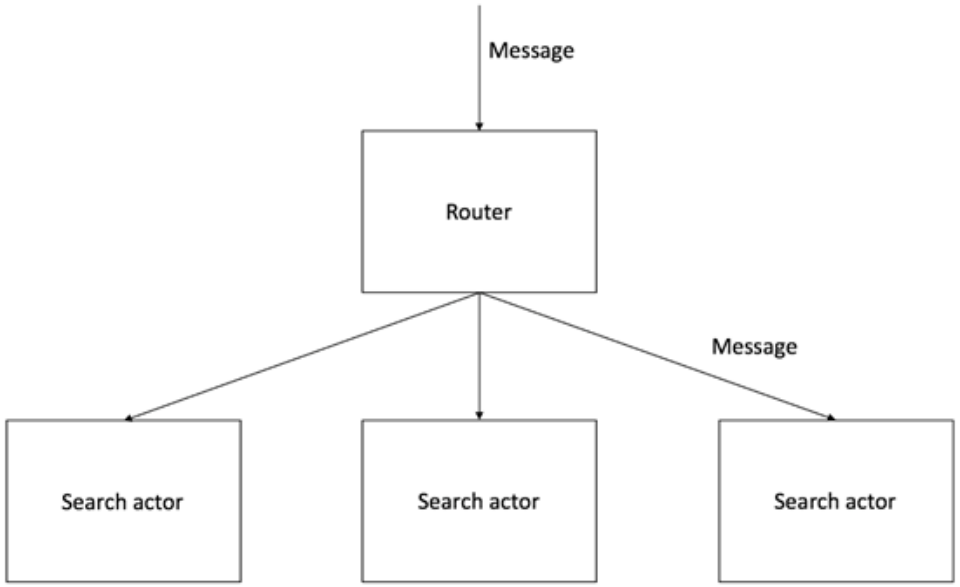


Figure 2.5 You can parallelize stateless operations by using multiple actors behind a routing proxy

The introduction of a router allows you to abstract away multiple individual actor instances and instead treat them all as a single actor, given that you simply direct messages to the router rather than to each actor sitting behind the router. The use of routers as a means of

parallelizing trivial workloads is a common pattern, and we'll see how you're able to use it in chapter 6, when we look at how the introduction of routers allows you to build more scalable applications that are able to respond to an increased number of messages by forwarding messages on to other targets.

2.2.6 Reactive application design patterns summary

These are just some of the most basic design patterns you're likely to encounter when designing applications using the fundamentals of the Reactive Manifesto. In figure 2.6, you can see a larger picture of the checkout system as a whole, which features the individual components and how they communicate with each other. You can see the flow of the customer to their shopping cart, where they add more items as they browse through the store. After a while, the user decides that they have finished shopping and want to complete their purchase. The shopping cart then transfers into the completing purchase state, which passes their payment details through to an external payment gateway. Once you get a response from the payment gateway, you're then able to complete the purchase internally within your application.

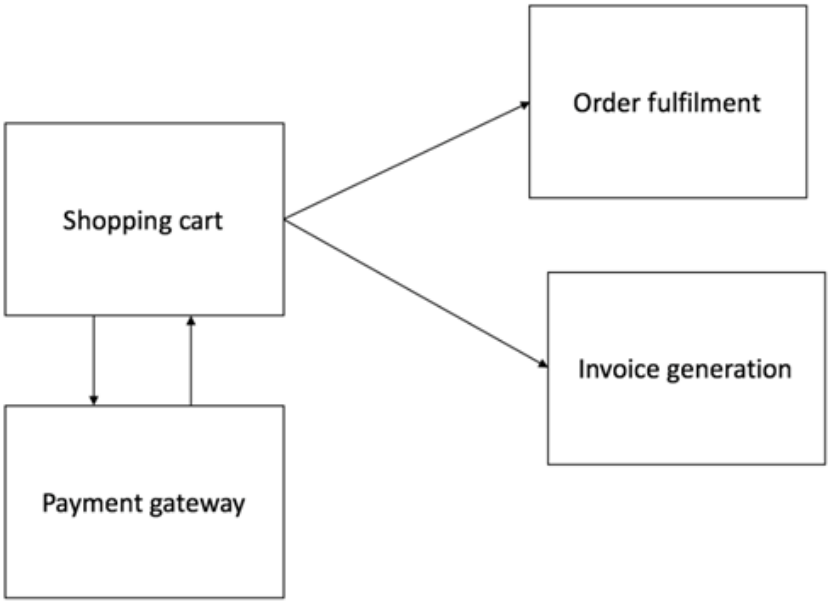


Figure 2.6 A simple e-commerce checkout flow has a number of interconnected components which can be built up from the design patterns you've seen so far

Here, we have considered only some of the more basic design patterns, however, you can see how even the simpler Akka.NET functionality forces you to consider many of the traits which make up a reactive application. The use of message passing pushes you down a path

which allows for systems which simplify the process of scalability and fault tolerance, actors force you to think about concurrency boundaries and how you're able to think about which tasks are able to be performed simultaneously, and supervision makes you think about what will happen in the event of the failure of other systems, whether they are internal or external. We'll progressively consider deeper design patterns which help simplify the process of development of larger and more complex systems throughout the rest of the book.

2.3 Building on top of the reactive foundations

Although these few components allow you to build an e-commerce system with a wide array of features, there are still many potential enhancements available for you to take it further. On top of the basic components provided by Akka.NET, there are a number of additional features which allow you to build in more advanced functionality to your e-commerce application. In this section, we'll consider how you're able to extend your application using these features, and how these enable you to more easily build larger and more complex systems which continue to follow the traits of the Reactive Manifesto.

2.3.1 Publishing the e-commerce application to the world

As it currently stands, the e-commerce application you've designed exists solely as an Akka.NET application. A typical system is consumed by a number of different clients including web browsers and mobile apps, this means that you'll also need to add some degree of integration with existing systems. Akka.NET allows you to expose an actor system onto a network using the Akka.Remote functionality. This then allows you to consume your e-commerce application from a number of other clients. Let's consider one of the most typical ways for how you might consume an e-commerce application, through the use of a website within a web browser. In this case, you need to be able to serve the contents of the application through a secure HTTP interface. The following figure shows the overall scenario for this, in which you have a web API that then communicates with the Akka.NET application. By using an HTTP-based web API, you're able to consume the application from a number of alternative clients and web browsers.

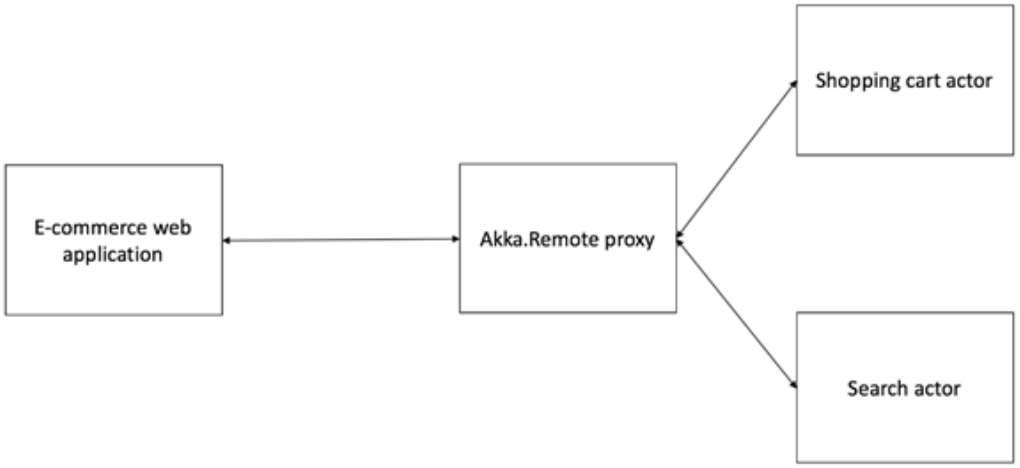


Figure 2.7 Akka.NET applications will typically need to be accessed from a number of alternative clients. This can be achieved through the use of a web proxy in front of the application.

In order to achieve this level of integration, Akka.NET provides the Akka.Remote functionality, which allows an Akka.NET application to be exposed over a network connection to a number of other clients. This allows a web proxy to communicate with the actors you have defined as part of your web application, such as the shopping cart or the actors responsible for integrating with external payment gateways. We'll see in chapter 8 how you're able to use Akka.Remote to work with existing web API-based projects.

2.3.2 Storing state within actors

The application you have created so far has no persistence of state and instead simply stores everything in memory. You'll usually want to persist changing data, so that users are able to modify it. We considered your shopping cart, where users are able to add more items as they browse through the store. In an e-commerce store, you want to ensure that there is as little friction in the overall buying process as possible. If you were to lose the user's shopping cart, then it's a potential loss of revenue for the business. In order to ensure that there's no data loss, it's important to persist data to a persistent data store such as a database or a filesystem. In chapter 11, we'll see how you're able to use a database to back up the data stored in your actors so that you're able to create even more resilient applications through the use of Akka.Persistence.

2.3.3 Scaling out across a cluster of machines

As e-commerce sites continue to grow in popularity, it's likely that you'll start to see increased volumes of traffic visiting your website and using the e-commerce application. This may even become too much for a single server to handle, so you'll need to use multiple servers

instead. In order to make this easier, Akka.NET provides the Akka.Cluster extension, which allows you to treat a number of machines as a cluster and run your e-commerce application across all of the machines in the cluster, offering the ability to scale the application up above the limits you might encounter in an individual application. In chapter 12, we'll see how the use of Akka.Cluster allows you to build elastically scalable services and systems that are able to scale on demand across multiple machines, as dictated by the overall load.

2.3.4 Continuing to react to environmental changes

The Reactive Manifesto states that for an application to be considered reactive, it should react to changes within its environment. In many cases within an Akka.NET application, this means responding to a message from another actor. We've seen this with the way multiple actors communicate within the checkout flow, but sometimes you are likely to want to automatically respond to changes as soon as they happen. For example, in your e-commerce application, you may have a number of peripheral components that rely on knowing when a customer completes a purchase. Such components might include systems that autosuggest new products based on historical purchases, or internal systems that adjust pricing automatically based on the number of purchases within a given time period. In these cases, responding to events as soon as they occur allows you to build more reactive applications that automatically respond to changes. Akka.NET provides publish and subscribe functionality, allowing decoupled components to register to receive any messages that are published onto an event bus. In chapter 12, we'll see how the distributed publish and subscribe functionality can be used to respond to changes in the cluster as soon as they happen.

2.3.5 Building on top of the reactive foundations summary

Akka.NET provides a rich ecosystem of additional functionality, much of which is beyond the scope of a simple checkout in an e-commerce application. But, given increasingly demanding customers and a more competitive marketplace, it's important to consider how some of this additional functionality can be applied to a simple shopping cart to create a richer experience for users, which hopefully leads to increased spending within your e-commerce website. These more advanced features will be addressed later in the book, where you look at the likes of clustering across machines, persisting actor state to external datastores and integrating an Akka.NET based application within other applications, whether these are new applications or existing legacy applications.

2.4 Summary

In this chapter, you learned:

- How you're able to design applications which have the traits of the Reactive Manifesto
- The core design patterns which can be used to simplify reactive application development
- The considerations you need to make when designing a system to effectively utilize the benefits presented by Akka.NET

3

Your first Akka.NET application

This chapter covers

- **Setting up an actor system**
- **How to define an actor**
- **How to send a message to that actor**
- **Alternative actor implementations available to use**

The first few chapters have so far covered the key reasons why you'll likely want to use a reactive architecture, as well as what a reactive architecture means. We've seen how the overall aim of a reactive system is to create applications that are responsive to the end user and how this requires applications to work, even when struggling with the demands of scale or malfunctioning components. We've also covered the key things you need to consider when you start to design a reactive application, so that you're able to truly ensure your application includes the traits of a reactive application.

From here on, we'll be considering how you can start to write reactive systems that follow the traits laid out by the Reactive Manifesto. As we saw, the Reactive Manifesto is a series of guidelines designed to suggest solutions that many organizations have found effective at solving their problems. There are many means of developing reactive systems, but you'll be focusing on one in particular. You'll be using the actor model as the underlying basis for your reactive systems, and the implementation you'll be using is in the form of Akka.NET, a framework designed for writing concurrent applications using the actor model in .NET.

In order to build these reactive systems, you'll be starting to write code. Akka.NET runs on the .NET framework, and although any language that runs on the .NET framework can use Akka.NET, the main content of this book will use C# to write applications. But, because Akka.NET provides a pragmatic API for F#, which features some key differences to the C# API, these will be covered in the appendix at the end of the book. All of the concepts you'll pick up

throughout the book will be the same, regardless of whether you use C# or F#, but the implementation of these concepts will depend upon the language used.

By the end of this chapter, you'll have a really basic actor created that is able to receive messages, and you'll send this actor some messages. You'll then be able to adapt this actor and start to build your own, capable of performing more complex functions.

3.1 Setting up an application

Akka.NET feels very much like a framework, but it markets itself as a toolkit and runtime that forms the basis of concurrent applications. Akka.NET requires no special application configuration to run and can be hosted in any of the normal .NET runtime environments, whether console applications, Windows services, IIS, WPF, or Windows Forms applications. Throughout this book, examples will be given in the form of console applications unless specified otherwise.

All of the components required to run Akka.NET are distributed through the NuGet package management system. As Akka.NET relies on many modern features of the .NET runtime, it requires a minimum of .NET v4.5 to run. Akka.NET also has full Mono support, allowing it to run in Linux and Mac OSX environments.

To install the libraries, a NuGet client is required. There are several options available for dependency management with a NuGet client:

- *Visual Studio package management GUI*—If you're developing applications using Visual Studio, then dependencies can be managed directly through the references node of a project in the Solution Explorer.
- *Command-line tooling*—In environments where you don't have access to Visual Studio, a number of command-line tooling options are available, including the official NuGet client or third-party alternatives such as Paket.

To develop applications in a single-machine scenario, the only NuGet package required is the Akka package. This provides all of the core functionality required to create and host actors, and then send messages to these actors.

3.2 Actors

When considering Akka.NET, it's important to realize that the underlying ideas surrounding the framework are those relating to concurrency. Ultimately, the actor model is designed to allow multiple tasks to operate independently of each other. The actor model is designed such that it abstracts away many of the underlying multithreading constructs that are required to ensure that concurrency is possible. At the heart of this is the concept of an actor.

3.2.1 What does an actor embody?

The concept of an actor is something that has been discussed several times so far, but now we can consider what an actor is in the context of Akka.NET. The actor model is a model of computation designed to make concurrency as easy as possible by abstracting away the

difficulties associated with threading, including the likes of mutexes, semaphores, and other multithreading concepts.

We can think of actors much like how we think of people. Every day, we communicate with hundreds or thousands of people using a variety of methods. People send messages to those surrounding them and then react to messages they've received. This communication is all in the form of message passing, where a message can be any of a number of types, such as body language or verbal cues. When a person receives a message, they can process the information and make decisions as a result of something. The decisions a person makes might include sending a message to the person who originally communicated with them, such as saying "hello" in response to a greeting, or it may be to interact with other parts of the world, such as tasting or feeling something in order to get more information. Finally, a person is able to save memories or information in their mind. For example, they're able to recognize faces and names, or store facts for later recollection.

When we talk about actors, this is the simplest idea we can use. The overall idea is that people can be broken down into three key concepts, which form the basis of the actor model. These three concepts are communication - how they send messages between each other; processing - how the actor responds whenever it receives a new message; and finally, state - the information that an actor is able to store when processing.

COMMUNICATION

When considering the principles of reactive applications, we saw the advantages of using a message-passing architecture in order to help build systems that are scalable and fault tolerant. By default, all actors in the actor model communicate asynchronously through the use of message passing.

Each actor within an application has a unique identifier with which it can be contacted. You can think of the actor's address in exactly the same way you think of an email address: it provides you with a known endpoint you can send messages to. The end user can receive their email at any address, and the same exists with an actor's address. You can simply send a message to an address and it automatically gets routed through to the intended processing for that actor. This address is then connected to a mailbox, which is simply a queue of the messages an actor has received at its address. This mailbox is where every message is added as it's received, until the actor is able to process them sequentially.

When you think of an email, it can be one of several different types. It can contain the likes of text, media, or even contact information. Akka.NET has a similar concept, but it relies on using data types as the basis of messages. You can choose to use any type you like as the basis of your messages, but there is one requirement: messages must be immutable. If a message isn't immutable, then you could potentially modify it either while it is in the processing stage, or while it is in the queue. In either scenario, this breaks the concurrency safety guarantees provided by Akka.NET.

PROCESSING

Once a message has been received, an actor needs to be able to do something with that data. This is the job of the processing component of an actor in Akka.NET. As a message is received, the processing stage is started by the framework, which then invokes the appropriate method on the object to handle it. Akka.NET provides guarantees that at most one message will be processed at any one time and, due to the queue provided by the framework, the processing stage receives the messages in the exact order they were sent to the actor.

Due to the different programming methodologies supported by Akka.NET, there are a number of different techniques for using the APIs to best fit the paradigm being used. For example, the C# APIs revolve around the use of inheritance.

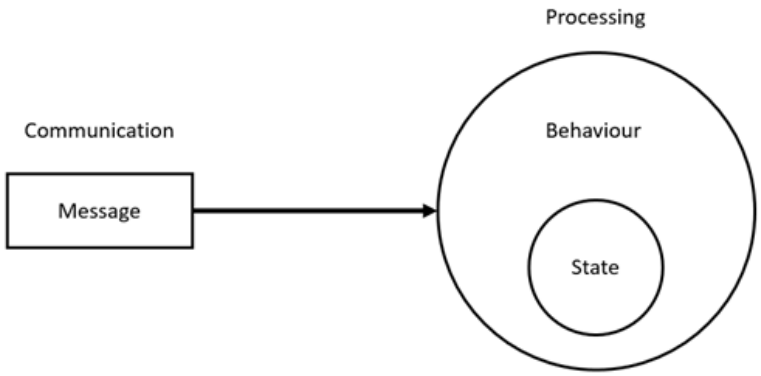
STATE

When we think back to the analogy of actors as people, we touched on the notion of memories and information saved in the brain. If you want to access this data, you can't directly query it from somebody else; instead, you need to ask them about the data they know about. The same concepts apply with actors. An actor is free to store whatever state is appropriate, and it then forms a sealed boundary around it. The only thing in the application that has access to the data stored within the actor is the processing element associated with that actor.

The primary reason for this is due to the ultimate aims of using actors. Actors are a construct designed to reduce complexity when writing multithreaded applications. Removing shared access to data reduces vast numbers of potential concurrency bugs, such as deadlocks or race conditions. It also means you can quickly scale an application built on actors, because you can simply deploy actors into entirely new locations when required.

COMBINED RESULT

When these three constructs are combined, you are left with the concept of an actor: a high-level approach to dealing with concurrency, whether the tasks running concurrently are on separate threads or in separate datacenters. Figure 3.1 shows the interaction between the three key concepts and how they relate to each other. As you can see, the state is entirely enclosed within the bounds of the actor and is not accessible from outside that actor instance. The only means you have of manipulating or retrieving the data is through the use of behavior, which you define to run within the bounds of the actor. This behavior is then only invoked as required once a new message is received by the actor's inbox.



3.2.2 What can an actor do?

You've seen that actors are very small isolated entities that share nothing with the world outside of them, and each of them is then scheduled to process the messages in its mailbox. You can then think of actors as really small applications that have a built-in communication channel. Because of this, actors are able to perform any operation an application may normally perform. But you can generalize the actions that an actor is likely to perform into one of three categories:

- *Sending a message*—When you designed a reactive system, you saw that applications are typically built as a dataflow, whereby applications propagate events that they've received and responded to. In order to manage this, actors need to be able to send messages to other addresses within the actor system. This task is not necessarily related solely to sending messages to actors within the actor system; it could also include communication through external services with other transport protocols such as HTTP.
- *Spawning other actors*—In the case of actors that perform long-running computations while also needing to process large numbers of messages, it's common to spawn a new actor that is responsible for handling all of the significant processing. For this to happen, actors need to be able to spawn new actors. This also serves uses in other areas, such as having a supervisory actor spawn new children to perform dangerous work that may lead to errors.
- *Setting behavior for the next message*—A key role of an actor is to be able to respond to any messages it receives, because reactive applications strive to react to changes in their environment. Changes in an environment are likely to lead to changes in the way messages need to be processed, so actors should be able to set how they should process new messages.

These are just some of the most common tasks that actors can typically perform, but it's likely that actors will be performing other tasks as well. This might include jobs such as connecting with external web services, interacting with devices such as the graphics on the

host machine, or potentially interacting with external input and output on the machine it's running on.

There is, however, a restriction on the type of work that an actor is capable of performing. Actors should try and avoid performing long-running blocking operations, particularly in cases where a number of actors may all be performing blocking operations. This prevents the Akka.NET scheduler from running any of the processing for other actors. The work that actors do should aim to be asynchronous and operate primarily through message passing. An example of a blocking operation is waiting for a user to enter some text into a console window through the use of `Console.ReadLine`.

3.2.3 Defining an actor

Having now seen the underlying principles behind actors, you're able to see how the core components fit together. We can now look at how you're able to define your actor. Let's think back to the original actor analogy, which looked at its similarity to how we, as people, communicate. Let's build up an example of how we can model this interaction through the use of actors. You'll create an actor that represents the sort of actions a person might take upon receiving a greeting.

When writing an actor in C#, you rely upon the inheritance of certain actor classes and override certain methods that get called whenever a new message arrives. The simplest possible means of implementing an actor is through the use of the `UntypedActor` class. Using this approach, it's possible to simply execute a single method any time a new message arrives, similar to the following:

```
class PersonActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
   Console.WriteLine("Received a message: {0}", message);
    }
}
```

Although this example is the basics of how you can write an actor using Akka.NET, it's likely that you'll want to do something with the actor whenever it receives a message. You can use any type within the CLR as a message, with the only requirement being that the class must be immutable. You'll create two potential messages that a person can receive: either a `Wave` message or a `VocalGreeting`. In the following message classes, I have omitted the likes of constructor guards, which should be used to verify that you're not, for example, passing a null or empty value where you should be passing an actual value. In a production-quality application, as opposed to demo code, additional checking should be performed to ensure the application consistently stays in a valid state:

```
class Wave {}

class VocalGreeting
{
    private readonly string _greeting;
```

```
        public string Greeting { get { return _greeting; }}

        public VocalGreeting(string greeting)
        {
            _greeting = greeting;
        }
    }
```

These are the two message types your actor is now capable of receiving. Your original actor can now be changed to perform different actions when it receives a message of a given type. For example, when you receive a `VocalGreeting` message, you can print a message to the console:

```
    class PersonActor : UntypedActor
    {
        protected override void OnReceive(object message)
        {
       if(message is VocalGreeting)
            {
                very msg = (VocalGreeting)message;
                Console.WriteLine("Hello there!");
            }
        }
 }
```

When you create a message for each type, you end up with a lot of duplication in the handling of the message. For example, in the example, you've got two types of message, and in each instance, you need to check whether the message is of a certain type and then cast it to that type. You can also end up with a lot of code duplication when you want to check a condition within the message itself. In order to prevent this, Akka.NET provides an API that allows you to pattern match on the message type. The following example shows how, using the Akka.NET pattern matching API, you can invoke a handler depending upon the message received:

```
    class PersonActor : UntypedActor
   {
        protected override void OnReceive(object message)
        {
          message.Match()
                   .With<VocalGreeting>
                       (x => Console.WriteLine("Hello there"));
        }
}
```

Akka.NET also provides a further abstraction on top of the basic actor, which you can use to declaratively handle messages. The `ReceiveActor` combines many of the aspects of pattern matching while continuing to abstract away as much of the logic surrounding message type handling as possible. Whereas, with the simple `UntypedActor` you had to override a certain method that would then be executed upon receipt of a message, the `ReceiveActor` requires you to register a message handler for each of the message types you want to support. The

following example shows how the previous example using an `UntypedActor` can be converted to the `ReceiveActor` implementation:

```
    class PersonActor : ReceiveActor
    {
        public PersonActor()
        {
            Receive<VocalGreeting>
                (x => Console.WriteLine("Hello there"));
        }
 }
```

Akka.NET is essentially a model for concurrently performing asynchronous operations, so is an alternative to the .NET Task Parallel Library (TPL). When dealing with asynchronous operations, you'll typically pipe the results back to the actor's mailbox as a message, but the `ReceiveActor` provides the ability to interoperate with the TPL through the use of asynchronous message handlers. An asynchronous message handler works exactly the same as a regular message handler, except it returns a `Task` instead of void:

```
    class PersonActor : ReceiveActor
    {
        public PersonActor()
        {
            Receive<VocalGreeting>(async x =>
                {
                    await Task.Delay(50);
                    Console.WriteLine("Hello there");
                });
        }
}
```

The approaches shown so far for creating actors have relied upon the use of delegates as a means of handling messages, but Akka.NET provides an additional means of creating actors in the form of the `TypedActor`. The `TypedActor` allows for stricter contracts to be built up for the types of messages an actor should be able to receive, by implementing an interface for each of them. Upon receiving a message of a given type, the method implementing the interface for that message type is executed with an instance of the received message:

```
    class PersonActor : TypedActor,
                        IHandle<VocalGreeting>
    {
        void Handle(VocalGreeting greeting)
        {
            Console.WriteLine("Hello there");
        }
    }
```

All of the actor definitions here allow you to build up bigger and more advanced actors, capable of performing more complex operations. You saw in the definition of an actor that you're able to store state within an actor. As you've seen, the actor definitions are simply classes in C# that override specific methods. You're able to store state within an actor using either properties or fields on the class.

When you store any state within an actor, it's only accessible from within that actor. It's impossible to access any properties or fields from outside the actor's boundaries. This means that, regardless of where an actor exists, there's no need to worry about synchronizing access to the state, because messages are only processed one at a time.

```
    class PersonActor : ReceiveActor
    {
        private int _peopleMet = 0;

        public PersonActor()
        {
            Receive<VocalGreeting>(x =>
                {
                    _peopleMet++;
                    Console.WriteLine("I've met {0} people today",
                                      _peopleMet);
                });
        }
}
```

Upon receiving a message, it's common to require some metadata about either the message received, such as the original address of the sender, or about the actor processing the message itself, such as the address behavior stored within the actor. In any of the actor types, you can access this through the `Context` property within the actor. For example, if you wanted to retrieve the original sender of the message, you can access this through the `Sender` property on the context. Given the sender, you can send messages in response to a message you received. For example, if somebody waves at you, then you'll say hello to that person in response by sending them a `VocalGreeting` and also waving at them by sending a `WaveGreeting`:

```
class PersonActor : ReceiveActor
    {
        public PersonActor()
        {
            Receive<Wave>(x =>
                {
                    Context.Sender.Tell(
                        new VocalGreeting("Hello!"));
                    Context.Sender.Tell(
                        new WaveGreeting ();
                });
        }
     }
```

There are many more ways of defining actors that are specific to certain aspects of Akka.NET, but we'll cover those in later chapters when you need to.

3.2.4 Summary

When we discussed design considerations within a reactive system, one of the key considerations was that operations should be done on the level of the smallest unit of work. In the context of Akka.NET, the actor is the encapsulation of that smallest unit of work. One of the

key takeaways when dealing with actors is that, due to its original design intentions as a concurrency model, any operations within the confines of an actor are thread-safe. This ensures that you're able to automatically scale out your application across as many threads, machines, or datacenters as you like, and the framework will be able to handle any and all scaling issues. These are handled by messages being processed one at a time in a queue, ensuring that messages are processed in the same order in which they're received.

3.3 Spawning an actor

Having defined an actor, you need to be able to start it running within your application. In order to do this, we'll start to dig into the underlying framework and look at how you can use Akka.NET to start instances of actors that can react to messages you send to them. In order to do this, we'll look at the concept of an actor system and what needs to be done to deploy an actor into this actor system.

3.3.1 The actor system

If actors are people, then actor systems are the countries in which they live. An actor system is essentially the host within which all of your actors will be deployed. Once actors are deployed, they're able to perform any tasks that have been assigned to them. Like people and governments, actors need some form of management and restrictions in place to ensure that they are good and valuable citizens in society. These tasks fall within the realm of the actor system, which is not only your actor host, but also the scheduler and routing system. You don't need to know about the internals of the actor system to be able to develop applications with Akka.NET, as it abstracts all of that away from the user. There's much more to it than just those few elements, but some of the key roles the actor system is in charge of include:

- *Scheduling*—Actors as a multithreading construct run at a higher level than a regular thread, and as such, there needs to be some means of coordinating these actors. The actor system ensures that all actors have a fair chance of processing their messages within a reasonable amount of time. It also ensures that heavily-used actors aren't able to starve the system of resources, which prevents less frequently used actors from being in a situation where they're not processing data.
- *Message routing*—All of the messaging through Akka.NET is location transparent, meaning the caller doesn't need to have any knowledge of the location of the recipient. But there must be some part of the system that does have knowledge of message locations, and this is the actor system. The actor system is capable of routing messages to a large number of different locations, whether they're on a separate thread, running on a remote system, or running on a machine in a cluster.
- *Supervision*—The actor system also acts as the top-level supervisor of your application, so that, should any component crash, then it's able to recover it. We'll look into this in a later chapter, as you look to incorporate the notion of fault tolerance into your application.
- Extensions—Akka.NET supports a vast range of extensibility points at all stages in the

processing pipeline. The actor system is responsible for managing all of these extensibility points and ensuring that any extensions are correctly incorporated into the application.

This is only a small subset of the large number of tasks the actor system is responsible for. It's common to have only one running per application, but actor systems are identified on a machine through the use of a unique name, meaning that there is the possibility for more than one actor system to exist on each machine.

Actors in Akka.NET operate under the concept of a hierarchy, whereby all actors are the children of another actor in the hierarchy. The reason for this is to provide better fault tolerance when developing applications; the intricacies of this will be covered in a later chapter on fault tolerance. When instantiating an actor system, Akka.NET initially creates a number of actors used by the system. These top-level actors are:

- `/user`—This actor holds all of the actors you spawn into your actor system. Even if you spawn your actor without a parent, it does in fact have a parent in the form of the user actor, which supervises all of your top-level actors.
- `/system`—This actor is the top-level actor under which all of the system-level actors are stored. This is typically those actors that are used for tasks such as logging, or those deployed as part of some configuration.
- `/deadletters`—As actors are free to send messages to any address at any stage of the application, there is always the possibility that there is no actor instance available at the path specified. In this case, the messages will be directed to the `deadletters` actor.
- `/temp`—At times, Akka.NET spawns short-lived actors. This is typically for scenarios such as retrieving data, which will be covered later in this chapter.
- `/remote`—When joining multiple actor systems using Akka.NET remoting, there are some scenarios in which Akka.NET needs to create actors to perform the task of supervisors when a supervisor exists on a separate machine. In these cases, the `remote` top-level actor is used. This will be covered in a later chapter.

These actors all form part of the hierarchy. in figure 3.2, you can see their deployment in the hierarchy. The actors themselves form a tree structure, similar to a filesystem, with files and folders. The following figure shows an example deployment of a relatively simple actor system. In this case, the user has deployed three actors into the actor system, `actorA`, `actorB`, and `childA`, which is a child actor spawned beneath `actorA`:

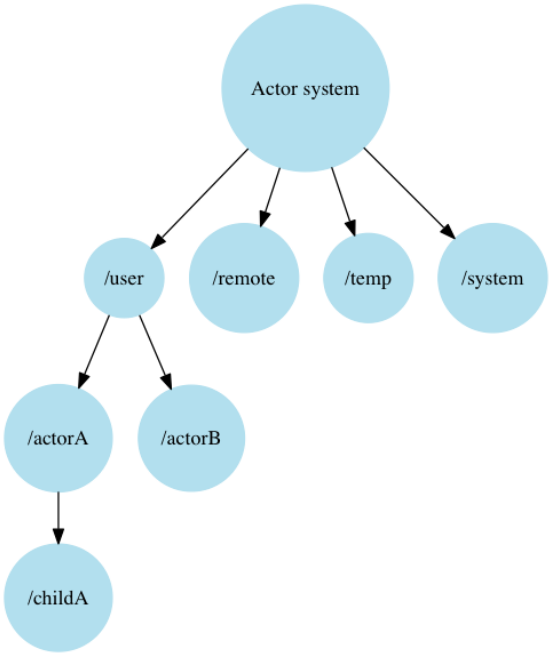


Figure 3.2 An example actor hierarchy within an Akka.NET application

The decision to use actors for all top-level work in Akka.NET ensures that a uniform interface exists throughout the application. For any of these system-created actors, users are free to send a message to them in the same way that a user might expect to send a message to an actor that they have instantiated.

3.3.2 Spawning an actor

Now that you have defined an actor that is able to do some work, you need to deploy it into an application so that it can be used. Before you're able to deploy your actor, you need something that is capable of hosting an actor. In order to get this, you need to initialize an actor system. As you've seen, the actor system is the component of Akka.NET responsible for many of the tasks related to how actors are run within the framework.

Instantiating an actor system in which you can host actors is a simple task and simply requires calling the `Create` static method on the actor system. The only requirement when creating an actor system is to name it so that actors can be identified based on which actor system they live in:

```
var actorSystem = ActorSystem.Create("MyActorSystem");
```

An actor system can also be created with a configuration section, in order to customize certain parts of the framework. This will be covered in a later chapter. For now, you create an

actor system without a configuration file in C#, which causes a fallback onto the default configuration.

As we've seen, the actor system is responsible for many of the internal tasks and scheduling associated with the Akka.NET framework. As a result of this, the actor system ends up becoming a heavyweight object, and so, you typically only spawn one actor system per application. The actor system is also the main means you have of interacting with the actors that are operating within the framework. In the majority of scenarios, it's typical for the actor system to either reside in a static class or singleton object, or be injected as a dependency into those methods that require it.

Once an actor system has been created, you are then free to use it and deploy new actors into it. In order to deploy an actor into the actor system, you use the `ActorOf` method, which simply requires the actor type to instantiate as a generic type argument. The following example shows how you can now deploy your actor from earlier into the actor system, so that you can start to interact with it:

```
var actorRef = actorSystem.ActorOf<GreeterActor("actorA");
```

Once this method has been called, Akka.NET will create and initialize this new actor in the actor system. You also pass it a string which you can use to uniquely identify a given actor instance within the actor system. In this case, you have chosen to refer to the actor as `actorA`. With this name, you're able to retrieve references to it directly from the actor system.

3.3.3 Summary

The actor system essentially forms the basis of your host, within which all of your actors will live. Although you don't need to understand all of the intricacies of what happens deep within the framework, it's beneficial to have an understanding of some of the features provided by the actor system. The actor system is also the key extensibility point of an Akka.NET application and allows more advanced features to be implemented, many of which you'll look at in later chapters.

3.4 Communicating with actors

Once you've spawned an actor into your actor system, you'll want to be able to communicate with it. Although you have an actor deployed into your actor system, it's currently doing no work and simply sits in the memory doing nothing. By communicating with it, the framework will invoke the message processing on that actor. You saw that the actor model relies upon message passing as a means of communication between actors. A message is essentially a generic term for a collection of data that is then packaged and sent to an actor instance, represented by its address, somewhere in the actor system. You saw in the example earlier that your messages will be a data type you've created.

3.4.1 Actor addresses and references

Upon spawning your actor, the actor system returned a direct reference to the actor through an `IActorRef`. This actor reference isn't a direct reference to the actor's location in memory, but it is a reference to the actor as used by Akka.NET. Its ultimate use is to facilitate sending messages to the inbox of the referenced actor. The Akka.NET framework provides a number of built-in means of referencing actors out of the box. These include the likes of actor references for clusters and remote actor systems, but we won't be seeing these until later chapters.

The most commonly used actor reference is the simple `LocalActorRef`, whose job is to operate on actor systems that only operate on a single machine. The key component of the actor reference is the storage of the address of the actor itself. Upon deployment, every actor is given a unique address, through which the actor is reachable. The address is reminiscent of a simple URI that might be used to identify files in a filesystem or web pages on a website. But, in this case, it represents the address of an actor in an actor system. Figure 3.3 shows the components of an address. An actor address is made up of four key components:

- *Protocol identifier*—The protocol identifier is used to reference how a connection is made to that actor system. This is similar to how http and https are used in web addresses to identify which system should be used. For a single machine, this is typically through an identifier similar to `akka://`, but in order to handle concepts such as remoting, there are other commonly used examples, such as `akka.tcp://`.
- *Actor system name*—When you created an actor system, you gave it a unique name to refer to that actor system instance. This part of the address relates to that name.
- *Address*—This is only used when dealing with the concept of remoting, but it still forms a key part of the actor path and is used to identify the machine upon which an actor system resides.
- *Path*—The final part of the address is the path, which is used to identify an actor. All user-defined actors will start with `/user/` for this part of the path, but there are other system-defined actors that inhabit other root addresses.

The concept of an actor reference starts to ensure that your application is more loosely coupled, but it still causes problems. In order to send a message to a given actor reference, you'll need to pass the actor reference around the application. When you considered the benefits of a message-driven architecture, one of the more important benefits was the ability to

have loosely-coupled systems that didn't rely upon intimate knowledge of other actors. In order to solve this problem with Akka.NET, you're able to send messages to an address rather than to an actor reference directly. Given an address, you're able to then send a message to that address. For example, in order to send a message to an actor known as `ActorA` in your actor system, you are able to retrieve a reference to its address:

```
var address = system.ActorSelection("/user/ActorA");
```

When you deployed your actor, you saw that it gets deployed into a hierarchy. If you deployed your actor as the child of another actor, then you can continue to address it, similar to how you find files that are in a folder in a filesystem. If `ActorA` has a child actor called `Child`, then you can send messages to it as follows:

```
var childAddress = system.ActorSelection("/user/ActorA/Child");
```

The addressing system in Akka.NET also respects the usage of certain path traversal elements that are typically associated with URIs. For example, a common case is to retrieve the parent of the current actor, so that messages can then be sent to a sibling of the current actor. This can be achieved by using the `..` syntax to retrieve the parent within an actor:

```
var address = Context.ActorSelection("../ActorB");
```

Although it might seem that the concept of an actor selection and an actor reference are the same, there is a significant difference in that an actor reference points to a specific incarnation of an actor, whereas an actor selection simply points to an address. This address may then be shared with multiple instantiations of an actor. For example, given a reference to a specific actor, if that actor is destroyed and recreated, then any messages sent to that actor reference won't be delivered to the target, even if they both share the exact same path across instantiations. But, given an actor selection, messages can be sent to it; even if an actor is destroyed and recreated, all messages will be delivered.

This distinction allows for more complex paths to be used in the context of an actor address. An example of this is the use of wildcards in the path to a given actor in order to select a large number of actors at once. Once actors have been selected, it's possible to send the same message to all in the wildcard with a single method call. Paths in Akka.NET support two kinds of wildcard in an actor address, based on a standard wildcard syntax common to other languages and tools:

- `?`—The question mark replaces a single instance of any given character in a path. For example, the path `c?t` would match paths such as `cat`, but not `coat` or `cost`.
- `*`—The asterisk matches any string of characters usable as a path. For example, the path `/parent/*/` would send a message to all children of the actor called `parent`.

On occasion, it is beneficial to have a direct reference to an actor instance rather than a generic address. In order to cater for these situations, Akka.NET provides a number of different means of retrieving a reference from an address:

- *Calling* `ActorOf` *to spawn a new actor*—Upon spawning a new actor, a direct reference

to that actor is returned, which represents the incarnation that has been spawned.

- *Sending a message to an actor*—By sending a message to an actor, it's possible to use the sender property of a received message to identify which actor replied to the request for information. Akka.NET provides built-in support for this through the built-in `Identify` message, and through an abstraction over the top of this on the `ActorSelection`, which can be used to resolve an instance.

Although there are many cases in which it's appropriate to send messages to an address, it can frequently be beneficial to pass around a reference to a specific actor. For example, for a long-running actor that is valid throughout the lifecycle of the application and performs a specific purpose, it is typical to pass an actor reference in the constructor of those actors that depend on it.

It's important to understand the difference between an actor reference and a simple actor address due in part to the actor lifecycle, which is something covered in a later chapter. But for your purposes, either is an appropriate means of messaging a specific actor.

3.4.2 Sending a message

Upon spawning your actor into the system, you're able to start to communicate with it by sending messages to its mailbox. In order to send a message to it, you need something capable of receiving a message. As you saw when discussing the differences between an address and a reference, you're able to send a message to either. Once an actor is spawned, the actor system returns a reference to that actor instance, which you can then send a message to. The actor reference defines a method called `Tell`, which takes an instance of any type and passes it through the Akka.NET framework. If, however, you're using F#, there is a custom operator defined for sending a message. For example, if you wanted to send a vocal greeting message to the actor you defined earlier, then you can do this as follows:

```
actorRef.Tell(new VocalGreeting("Hello"));
```

There may be times when you don't have an actor reference; on those occasions, you'll look up an actor by its address. In order to look up an actor, you need something capable of providing references to other actors. This may be the actor system that is hosting the actor, or it may be the Context associated with a specific actor. To select the actor you deployed earlier, you can use the actor system and select the actor by its address:

```
var selection = actorSystem.ActorSelection("actorA");
```

In each of these cases, the actor system provides the root location from which actors will be retrieved, which for the actor system is directly beneath the user actor. But if you had a second actor deployed alongside your first, you could use your first actor reference as your anchor to other actor locations:

```
var selection = actorRef.ActorSelection("../actorB");
```

Once you've got an address, you can then pass messages to it in the exact same way as if it was an actor reference:

```
selection.Tell(new VocalGreeting("Hello"));
```

Actors are designed to completely encapsulate any state, to ensure that nothing outside of the system is capable of mutating it. This ensures that Akka.NET retains full control over the processing stage, in order to only allow one message to be processed at a time. This leaves all code thread-safe, but it makes it more difficult to access the data. In order to access data from outside the system, you need to send a message specifically requesting the data be sent back. Akka.NET provides another method that allows for request-reply scenarios to be used: Ask. Ask is an asynchronous method designed to form a layer of abstraction over the top of the messaging that is required:

```
var response = await selection.Ask(new Wave());
```

As Ask is an asynchronous construct, you'll need to factor this into your code so that you await the response to your Ask. By default, Ask has a timeout of 10 seconds, within which the actor needs to respond to your initial request message; otherwise, the request will simply time out with an exception. It's important to realize that your actor which is being asked has no way of knowing that the sender is expecting a reply and it's down to how you, the developer, handle this scenario.

3.4.3 Summary

Messages form an integral part of the design of a system using Akka.NET, and are the key to communication between multiple actors, or even other entities outside of the actor system. As such, it's important to model your domain effectively, through the commands that actors will need to be able to respond to. In later chapters, we'll look at techniques such as event sourcing and domain-driven design as a means of modeling certain interactions between actors. At this stage, however, it's likely that most actors will either be reacting to events or responding to commands.

Although the name "message" is used, Akka.NET doesn't require anything special with regards to the design of a message, and they can be simple .NET classes or structs. The only requirement when designing these messages is that they should be immutable, in order to ensure that the thread safety guarantees specified by Akka.NET can't be broken anywhere within the application.

3.5 Case Study: Actors – Concurrency – Phone billing

Many modern mobile games operate on a freemium model, whereby users are able to play the game for free, but they require in-game credits to perform certain tasks. These in-game credits can either be purchased using real money, or earned by performing certain operations within the game, and can then be used to purchase upgrades within the game in the form of visual changes or temporary performance boosts to get through challenging parts of the game.

In this case, you have multiple external sources attempting to credit the user's account; you also have the user, who will be trying to debit their account. The overall financial success of the company is dependent upon the ability to sell in-game credits to players, so it's important that users get the credits they're both entitled to through playing the game, and that they've purchased. You also need to ensure that you don't allow users to overspend their credits, and limit their spending potential to the number of credits they have in their account.

In these situations, dependent upon the user, you may have to deal with a large number of operations in which you try and either debit or credit the user's account. It's likely that many of these changes will be happening concurrently, with multiple components trying to access the user's credit balance. Because actors are primarily designed as a tool to eliminate many of the difficulties you face when developing concurrent, multi-threaded code, you're able to safely operate on the user's credit balance, without worrying about whether other components are also modifying the user's balance.

If the billing system is flawed, then it's possible that the business will suffer from lower revenue than expected. Actors operate on a serial stream of messages, and this guarantees that an actor can't modify the same state from two concurrent operations. In figure 3.4, you can see how you're able to model that in a game back end server. Here, you have multiple actors, where each actor represents a single user's account within the game. As multiple actors are able to process work concurrently, this ensures that every user is able to modify their balance with minimal waiting. Other components within the game can then send messages to modify the user's account credit balance by requesting that the amount is reduced when the user spends their earned credits, or by increasing the balance if the user purchases more credits.

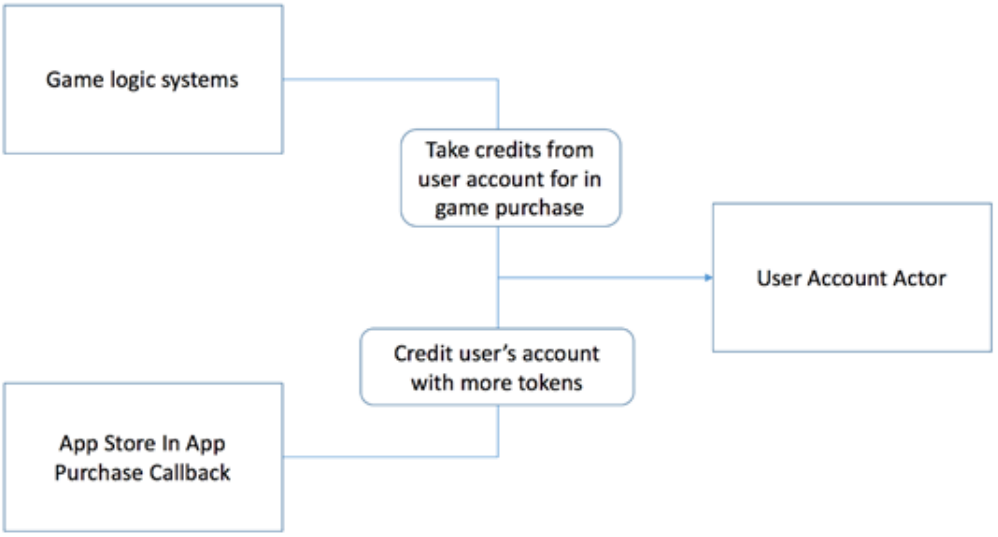


Figure 3.4 The user's in-app purchasing or financial situation can be updated from multiple different sources

By using actors, you've managed to greatly simplify the overall complexity surrounding concurrent operations on shared data, all thanks to the underlying principles of the actor model, which sits at the core of Akka.NET-based applications.

3.6 Summary

In this chapter, you learned:

- How to define an actor and how each part relates to the actor model
- How to deploy that actor within your application
- How to communicate with that actor by passing messages

4

State, behavior, and actors

This chapter covers:

- How an actor can selectively receive messages with switchable behaviors
- What a finite state machine is
- How you can model finite state machines using Akka.NET's switchable behaviors
- How you can build more complex finite state machines with Akka.NET's FSMActor

In chapter 3, we looked at exactly how you can create actors, as well as how you can send messages to these actors. We saw how, using the likes of the `ReceiveActor`, you can register methods that execute once a message of a certain type has been received. In this chapter, you'll be extending this further to allow you to change the methods that are executed upon receipt of a message at runtime. This allows you to build more complex actors that are able to truly react to their environment. This is what we were aiming for when we saw the Reactive Manifesto in chapter 1, and it allows you to use specific behaviors whether an actor is in a failing state or operating normally, and ensure that it remains responsive regardless of the current situation.

Furthermore, this chapter covers how you can generalize actors with lots of states into what's known as a finite state machine. You'll not only look at how you can diagrammatically create these, but also how you can then convert these diagrams into one of several different actor types, including introducing a new actor type available to you in the form of the `FSMActor`.

By the end of this chapter, you'll have the knowledge required to create actors that are able to truly react to changes in their environment, allowing you to build actors that continue to follow the traits laid out in the Reactive Manifesto.

4.1 Preparing for the next message

When we looked at the principles of reactive programming in chapter 1, we saw that the ultimate aim of a reactive application is to react to changes in its environment as quickly as possible. These changes in the environment may also mean the actor needs to either respond to completely different events, or respond to the same events but using different behavior.

As an example, you can consider an actor to represent a water depth sensor. For many people, living next to a stream or river can prove to be a concern in the event of heavy rain. In order to monitor the depth of the water in this river, you can install a sensor that measures the current depth of the water. In normal daily use, it's likely that you'll only want to log these values in a database somewhere for historical data storage. However, if the sensor begins to receive values that lie outside the expected range, for example if the river starts to fill due to heavy rain, then you'll instead want to alert the relevant people that a sensor has started to receive values that could indicate there's likely to be flooding should the water level continue to rise.

The simplest solution to this problem would be to maintain some variables within the actor to indicate the previous history of messages that have been received, as well as the current state. In the case of your water depth sensor, in order to monitor whether you're currently in a scenario where the water is at a higher level than expected, you'd need to store details such as how many messages you've received that contained readings over the alert level. Although this is a potentially viable option, it does cause a number of problems relating to how maintainable the overall codebase is. As developers, you aim to reduce the overall complexity of potential solutions, rather than increase it. When you have to analyze all of the variables stored within your actor to determine your current state, it requires more thought to understand the simplest parts of the application.

In chapter 2, we saw the pipeline of steps contained within an actor. In order for an actor to have some understanding of the process it should execute upon receiving a message, an actor has a behavior associated with it. Whenever the framework receives a message, this behavior is then invoked, providing the actor with a means of responding to the message. However, in order to simplify the process of behavioral changes within Akka.NET, the framework allows for the behavior to be hot-swapped at runtime.

This allows the actor to store an entirely separate handler, which is then invoked whenever a new message is received. When you then consider your depth sensor again, you're able to create multiple separate handlers for each of the possible states it can exist in. In this example, the two states might be a normal operating state and then an alerted state. When the actor is using the normal state behavior, it will process messages by simply appending the result to a database table, but if it receives a value higher than some specific amount, then it will need to switch into an alerted state. Whenever it's in an alerted state, you'll want to send an urgent notification to a user that there is possible flooding. At that stage, you'll need to process the messages, while also adding extra actions that will need to be taken.

4.2 Setting appropriate receive behavior in Akka.NET

As you've seen, the concept of behavioral changes forms a key part of building easy to maintain applications with Akka.NET. When you consider the key parts of an actor's usage lifecycle, you'll typically go through four stages. These are define, deploy, message, and become. You've seen how you can define, deploy and message actors in chapter 3, but you've not yet seen how you can perform the become operation. This section will focus on how the actors you define are able to change their behavior dynamically at runtime.

4.2.1 Switchable behaviors

In the previous chapter, we considered how an actor operates in a similar manner to how people operate, in that they both communicate asynchronously through message passing. We looked at how people respond upon receiving a message from somebody trying to communicate with them. Sometimes you aren't able to respond to a message in the same way that you would normally, due to you being in a different state. For example, if somebody waves at you while you're asleep, you won't know that it happened because you were asleep, so you end up ignoring the message.

The same is true of actors in Akka.NET. There may be times when an actor isn't in a situation where it makes sense to process a given message. This is the purpose of switchable behaviors in Akka.NET: to allow an actor to only process messages when it is in the appropriate state. Upon switching into this new state, you can continue to process the same messages as you would have received in the previous state, or you can receive an entirely different set of messages.

4.2.2 Become and Unbecome

When we saw the original actor definition in chapter 3, we saw the `UntypedActor`, which invokes a method upon receipt of a message. With Akka.NET, you can choose to change that message handler dynamically at runtime through the use of switchable behaviors. The only requirement for any new message handlers is that they must have the same method signature. Therefore, a new message handler must take an Object as a parameter and return `void`. In order to switch to a new message handler, you use `Become` to set it.

Let's consider an example of how you can use this. You can use an actor to provide access to a database, although in certain circumstances the database might not be reachable by the application. In these cases, where you can't reach the database, then any connections will timeout, leaving you in a situation where the application needs to wait several seconds every time it tries to access data. This then breaks one of the aims of the Reactive Manifesto, to ensure that applications remain responsive even in the face of failure. You can create an actor that has two possible states: operating normally or failing. If the database is unreachable, then you can either return cached data or a message informing the actor requesting the data that the database is unreachable. You'll create an application that responds to a `GetData` message. If the database is unreachable, then you return a `DatabaseNotAvailableMessage`, otherwise you return a `GetDataSuccess` message.

```
class DatabaseActor : UntypedActor
{
    protected override void OnReceive(object message)
    {

    }

    public void Reachable(object message)
    {
        message.Match()
               .With<GetData>(x =>
                   {
                       var data = Database.Get(x.Key);
                       Sender.Tell(new GetDataSuccess
                           {Key = x.Key, Data = data});
                   })
               .With<DatabaseUnavailable>(x => Become(Unreachable));
    }

    public void Unreachable(object message)
    {
        message.Match()
               .With<GetData>(x => Sender.Tell(
                                    new DatabaseUnreachable()))
               .With<DatabaseAvailable>(x => Become(Reachable));

    }
}
```

You can also use `Become` and `Unbecome` within the `ReceiveActor`. We saw in the previous chapter how the `ReceiveActor` registers message handlers that operate whenever it receives a message of that type. You can, however, choose to add each of the handlers within a method. As long as you then call this method from the constructor, it will still work as expected. Moving the handlers into a new method allows you to change the currently applied message handlers at runtime. When you call `Become` within a `ReceiveActor`, you need to supply a method that takes no parameters and returns nothing. When you then call `Become` with a new method, it calls that method to register all of the message handlers. You can also still use the same techniques as in the `UntypedActor` to add the message handler to a stack.

```
class DatabaseActor : ReceiveActor
{
    public void Reachable()
    {
        this.Receive<GetData>(x =>
            {
                var data = Database.Get(x.Key);
                Sender.Tell(new GetDataSuccess
                        { Key = x.Key, Data = data});
            });
        this.Receive<DatabaseUnavailable>(x => Become(Unreachable));
    }

    public void Unreachable()
    {
        this.Receive<GetData>(x => Sender.Tell(
```

```
                            new DatabaseUnreachable()));
            this.Receive<DatabaseAvailable>(x => Become(Reachable));
        }
    }
```

The circuit breaker pattern

Although this example of switchable behaviors might appear to be simple, it forms the basis for a frequently used pattern in the world of Akka.NET. We saw in chapter 1 how the overall aim of the Reactive Manifesto is to create applications that are responsive regardless of the circumstances, for example if the application is currently in a failing state. Taking your example of a database, you can only know if it is unreachable by trying to contact it. If it fails to respond within a given time limit, then you can say that the connection has failed and you want to avoid retrying every request.

The circuit breaker pattern is inspired by the principles of the circuit breaker, which is normally found in electrical wiring installations. In these systems, at the first sign of a fault being detected, circuit breakers automatically cut off the power supply, acting as a failsafe and potentially saving lives. The circuit breaker pattern follows the same principle: in the event of a failure, the circuit breaker automatically switches off. This ensures that timeouts do not begin to affect other actors that requested data from the database.

Although you won't be covering circuit breakers in depth here,[1] it's important to note that even more complex constructs can be built quickly and easily using some of the most basic elements of the Akka.NET framework.

The concept of switchable behaviors in Akka.NET forms an important part of building systems that follow the traits of the Reactive Manifesto. By changing the behavior entirely, it allows you to explicitly specify the messages you can receive in a given behavior, without lots of internal state which can hide the overall intent.

4.2.3 Switchable behavior summary

As you've seen in this section, switchable behaviors are a great feature of Akka.NET and allow you to perform a key operation. They allow you to write actors that not only react to the environment in which they're running, as per the aims of the Reactive Manifesto, but also allow you to write cleaner code than you might otherwise be able to. You've seen how you're able to write actors that are able to switch their message receiving behavior at runtime, to allow a completely different set of message types to be received. Switchable behaviors, although appearing to be a relatively small topic, allow you to do bigger things with cleaner code than you might previously have thought possible. In the next section, you'll look at a wider generalization of switchable behaviors in the form of finite state machines, and how these prove to be useful in a concurrency model like that of Akka.NET.

[1] A thorough guide to the circuit breaker pattern can be found on the Microsoft documentation website, available at https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker.

4.3 Finite state machines

When you're developing applications, you'll commonly end up in a situation where you need to identify as being in one of a number of key states. For example, if you're designing an application that directs calls to call center workers, you'll need to be able to identify what state any given agent is in. They might be in a call, on hold, finishing a call, on a break, offline completely, or many more scenarios. Although there are many scenarios possible, there are a finite number of states that they can be in and it's possible to model the states, as well as the events that cause transitions between states. For example, if you have a call center agent who's currently on a call, then they won't be able to answer the phone to a new incoming call. Finite state machines allow you to model this behavior effectively, using a common diagrammatic format, which makes it easy for people to see the events that cause transitions within a system.

4.3.1 Understanding finite state machines

You've seen how the example of a call center operator can be thought of as a finite state machine, but it can prove to be a relatively complex example to get started with. Instead, let's start with a relatively simple finite state machine. A finite state machine is composed of events and states. An event can be thought of as a signal that the current state uses to transition to a new state. A state is a group of actions that are then executed, depending on the event received. Once a state receives an event, it can perform one of a number of options: it can either ignore it, remain in the same state, or move to a new state.

Let's consider the example of a turnstile that you might find at an entrance to a theme park or sports stadium. The aim of a turnstile is to keep people out until they've provided either money or a ticket to cause it to unlock. When you approach a turnstile, you'll have a ticket, which grants you entry into the stadium. Once you scan your ticket at the turnstile, assuming your ticket's valid, then the gate will transition into the unlocked state. Once it's in the unlocked state, then you're free to pass through and get into the stadium. As soon as one person has passed through the turnstile, it automatically locks to prevent another person entering. Sometimes when someone walks up to the turnstile, they might not have a valid ticket; in that situation, you want to ensure they aren't able to enter the stadium with an invalid ticket. To do this, you stop the gate from unlocking, ensuring it remains in the locked state. On the other hand, sometimes you might accidently scan a ticket more than once; in that situation, you don't want to lock the turnstile before the guest has had the chance to enter.

We can represent these states in a state transition diagram. A state transition diagram shows all of the states that your system can exist in, as well as the events that cause the changes between states. When we consider the turnstile example, you've only got two possible states your system can be in: either locked or unlocked. When we consider how you can tranisition between those two states, you've only got two events within the system: either the user rotates the turnstile, or the user scans their ticket. When the user scans a valid ticket, you'll transfer the turnstile into the unlocked state. When the user then pushes the turnstile to

rotate it, it should let them through as well as also transitioning back to the locked state. If the user pushes on the gate when it is in the locked state, then it should stay in the locked state. Similarly, if the user scans their ticket multiple times, then it should remain in the unlocked state as it waits for the user to proceed through the gate.

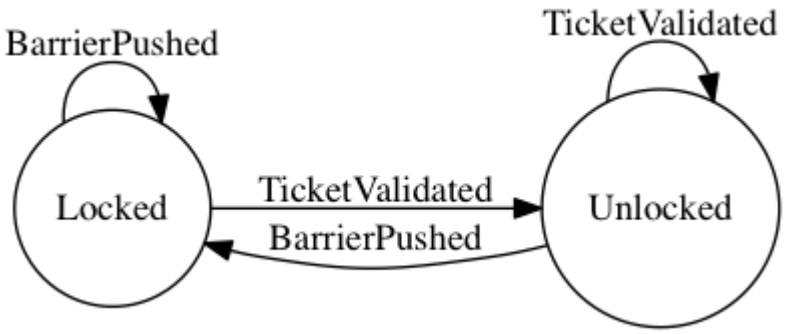


Figure 4.1 A simple finite state machine for a ticket barrier, showing the two possible states and the transitions between them

This text-based description of a finite state machine can prove to be difficult to fully comprehend, but it's possible to represent the state machine diagrammatically through the use of a state transition diagram. In a state transition diagram, you can draw the state machine as a directed graph, where each vertex is a state and each edge is an event. This provides a visualization of how each of the events causes transitions between states. Each of the edges is marked with a label that signifies the type of event received. When you look at the state diagram, you can see the transition by finding the current state vertex and then tracing the edge that has a label matching the event name. An event edge can transform the state machine either into a different state or into the same state.

Typically, when you're building a finite state machine, your system will automatically need to enter a certain state as soon as it starts up. We refer to this as the default state, and it can be represented on the state transition diagram as a small black spot with an arrow pointing to a state. This small spot symbolizes the initial entry location for the state machine and signifies that it should automatically move into the new space. Figure 4.2 shows your turnstile state machine with a default state marker to ensure it automatically enters the locked state when the finite state machine is started.

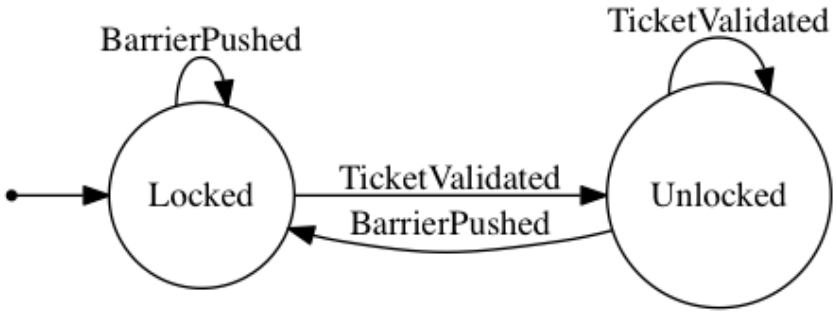


Figure 4.2 An enhancement of the previous finite state machine showing the initial state the system should exist in

Although state transition diagrams are a valid means of representing finite state machines, there are many other ways of showing them. The other most common means of representing them is through the use of a state transition table. This is a textual representation of the behavior changes enclosed within a state machine. It's also easier to include details of the effects that could occur between transitions. This might include, for example, some communication with an external information source, or in the case of the turnstile, telling a locking mechanism to unlock when the unlocked state is entered. You can represent your turnstile state machine as a state transition table, as shown in table 4.1. The information encoded in the diagram and the table is identical; they simply provide two different views of the available information.

Table 4.1 State transition table for a ticket barrier showing the current state and the next state it transitions to upon receiving a message

Current state	Input	Next state	Output
Locked	Ticket	Unlocked	Unlock lock
	Barrier pushed	Locked	Nothing
Unlocked	Ticket	Unlocked	Nothing
	Barrier pushed	Locked	Lock barrier

4.3.2 The use in a concurrency model

Finite state machines form one of the key building blocks, upon which you can build complex asynchronous systems. Let's take an example of an actor that is responsible for loading some state from a database. You've seen how important it is to ensure that the main

behavior of actors is asynchronous, in order to ensure that all actors are able to make progress. Given this requirement for asynchronous behavior, you can start to consider how you can handle the messages you receive.

You'll work with your persistent actor ideas a little more. As soon as your actor is initialized, it will immediately retrieve its internal state from a database. Once this asynchronous operation completes, it will pass the result back to the actor as a message. Unfortunately, this leads to a potential problem. What do you do in the event that another actor has sent your original actor a message before its received a result? In this scenario, you've received a message that you're not yet able to process because the actor is currently performing all of its initialization.

You can effectively split your actor into having two key states: initializing and initialized. You can then see the state transition diagram in figure 4.3. The actor is immediately set to be in the initializing state. When it's in this state, the only message that you care about is a response from the database containing your initial state. In this example, the input that causes a transition is a message containing a response from the database.

Once your actor has received the response from the database, it can parse the data and set its initial state. This actor is then in its initialized state, which ensures that it's able to respond to messages that it would normally experience during its lifecycle.

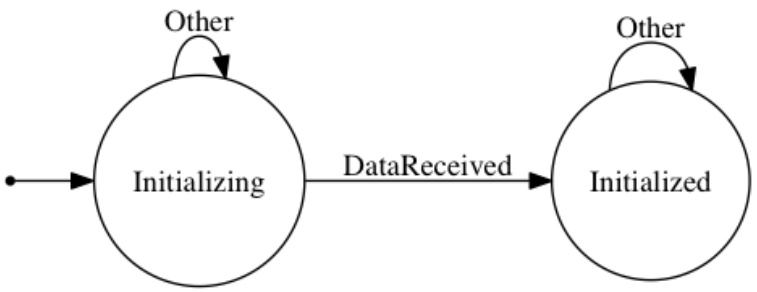


Figure 4.3 A finite state machine for an actor loading data into memory

Although this has shown how you can design an actor that can switch between states while it's performing an asynchronous operation, there is still a problem. The actor currently ignores any messages that were received while the actor was in the initializing state. Figure 4.3 shows a timeline of messages received during the different states that the actor was in. Currently, your initialization state will simply dismiss all of the other messages that are not the response of the database operation. In many cases, you typically don't want to ignore messages, as your application will assume that the actor is able to process any new messages upon creation.

STASHING

In order to counter this problem, you need to store any messages you have received but don't yet want to process. You could store these in a queue on the actor, as state internal to the actor. However, you then have to manually process them when you receive a response

message. Akka.NET provides a solution in the form of a message stash. A stash can be thought of as a temporary message store, where messages can be stored while also maintaining the order in which they were delivered. A stash works with the mailbox of the actor to ensure that you're able to temporarily stash away messages until a more appropriate time, in much the same way that some animals will stash away food until the winter when they're unable to get easy access to new food.

In order to create a stash on an actor, you simply need to ensure that your actor implements an interface. Currently, Akka.NET provides an unbounded stash, which doesn't set any limits on the maximum number of messages that can be stashed. In order to implement the interface, the only requirement is to add a single property to the actor. The simplest actor definition that also has a stash attached to it is shown in the following example. In this scenario, you've created an actor in exactly the same way as in chapter 3, but it's had one minor modification in the form of the addition of an interface implementation:

```
public class StashingActor : UntypedActor,
                             IWithUnboundedStash
{
    public IStash Stash { get; set; }
}
```

Now that you've got an actor with a stash attached to it, you can delay the processing of certain messages. Expanding upon your simple actor definition, you can now add two message handlers. Now, whenever the actor receives a message that isn't a database payload, it'll stash it for later. When it does receive the payload, it performs two operations. It changes its behavior to be a different message handler, in this case the `Initialized` message handler. It also unstashes all of the messages that had previously been stashed. When these messages are unstashed, they'll be prepended to the mailbox associated with that actor. Prepending these messages ensures that they'll be processed in the same order as that in which they were delivered:

```
    private void Initializing(object message)
    {
        if (message is ReadComplete)
            Become(Initialized);
            Stash.UnstashAll();
        else
            Stash.Stash();
    }
    private void Initialized(object message)
    {
        //Process messages normally here
    }
```

When you look to retrieve messages from the stash, you're able to retrieve them in one of three ways:

- Unstash—Prepends only the oldest message from the stash
- UnstashAll—Prepends all messages from the stash while also retaining order
- UnstashAll with predicate—Prepends all messages from the stash for which the predicate

returns true, while also retaining order

The use of stashing is not something that is limited to changing state. The stash is accessible within the actor at any time, and allows you to retrieve messages from the stash whenever you want to add your stashed messages back to the queue.

Mailboxes and stashing

Later in the book, you'll see how mailboxes can be customized to allow for different receive semantics. This is not something you need to worry about now, but stashing imposes additional constraints upon the mailbox used by an actor. In order to ensure that messages are delivered in the correct order, the stash needs to prepend messages into the mailbox. In order to manage this, the mailbox needs to be suppported by a double-ended queue, alternatively known as a deque. The deque allows messages to be added to the queue in the traditional way one might expect, but it also supports high priority messages by allowing them to be inserted at the front of the queue, ahead of all other previously enqueued messages.

Stashing forms one of the key parts of actor implementation when working with finite state machines, as it allows you to potentially delay the processing of some messages until it is in a fit state to do so. This allows you to build a more complex state machine, capable of dealing with significant amounts of asynchronous code, while also ensuring that it stays manageable and easy to understand.

4.3.3 Converting a finite state machine into an actor

As you've seen, finite state machines are an essential component when developing concurrent applications with Akka.NET. Although you could develop a finite state machine within the actor using state stored within it, you can use the behavior switching capabilities of an actor within Akka.NET. Given either a state transition diagram or a state transition table, you can port them over to be an actor.

Let's work with the turnstile example once again. In that, you had two states: locked and unlocked. You also had two events: push and ticket scanned. You can represent each of your states as your message received behavior when using Akka.NET. This means that if you were to use a `ReceiveActor`, for example, you can create two methods in your actor definition that represent your actor states. In this case, they'll be called `Locked` and `Unlocked` as per the states in your state transition diagram. This leads to an implementation similar to the following code example. The example shows the two states that your actor can exist in:

```
class TurnstileActor : ReceiveActor
{
    void Locked()
    {
    }

    void Unlocked()
    {
    }
```

```
}
```

Given your actor with its possible states, you need to now look at the events your actor receives and how they should affect the current state. An event in a finite state machine can be thought of as an external influence, designed to show a change in the world in which your system is running. This definition of an event in a finite state machine matches the definition of an actor when used in the context of the Reactive Manifesto. As such, you can model your events through the use of messages. Examining the state transition diagram, you can see you only have two events that cause transitions in your system: a guest has pushed against the turnstile or a guest has scanned a valid entry ticket. You'll create classes to represent the possible events; in both cases, they'll be class definitions with no data associated with them. You can call them `TicketValidated` and `BarrierPushed`. The following code example shows the events and how simple their definitions are. When naming events, you may have noticed a common trait. Events are always historical facts; they are things that have happened within your system, and as a result tend to be named as such:

```
class TicketValidated { }
class BarrierPushed { }
```

Now that you've got your states and events defined, you can look at the state transition diagram to see how your actor should react to the messages received. As we saw in the previous section, an arrow represents a state transition. An arrow from a vertex to itself can be thought of as a null operation; it won't have any effect on the current state, and therefore you can typically ignore it. Therefore, having looked at your state transition diagram, you can see that the `TicketValidated` event in the `Locked` state will cause a transition to `Unlocked`, and the `BarrierPushed` message will lead to the `Locked` state. In each of the states, you only have one message that you need to react to, so you can create them as a single message handler delegate. The following code example shows how you can set up your handler delegates to ensure that they only react to their specific message:

```
void Locked()
{
    Receive<TicketValidated>(msg => Become(Unlocked));
    Receive<BarrierPushed>(msg => { });
}

void Unlocked()
{
    Receive<TicketValidated>(msg => { });
    Receive<BarrierPushed>(msg => Become(Locked));
}
```

Continuing through the state transition diagram, you can see the entry state. As we saw in the previous section, the entry state is represented by a black dot pointing to a given state. In this case, it's the `Locked` state. Therefore, your actor should immediately be placed in the `Locked` state to ensure nobody is allowed in without a valid ticket. You can ensure that this

happens by placing a call to Become directly in the constructor of the actor. This specifies that an actor should use this specific behavior for the very first message it receives:

```
public TurnstileActor()
{
    Become(Locked);
}
```

Although that is all of the information from your state transition diagram converted to being an actor, there is still one other feature that needs implementing. We saw that the state transition table can contain more data than the diagram. In this case, it's an external change that needs to occur whenever you enter the Unlocked state. When you enter the Unlocked state, you need to communicate with your turnstile locking mechanism to unlock it for one turn to allow the user to enter. You can do that by adding any communication into the body of the Unlocked state method. In this case, you'll call a method on an object, which will inform it of the unlocking action:

```
void Unlocked()
{
    Barrier.Unlock();
    Receive<BarrierPushed>(msg => Become(Locked));
    Receive<TicketValidated>(msg => { });
}
```

4.3.4 Using the finite state machine actor

Finite state machines are incredibly useful components when writing asynchronous applications, and although you're able to replicate simple state machines using simple switchable behaviors, there are limitations involved with them. For example, you're not able to perform operations as you transition out of a given state. You also have one other problem with your current implementation of state machines. In Akka.NET, you have the concept of supervision, which you'll see in a later chapter. For now, you'll only need to know that when an actor throws an error, it restarts in a new state. This means that you lose any data and associated state upon restart.

In order to address these issues, Akka.NET provides an actor specifically for the purpose of creating a finite state machine. By using the FSMActor, you can develop actors as a finite state machine. The FSMActor is itself a generic class requiring two type arguments; you need to supply the type of the actor state as well as the type of data the actor stores within its boundary. The following example shows how you can start to work on converting your turnstile actor to use the FSMActor. This simply defines your actor as being a finite state machine that has all states deriving from an ITurnstileState and the actor stores data of type ITurnstileData:

```
class TurnstileStateMachine : FSM<ITurnstileState, ITurnstileData>
{
}
```

You can now start to define a number of data structures that are used to represent the states and data stored within the actor. You'll start by defining a base interface that all of your possible states can inherit from. Following that, you'll create two classes that implement the interface and represent your states, namely `Locked` and `Unlocked`. You'll also create a class, within which you'll store any state. Although you don't need to store any data within your finite state machine, you'll create a class that you could use as a data storage location:

```
interface ITurnstileState { }
class Locked : ITurnstileState
{
    public static readonly Locked Instance = new Locked();
}
class Unlocked : ITurnstileState
{
    public static readonly Unlocked Instance = new Unlocked();
}

interface ITurnstileData { }
class TurnstileData : ITurnstileData { }
```

When you create your states, an important consideration is how Akka.NET compares the states within the application. By default, C# uses reference equality to compare the two states. In order to ensure the states are compared correctly, you should implement `Equals` on each of your states to ensure you choose the correct state.

Once you've implemented `Equality`, you can register each of the states and a handler in the constructor of the actor. Use the `When` method on the `FSMActor` to register a handler that executes whenever a message is received and the actor is in that state. The handler is invoked with two pieces of information: the current actor state and the received message. Once you receive a message, you can then pattern match on the possible messages received and handle them appropriately. Every message handler has to return the action that should be undertaken as a result of the message. This is commonly one of two methods: either `GoTo`, which transitions the actor into a new state, or `Stay`, which keeps the actor in its current state. You can continue to build up your state machine further by using these features to register the two handlers you'll need for the `Locked` and `Unlocked` states:

```
public TurnstileStateMachine()
{
    When(Locked.Instance, @event =>
    {
         if(@event.FsmEvent is TicketValidated)
         {
              Console.WriteLine("Ticket was validated");
              return GoTo(Unlocked.Instance);
         }
         return Stay();
    });

    When(Unlocked.Instance, @event =>
    {
        if(@event.FsmEvent is BarrierPushed)
        {
```

```
            Console.WriteLine("User pushed barrier");
            return GoTo(Locked.Instance);
        }
        return Stay();
 });
```

Once you've registered each of the handlers, you need to perform a number of other operations before the actor is usable. You first need to tell the actor what the initial state and internal data should be, which you manage with the `StartsWith` method. Once you've performed all of your configurations, you need to initialize the actor and ensure it's ready to receive any messages you send to it. This is managed through the use of the `Initialize` method:

```
StartWith(Locked.Instance, new TurnstileData());
Initialize();
```

When developing applications using finite state machines, you're also provided with the ability to have a timeout on an individual state. This timeout says that if a message has not been received within a fixed timespan, then the actor will send itself a `StateTimeout` message. This message can then be handled like any other, and be used in pattern matching. In order to use the `StateTimeout` functionality, you simply pass a `TimeSpan` to the `When` method, which specifies how long the actor should wait before sending a timeout message. For example, when developing your turnstile, you may want the turnstile to only stay unlocked for a limited period of time. In order to do this, you can create a timeout on the `Unlocked` state and handle the message within the state message handler:

```
When(Unlocked.Instance, @event =>
{
    if(@event.FsmEvent is BarrierPushed ||
       @event.FsmEvent is StateTimeout)
    {
        Console.WriteLine("Barrier will now lock");
        return GoTo(Locked.Instance);
    }
    return Stay();
}, TimeSpan.FromSeconds(10.0));
```

We saw that one of the problems when using switchable behaviors was the limited potential for performing operations on a state change within an actor. The `FSMActor` allows you to solve this problem by registering a function that executes every time a transition occurs. When this function is called, it receives the previous state as well as the next state. This then allows you to perform certain operations that happen between state transitions. For example, in the turnstile actor, you'll want to tell the locking mechanism that it needs to unlock upon transition from `Locked` to `Unlocked`, and lock on the transition between `Unlocked` and `Locked`.

```
         OnTransition(OnTransition);

 private void OnTransition(ITurnstileState prevState,
                           ITurnstileState newState)
        {
            if (prevState is Locked && newState is Unlocked)
```

```
            {
                BarrierLock.Unlock();
            }
            else if (prevState is Unlocked && newState is Locked)
            {
                BarrierLock.Lock();
            }
            else if (prevState is Locked && newState is Locked)
            {
                Console.WriteLine("The barrier gate is still locked");
            }
            else if (prevState is Unlocked && newState is Unlocked)
            {
                Console.WriteLine("The barrier gate is still unlocked");
            }
    }
```

The use of the `FSMActor` has allowed you to develop more complex finite state machines that are easier to scale when using more states with more complex transitions. The `FSMActor` tends to be less frequently used than the likes of the `ReceiveActor`, but it can form a solid foundation upon which you can build more complex state-based asynchronous systems.

4.3.5 Finite state machine summary

This section has covered a lot of content. You've looked at how you can generalize the ideas surrounding the states and behaviors that an actor can exist in at any given time through the use of finite state machines. You've seen how you can represent these states and transitions both textually and diagrammatically, through the use of state transition tables and state transition diagrams. Finally, we saw how these representations can be converted to Akka.NET using features built into actors to help manage complexity.

4.4 Case Study: State machines – States + events – Marketing analytics campaign

In this chapter, we considered how you're able to model some common day-to-day objects using state machines. Notably, we saw how you can represent a turnstile easily as a set of finite states, and events that cause transitions between these states. These principles can be extended to many larger complex applications that have a limited number of states. One such example is the management of complex marketing campaigns.

In order for businesses to thrive, they need to ensure that they're able to retain customers and potentially convert them from an occasional user to a recurring user who pays for the service each month. One way to achieve this is to incentivize the user by offering promotions to them, based on their historic usage of the application or service. For example, if a user signs up for a service but doesn't use it, then the business might want to target the user and provide them with a user guide to help them make the most of the service. Similarly, if the user signs up as a free user but doesn't convert to a paid user, then the user could be targeted with a promotional campaign offering them a discount code. Although these marketing campaigns may start small, they can quickly grow into more complex systems depending on a number of

different variables, ranging from the date through to how long it is since the user last used the service.

As these campaigns are typically made up of a number of states, along with events that cause transitions between the states, you can think of a marketing campaign as a state machine, which ultimately means you can encode the logic within an actor. In figure 4.4, you can see how you push events to the actor representing a marketing campaign. These events will include information relating to how the user uses the application, such as how long it is since they last used it, or whether they've chosen to pay for the service. The actor can then use its current state to work out how to process the incoming events, and whether the user should be notified of a discount available to them or sent more information on the service.

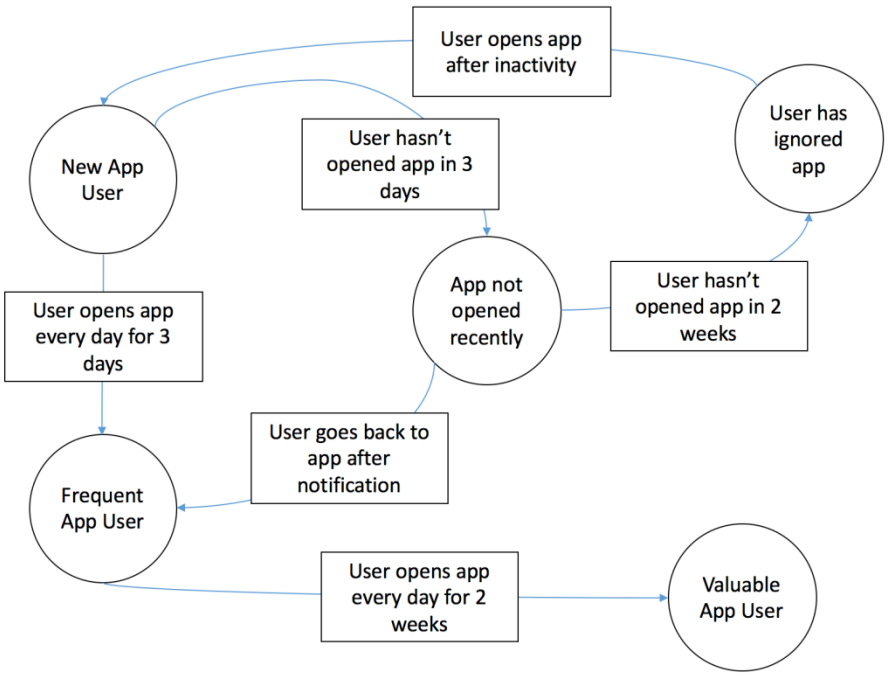


Figure 4.4 An app marketing campaign can quickly become complex. Marketing departments may want to provide targeted information to individual users based on how they've been using the app.

In this example, the overall state of the marketing campaign would be difficult to represent in a database table, but by using an actor that allows for on-the-fly configuration of its behavior, it vastly simplifies how you process incoming messages.

4.5 Summary

In this chapter, you learned:

- How to model real-world situations using state machines
- How to diagrammatically explain a state machine using state transition tables and state transition diagrams
- How to create a simple state machine using the behavior switching functionality of Akka.NET
- How to create more complex state machines using the finite state machine actor type

5

Configuration, dependency injection, and logging

This chapter covers

- Configuration of individual actor deployments
- Configuration of the actor system as a whole
- Using Akka.NET's logging system

We've seen in the past couple of chapters how you can deploy relatively simple actors that don't use the more advanced features of Akka.NET. From here on in, the rest of the book will start to build upon these concepts as we look at how Akka.NET allows you to write actors capable of solving more complex problems in the simplest way possible. To achieve that end, you first need to understand how you're able to inject configuration into Akka.NET in order to manipulate how it handles certain scenarios within the framework.

In this chapter, we'll look at how you're able to instrument and operationalize an Akka.NET application. Although you have only just seen how to create an actor, it's vital to understand how important it is to have thorough instrumentation and logging in place, even when developing simple actor-based applications. Akka.NET provides abstractions that help you to develop distributed applications, but this comes with inherent downsides. We saw in chapter 1 that one of the building blocks for the reactive manifesto is the use of asynchronous message passing as a means of communication between components within a reactive system. Because messages are passed asynchronously, there is no notion of the typical flow control that you get with applications that rely on synchronous method calls. Although this provides you with the foundation upon which you can build reliable and scalable systems, it does prevent you from effectively debugging applications using a debugger. Instead, it's important to have thorough logging and instrumentation in place, which allows you to understand the system as a whole,

rather than in isolated components. This chapter focusses on how you're able to use the functionality provided by Akka.NET to monitor and instrument an application so that you end up with centralized logging. We'll also look at some of the functionality within Akka.NET that allows more effective tailoring of an application to different environments by configuring the overall actor system, as well as the context within which the system operates.

This chapter won't serve as a comprehensive guide to all of the means of configuration within Akka.NET, but rather a baseline from which later chapters will build upon when it comes to introducing their underlying configuration options. For example, in later chapters, we'll see how you can quickly create a pool of single actors to handle more requests or even create a cluster of actor systems spread across multiple machines. In all of these cases, the underlying configuration model follows the same techniques as laid out in this chapter, and so the lessons learned in this chapter will prove to be vital for ensuring a thorough understanding of how actor systems in later chapters work.

5.1 Why do you need configuration?

Up until now, the actor systems you have used have all been relatively simple, with only a few actors within the system communicating and with a small number of messages being passed between them. After this chapter, the number of features in the systems we look at will increase, which also means an increase in the complexity of those systems. This chapter is designed to help alleviate some of the difficulties you might encounter in subsequent chapters. By ensuring you're prepared now, you'll be able to approach problems head-on later.

Let's consider for a moment the actors you've deployed into your system. They have so far required no customization at all. Ultimately, the simplest actor configuration will start to fail once you start to build more complex actors into the system. For example, how do you handle a case where you want an actor to distribute its messages to other actors in an effort to increase the amount of work that can be done in parallel? Or, alternatively, how does an actor cope when it encounters a failure? These are just two of the things you are likely to want to configure about an individual actor's deployment within your system. We'll see more use cases for this kind of configuration throughout the rest of the book as we look at the likes of remote actors and failure handling within actors. In this chapter, we'll just look at cases where you want to create more complex actors, such as those with many dependencies.

Not only can you configure actor deployment, you can also modify the way the actor system itself runs, without the need to recompile the application. You're able to change how Akka.NET deploys an actor into the actor system; for example, you may want to create a cluster of actor systems, or even change where actors are deployed within the cluster. These are more in-depth examples that you won't see until later in the book. For now, we'll look at one of the most essential parts of an Akka.NET project, which is the use of logging. In this chapter, we'll see how you can use the Akka.NET configuration to customize the loggers used and see how those can be used within an application.

5.2 Configuring an actor deployment

In each of the actor examples we've seen so far, in chapters 3 and 4, all of the deployments into the actor system have been through the generic method called `ActorOf<T>`. In these cases, `T` is the type of the actor you're telling the actor to instantiate. There are, however, times when you want to change how the actor is deployed by Akka.NET. Some of the common cases you have when you want to change the deployment include the likes of:

- *Custom constructor*—When Akka.NET deploys an actor using `ActorOf`, it simply uses the default constructor of the actor. If you have something more complex, you may need to pass some dependencies into the constructor. In this situation, you're going to need to tell Akka.NET what it needs to do in order to instantiate the actor.
- *Custom mailbox*—The mailbox is the internal component responsible for receiving messages for an actor. Although the mailbox isn't directly used by the developer, it forms a key component of the actor itself used internally within Akka.NET. There are frequently occasions where an actor will need to use a mailbox other than the default one. An example of this is if certain messages have a higher priority than others and should be processed first.
- *Deployment location*—When an actor gets deployed, it typically gets deployed into the local actor system, but as we'll see later in the book, an actor can be deployed to several other locations, including a remote system or even into a cluster of actor systems.
- *Custom dispatcher*—The dispatcher is responsible for the thread on which an actor processes its messages. For example, if you're writing an actor that processes a large number of messages, it may need more resources than other actors. In this example, the dispatcher for that actor can be set to use a dedicated thread to process messages.

These are just some of the more common requirements you are likely to need when configuring an actor. Whereas the custom mailbox, deployment location, and custom dispatcher are only used in more advanced actors, the custom constructor is frequently used, as it allows you to directly inject your dependencies into the deployment itself. Throughout the rest of this section, we'll be looking at how Akka.NET allows you to configure how an individual actor is created by the framework upon deployment into an actor system.

5.2.1 Understanding Props

In previous chapters, we've seen that having initially defined an actor, you then need to effectively deploy that actor into an actor system. Deploying an actor means that it's given an address and mailbox that are able to receive messages from other actors in the system. In order to deploy an actor and instantiate it, we've so far seen the use of the `ActorOf<T>` method on the actor system instance. For all of the examples so far, `ActorOf` has been sufficient, as it simply takes an actor type as a generic parameter and uses the default constructor.

Whenever you use this generic method, though, it doesn't automatically spawn the actor into the system. Instead, it creates a template for how it should be deployed. By creating a template for the actor, you're able to form a reproducible way of creating an instance of an actor. You can consider the template to be similar to a recipe that you might encounter when cooking food. Your recipe isn't a meal in itself, but a series of reproducible steps that are used to enable you to create a meal. By providing food and something capable of using the food, such as a chef, you're able to repeatedly create the same meal every time. Now, let's consider when something goes wrong in a restaurant, such as a waiter dropping the meal; you're able to automatically create a new meal because you have the recipe available to you.

The same principles apply when you're using actors; for example, let's consider the scenario where an actor unexpectedly quits because it encounters an error. We'll look more into how Akka.NET responds to failures in a later chapter, but for now, we'll consider possibly the easiest and most common approach to handling failures. When we saw the principles of the Reactive Manifesto, we saw that you want to build applications that are able to respond to failures within the system. One way of responding to an actor's failure is to simply restart it in the hope of getting into a known working state. In order to ensure it is restarted into the same state, you'll need a template to provide you with the reproducible deployments. The term *reproducible* in this scenario means that it must end up in the same state as when you deployed it the first time.

This recipe is known as the `Props` for an actor in Akka.NET. You'll see throughout the rest of the book that `Props` is used heavily in Akka.NET and is a requirement in the case of more complex actors. For now, though, we'll focus on the more simple aspects of the usage of `Props`, such as creating actors with more complex constructors.

CREATING PROPS

Due to the multithreaded nature of the code in which `Props` is used, every `Props` instance is immutable and requires us to use the factory functions in order to create an instance. There's a number of different options available. We'll take an actor example from chapter 3, where we discussed how to define an actor. Let's take a look at how the actor was defined. To remind ourselves, this actor defines a simple behavior for what happens whenever it receives one of two messages. If it receives a wave, then it sends a vocal greeting back to the original sender; if it receives a vocal greeting, then it writes the content of the vocal greeting to the console.

Normally, when spawning this actor, you'd use `ActorOf` with the actor type as the generic parameter, but you can create the `Props` object for this actor in a similar way. You use the generic factory method that is used to create `Props` for this type. Using this method assumes you have a default constructor on the actor that takes no parameters. After this, you receive the `Props` object, which you can then use to spawn an actor internally:

```
var personProps = Props.Create<PersonActor>();
var personActor = system.ActorOf(personProps);
```

But this hasn't really changed anything for when you want to deploy that actor, and you've ultimately had to write more code to do the exact same job. Custom `Props` become more

useful when you want to do something more complicated that is outside the scope of what Akka.NET can manage by default. For example, let's modify the original actor definition to take a dependency on a string that the actor will use to communicate with another actor. In this case, Akka.NET needs some way of knowing what string you want to pass into the constructor. You can manage this in one of two ways. You can either provide an expression that is used when an actor instance needs to be created, or you can specify the type name along with an array of parameters.

The simplest way of providing this string is by creating `Props` with the type of the actor and an array of the parameters you'll be using. For example, if you want to use the string `"Hello from Props!"`, you can create `Props` in the following way. In this case, you provide the type of the actor to `Props` using the `typeof` keyword. You then also create an array of the parameters you'll be using, which in this case is the string constant `"Hello from Props!"`:

```
            var personProps = Props.Create(typeof(PersonActor),
  "Hello from Props!");
```

This approach is by far the simplest, but you give up the potential for compile time to check that you're providing either all of the required parameters or that all parameters are the correct type for the actor definition. In order to alleviate this difficulty, Akka.NET allows you to specify an expression that is then used to create an instance of the actor. Whenever you want to deploy or redeploy that actor definition, Akka.NET will evaluate that expression and use the returned instance in the actor system.

```
            var personProps = Props.Create<PersonActor>(
() => new PersonActor("Hello from Props!"));
```

SPAWNING AN ACTOR WITH PROPS

Having created a `Props` object, you're then able to spawn it in a very similar way to how you spawned actors before using just the actor type. In this case, though, rather than using a generic method, you can just use the regular `ActorOf` method and pass in the `Props` object. You can also create it with a name, exactly as you saw when you deployed actors previously. The following example deploys your more complex actor using the `Props` object you built earlier. As you can see, there is very little difference in the process when creating actors using `Props` rather than directly deploying actors:

```
var personActor = system.ActorOf(personProps);
```

A WARNING ABOUT PROPS

As you've seen previously, when you use `Props` you can use an expression that is used to create an actor instance. You can then create a `Props` object with this expression and use that to spawn an actor. But there's no guarantee that you spawn the actor as soon as you create the `Props` object. In order to remain performant, many operations within Akka.NET are evaluated lazily. One such example of this is when you want to retrieve the original sender of the current message using the `Sender` property. This is only evaluated when it is needed, which means that if you pass the `Props` object, you create a different actor to spawn it, and

then the Sender will be different to what you expect. You can solve this problem by retrieving a reference to the current value first, as shown in the following code:

```
var sender = Sender;
var props =
    Props.Create(() => new LoadTestingActor(sender));
```

There are other scenarios within the C# language where you might also encounter this problem, most notably when using an index variable in a loop. In all of these situations, it's important to realize that the Props are not evaluated immediately and so might lead to the use of unexpected values.

5.3 Actor configuration summary

In this section, we've seen how Akka.NET is able to spawn an actor through the use of a template known as Props. We've seen how you can create these Props for a given class with a more complex example. In the next section, we'll look at how you can spawn these complex classes using a dependency injection framework. Although the use of Props might seem quite obtuse at the moment, we'll see in later chapters just how important Props is when we deal with how Akka.NET handles scaling up actors and how it handles failure. At this stage, however, the main use of Props is as a means of deploying actors with more dependencies provided through the constructor, and we'll see in the next chapter how you're able to use a dependency injection to automatically inject these dependencies into Props.

5.4 Dependency injection

In previous chapters, we saw actors that had no external dependencies and that were simple to deploy into an actor system. Then we saw how Akka.NET creates a template for these deployments, and how you can interact with these templates to provide additional dependencies into an actor's instantiation. Sometimes, actors can grow and require more and more dependencies upon external services. As these dependencies grow, you start to generate more complex chains of dependencies, for example, if one dependency requires others. Typically, you'd approach the problem by using a dependency injection framework, which is then used as a means of automatically creating any dependencies that you require.

The same principles apply to Akka.NET, and you're able to use dependency injection to create the Props object that you used as a template. In this section, you'll see how you're able to manage this. You'll see how you can create an actor using the basic principles of dependency injection by providing a number of external dependencies that you then consume within the actor instance.

5.4.1 Introducing dependency injection[1]

When you consider the tasks an actor is likely to perform, in many cases they are likely to include working with external dependencies. For example, you may have a database actor that forms the basis from which other actors in the system will communicate with the database. Or, alternatively, you may have an actor that is in charge of performing interactions with a web service. In all of these cases, you're dealing with things that you may want to quickly and easily configure so that you can replace the dependency with something else.

Let's consider your database actor again. When it communicates with the database, it probably performs the operation with a well-known abstraction over the top of it. For example, you may already be using the likes of Microsoft's Entity Framework or even a custom-written API using raw SQL queries. Regardless of the technology used, you tend to use a built-up abstraction because of its simplicity. But there may be times when you no longer want to use that abstraction to retrieve data from the database. A common situation where you might want to change these abstractions is during testing. When you write tests, you don't want to interact with external services, for a number of reasons:

- *Time to execute*—These dependencies will typically be served over high latency connections such as a network or internet connection. If every test needs to use this connection, it can quickly lead to a significant increase in the amount of time a test suite takes to execute. This is something you don't want, as you seek quick or immediate feedback from tests.
- *Difficulty in setting up data for tests*—Once you've got a test with a dependency on an external service, in order to achieve predictability, you need to have an understanding of the data in the source. In order to have this understanding, you need to configure the data source to add new test data or retrieve stale data before you're able to run a test.

In the case of testing, as soon as you have a dependency on an external service, you should ideally replace it with something that can serve the data better to your needs. Although this can be configured manually, the use of a dependency injection framework allows for the easier resolution of more complex dependency graphs. A dependency graph is the term given to the chain of dependencies that is built up as dependencies depend on other services. For example, you may have a dependency upon Gmail within your application, but for that Gmail dependency to operate, it might require an abstraction over the top of an email client.

The usages stretch far beyond just testing; for example, if you have an application that could be running in a number of different environments, you might want to switch the dependencies on a per-environment basis. If an application can run in a number of different cloud hosts, such as Amazon Web Services or Microsoft Azure, you might have different logic available for performing certain operations, such as retrieval of the current machine's IP address.

[1] This book is not intended to be a thorough introduction to dependency injection, but only provides the basics needed to begin using it. If you're interested in learning more about dependency injection and best practices, either in general or for a specific framework, a number of books on the topic are available from Manning.

Scenarios such as these are where dependency injection can prove to be useful. By registering a concrete instantiation of a dependency, such as a class, against a template of the dependency, such as an interface, you're able to separate the implementation of the dependency from how you use it.

You'll see throughout this section how you can manage more complex dependency graphs in Akka.NET through the use of a dependency injection framework. We'll introduce containers and show how they can be used to provide dependencies to an actor upon spawning it.

Do you need dependency injection?

Many developers swear by dependency injection as a means of creating more testable code, but before simply deciding to use a dependency injection framework, you should consider whether it is truly necessary. Akka.NET presents a different approach to concurrency through the use of actors, which were typically not considered when previously using dependency injection.

Due to the nature of actors, you have to be cautious of anything that can be a potential source of state sharing. You saw in chapters 2 and 3 how you should design actors so that they don't share any state between them in order to achieve better scalability and fault tolerance. This goes against the grain of how dependency injection frameworks work: they try to use a dependency for as long as possible, which is then shared across many instances. You also don't know how long an actor might be in use. Some actors may be created to simply reply to a single message, and have a lifetime of potentially less than a second, whereas you may have other actors that stay up for long periods of time with potentially no downtime. This poses a different series of challenges that some dependency injection frameworks may not be tailored towards.

Before using dependency injection in Akka.NET, you should strongly consider whether it brings significant advantages to your codebase when compared to the need to be more careful about the dependencies that are used by it.

5.4.2 Configuring an inversion of control container

When using dependency injection in Akka.NET, you're able to use one of a number of different dependency injection libraries, with the option of including adapters for others, if they're not already provided. All of the adapters are available through NuGet, along with a number of community-contributed alternatives. Some available adapters include Ninject, Castle Windsor, and Autofac, although throughout this section, we'll be focusing on using Autofac. The only key difference within the API is how you register dependencies within your framework.

The first step is to install the Autofac library from NuGet, along with the adapter that's needed to use it within Akka.NET. In the same way as you installed the Akka.NET project in chapter 3, you need to install the Akka.DI.AutoFac NuGet project. This will also add the dependency injection library to your project, if it's not already included. Before you're able to create actors with automatically resolved dependencies, you need to configure the container. The container is the component that is responsible for mapping a required type onto a type instance. We'll continue with how you can insert extra dependencies into your actor system. In this case, you'll register every request for a string to provide a simple instance of a string. This is something you would not typically do with a dependency injection container, and instead you

would tend to provide more complex type definitions. Within AutoFac, you can quickly create a container and supply an instance to use for a given type[2]:

```
var containerBuilder = new Autofac.ContainerBuilder();
      containerBuilder.RegisterInstance<string>(
    "Hello from a DI container");
    var container = containerBuilder.Build();
```

Now that you've got a container, you can create instances of types from this. Although you are able to create types with it, you need the Akka.NET framework to be able to use this to create instances of actors. In order to manage this, you need to register the container with the framework. To do this, you need to create a dependency resolver specific to your chosen dependency injection framework. In this case, you need to create an `AutoFacDependencyResolver`. Upon creating the resolver, it automatically gets registered to the actor system so that it can be used to create actors:

```
var propsResolver = new AutoFacDependencyResolver(container, system);
```

As you saw earlier, you need a `Props` object in order to spawn an actor. In cases where you want to create an actor using the dependency injection framework, you use an extension method provided on the `ActorSystem`. Once you have retrieved a reference to the DI extension, you simply call the `Props` method with the type you want to create as a generic type argument. The following example creates the `Props` for the person actor by retrieving all the dependencies from the provided container. To create the `Props` object, you simply retrieve the `Props` from the context using the Akka.NET dependency injection features.

```
var props = system.DI().Props<PersonActor>();
```

Once you've retrieved the `Props`, you're then able to use them in the same way as you have done previously. This ensures that you're able to use them in later chapters, when we're looking at configuring more advanced features. You're also able to use it to deploy the actor in the same way as you did previously, when manually generating the actors.

5.4.3 Dependency injection summary

This section has focused heavily on how to configure and manage complex dependency graphs for actors within Akka.NET through the use of a dependency injection framework. Although you've only seen how to use one dependency injection framework, there are a number available through NuGet, for which bindings are available for Akka.NET with thorough documentation. The same principles apply regardless of which framework is used, allowing you to use your preferred dependency injection framework.

This section also sees the end of our introduction to independent actor deployment configuration with `Props`. Throughout the rest of the book, we'll go through more and more examples of how `Props` is used to provide additional functionality to an actor's deployment,

[2] For a more in-depth guide on how to use AutoFac as a dependency injection framework, see the official documentation available at http://www.autofac.org

such as with remote actors or clustered actors. Throughout the rest of this chapter, we'll look at further configuration within Akka.NET using HOCON, and how you can use it to configure actor system-wide settings.

5.5 HOCON – A human-readable configuration file format

Many libraries and frameworks today tend to use the likes of XML or JSON as a means of storing configuration data from which data is read. Typically, this includes configuration variables that might need to be changed depending upon the environment at runtime, or need to be frequently changed without requiring a re-compilation before it can be changed again. The main difficulty of these means of storing configuration data is they are far from ideal for the task at hand.

Both XML and JSON were originally intended as a data interchange format for sending data between multiple applications. In these examples, the formats are typically designed for high-speed serialization and deserialization, but they tend to be more difficult to read and write by a human. For example, in the case of XML, it's a largely verbose format with significant amounts of repetition that the user is required to write. Also, it is far from intuitive for the user when it comes to writing it, due to the potential confusion of when to use attributes on data or when to use nested elements. JSON removes large amounts of the verbosity associated with XML, but it still suffers from problems. The lack of comments can cause issues, particularly when documenting any complex configuration. Furthermore, the format can at times be difficult to comprehend when dealing with data that doesn't naturally map onto the types provided by JavaScript.

In order to address many of these issues, Akka.NET uses a configuration file format known as HOCON, which stands for Human Optimized Configuration Object Notation. The key point here is that it's human-readable and has been designed from the ground up to be easier to read and write than many other formats.

5.5.1 What is HOCON?

HOCON is a configuration file format that was designed as a superset of JSON. As such, you may notice some similarities between a HOCON file and a JSON file. But HOCON makes some changes to JSON to remove a lot of the noise that doesn't add any value for a human reading it. It removes the likes of the leading and trailing braces found in JSON and adds other features, such as comments, which ensure that developers are able to express the quirks or explain certain decisions within their configuration files.

Let's now take a look at a simple HOCON file and see how to express certain constructs within the configuration file. The following code listing shows a file that has been tailored to give demonstrations of many of the features available within HOCON. The first noticeable point of HOCON is that it resembles a JSON file. In fact, since HOCON is a superset of JSON, valid JSON can also be used to configure Akka.NET.

As you can see, there are a lot of similarities between JSON and HOCON, but some key changes have been made in order to ensure it's as human-readable as possible. For example,

there's no need to supply opening and closing braces. The separator between keys and values has also changed from a colon to an equals symbol. These are the two most noticeable changes, but there are many other features designed to assist readability. In the example above, there were a number of nested sections, each of which had only one key within it. In order to simplify cases such as this, HOCON allows you to specify paths where each path segment is separated by a full stop. The two following examples both resolve to the same configuration object:

```
akka {
 cluster {
  roles = []
 }
}

  akka.cluster.roles = []
```

From time to time, you may need to convey additional information about certain decisions within the configuration. For example, you might need to specify a detail about the overall structure of a configuration string for whoever might be looking to change it at a later date. When you have situations like this in a codebase, you can use a comment, which can be used to express more intent than you'd typically have available. It's no different in Akka.NET, and you're free to use one of two commenting styles. Comments can either start with a double forward slash, similar to C-style languages, or a hash character, similar to languages such as Python and Bash. In the following example, you add a comment above a key value, which is used to describe the intent of the definition. Comments extend through the length of a line, and a comment is deemed to have ended upon reaching a line separator.

```
#Specifies the roles which this node belongs to
akka.cluster.roles = []
```

Two of the most common things you'll want to configure within Akka.NET are timeouts or the intervals between certain operations. Although the other common configuration formats typically approach the problem by simply using milliseconds or seconds as the value and leave you to work out which is required from the documentation, HOCON has support for certain units on values. For example, the following time values ultimately lead to the same value being made available in the configuration:

```
akka.cluster.seed-node-timeout = 120 s

  akka.cluster.seed-node-timeout = 2 m
```

These are some of the features of HOCON that we'll be using throughout the rest of the book; nut there is plenty more to discover and use, in particular if you

5.5.2 Loading configuration into an actor system

Now that you've seen what the HOCON format looks like, you can start to use it within an actor system. You can retrieve the configuration from a couple of different places within an

application. At the simplest level is the `ConfigurationFactory.Parse` method, which takes in a string containing the HOCON definition. As a layer of abstraction on top of this, you can also choose to retrieve the configuration from a resource file embedded within the application. The final option is to retrieve the configuration from an Akka element stored within the application's App.config file as CDATA. For now, we'll use the .NET file APIs and `ParseString` as a means to retrieve the configuration from a file stored in the same directory. Assuming that the configuration file is in the same directory as the executable, you can load it in as follows:

```
var configString = File.ReadAllText("actorsystem.conf");
var config = ConfigurationFactory.ParseString(configString);
ActorSystem actorSystem =
    ActorSystem.Create("configuredActorSystem", config);
```

Having now created an actor system along with a configuration file, because Akka.NET starts the components within the actor system, such as logging and many others you'll see later in the book, the components will overwrite their default values with the values provided in the configuration file. Because all components within Akka.NET use default values, this means you don't need to create a configuration file with every single value for every single component, but instead you just fill the configuration with values that you need to change.

5.5.3 HOCON summary

In this section, you've seen some of the features of HOCON and how it differs from other file formats typically used by applications for configuring them. You saw how the features provided by Akka.NET and HOCON allow you to write application configuration that is tailored towards the environment within which it's running. Finally, you saw the Akka.NET API, which allows you to load the configuration into an actor system when you create it. HOCON is a really feature-rich configuration format, providing many more features outside the scope of this section, including the likes of merging keys and more advanced key replacements. Up until this point, you haven't needed to configure some of the internals of the actor system because Akka.NET provides a configuration file of sane defaults that it uses as the backup if you don't provide a setting. As you use more advanced features of Akka.NET, you're likely to need to modify certain settings, such as how actors get created if they're in a cluster or a remote actor system. The next section shows a simple real-life example of using HOCON to modify a simple section of the framework, the logger.

5.6 Logging

You've now seen how you can configure the actor system within which your actors are running, thanks to the HOCON configuration available in Akka.NET. You'll now see how to create a configuration file and alter how key parts of the system operate. In this section, we'll focus on the logging capabilities of Akka.NET and how to use it in order to gain a deeper insight into the overall state of the actor system. You'll see how to access the logging utilities provided by Akka.NET, as well as how to configure the log sink to use a different logging system.

Logging is one of the most important operations you're likely to use in any Akka.NET application to counter some of the problems that arise with asynchronous, potentially distributed systems. In this section, we'll look at how to configure the logging within Akka.NET and how to use this logging functionality from within an actor to send messages to your configured log sink.

5.6.1 Why do you need logging?

Logging plays a valuable part in any application you build. It provides you with valuable insights into how our system is running at any time. But it's especially important in the case of actor systems, or more generally, asynchronous systems. This is due to the disadvantages that come when writing asynchronous systems. A synchronous system provides a linear flow through the system at the cost of a reduction in scalability and fault tolerance, as we saw in chapter 1. But as asynchronous systems allay this determinism, you're left in a situation where you need to be able to truly get a thorough, deep understanding of your system.

For example, let's consider the example flow of a message as it proceeds through a number of stages in a processing pipeline. Given the message flow shown in the following diagram, messages are processed or modified in one actor before being passed onto the next one. Within a synchronous system, you would be able to iteratively step through each of the processing stages before arriving at the final result. At each stage, the workload would be visible and you'd be able to see the results of that stage. Asynchronicity forfeits this and gives you benefits related to scalability and fault tolerance in its place. By using logging in its place, you can see the flow of messages through the chain of actors, allowing you to see potential sources of failure when the system is running. By using logging within messages, you're able to see cases where messages might be getting directed to targets other than their intended destination, or even where messages aren't reaching any target.

```
<INSERT MESSAGE FLOW DIAGRAM>
```

This is just one example of when logging can be used. There is a vast number of other uses, but they all relate to the same concepts, which is that logging is designed to provide better visibility for cases if or when something goes wrong in an environment to which you can't attach a debugger. In the event of a failure in the system, you can consult the history stored within the logs and actively step through the circumstances that lead to the system failure.

5.6.2 Writing to the log

Before you can write to the Akka.NET log, you need to retrieve access to the system logger. The logger is retrieved through the static `GetLogger` method on the `Logging` class. From there, you need to pass in a logging target. In this case, you use the actor's context. Internally, the factory method will retrieve the

```
private readonly ILoggingAdapter _log = Logging.GetLogger(Context);
```

Having retrieved the logger, you can then simply write to the log by using any of the methods provided. For example, if you want to write a message at the Debug log level, you can simply call the Debug method with a string that you want to print. Akka.NET then formats the string to provide additional information before it is written to the log. For example, it adds information such as the thread on which the actor was running, as well as the address of the actor that logged the message.

Logging is asynchronous

Like many other components within Akka.NET, logging is completely asynchronous. Within the logging system, an actor is responsible for receiving log messages that it then writes to the log output. This means that, sometimes, as your system is being shut down, messages going into the log may not reach the dinal destination, leading to the apparent loss of messages.

5.6.3 Customising a logger deployment

As we saw in the configuration section, Akka.NET uses sane default values in situations where no other values are provided. This is no different for cases when a logger is not provided. By default, Akka.NET creates a logger that prints all received messages to the console. So, if you want to just direct any logged messages to the console, you can leave the configuration as is. When it comes to a production deployment, though, you'll want to log all of your messages to a centralized server somewhere, where they can be more easily processed or read. Although Akka.NET doesn't provide the infrastructure to directly log messages to a centralized system, it provides adapters that are able to perform the task instead. In this example, you'll use the NLog logging library, which is able to append log messages to a number of different sources, including files on the filesystem, databases, and even emails. For now, we'll only be using a simple logger that directs the output to an output file.[3]

The first step is to add the logging library to the project. You can do this from NuGet by simply adding `Akka.Logger.NLog`. In order to customize some of the core components, you need to create a file called `NLog.config` and add it to the project. The `NLog.config` file provides details of how it should handle certain log messages. For example, it can specify that debug messages go to one location, such as a database, and error messages go to another, such as a text file. For now, you'll be using a simple config file that simply sends all log messages to a text file. For more information on configuring and using NLog, see the project documentation.

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <targets>
    <target name="logfile" xsi:type="File" fileName="file.txt" />
```

[3] This is a relatively simple example of a logging output. More complex outputs are available, but they won't be covered here. For more information, see the NLog project site.

```
  </targets>
  <rules>
    <logger name="*" minlevel="Info" writeTo="logfile" />
  </rules>
</nlog>
```

Now that Log4Net is configured within your application, you can configure Akka.NET to use it as the sink to which all logs are sent. This is done by modifying the configuration object that you pass into the actor system when you create it. To change your log providers, you can add an array of possible Akka.NET loggers:

```
akka {
    loggers = ["Akka.Logger.NLog.NLogLogger, Akka.Logger.NLog"]
}
```

There are further settings that you can modify that also affect the logging behavior of the application. For example, sometimes you may not have an actor instantiated at the address to which you sent a message. In this case, the message would get diverted to a special actor known as `DeadLetters`. You can configure the logging functionality of Akka.NET so that it outputs a message to the log whenever messages are undelivered.

```
akka {
    actor.debug.unhandled = on
}
```

Creating custom loggers

Although Akka.NET provides a wide variety of different adapters for various commonly used logging libraries, there may be a situation in which it doesn't support your current choice of logging library. For example, you may be using a custom developed solution for your business or a less commonly-used library. Akka.NET provides an extensibility point that allows you to create a custom logging adapter. This custom logging adapter then receives messages from your Akka.NET application and outputs them to your logging library. This is out of the scope of this chapter, but we'll come back to it later on in the book, when we talk about how to extend the Akka.NET library and the components within it.

5.6.4 Logging summary

You've seen in this section the benefits of using logging when you're dealing with an asynchronous system, and how it allows you to gain better insight into how an application is working and whether it is operating correctly at runtime. You saw how logging works in Akka.NET through the actor system and how to send messages to it from within an actor. Finally, you saw how to change the output destination for any log messages you write within the actor system.

5.7 Case Study: Configuration – Distributed systems – Docker?

In any modern application development workflow, it's common to have a variety of environments that are used at different stages of the development pipeline. For example, it's

common to have a development, test, and production environment. But, it's unlikely that configuration is common across all the environments. When logging in a development environment, you can simply log to the local development machine, but in a production environment, you'll probably be aggregating all of your logs into one centralized log management service, allowing for simplified problem finding in the event of production issues. Similarly, if you're persisting data into a database or alternative data storage location, you'll want to ensure that production data and test data are stored in separate datastores, and for testing purposes you may not want to persist data at all.

In this chapter, you saw how the configuration components work within Akka.NET; the configuration tooling allows you to change application parameters without having to recompile the application. This ensures that you're able to simply change the configuration file for the different environments, and the changes will be reflected within the application.

As part of application configuration, there are typically two categories of configuration parameter: application settings and environment settings. Application settings are the parameters that are responsible for driving business logic within the application. For example, in a machine learning component, this will include the configuration of the machine learning models. These are typically things that might change frequently as the application is used, but will remain stable across environments. But, environment settings are the settings that describe how the application interacts with other systems in the environment. This includes the likes of connection strings to databases or the keys needed to consume external APIs. These are configuration parameters that should change between environments.

The Akka.NET HOCON configuration simplifies this usage by allowing you to overwrite configuration parameters depending upon the environment. In the following diagram, you can see that there's a common configuration file shared across all of the environments. This will include the common configuration elements that are shared across all environments, notably the application settings. You can also see that there is an additional configuration file that is different for each environment and contains the settings for each independent environment. Internally, the HOCON configuration will then merge the two files, the shared configuration and environment-specific configuration, into a single configuration file.

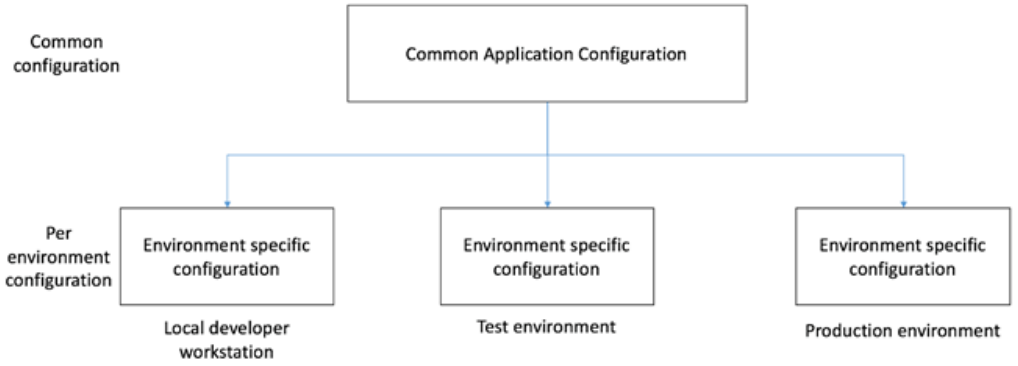


Figure 5.1 A modern application needs to run in multiple contexts, either local, test, or production. It's important to be able to configure the application for each environment without needing to recompile.

This ensures that you're able to maintain consistency across environments for configuration parameters within applications that still need some simplified runtime modification. You're also able to easily modify the environment-specific settings and ensure that there is no possibility of settings from one environment making their way into another environment.

5.8 Summary

In this chapter, you learned:

- The importance of monitoring and logging in asynchronous systems
- How Akka.NET provides a centralized logging point that's capable of collecting both system and application logs
- How to configure an Akka.NET application to suit the environment

6

Failure handling

This chapter covers

- **Where failures can happen in asynchronous systems**
- **How actors in Akka.NET are able to handle failures**
- **Dealing with failures in Akka.NET**

So far, we've seen how you can create simple actors in Akka.NET in chapter 3, then we saw how you can create more complex actors that are able to react to changes in their environment in chapter 4, through the use of state machines. In chapter 5, we saw how actors are configured in Akka.NET through the use of Props for individual actor deployments and the use of HOCON for configuring the internals of the actor system. Through those chapters, we saw how you can start to implement the traits of the reactive systems we saw in chapter 1. Notably, in chapter 3, we saw how you can build actors that communicate asynchronously through the use of message passing, which forms the building blocks of reactive applications.

In this chapter, we'll look at how you can implement one of the blocks that sits on top of the message passing layer, and how you can respond to service failures within an Akka.NET application. Throughout this chapter, we'll look at what a failure typically involves, especially in the context of distributed environments in which Akka.NET might be running. Given these failures, we'll then see how you're able to react to them in the best possible way to ensure that any applications you write are able to operate independently for as long as possible without the need for frequent human intervention.

6.1 Understanding failures

As the applications you write become more and more complex, with more moving parts than has previously been considered, there is more potential for errors to occur at many different points within the stack. If you want to write applications that are able to operate for

extended periods of time with minimal downtime, then it's important that the applications you write are able to withstand these failures.

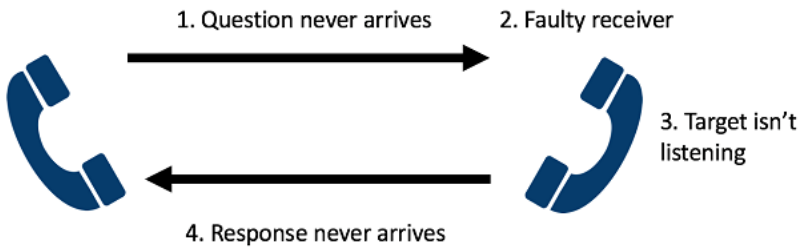


Figure 6.1 In a simple phone call, there are numerous possibilities for how you can fail to receive a response to a question.

Once you make applications asynchronous, you potentially increase the number of failures within the system, while simultaneously making it harder to find the root cause of the issue. For example, let's consider the scenario where you make a phone call to another person when you can't directly see the other person. During the phone call, you might ask the other person a question. In an ideal situation, the person will hear the question and respond directly, but if you don't receive a response, then you're presented with a wide range of possible circumstances that may have lead to this scenario:

- *The other person didn't receive the question.* In this case, although you may have asked the other person a question, there's no way of knowing that the other person did ultimately receive the question.
- *The other person didn't understand the question.* You might have asked the other person something that they don't understand either at all or in the given context. In this scenario, the other party might not know how to respond to the question, meaning that they ultimately choose not to respond.
- *The other person is preoccupied with something else.* When you're talking to a person on the phone, you can't directly see that person and so you don't know what they're doing at the moment you ask them. For example, they may have had to temporarily put down the phone to respond to a more urgent matter.
- *Something serious might have happened to the person.* Once again, you can't see what the other person is doing at the time you ask them a question. If you ask them to do something as part of the question and it causes them harm, then they won't be able to respond to you and will need attention from somebody nearby before they can respond to your question.
- *You never received their reply.* The other person might have received the question, formulated the response, and then told you the response, but the response might have been lost if the phone signal was too weak to transmit the response.

These are just some of the issues you may face when trying to have a phone conversation with someone else; but all of these potential failures directly translate to failures that you may see in the world of asynchronous systems development. Ultimately, these are all scenarios that you should cater for and attempt to mitigate throughout the design process of any applications that you develop. Because these failures are likely to be faced by applications developed with Akka.NET, the framework provides features designed to help developers create applications that do stay responsive, even in the face of these forms of failure. Throughout the rest of this chapter, we'll be looking at three key elements of writing failure-resistant applications, including supervision trees, failure recovery, and message delivery. It is important to realize that these issues are faced on a daily basis by all asynchronous applications and not just those written with Akka.NET.

6.2 Handling application-level failures

The first of the failure cases we'll look at in this chapter is failures that your application can directly cause. These are the failures of application logic caused by attempting to perform an operation when the system is not in a valid state. Some common examples of these sorts of operations include trying to access data stored on objects that are currently null references, or invoking a method at the wrong stage in the application's lifecycle. There are, ultimately, many things that can lead to these situations. But, when we consider the lifecycle of an application, it can be thought of as an original starting state with a number of transitions leading to a finished state. The following diagram shows how you modify a state by performing operations on the data.

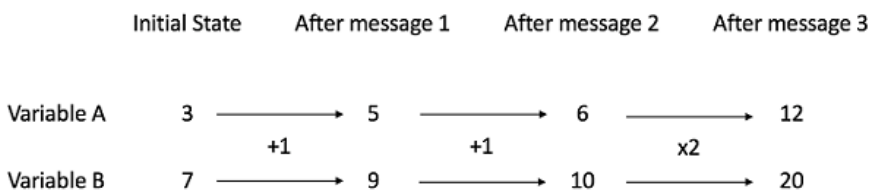


Figure 6.2 A variable's state can be considered as a series of events applied to an initial state.

But there may be times when you encounter problems with transitioning into a new state. In the case where you're performing a number of potentially dangerous operations, you don't want to leave the system with only half of them completed. In this scenario, you can quickly end up with a system in an indeterminate state. The following diagram shows a number of operations that are applied to an object until it unfortunately encounters an error. The object has then ended up in a situation where it's set a local state before encountering an error, which it tries to recover from. After this, there's no guarantee that any operation that is executed can be deemed valid, because it existed in an indeterminate state.

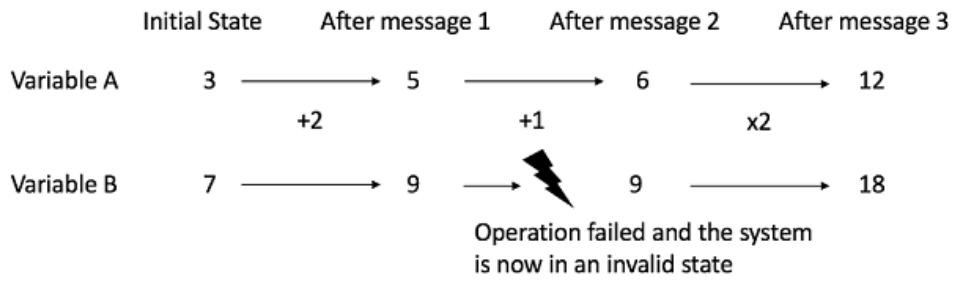


Figure 6.3 An event that fails to apply a change to a variable's state then leaves the system in an invalid state.

Throughout this section, we'll be looking at how severe these kinds of issues are, as well as how you're able to recover from them normally. We'll see the Akka.NET approach to dealing with these sorts of errors and the shift in mindset that is required when designing applications that are truly capable of recovering from errors on their own without the need for human interaction.

6.2.1 Responding to application errors

Programming languages have always provided options for dealing with errors within them. Relatively simple languages, such as C, simply relied on the use of an integer to represent an error status returned from a function. But, more recent programming languages have provided more advanced means of handling scenarios where the application has ended up in an invalid state. One example of this is the use of exceptions as a means of conveying more in-depth information relating to an error. For example, one of the more common exceptions that you might see when developing applications in C# is the `NullReferenceException`. The issue that causes this is trying to use an object that has not yet been initialized and is still null. When the application runs, it tries to retrieve the data stored at that address and is unable to because nothing exists at that memory location. We can say at this stage that this exception was probably encountered because the application ended up in an invalid state. This is a key example of what an application error is. There are many potential causes of such an error, but the root cause is that the application has performed an operation that has forced the system into an invalid state at some stage in its lifetime.

This is a scenario that you will probably encounter relatively frequently in the world of technology. Let's consider for a moment the sheer complexity of a modern-day application in relation to the universe in which we live. For an application that has six 32-bit integers, there are more than 6×10^{57} possible states that those six integers can exist in (2 ^ 32-bit arrangements per number with six possible arrangements, giving $2^{32 * 6}$), but the earth on which we live has only 1.33×10^{50} atoms. Given just how many potential states there are for six simple integers to end up in, it's not unfeasible to think, therefore, that a more complex application could end up in a broken state, especially because you have neither the computational resources to check every state, nor an understanding of what should happen in

each state. But, there is one state in which we know it's almost guaranteed to work, which is the very first state that it exists in once the object has been created. This is an approach to solving technological problems that is used fairly frequently. For example, if your computer starts behaving erratically or slowing down, the first thing you do in order to try and eradicate the problem is to simply restart the machine in the hope that it will return to a known good state, typically the state in which the system was started. After a number of tries, this might not work and the problem might not be solved; you can then try to freshly install the operating system in an effort to return the whole machine to a known state.

Due to the isolated nature of components written within Akka.NET, this is an approach to maintaining fault tolerance that proves to be viable. With more tightly coupled applications, it's unlikely to be possible to selectively recreate selected components within that system. But due to the loosely coupled message passing architecture found within Akka.NET, it ensures that you are able to remove components temporarily in an effort to fix them. You typically need some way to signal that the component has faulted to the other system in charge of monitoring for failure. Fortunately, such a system is provided by the .NET framework, as we discussed earlier, in the form of exceptions. Due to the nature of exceptions, they are considered to be a fatal error and will lead to the whole application crashing in the event that they are left unhandled. But within Akka.NET, any unhandled exceptions are considered to be a crash of the internal logic of the application.

So far, we've simply considered that an invalid state is a potential source of errors; but you can include a number of other categories of problems that may also result in errors. The first of these is a logic error when processing a certain message that an actor receives. In this case, there's little you can do to manage this situation at runtime, other than completely failing the entire application. An example of such a failure might be that you have used some specific hardcoded logic that might divide a number within the message by a constant number within the actor. If this constant number has been initialized to 0, then it will always lead to problems whenever that type of message is received by the actor. In this scenario, there's no operation you can perform that could potentially alleviate this situation. Instead, you need to redeploy the application with the required fixes in place.

Another potential failure scenario is the case of transient system failures. A transient failure is one that may only last for a certain period before being fixed again without the need for any external interaction. An example of this is if you communicate with an external service, such as a web API or a database. In both of these cases, the external services may be afflicted by errors that then propagate through into your system. We saw in chapter 4 how you can address these sorts of issues, relating to transience, by using finite state machines to create objects resembling circuit breakers.

Ultimately, from a black box perspective, there's no single means of determining the source of the application error; instead, you take the same approach across all three scenarios in the hope that the issue is not threatening to the overall integrity of the application. In this case, you can simply attempt to restart the failing actor in an effort to try and refresh it back into its original state. This is not to say that continuous restarting of the actor is the solution

to the problem. If it's a continuous logic problem that is leading to issues, then you need to ensure that the logic bugs are addressed. This requires the thorough instrumentation of your codebase, through the logging functionality we saw in chapter 5, so that you're able to understand why actors are restarting and how you can fix the application in the future.

6.2.2 The Akka.NET supervision tree

We saw in chapter 3 what happens when you deploy an actor into an actor system and the hierarchy of actors that is formed as you deploy each new actor as a child of another. The benefits we saw at the time related to scoping actors into related groups so that you could build up an effective hierarchy. But this is only one advantage of designing systems in such a manner.

Upon spawning an actor into a given context, whether that's into the actor system's root level or as the child of another actor, that actor is configured to have a parent, which is responsible for monitoring the status of that actor. Let's take an example from the world of business to show how this process works in the real world. When it comes to finding a job, you have one of two options: you can either create your own company or you can join an existing company. Regardless of which option you choose, you ultimately have somebody who sits in the position above you in the hierarchy, who you directly report to. If you've joined a company, then you probably have a manager, who is given control over one area of the company. If something that you're working on goes seriously wrong, then you tell your manager what happened and look at what steps can be taken to get the work back on track. This might involve several different options, including firing you or the whole team or even having to report the issue to his boss. There'll be times when you're at the top of the tree, either from starting your own company or being promoted up the hierarchy so that you don't appear to report to anybody, but you still have to report to the government and its offices about your activities as a company, who are then in charge of sorting out any serious problems that may arise.

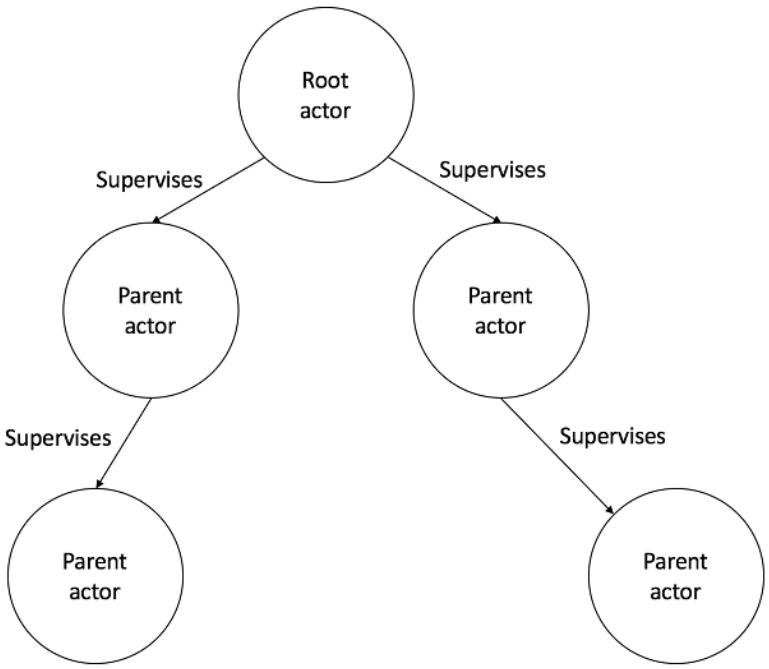


Figure 6.4 The Akka.NET deployment tree also leads itself to a natural hierarchy of supervision in the event of errors.

The same idea is used in Akka.NET. When an actor encounters a serious error, it tells its immediate parent that something has gone wrong. Upon receiving this notification, the parent is able to decide on the best choice of action for the given error. By default, Akka.NET will just restart the child in the hope that this will fix the issue; but if the error continues to be received, then it will escalate the issue to its parent. This will be continuously executed until the problem is solved - a technique that is also used within many operating systems in order to ensure that a single application failing doesn't cause the entire operating system to crash as a result of a minor error in one application. The most notable operating system kernel that uses this method is the Linux kernel.

Internally, Akka.NET handles this behavior through the use of supervision strategies, which inform the framework how it should respond when it detects the failure of an actor's processing stage. A supervision strategy consists of four key components, which are used in order to make decisions as to what should happen following an error: the actors upon which to perform the action, the action to take depending upon the failure, the maximum number of the same type of error, and the timespan within which those errors can occur.

SPECIFYING ACTORS TO RESTART

The first component of the supervision strategy is the actors that should have the actions performed on them in the event of failure. This is known as the supervision strategy itself, and Akka.NET provides two key options within the framework. There is the opportunity to add more, but the two provided cover almost all of the common actions that are frequently required. The two supervision strategies provided are the One for One strategy and the All for One strategy. In the case of the One for One strategy, only the failing actor and any actors that have also been deployed under that actor are restarted. The following diagram shows which actor gets restarted in the case of failure. As can be seen, once ChildA is deemed to have failed, it gets the action performed on it.

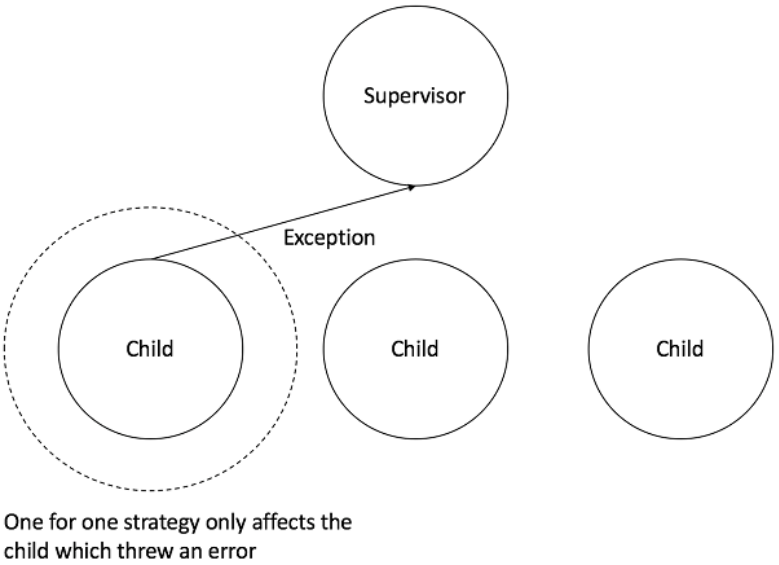


Figure 6.5 The one for one strategy will apply the given action only to the child that encountered a problem.

The All for One strategy, however, performs the action on the failing actor as well as all of its siblings. The following diagram shows the case where ChildA fails. Once ChildA is considered to have failed, the resulting action is performed on it; but it is also applied to ChildB and ChildC as well.

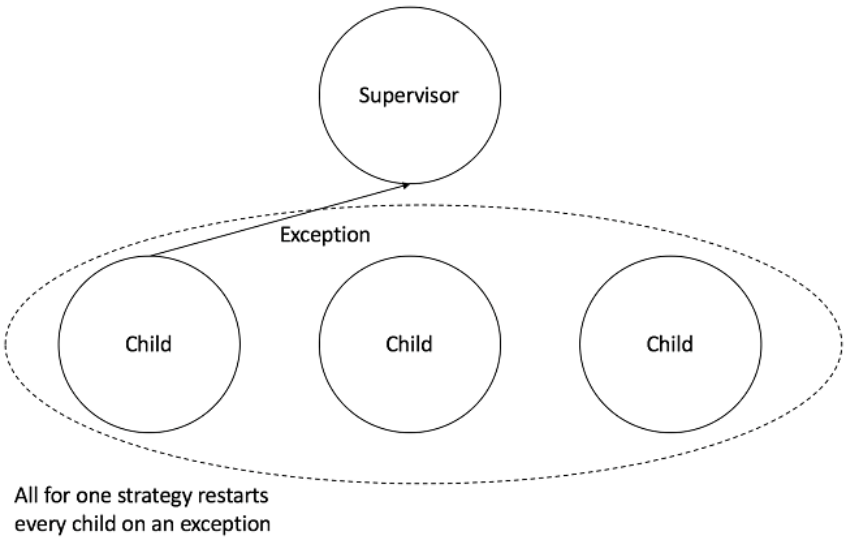


Figure 6.6 The all for one strategy will also apply the given action to all of the siblings of the actor that encountered an error.

Typically, the only supervision strategy you will need to use is the One for One strategy. The All for One strategy is only really used by actors that have a tight coupling on certain key components with their siblings. This might, as an example, include the need to recompute a shared resource that all actors need in order to process messages. We saw in chapter 3 how Akka.NET tries to provide sane defaults for configuration settings, and it's no different in the case of supervision. By default, Akka.NET uses the One for One strategy for any freshly deployed actor.

CHOOSING WHICH ACTION TO TAKE

Upon detecting a failure, it's important for Akka.NET to understand what to do next. This is determined through the use of the `Decider` component of the supervisor strategy. The `Decider` is a simple component that supplies an instruction back to Akka.NET, directing it as to what the next action to take should be. This is achieved through the use of what Akka.NET refers to as a `Directive`. Akka.NET provides four directives that you're able to use to indicate to the framework how it should respond to failure:

- *Resume*—This will cause the actor to simply ignore the error and continue on to the next message in its mailbox without performing any special operations relating to the error.
- *Restart*—The actor will be restarted in an attempt to resolve the error, but the actor won't attempt to reprocess the same message and will instead restart with the next message in the mailbox as the first one it will try and process.

- *Stop*—The actor will be shut down safely and will not receive any more messages. Any messages in its queue will be considered undelivered, and the undelivered message error will be written to the log if it's enabled.
- *Escalate*—Sometimes an error might be so serious that the supervisor has only one possible action, which is to tell its supervisor that something has gone very wrong. In this case, it will be left up to the parent to decide on the best course of action for the error. Then, the supervision strategy of the parent will be used, depending upon the error.

Within Akka.NET, `Decider` is a simple function that you can pass into your supervision strategy that simply returns a `Directive` when presented with an exception. Within a `Decider`, you simply have to check what exception has been invoked within the actor and return the appropriate `Directive`. As an example, you can create a simple `Decider` that responds to some of the typical sorts of failure you might experience.

Let's consider the example of the shopping cart that we saw earlier. In that case, each shopping cart has an associated actor responsible for communicating with the external payment gateway to settle the final bill for the contents of the shopping cart. There might also be other actors deployed as children of that shopping cart. For example, you might have another actor that is responsible for applying any discounts to the entire contents. Because your payment gateway is typically provided by an external entity, the data you get back might not always be in the form you expect. In this case, it will probably cause an issue within the actor that leads to an exception. As we've seen in this chapter, an exception will cause the actor to crash and a message to be sent to the parent asking how it should respond. You have a choice of supervision strategies you can use, either restarting the faulty actor or restarting all child actors. This scenario is an ideal case for using the One for One restart strategy, as the other child actors will not then be restarted as a result of the payment gateway actor failing. But if the child actors communicated with each other and shared state amongst themselves, then you'd want to restart all of them, as they could have an invalid shared state.

But there are sometimes errors that don't affect the overall operation of your actor. For example, when using a payment gateway through the network, you might encounter a response timeout. Although you could attempt to restart the actor, it's likely that any other services will simply resend the request if it doesn't happen within a predefined time period, ensuring that you don't need to restart the actor. You can then build this logic up into a `Decider`. If you get a `TimeoutException`, then you simply resume processing messages through the actor, but if you get a `MalformedDataException`, then you completely restart the actor.

```
Decider.From(exception =>
    {
        if (exception is ArithmeticException) return Directive.Restart;
        else if (exception is NullReferenceException) return Directive.Resume;
        else return Directive.Resume;
    });
```

The `Decider` is ultimately the core of the supervision strategy and allows you to narrow your issues down to a point and attempt to remedy them in the most appropriate way possible. This, along with the supervision strategy, is all that is required to effectively handle errors within your actors and it can lead to an easier way of managing the issues that your applications are likely to encounter. Although you can choose to create your own custom `Decider`, if you are using the default supervision strategy within Akka.NET then it simply uses a `Decider` that restarts every actor regardless of which exception was raised by it.

CUMULATIVE ERRORS

So far, we've simply considered what happens for each independent error that may be caused by the state within an actor becoming corrupt; but there are more categories of errors, as we saw earlier. Notably, what happens if the state of the failing actor is corrupted because of a corrupted state in the parent? In this case, there's no possiblity of the actor recovering from the failure, because it's been incorrectly configured by its parent. Here, you need to restart the parent as well. In these cases, you also need to be able to escalate the issue. In order to do that, you can specify that if the actor is made to restart a given number of times within a given time period, then it will escalate the error to the parent. By default, Akka.NET uses 10 restarts within a one-minute period to determine whether the next exception should be propagated up the hierarchy.

PIECING IT TOGETHER

We've seen the components that make up the supervision strategy of an actor. You can now start to piece these together and show how to use them within an actor. We'll continue to use the machine learning example, where you have already designed the `Directives`. Now you can start to add together all of the components. In order to do this, the first step is to create a `SupervisorStrategy` based on all of the parts we've seen so far. In this example, you'll be telling your supervisor that it should only restart the failing child and leave the siblings alone. You'll also say that, in the event that it restarts more than 10 times, the issue should be escalated to the supervisor's parent. Finally, you only want to restart in the event that the child actor throws an `ArithmeticException`; if the actor throws an `ArgumentException`, then the supervisor should simply ignore the message and continue as though it never received the message. The following code example shows how to define this `SupervisorStrategy`. You can see the use of the `OneForOneStrategy` as the basis from which you build up the rest of the components. The `Decider` is created simply from a function

passed into `Decider.From`. This function simply requires you to return a `Directive` based on an `Exception` that is passed to the function:

```
new OneForOneStrategy(10,
                      TimeSpan.FromMinutes(1.0),
                      Decider.From(exception =>
    {
        if (exception is ArithmeticException)
            return Directive.Restart;
        else if (exception is NullReferenceException)
            return Directive.Resume;
        else return Directive.Resume;
    }));
```

Now that you've created a supervisor strategy, you need to associate it with an actor. In order to achieve this, there are two key approaches you can use: you can associate the strategy with the actor directly itself, ensuring that it's responsible for its own supervision settings; or it can be left to be configured at the point when the actor is deployed, leaving the deployer responsible for configuring how its child should act in the face of failure.

The first case is by far the simplest, requiring you to simply override the `SupervisorStrategy` method on an actor. This method is then called in the event that an exception is thrown during the processing of a message. In the following code example, you take the `SupervisorStrategy` that you created before and leave it to the actor to be responsible for what it should do in the event of failure:

```
protected override SupervisorStrategy SupervisorStrategy()
{
    return new
        OneForOneStrategy(10,
                        TimeSpan.FromMinutes(1.0),
                        Decider.From(exception =>
        {
            if (exception is ArithmeticException)
                return Directive.Restart;
            else if (exception is NullReferenceException)
                return Directive.Resume;
            else return Directive.Resume;
        }));
}
```

We saw in chapter 5 how to change specific details of an actor's deployment through the use of `Props`, which you pass into the actor system when you want to spawn an actor. You can choose to specify the supervisor strategy on the `Props` so that it's decided by the actor responsible for deploying it. In the following example, you specify the supervisor strategy as part of the `Props`. If you've got a simple actor responsible for communicating with a database, called a `DatabaseCommunicationActor`, you can create the `Props` for it by using the `Create` factory method. Once you've got the basic `Props` responsible for creating the actor, you use the fluent API to create a new `Props` object with the supervisor strategy on it. The supervisor strategy is declared in exactly the same way as in the previous example:

```
Props.Create<AnomalyDetector>()
```

```
    .WithSupervisorStrategy
    (new OneForOneStrategy(
       10,
       TimeSpan.FromMinutes(1.0),
       Decider.From(exception =>
       {
           if (exception is ArithmeticException)
               return Directive.Restart;
           else if (exception is NullReferenceException)
               return Directive.Resume;
           else return Directive.Resume;
       })));
```

Both of these approaches have advantages, and the decision on when to use them is ultimately influenced by the overall usage of the actor. Some actors may be designed to be more generalized and able to be used within multiple different components within the same system. In this case, it's likely that it would impose differing constraints on how it should react in the event of failure, depending on its role in that key area of the system. As such, it's likely that an actor would not want to be responsible for specifying how it should react to failure, instead pushing that responsibility onto the component that wants to spawn the actor into it's context. But, in the event that you've written an actor for a very specific purpose, which relies upon restarting in a specific way, then you're more likely to associate the supervisor strategy with that specific actor. This then ensures that all of the actors of that type that are spawned will be spawned with that same supervisor strategy, leaving it suited to more homogeneous collections of actors.

6.2.3 Failing fast

As a programmer, you're likely to interact with a number of APIs on a daily basis, many of which are likely to perform some potentially unsafe operations in the course of their lifecycle. There is a large number of potential errors, and we'll see some of these throughout this chapter. In every case where these result in an unrecoverable or unexpected error, every API in high-level languages, such as C#, will typically follow the same approach and throw an exception.

This usage of exceptions is designed so that the caller of the API has its control flow diverted and stops it from the intended order of execution. In order to try and regain control of the situation, the approach taken is to simply wrap the calls to the API in a try-catch block. Following this, any exceptions thrown are handled by a specific exception handler for the type of exception thrown. Then, within the exception handler, you'll perform operations that will usually try and recover from the exception by either retrying the operation once again or simply logging some details of the exception. Then, it's likely that the exception will get wrapped within an enclosing type so that you provide potentially more meaningful information to the user. Finally, you either throw this new exception or rethrow the original exception.

This approach has, however, a number of disadvantages associated with it. The first of which is the potential obfuscation of the original intent of the business logic expressed within the code. By having to write error handling code, you end up needing to surround your

originally cleanly factored code into code that provides a number of functions that will only be executed in exceptional circumstances rather than allowing developers to focus on the core intent of the code. Ultimately, by adding the error handling code, you may end up creating a codebase that is more confusing for developers to understand because it imposes an understanding on the implicit details of API methods and where they can fail. This is information that is typically not made immediately obvious within documentation.

Another source of pain when using this approach is the potential for any logs to quickly become polluted with multiple messages reporting the same source of errors but at different points in the error handling hierarchy. For example, a log message may be generated at the most deeply nested point where the error is generated, as well as the place where this rethrown exception is then caught. The log will then contain two messages with different information relating to the same error.

Also, in the event that an exception only gets logged at the top of the error handling chain, there's the potential for lost context. If the original exception is generated but not included in any more generalized API exceptions, this presents the problem that it becomes significantly more difficult to drill down to the original source of the error, instead only knowing which API method led to the failure.

The final issue is potentially the most significant issue. If you perform a number of method calls, one following the other, and following each method call you modify state, in the event that the final method call throws an exception, this will prevent it from setting its final state. This ultimately leaves the system in an indeterminate state, where the application has left one state but has not entered the new state. This causes difficulties further down the line. If you now call that API again, there's no guarantee that it won't provide invalid results every time it's called.

We've seen how Akka.NET provides supervisors that prevent the whole application from being brought down in the event that a specific exception is not caught by an exception handler somewhere. As a result, Akka.NET encourages the decision to not use any behavior blocks at all, instead relying on the use of supervisors to deal with errors. This approach is more commonly known as fail fast programming, because you look to inform the system of failure as soon as possible after the error occurs. Then, once there's a failure, you let the supervisor deal with the specifics of what to do to try and recover from the failure. This ensures that, once the supervisor receives the error, it is the only thing that logs the error right from the place where it originally occurred, as well as the specific reason for the error, whether it was the inability to connect to an external database or an attempt to divide by zero. Then, assuming the error can lead to an invalid state, the actor can be restarted to a known good state in an effort to prevent knock-on effects.

This approach to programming has seen significant benefits historically when designing applications built upon actor systems; but it does involve a significant change in the fundamental concepts of how to write code that can lead to errors.

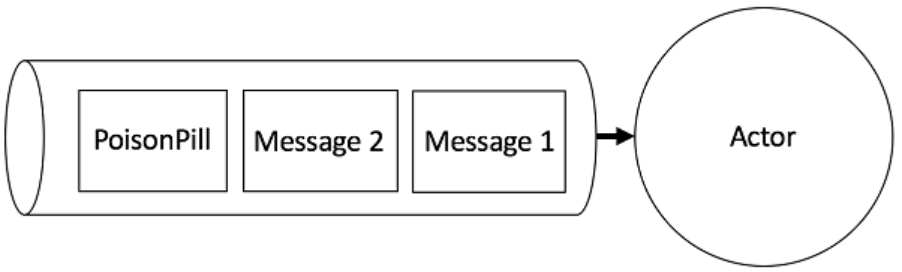
THE ERROR KERNEL PATTERN

This approach to simply restarting actors as soon as an error is encountered can lead to potential issues, in particular relating to actors that are typically required to store lots of state. Because Akka.NET actors exist wholly in memory, if the actor gets restarted, then it loses all of it's associated state. This causes problems because you rely on restarting actors so frequently within Akka.NET. The common approach to dealing with this problem is to use the error kernel pattern. The error kernel pattern forces dangerous work down the actor hierarchy to child actors. Then, in the event that the child fails, the error is isolated, ensuring that the parent doesn't end up losing it's more long-term state.

6.2.4 The actor lifecycle

We saw in chapter 3 how to spawn a new actor into the actor system that internally creates an actor instance. We've also seen in this chapter how actors are also able to be shut down and then started up again whenever a failure is detected. Actors can also perform other operations throughout their lifetime, including shutting down. In fact, there are two different ways of shutting down an actor depending upon the severity of the situation:

- *Passing a poison pill message*—Akka.NET internally provides a number of messages that can be sent to actors and processed before reaching the internals of your actor. One example of this is the poison pill message that is identified by the `PoisonPill` class. With this, the actor will process any messages in it's mailbox until it reaches the message. Once the message is reached however, it will shut down the actor before it processes any messages that arrived in the queue after the poison pill. By using the `PoisonPill` you can send messages to a single actor or a group of actors as you would send any other messages.
- *Using* `Context.Stop`—If an actor needs to stop immediately after processing the current message, then it can achieve this by using the `Stop` method on its internal actor context. Here, it passes in the `ActorRef` relating to what it wants to shut down, and then the framework will handle the shutdown of the actor.



THE LIFECYCLE

We've so far seen how an actor can be created by calling `ActorOf` within the framework, which will cause your actor to be instantiated. We've also seen how an actor can be killed

either programmatically by itself or other actors, or by the framework once it detects a failure. Actors in Akka.NET operate and transition through a number of key states during their lifecycle. You can see in the following diagram the states an actor can exist in, as well as how to move between those states:

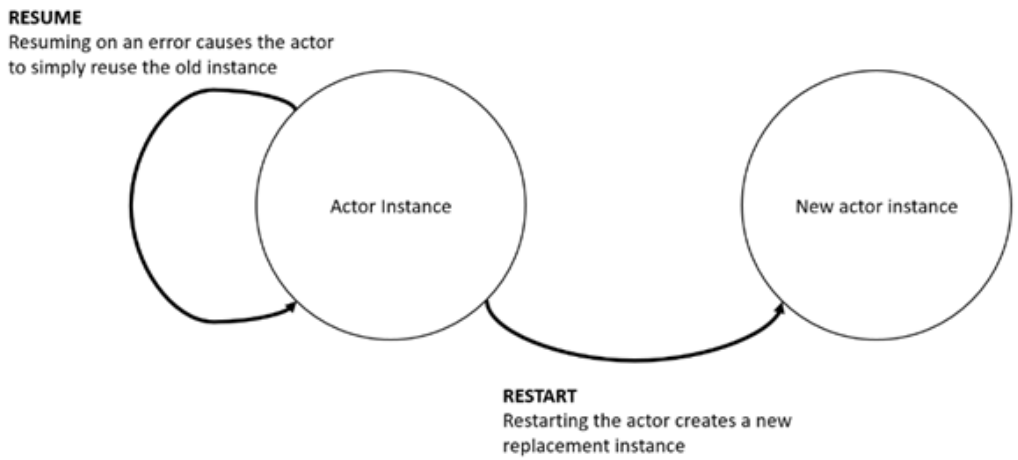


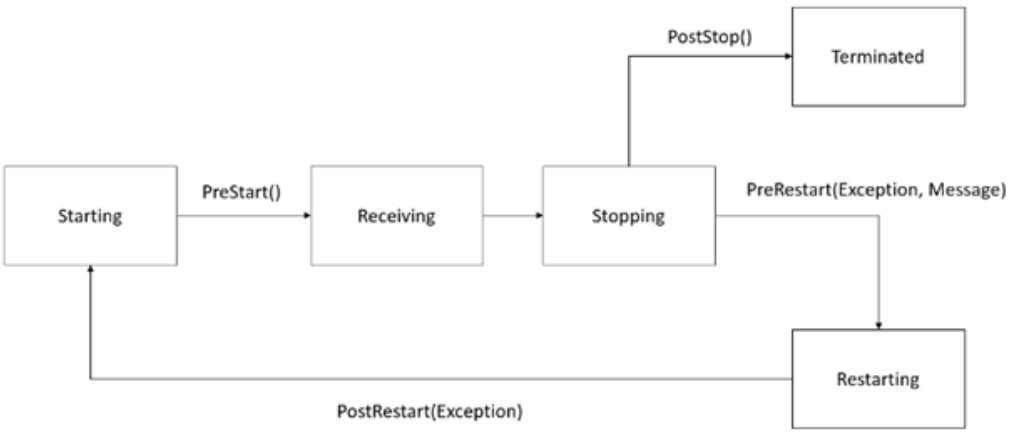
Figure 6.7 The two most common options for what to do in the event of an error will cause the actor system to reuse the old actor instance or create a new one in its place as a replacement.

The core state is the actor's regular operating state, but there are also a number of steps it must take along the way to get into such a state. Upon an actor being spawned into the actor system, it starts in the starting state. Its at this point that any configuration the actor needs during it's lifecycle is prepared and ensures it is ready to start receiving messages. From here, it moves into the running state. It's here that the processing loop is invoked any time a message is received in the mailbox of the actor. Eventually, the actor is likely to be terminated, either gracefully by choice through a decision to stop, or forcefully by a supervisor. Following either of these cases, the actor then transitions into the terminating state before ending up in the terminated state. When the actor is in the terminated state, any messages addressed to it won't be delivered and instead are passed on to the `DeadLetters` actor that logs these events.

SEEING THE TRANSITIONS AN ACTOR TAKES

When you create a class for an actor, you're able to specify operations that should be undertaken at certain points during the actor's lifecycle. For example, it can be useful to perform actions before an actor starts up, such as sending a message to other actors in the actor system. This can be achieved by overriding certain methods on the base actor class that then get invoked by the framework at the stages in the lifecycle. In the following diagram, you

can see the state transitions which you see throughout an actor's lifecycle, starting with the starting phase and moving onto the receiving stage of it's lifecycle:



In this chapter, we've seen how the framework is able to automatically restart an actor once the framework decides that it has failed. It can be beneficial to know whether an actor has been restarted, as well as the reason for that. This can be achieved by overriding the `PreRestart` method on the actor. The `PreRestart` method then provides an Exception that caused the actor to restart, as well as the message that lead to the exception. This then provides a number of key advantages. Most notably, it presents the opportunity to reschedule the failing message so that the actor can have another attempt at processing the message. We have seen in this chapter that the common approach to dealing with failures is to simply restart the actor that is currently failing because there's a chance that a known good state could allow it to process the message successfully. By supplying the message that led to the failure, you can send the message to yourself and attempt to process it in its new state. This is what you do in the following example, in which you receive the message that caused the crash and schedule it to be added to the mailbox, ready to be processed again:

```
protected override void PreRestart(Exception reason, object message)
{
    Self.Tell(message);
}
```

When you're using an actor that does a periodic job, you're likely to use the scheduling capabilities provided by Akka.NET to deliver a message at periodic intervals to invoke certain behavior. For example, you may want to perform synchronisation between the actor's internal state and an external service once every second. To achieve this, you'll want to send a message to yourself once every second so that you can subscribe some logic to this message, such that any state within the actor is accessed safely with the same concurrency guarantees as other messages. You want to ensure that this message gets scheduled once the actor starts

up. To manage this, you can use the `PreStart` method of the actor. This gets called following the actor's construction, before it starts receiving messages. It gets called every time the actor starts, whether it is restarting or a fresh start. It's a common pattern to initialize any resources in here that are likely to last the lifetime of the actor. One example of this is your timer, which you need for your synchronisation. The following example shows an actor, like the one described previously, that is responsible for performing an operation on a recurring schedule. You create a scheduled message in the `PreStart` method and, following this, it will start to receive that message on a regular schedule:

```
ICancelable _synchronisationTick;

protected override void PreStart()
{
    var scheduler = Context.System.Scheduler;
    scheduler.ScheduleTellRepeatedlyCancelable(
        0, 500, Self, SynchronisationTick.Instance, Self);
}
```

After having registered your continuously scheduled message, you'll want to stop it when your actor stops in order to ensure the scheduler doesn't try to keep sending the message to it. The `PostStop` override is available for this purpose of disposing any resources that are no longer needed. The last example saw the actor create a recurring task and store a handle to it in the actor's state. Now that the actor has finished its work, it's able to dispose of the scheduler sending it a message. The following example shows how, by using the `PostStop` method, you can safely access the state of the actor and modify it, in this case by disposing of the scheduler. It's not just disposing of unnecessary data that you can perform in the `PostStop` method; you can just as easily send a message to other actors to notify them that this actor is currently terminating:

```
protected override void PostStop()
{
    _synchronisationTick.Cancel();
}
```

These are the core functionalities of the actor lifecycle within Akka.NET. As well as seeing how, by simply overriding the methods in the actor definition, you can easily gain insight into the operation of the actor; we've also seen how you can safely access the internal state of the actor to ensure it is able to initialize and dispose of the resources used throughout its lifecycle.

6.2.5 Watching for the deaths of other actors

Within Akka.NET, it is not just the parent of an actor that can watch for the failures of an actor, through a concept known as DeathWatch, other actors are able to monitor an actor to be notified if it should happen to fail. Using DeathWatch, when the framework discovers that an actor has failed, it sends a message through to all the actors who have subscribed to the notifications.

Actors are able to sign up to DeathWatch notifications through their own context within the actor instance. We saw earlier, in chapters 3 and 4, how use the `Context` within the actor to perform certain core operations; DeathWatch is no different. By using the `Watch` method, you're able to supply an `ActorRef` to use, which will be monitored, as will the `ActorRef` of the subscriber. Following this, if the watched actor fails, then the subscriber will receive a Terminated message in its mailbox for it to process, containing the address of the actor that was terminated.

DEATHWATCH AND THE REAPER PATTERN

Within Akka.NET, one of the common operations you're likely to perform is to wait until a number of core actors within the actor system have shut down before shutting down the entire actor system. It's important to understand what we mean when we say that an actor is done. The simplest answer is that its mailbox is empty, but this poses a number of problems. Notably, you might think that an actor is finished because its mailbox is empty, but it might be in either of two other situations:

- *Still processing the last message*—The actor might have emptied its mailbox, but if it's doing some particularly intensive computation as a result of this final actor, it may not be safe to shut down the actor system just because the message queue of this actor is empty.
- *Not received all of its messages yet*—In a similar vein to the previous state, if an actor depends on receiving a message from another actor before it in the chain, then it may not be safe to shut down the actor just because its message queue is empty, because it may not have even processed one message yet.

Earlier in the chapter, we saw the various ways in which you can shut down an actor, one of which was through the use of the `PoisonPill` message, which causes the actor to shut down once it is processed. This presents you with the potential to send an actor a number of messages it should process and then a `PoisonPill` message after all of that work. You can then say with a degree of certainty that once an actor has died, it has managed to complete all of the work it was sent to process. The diagram below shows an example message queue of all the work you need an actor to do and then shutting it down straight after it has completed that work.

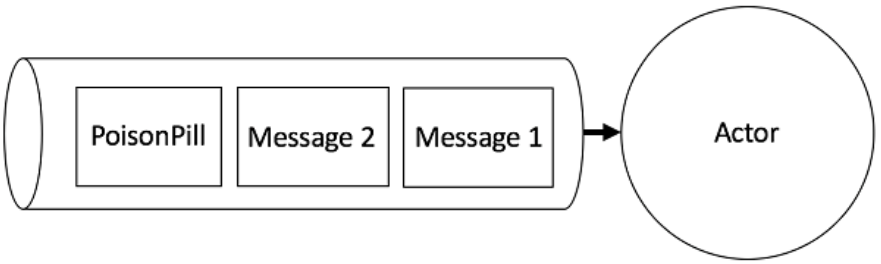


Figure 6.8 The PoisonPill is a means of ensuring all messages in the queue are processed before an actor is shut down.

But it's likely that you have more than one actor that sits at the core of the actor system and you can only safely exit once all of these actors have finished. Ultimately, the aim is to form a barrier that you can only pass once all of these core actors have finished their work. You can build this up from the approach you've seen so far, where the actors do some assigned work and then shut down once that work is complete.

In order to manage this, you can use the DeathWatch concept outlined earlier. You can therefore create an actor whose job it is to DeathWatch each of the core actors that you're interested in. Once this actor gets a message informing it of the death of all of those actors, then it's safe to pass the barrier because they've all completed their work. You can see the core idea of the reaper pattern in the following diagram. You have the regular hierarchy of actors built up below the guardian actor; but you also have your reaper actor, waiting for the deaths of the other actors:

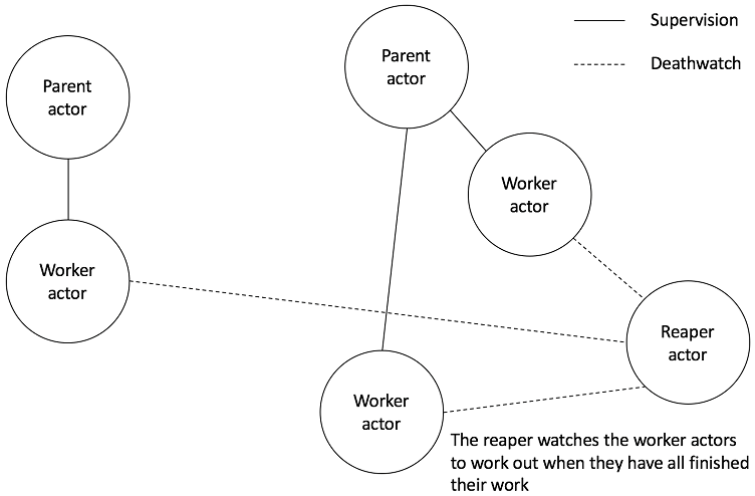


Figure 6.9 The reaper pattern is an application of DeathWatch that allows you to wait until all actors have finished working.

You can now build up an actor capable of safely watching these actors and then safely executing operations once all the work is complete. Your actor will work by taking a collection of actor paths that it will watch for deaths. It then subscribes to DeathWatch notifications for all of these actors, before listening for the terminated events. Whenever it receives a Terminated message, it removes that actor from the collection of watched actors and checks to see whether it needs to wait for any other actors. If it doesn't, then it exits the barrier.

The first thing to do, then, is to create an initial reaper actor that is responsible for watching all of the core actors. The following code snippet shows the initial actor definition with the constructor. The only data you need for this actor is the list of actors you'll be watching, which you supply as a set. Then, in the constructor, you also subscribe to the DeathWatch notifications for each of the actors you're interested in. As shown in the example, this is done simply by calling `Context`. Watch with the `IActorRef` for the actor you're interested in watching:

```
public class ReaperActor : ReceiveActor
{
    readonly HashSet<IActorRef> _watchedActors;

    public ReaperActor(HashSet<IActorRef> watchedActors)
    {
        _watchedActors = watchedActors;
    }

    protected override void PreStart()
    {
        foreach(var actor in _watchedActors)
        {
            Context.Watch(actor);
        }
    }
}
```

You now need to add the correct handler for the Terminated message. Upon receiving a Terminated message, you need to remove the actor references from the set of references you're watching. If that set is now empty, then you can safely say that the all of the work has been successfully completed. Because all of the work has now been completed, you can safely stop the whole actor system without worrying about whether there is any work still going on:

```
Receive<Terminated>(terminated =>
{
    _watchedActors.Remove(terminated.ActorRef);
    if (_watchedActors.Count == 0)
    {
        Context.System.Terminate();
    }
});
```

This simple pattern shows how you can use the built-in lifecycle monitoring tools as a means of watching for the completion of actors. The reaper pattern allows you to safely perform actions once all of the work has been completed. The key benefit of using

DeathWatch, as opposed to other potential techniques, is the built-in consideration for failure. If you were to design such a pattern using simple message passing instead, you'd have to factor in the potential for the actor to fail before ever sending a completed message, meaning you'd need to add timeouts to ensure your barrier actor didn't get stuck in a state where it was waiting for a message it was never going to receive.

6.2.6 Interface-level failures

So far, we've seen what happens in the event that your actor encounters an error owing to it transforming into a faulted state; but there's another error state you need to consider, which is caused by a user providing an invalid input to your actor.

A common pattern typically seen when developing APIs is something resembling the following. A method or function takes a number of arguments; the API then progressively checks each parameter to the method to ensure it is deemed as being valid. Typically, this might include checking to see whether a parameter is null or matches a validation scheme. In the event that a parameter is not valid, then an exception is thrown, such as an `ArgumentNullException` or just an `ArgumentException`.

```
public void RegisterUser(string email, string password)
{
    if (email == null)
        throw new ArgumentNullException("email");
    if (password.Length < 8)
        throw new ArgumentException(
            "Provided password is too short", "password");
}
```

But this approach falls apart when you consider the message passing approach of Akka.NET. In this case, the interface is called by simply passing a message rather than directly invoking it. If you then request data from the actor, you could still be passing invalid data. As such, you need to change the way you consider these errors.

Earlier in this chapter, we already saw what happens when an application error occurs, in that the supervisor is contacted with a notification of the failure, which is then responsible for deciding on the appropriate course of action. Ultimately, the sender of the message doesn't know whether or not an exception was thrown, leaving you in a situation where you aren't sure whether you succeeded in processing the message.

This presents problems with knowing whether the information that you sent to the target actor was valid for the API. We saw in chapter 3 that when you send a message to an actor, you can either send it in the form of a fire and forget manner through the use of `Tell`; but it's also possible to use `Ask`. `Ask` allows you to asynchronously send a message to an actor and await a response. You can use this communication channel to surface any validation errors to the party that can deal with them in the most appropriate way.

Akka.NET provides the required classes to be able to start using this pattern through the use of the `Success` and `Failure` classes. These two classes are used to encompass all of the possible outcomes of calling an API through a message-based protocol. Your actor can either

return a Success message with a result that indicates that the sent data is valid and everything has executed successfully, or the actor can return a Failure message that can contain any exceptions providing more details of the validation failure.

Let's take an example of how you can design actors that are able to respond with the details of any user specific errors you may encounter. We mentioned earlier in this section a potential use case for such a design, notably an API used for registering a user with an email address and a password. The actor will first validate that the information the user has provided passess all the checks you have in place. In this case, you'll ensure that the email address provided by the user is valid and the password matches some basic rules.

Your actor receives a UserRegistration message containing the desired username and password. Whenever it receives such a message, the actor will perform some simple validation and in the event that the email isn't in the correct format or the password doesn't match your rules, then it will reply with a failure message. Otherwise, it will continue with the execution and process the request by storing the message in a database of users. After the execution completes, it returns a Success message informing the caller that their response was successful:

```
Receive<UserRegistrationInformation>(registration =>
{
    if (ValidInput(registration))
    {
        var accountInfo = RegisterAccount(registration);
        Sender.Tell(new Status.Success(accountInfo));
    }
});
```

A client is then able to use this actor to understand whether any information passed to it was correct and in the expected format by asking the actor for a response. Upon receiving the response, you're then able to see whether it was successful or look deeper into the cause of the error the actor saw:

```
var response =
    await userRegistration.Ask(
        new UserRegistrationInformation(
            "newuser@google.com",
            "P4ssw@rd"));

if(response is Status.Success)
{
    //Handle successful account creation
}
else if(response is Status.Failure)
{
    //Handle invalid input case
}
```

By using this approach to error handling combined with that of the previous section, it implicitly creates two error channels, which presents you with further benefits. If an error is encountered due to an invalid state within the actor or by failing to communicate with other

services, then these are errors that the user is unable to deal with and that will ultimately degrade the user experience when using your service or application. But if the user provides an invalid email address, then this is something that you can use to stop this.

6.2.7 Application-level failures summary

In this section, we've seen how Akka.NET can help you reduce the impact that an application-level failure is likely to have on the subsequent lifecycle of the application by isolating errors and encapsulating them within the smallest unit of work. By having all of this potentially error causing logic hidden deep within one actor, it then allows you the freedom to simply fail if something goes wrong, because you have an actor responsible for watching the actor and restarting it to a known working state. We've also seen how you can deal with errors caused by other users using the API in a way other than was intended, such as what happens when a user supplies an address in an invalid format.

6.3 Understanding transport-level failures

So far in this chapter considering failures, we've seen the large number of failures that can occur in the code that you write; but there's more to actor systems than just the code. You also need to consider how your systems handle failures induced by the distributed environments in which they are running. The most pertinent failure you need to consider is whether there's any guarantee that two actors are able to communicate with each other.

Let's consider the case of what happens when you send a letter through the postal system. You wrap your message in an envelope that informs the delivery mechanism of the target, and you leave it in a known location for the delivery system to pick up. From there, you have no knowledge of whether or not the target received whatever you sent. This is one of the key disadvantages of asynchronous systems such as Akka.NET. Due to the large amount of infrastructure in place between the point from where a message is sent and the point where a message is received, there's the possibility that the message might not reach its intended destination safely and may end up getting lost in the postal system.

These sorts of issues are equally common in software development as in the physical message delivery world, so it's important that you write systems that are able to cope with such failures in appropriate ways.

In the vast majority of cases, you're unlikely to lose a message, especially when we consider the scenarios that we've seen so far, which have all featured actors running across multiple threads on a single machine. When we looked at how you can design a reactive application in chapter 2, though, we considered that there might be a message sent between a sensor and the system that fails to be delivered. In this case, you end up having to use a potentially low-quality network connection through which you send messages. This vastly increases the probability of message loss due to the additional levels of complexity that you see through all stages of the pipeline.

For the case of most sensor data, such as light or humidity, it's unlikely to matter if a message is lost, especially if the volume of data is sufficiently high that you can afford to lose

one out of every one hundred data points. But certain sensors could prove to be more important. For example, if a motion detector picks up movement within a house, then this could suggest that an intruder is in the house, meaning an immediate reaction would be wanted. In this case, you want to ensure that your message gets delivered to an actor that is capable of dealing with such an event.

When we look at the systems you have built so far, you've relied on Akka.NET to guarantee that a message gets delivered for us; but such guarantees are non-existent in Akka.NET as it simply follows the at most once delivery guarantee. This essentially says that a message will get sent and passed through the system. If it gets lost at any stage in the pipeline, then the sender won't receive any feedback about the failure. Ultimately, you hope to see that each message is only processed once, but this poses difficulties for you.

Throughout this chapter, we've seen the vast array of possible failures that an actor can face during its lifecycle. We've also seen some of the potential failures in a more generalized asynchronous system, such as a phone call. In all cases, we've seen that the key problem is that you don't know what state the remote actor is in. It could be in a faulted state, in which it's incapable of processing messages; the communication link between the two actors might have failed, or it might have only failed in one direction. These problems all combined mean you are unable to determine the state of remote actors. This then presents you with the difficulty that you can never see whether an actor has received a message, whether it successfully processed the message, or even whether it failed to send a successful acknowledgement in response.

To get around this, instead of using the at most once delivery strategy, you can switch to using the at least once delivery strategy. Using this technique, you'll send the same message to an actor multiple times, until the sender eventually receives a message informing it that the message has been successfully processed and it should now stop sending the message. This ensures that the target will eventually receive the message, although it might end up receiving several copies of the same message. By sending the message a number of times, you can counteract the issue, with any transient errors across the communication link being resolved, if in only briefly, to allow at least one of the messages to reach its intended destination.

6.3.1 Writing applications that handle message loss

The actor systems that you have written so far have all dealt with any failures that might be induced as a result of the code you write; but a modern application ultimately sits on top of several other layers of infrastructure, including the likes of the CPU internals and the operating system, and is potentially dependent upon the role of the application, a network connecting multiple machines. So far, we've seen the problems that can arise from having all this supporting infrastructure, which is also capable of inducing failures. We've also seen how Akka.NET handles the sending of messages and how you simply send a message without any idea of the likelihood of failure.

Although the underlying delivery guarantees of Akka.NET can't be changed, given an at most once delivery guarantee, you can turn this into an at least once delivery guarantee. To

create systems that allow you to reliably send messages, you can create a system that guarantees the delivery of messages. In this section, we'll look at how you can build such a tool, allowing you to communicate freely without the need for worry about the underlying communication layer.

In order to create such a delivery system, you'll need an API capable of repatedly trying to send a message until you eventually send it an acknowledgement informing it that it has completed. Throughout the book, we've seen many instances where the work can be expressed simply through an actor, and this scenario is no different. In order to repeatedly attempt to deliver your message to a target, you can create an actor that is responsible for sending the message and attempting to receive an acknowledgement message. In the event that it doesn't receive a response within a fixed timeout, then it will automatically resend the message, once again repeating the process if it doesn't receive an acknowledgement in time, until it does eventually receive an acknowledgement or it attempts to send a configured maximum number of messages without acknowledgement.

As you have done with many of the actors you have sent so far, you will first consider the states that your message delivery actor can exist in. The main state within which it will typically exist is the state when you send a message and await a response. In this state, it's capable of accepting one of two messages: either one from the intended recipient notifying you that it has successfully received the message, or a timeout message informing it that it should send another message. If it does receive an acknowledgement message, then it should shut down because it has nothing else to do. If, however, it fails to receive any acknowledgements across several timeouts, then it will respond to the original sender with a message informing it that it has failed to send the message to the intended target. You can see these states and associated transitions in the following diagram:

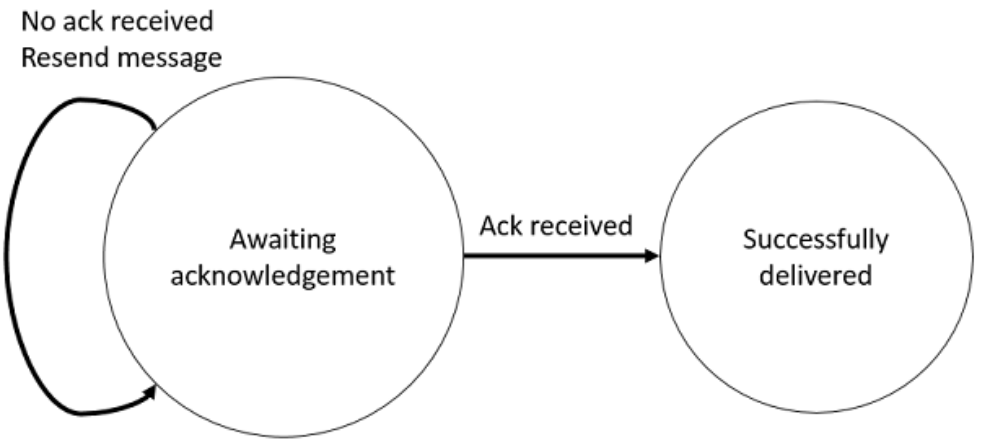


Figure 6.10 The state machine for an actor providing at least once delivery semantics is very simple, requiring only two states with two possible events.

The first thing to consider is the messages that you'll be sending between the actors. As you can see in the diagram, there is a number of events that cause state transitions, notably the `Acknowledgement` and `ReceiveTimeout` events. These two messages allow you to see most of the state transitions, as we saw in the last section, where we discussed interface-level failures. We also have `Success` and `Failure` messages provided by the framework, allowing your guaranteed delivery actor to inform the original sender of whether it did manage to successfully deliver the message.

Due to the potential for failures in asynchronous systems, as we've seen so far in this chapter, Akka.NET provides an API as part of the actor that allows you to specify that, after a certain period of inactivity, a `ReceiveTimeout` message is automatically sent to an actor. This then provides a means of developing actors that are able to respond to situations where the actor needs to be aware of how long it has been since it last received a message.

You now just need a message that allows you to tell the repeated sender that the target has successfully received the message so it should stop processing messages. In order to do this, you can define a simple class that is just used to inform the target that it has successfully received a message. In the following code snippet, you define such a class, simply called `Ack`, which doesn't need to hold any additional data associated with the received message:

```
public class Ack
{
    private static readonly Ack _instance = new Ack();

    public static Ack Instance { get { return _instance; } }
}
```

You've now got all of the messages you use as state transitions in your state machine. From here, you need to implement each of the states. As is the case with the actor model, you'll define an actor that is capable of doing this resending work, allowing the original sender to be freed up to process other work whilst you attempt to communicate with a target. When you create the actor, you'll need to take in the destination of the message, the message you want to send, the maximum number of retries you should attempt, and the timeout between retries. Because your actor will simply be single-purpose designed to send one message and then stop, you'll just rely on the use of the constructor to pass these arguments in.

In the following code example, you can see the initial actor definition with the constructor. The first thing you might notice is the decision to take in a simple actor selection for the target you'll be sending the message to, rather than a direct reference to an actor in the form of an `IActorRef`. We saw earlier, in chapter 3, the difference between an `IActorRef` and an `ActorSelection` in that the `IActorRef` also contains information about the incarnation of an actor, so if it restarts then this reference will change. In this scenario, you want to guarantee that you can send the message to the target regardless of any failures that occur, such as the network failures you're accounting for here. But by using an `IActorRef`, you wouldn't be accounting for the possibility that the given actor incarnation fails and is restarted. In the constructor, you'll also notice the use of the `SetReceiveTimeout` method, which simply tells the

framework that the actor should receive a `ReceiveTimeout` message after the period specified in the timeout:

```
public class GuaranteedDeliveryActor : ReceiveActor
{
    readonly ActorSelection _target;
    readonly object _message;
    readonly int _maxRetries;
    int _retryCount;
    readonly TimeSpan _messageTimeout;

    public GuaranteedDeliveryActor(ActorSelection target,
                                   object message,
                                   int maxRetries,
                                   TimeSpan messageTimeout)
    {
        _target = target;
        _message = message;
        _maxRetries = maxRetries;
        _messageTimeout = messageTimeout;
    }

    protected override void PreStart()
    {
        SetReceiveTimeout(_messageTimeout);
        _target.Tell(_message);
    }
}
```

Because this state machine ultimately only has only one core state, you don't need to use any of the finite state machine features; you can simply use a basic actor instead. You saw in the state transition diagram that there are two core events that you need to handle: you either receive a ReceiveTimeout message or an Ack message. If you receive a ReceiveTimeout message, then you need to do two things. First, you need to check whether you have reached the maximum number of retries for sending the message. If you have, then you need to notify the original sender that you have failed to send the message, cancel the message, receive timeout messages, and shut the actor down. If, however, you do have retries remaining, then you can attempt to send the message to the target again and increment the retries counter. This means you have a receive handler for the timeout messages, which looks like the following code example:

```
Receive<ReceiveTimeout>(_ =>
{
    if (_retryCount >= _maxRetries)
        throw new TimeoutException(
            "Unable to deliver the message to the target within the specified number of
       retries");
    else
    {
        _target.Tell(_message);
        _retryCount++;
    }
});
```

You also need to deal with the case where you receive an Ack message in response from the target. In this case, you need to inform the sender that you have successfully sent the requested message, then you need to cancel the ReceiveTimeout messages, before finally shutting down the actor. Once again, this leads to a fairly simple receive handler, which you can see in the following code example:

```
Receive<Ack>(_ =>
{
    SetReceiveTimeout(null);
    Context.Stop(Self);
});
```

Finally, having completed such a system, you need to deal with the guaranteed delivery at the receiving end. Ultimately, you simply need to tell the actor that it has managed to contact the required target. To do this, all you need to do is send the Ack message to the Sender of the message it received.

```
public class BillingActor : ReceiveActor
{
    public BillingActor()
    {
        Receive<RequestNewPayment>(payment =>
            {
                Sender.Tell(Ack.Instance);
            });
    }
}
```

It's important to consider the operations the actor undertakes upon receiving a message that is sent using your actor. In this case, there's the possibility that your target could end up processing the same message multiple times over multiple timeouts if it doesn't send its acknowledgment back within the timeout period. To get around this problem, you need to either filter out messages that the target has already processed, possibly by passing an identifier to uniquely identify a message and then storing a set of processed messages within the actor. Alternatively, you could choose to design your target actor so that receiving the same message twice will lead to the same outcome. This property is known as idempotence.

You have now seen how to define an actor, which will provide you with a means of a best effort at least once delivery guarantee and will repeatedly attempt to send a message to a given target.

6.3.2 Transport-level failure summary

This section has focused on how to build applications that stay resilient and reliable even in the case of failures outside of your control, in this case, in the underlying infrastructure that connects actors together. We've seen how Akka.NET typically deals with sending messages and the limitations that this brings. We've also seen how you can attempt to solve these problems in order to ensure that you can successfully transmit messages between actors. This is a topic we'll come back to in later chapters as you look to build even more reliable systems,

and we'll see an implementation available within Akka.Persistence, one of the many Akka.NET plugins, which provides additional features to the framework.

6.4 Case Study: supervision – failure – chat bots

Modern applications have become increasingly complex and dependent upon a variety of other systems in order to successfully operate. Even a relatively simple application will need to interoperate with a database and, potentially, external APIs. More and more APIs are being developed to help open up more complex domains and techniques to a broader audience in an effort to simplify the development of more enhanced projects.

One such example is the recent proliferation of conversational user interfaces in the form of chat bots. Due to the conversational nature of these interfaces, they require an understanding of the way natural human language works. Natural language processing is a fundamentally difficult problem to solve due to the complexities of human language. On top of the difficulties of processing the text itself, you need to understand the intent of the text and understand what the user wants the system to do.

In order to counter this problem, a number of companies have started providing APIs capable of figuring out the intent of text so that you can compress the user's original text down into a more manageable data set. But it's also possible that these external services could fail at any stage. If they do fail, then you need to continue to work and try and process the user's request. If you don't handle the failure correctly, then you could end up in a situation where the user is presented with internal error messages or, even worse, the user could end up without a response and the appearance that the application is hanging.

By using the supervision components of Akka.NET, you can allow Akka.NET to handle the exceptions you don't expect. In the case of these natural language intent APIs, the common means of communicating with them is by using a HTTP API. When dealing with a HTTP API, there are a variety of issues that you might encounter: there may be network issues that cause timeouts or no responses to be returned, there may be authentication issues if you fail to supply passwords when they are required, or there may be errors generated by the remote system if the text that you supply is invalid. In all of these situations, you still want to tell the user something meaningful in response to their questions, rather than simply leaving them without any further information.

In figure 6.11, you can see how you're able to dedicate work to an individual actor that is then responsible for performing the work and communicating with the external service. In the event that an unexpected error is encountered, Akka.NET will automatically process the faulted actor. In a typical situation, this will involve restarting the actor and hoping that the issue you encountered was just a transient issue. But, given that these APIs are commonly billed based on the number of operations performed against the API, you will shut down the actor calling the API if it encounters an exception caused by invalid text, because you know that this isn't something that you can recover from.

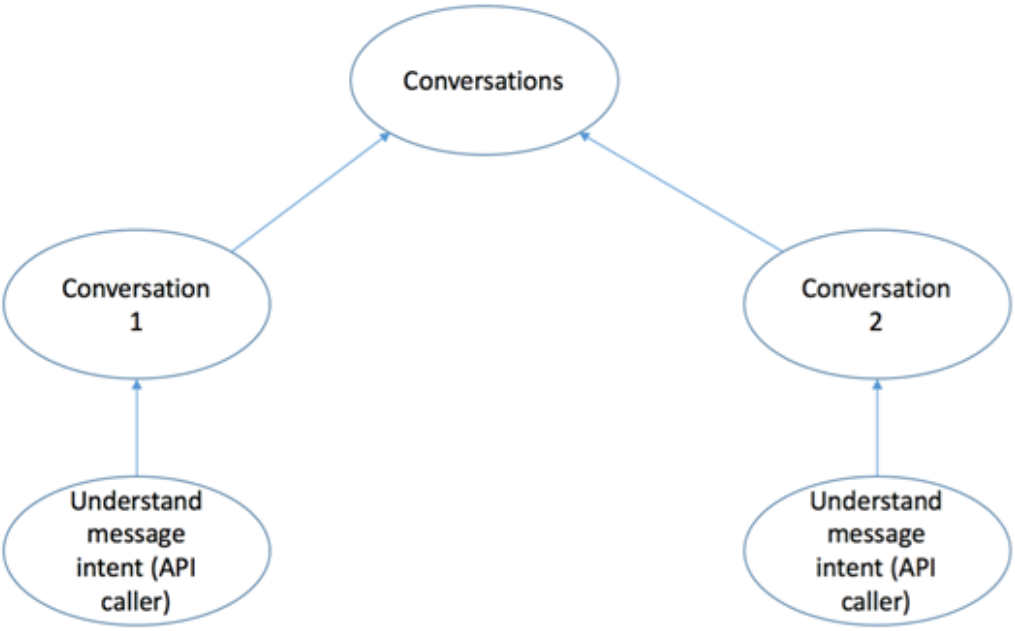


Figure 6.11 Designing a strong actor hierarchy simplifies the process of reacting to failure in such a way that it doesn't affect other parts of the running application.

Supervision ensures that you're able to write systems that aren't overly dependent upon external services, as the handling of failure is a component at the core of the Akka.NET API.

6.5 Summary

In this chapter, you learned:

- How the complexity of modern systems leads to an increased likelihood of failures
- How Akka.NET helps you to reduce the likelihood of system failure by isolating component failures
- How Akka.NET allows you to intelligently recover from errors
- How failures are not just limited to your systems but the systems with which you interact and how you need to consider the

7

Scaling in reactive systems

This chapter covers

- The difficulties involved in traditional scaling approaches
- How the Akka.NET approach to scaling differs from the traditional method
- Using routers within Akka.NET
- Dynamically scaling actors to react to load changes

In chapter 1, when we were looking at what a reactive application means, we saw that, ultimately, you want to build systems that are responsive to ensure that the end users of your application or service have the best experience possible. In chapter 3, we saw how the use of a message passing architecture allows you to break free of systems with a heavy reliance upon blocking API calls. We also saw, in chapter 6, how you can ensure your applications are able to continue to work in gray-sky conditions as well as blue-sky conditions. We saw how, by considering where failures are likely to occur and leveraging the failure detection and recovery tooling within Akka.NET, you're able to actively respond to errors within your application before they significantly impact the performance of the rest of the service and negatively affect the end user's experience. But, for an application to maintain responsiveness, it's important that you have the ability to respond if you see a vast increase in traffic, to prevent that increased pressure affecting the overall performance of the application.

For example, in recent years, due to the increased use of computers and the internet in the home, many traditional retailers have started to offer their products for sale through the internet, and many retailers such as Amazon operate exclusively through the internet. But, for many retailers, the number of customers attempting to access their site is not linear over the course of the year, with many seeing significant spikes around holidays or during large sales, for example, those around Black Friday or post-Christmas sales. In the example of sales, many retailers offer significant discounts on products in an effort to draw in new customers and

encourage them to spend money. If these sales offer significant discounts, then it's highly likely that there will be a significant increase in the number of users trying to access the online store. In this case, the vast majority of users trying to visit a retailer's website likely want to buy something that retailer has available for purchase. If the website struggles under the increased pressure, then it's likely that many buyers will simply stop trying to access it and instead buy the product from an alternative retailer. By providing a degraded experience, there is the possibility that many users will be driven away, leading to a potential loss of earnings for the retailer.

Providing users with a solid experience even when the system is under an intense load is just one potential benefit to creating a scalable system. There are benefits to be found on the micro level as well as the macro level. Many applications will typically have a relatively fixed number of users at any point of the day. For example, an application designed for employees in the UK is likely to see its full usage during the normal working hours of 9A.M. to 5P.M. GMT; but outside of these hours, there might be only a handful of users using the application. By providing a scalable solution, you're able to scale the application down when you see periods of extended quiet and, as a result, have the potential to save both money and resources.

7.1 Scaling up and scaling out

There are a number of markets where it's likely that there will be frequent large spikes in the number of users at any given time. One example of this is in the e-commerce world, where an online retailer may be about to reveal a new product for purchase, start a major sale with signficant discounts, or even just enter a gift-giving period where more people will be buying gifts for others, such as Christmas. In order to handle any of these spikes in users, they need to provision the resources capable of serving an increased number of requests to their website. The simplest and most frequently used approach would be to simply purchase a faster server for their website to run on. If the number of requests is limited by the number of CPU cycles it takes to generate a response, then by using a CPU that can go through more cycles every second, you're able to reduce the time it takes for a response to be generated.

Although this approach might be the simplest, due to its potential for application to existing legacy code bases, there are many disadvantages to it. The key disadvantage is that this approach will eventually hit a limit, as we saw in chapter 1. Throughout the history of the semiconductor, and the CPU in particular, the number of transistors has increased every year in line with Moore's law. But, in recent years, the increase in transistor count hasn't led to an increase in speed. We've encountered the point at which we're no longer making CPUs faster. As a result of this, if you rely on being able to buy a faster CPU in order to scale your applications, you're ultimately going to reach the point where you've bought the fastest processor available, with no room for growth and scaling above that. We typically refer to this approach as scaling up, where you simply make the resources that you currently have available better.

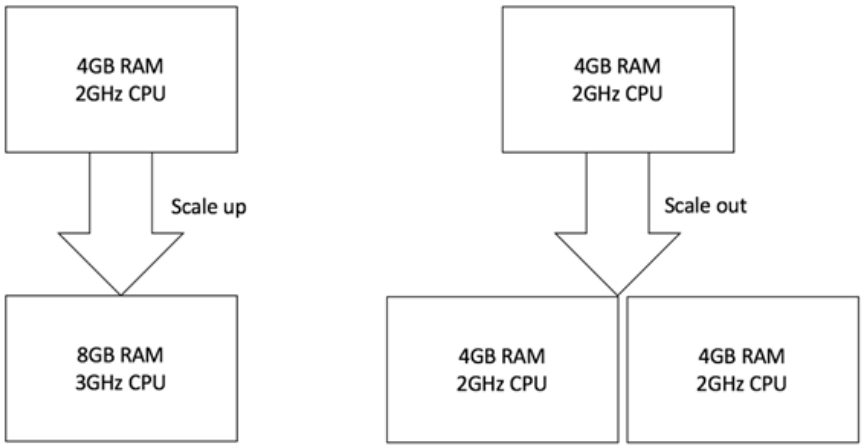


Figure 7.1 - Scaling up means you use bigger machines while scaling out means you use larger numbers of smaller numbers.

You also need to consider how you're able to move your application over to a faster machine in such a way that it doesn't negatively impact the overall availability of the application. If you only have one instance of the application and simply rely on moving to a faster processor every time it needs to be scaled, then you'll incur an amount of downtime. This will be due to the time the application is shut down, as either the existing machine the application is deployed onto is upgraded, or a virtual machine is shut down and restarted. This then presents an even worse experience for the user, as, rather than a slow system that they might be able to use, they're instead unable to access the system at all, leading to more frustration.

This approach is also either very difficult, or downright impossible, to run in an automated manner, depending upon the underlying changes that have to be made to the hardware upon which the application is running. We saw in chapter 1 that the defining characteristic of reactive applications is that they are able to react to the changing environment in which they're running. In this case, the environmental change is the sheer pressure the application is being put under as a result of the increased load. If you want to maintain a truly reactive application, then it should be as autonomous as possible and handle this increased load dynamically of its own accord. The scaling up approach makes this incredibly difficult and ends up needing system administrators to monitor the system and make pre-emptive changes based on guesswork or assumptions.

An alternative approach that can be taken in an effort to counter the problems shown with scaling up is, rather than simply trying to do the work faster in an effort to free up resources for the next batch of work when it arrives, you can simply provide more resources in order to process more work concurrently. This approach relies on you simply scaling out the resources you have available rather than having to modify already provisioned resources. For example,

going back to the online retailer example, if they see an increased load, then they simply make a second server available to serve a request, with clients being directed to whichever server is least busy. Although this won't result in requests being processed any faster than they were previously, twice the amount of load will be able to be handled. This approach, known more commonly as scaling out, provides you with an easier means of scaling resources, such that they're able to handle an increased load.

We saw that the scaling up approach could potentially lead to downtime due to the need to transfer the application over to a new machine. This is a problem that you won't encounter when scaling out because you are simply adding more resources to a pool of workers, so you still have a worker available to service requests while new resources are allocated. Although the service may start to experience delays as the scaled down service copes with the increased load, once the new workers are available, the load is balanced and the response times drop down to the values expected prior to the spike.

When scaling up resources, we saw that it's incredibly difficult to dynamically allocate them as and when they are required. This was due to the lack of automation available when you want to modify the physical hardware upon which an application is running. But if you're in a situation where you simply need to increase the number of services running at any one time, this ends up being significantly easier to automate and more manageable.

Although scaling out does present many benefits, it also poses a number of challenges in how you're able to handle the increased concurrent workflow from a programming perspective. For example, when scaling up, you don't need to worry about two operations modifying a piece of data simultaneously because you're not running two operations together. Fortunately, Akka.NET's underlying actor model, which as we saw in chapter 3 is a tool designed to abstract away many of the difficulties of concurrent programming, helps resolve this core difficulty. By isolating a state to within a single actor, you're able to ensure that there's no contention between multiple instances for a shared resource, which might ultimately end up leading to a bottleneck.

This chapter will focus on how, using Akka.NET, you're able to scale out actor instances effectively, such that it's able to follow the Reactive Manifesto's aim of being responsive even when you encounter significantly higher amounts of traffic than you originally planned for. The key component that will help you in this regard is using routers to distribute the work across multiple actors.

7.2 Distributing work

While scaling up the machines on which you run Akka.NET is a perfectly viable option, it doesn't tend to be the preferred option for the reasons we saw earlier in the chapter. Ultimately, you want to build systems that are able to cope under any conditions they are persented with, and if the status quo changes, then they should react to their environmental changes so that they're able to continue to provide the best experience possible for users. As we've seen so far, the scale out option allows you to manage that. This is especially true in the case of Akka.NET, where you have a very isolated state that is inaccessible by any other actors

in the system. This means that you can safely scale the actors out without the worry of potentially ending up causing concurrency bugs.

If you are to scale out your workload, then you must be able to do as much work concurrently as possible. We've already seen one example of this so far within the book, namely in chapter 2, where we looked at how you could potentially design an e-commerce system that followed the key traits of the Reactive Manifesto. We considered how you could effectively design the system such that as much work could be done simultaneously as possible. The key things we considered were to split work on concurrency boundaries, where we knew it was possible that multiple workloads could run at once. In that example, we saw how to prevent a situation in which you encountered a bottleneck due to the large number of sensors that could exist in a single room. Your solution to that problem was to create an actor per sensor, which theoretically allowed every sensor in every room to run at the same time and process messages.

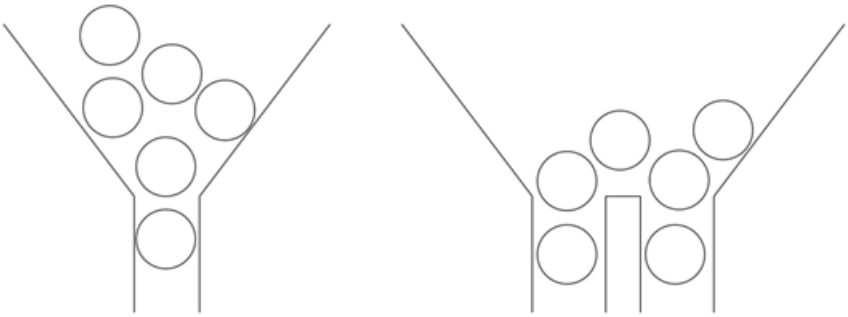


Figure 7.2 – Actors can be thought of as funnels, scaling up is the equivalent of a faster flow rate, whereas scaling out is similar to adding multiple channels to a funnel.

This is the core idea that you need to use within Akka.NET, which is to identify the core responsibility of an action and try and scope down to that level to an extent that makes sense. For example, let's consider the example of services that you might use. Using your smart home idea from chapter 2, you might choose to interact with the homeowner through text messages to alert them to any serious issues that are detected by sensors in their home. In order to send a text message, you're going to need to use an external service that interacts with existing mobile network providers to allow you to send text messages through a web API. Using Akka.NET, you can quickly and easily create an actor that wraps this API and allows you to send a message within Akka.NET representing a text message to send to the user. This actor requires only two parameters in a message whenever it needs to send a text message: the text content and the phone number to which it should be sent. Due to the nature of the content contained within the SMS, namely that it's an alert about something potentially urgent happening in their house, you want them to receive it as soon as possible. You don't want the

request to wait in a queue for an extended period of time, because it could be an urgent message relating to an issue in their house, for example, a fire.

7.2.1 Routers

In order to ensure that you're able to handle work at the same time, you need to be able to perform work in parallel. The easiest way of achieving this is by having multiple instances of an actor able to process messages directed to the same target. We saw earlier in the book that, when choosing where to send messages to, you can use wildcard paths, which will send a message to all of the actors whose paths match the wildcard address. But, frequently, you may want more control over how you process parallel workloads. Let's consider a further example: a tool designed to stress test a website by putting an increased load on it. In order to achieve this, you would typically send as many requests as possible simultaneously. Although you could use the wildcard address-based approach, it's not without its problems; it can be quite slow in cases where there are complex paths to match, it only provides you with very basic message delivery techniques, and it ends up being tightly coupled into the overall architecture of the actor hierarchy.

You are left needing a way to distribute a workload evenly across a number of potential worker processes. This is provided within Akka.NET through the use of routers. The concept of a router is quite simple, in that it simply wraps a number of actors within the system as a single target to which messages can be sent. Messages can then be distributed to all of the workers, as specified within the routing strategy. Routers can be configured in a number of different ways, either mostly through configuration or entirely within code. The simplest approach to distributing work is to send it to all of the actors referenced by the router. This is referred to as *broadcast* within Akka.NET, and the broadcast pool provides the underlying infrastructure required to distribute work in this way.

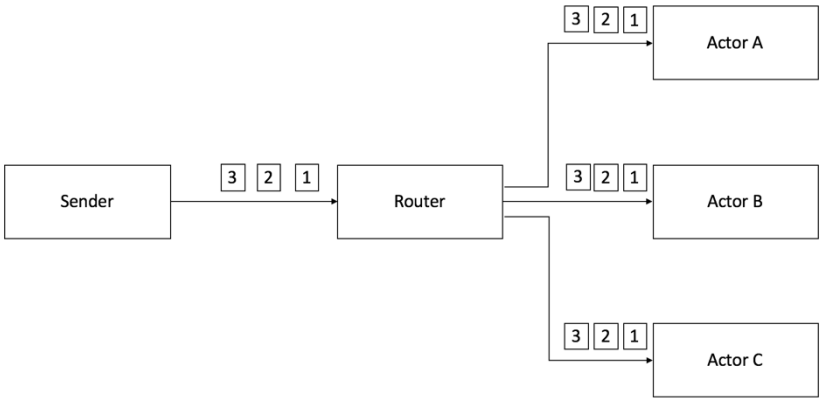


Figure 7.3 - The broadcast router sends the received message to every routee.

The first approach to configuring the router is through the use of the HOCON configuration. We saw the HOCON format and how it's used within Akka.NET to allow for the configuration of core Akka.NET features. In order to define a router in HOCON, you simply create a HOCON element, with the element key being the path to the router. This path has the same syntax as the actor selection we saw in chapter 3, and so you can use the wildcards in the address if the router is nested under multiple other actors. Every router configuration needs at least the type of router; but routers may also need other configuration. In the case of the broadcast pool, the router is simply `broadcast-pool`. This specifies that when the router gets created, it should use the specified routing logic. When using the broadcast pool, you also need to specify the number of workers that are deployed along with the router. In the case of the broadcast router, this is the number of workers that will be processing messages simultaneously. In the code example below, you can see that you've created a router at the `/LoadGenerator` route and specified that it should create 10 workers. The terminology used within Akka.NET for the workers in a router is `routes` and, as such, that terminology is used within the router definition.

```
akka.actor.deployment {
  /LoadGenerator {
    router = broadcast-pool
    nr-of-instances = 10
  }
}
```

It's not enough to simply define the router in HOCON; you also need to deploy it within the actor system. In order to do that, you need to create the `Props` for the given actor you want to deploy, as we saw in chapter 5. An example usage of the broadcast router is as a tool for load testing services, because you'll be able to execute a number of requests to the service simultaneously. To do this, you would have a load testing actor, for which you need to create the `Props` to deploy it. Once you've got the basic actor `Props`, you need to specify the router to use. You use the `WithRouter` method to achieve this and create a new `Props` object containing the extra routing information. In this case, because the routing information is referenced within the configuration, you need to tell Akka.NET that it should look in the configuration file for the specified definition. To do this, you need to pass an instance of the `FromConfig` class. It then looks up the definition based on the address to which the router is being deployed.

```
var loadTestingProps =
    Props.Create<LoadTestingActor>()
         .WithRouter(FromConfig.Instance);
```

It's also possible to create the router entirely in code without the need to use the configuration file. To manage this, when creating the `Props` for the actor and specifying the router, you can provide an instance of the routing logic to use. In the case of the broadcast pool, you can create an instance of the `BroadcastPool` router with the number of routes to use. You can then deploy that router in the same way as the configuration-based approach. The following code example creates a router instance and then uses that directly within the `Props`. The router is then deployed, with 10 routees to which it can distribute work:

```
var loadTestingProps =
    Props.Create<LoadTestingActor>()
         .WithRouter(new BroadcastPool(10));
```

A router is itself just a more generalized actor and exists in the actor system exactly the same as any actor that you might write. This means that you're able to send it a message and it will automatically forward the message onto the routes dependent upon the routing logic. In the broadcast example, it will distribute the message to every one of its routees. The router can be referenced in the same way as any other actor, and messages are sent to it in the same way as previously, simply using the `Tell` method.

```
loadTestingActor.Tell(new LoadTestingActor.WebsiteStressTestMessage("http://wwww.github.com"));
```

This is all that is required to instantiate a simple router that creates a number of routees to which messages are distributed. The broadcast logic is just one such example of a router, and there are many others included within Akka.NET, which we'll come to later in the chapter. But, for now, you're able to deploy a router into the actor system and then send messages on to the routees.

7.2.2 Pools and groups

In the last section, when you were deploying a simple broadcast router, you saw a different terminology used when you created a router. Rather than creating a `BroadcastRouter`, you created an instance of a `BroadcastPool`. The reason for this is that Akka.NET supports two different types of router, pools and groups. Although they both use the same underlying logic to route messages to their routees, they differ in the way that routees are managed.

When you created a `BroadcastPool`, you simply provided a number referring to the number of routees that it should distribute the messages to. Internally, the router creates the specified number of routees as its children and then passes any messages onto its children. This means that, when you're using a pool-based router, the router itself is responsible for things such as the supervision of the routees. The supervision of the workers only uses one supervision strategy though, which is to escalate all exceptions to the parent of the worker. As a result, if the parent then decides to restart the router, it will ultimately end up restarting all of its routees as well. This makes pools an ideal solution for when you're only interested in having a very basic set of workers that can respond to messages.

In contrast to this, if you're using a group, you need to explicitly specify the routees that the router is due to forward messages on to. As a result, the router expects the routees to already be deployed in the actor system at the provided paths. Because the router is no longer directly responsible for the routees it is communicating with, this means that it doesn't supervize the actors at all, and instead it's left to the original parents of the actors. You're not able to specify things like wildcards here either, and instead, you need to specify the concrete paths to the given actors. Router groups are ideal when you have a pre-existing hierarchy of

actors where a selection of actors within that hierarchy should be used as the routees. It's also beneficial if you have a more granular supervisor strategy you want to use.

In order to use a group instead of a pool, you simply need to instantiate it in the same way as you did with a group. But, instead of specifying the number of routees you use, you provide a collection of routee addresses. You can ultimately configure this in HOCON or code, exactly as you did with the pool setup. If you want to specify the use of a group in HOCON, you can use a snippet similar to the following. As you can see, the key difference is the use of broadcast-group as the router type and you specify the routee paths as an array of strings.

```
akka.actor.deployment {
  /LoadGenerator {
    router = round-robin-group
    routees.paths = ["/user/loadgenerator/w1", "/user/loadgenerator/w2", "/user/loadgenerator/w3"]
  }
}
```

You can then create an instance of it within the actor system, exactly as you did before, by specifying that the router used within the `Props` should be taken from HOCON. It's important to note that, when the router gets created, it does no validation to ensure that the actors exist at the paths provided. As such, if there are no actors available at those paths, as the router attempts to deliver messages, they will instead be delivered to the `DeadLetters` actor and logged as such:

```
var loadTestingProps =
    Props.Create<LoadTestingActor>()
         .WithRouter(FromConfig.Instance);
```

You can also choose to create the whole router group in code once again. In this case, though, instead of creating the `BroadcastPool` actor, you create an instance of the `BroadcastGroup` object. With this, you simply specify the paths of the routees that you should use:

```
var loadTestingProps =
    Props.Create<LoadTestingActor>()
         .WithRouter(new BroadcastGroup("/user/loadgenerator/w1",
                                        "/user/loadgenerator/w2",
                                        "/user/loadgenerator/w3"));
```

There is, however, one other core difference between pools and groups that we've not yet covered. Because pools are responsible for all of the workers that are able to receive messages, they're able to spawn new instances on demand to react to load and the message backlog. The pool can do this thanks to the availability of the concept of autoscaling, whereby you're able to specify the minimum number of workers that should be used, as well as the maximum number. You're able to specify that autoscaling should be used by providing a resizer when you create the pool, either in configuration or in code. In order to use it from within configuration, you can simply add a resizer configuration element to the router configuration, where you specify three values. You need to specify that it should be enabled; the lower-

bound, which is the minimum number of workers to use; and the upper-bound, which is the maximum number of workers to use. You can see in the following example that you initially deploy the router with five actors, but it can scale the number of children it has to a value between 1 and 10:

```
akka.actor.deployment {
  /LoadGenerator {
    router = round-robin-pool
    nr-of-instances = 5
    resizer {
      enabled = on
      lower-bound = 1
      upper-bound = 10
    }
  }
}
```

You can also specify it directly in code when you create the router by providing an instance of a resizer. The simplest resizer to use is the `DefaultResizer`, which is a simple autoscaler that uses message pressure as its reason for scaling. Once again, you can create a `BroadcastPool` with five workers by default, but the ability to scale between 1 and 10 workers in code, as follows.

```
var smsGateway =
    Props.Create<SmsGatewayActor>()
         .WithRouter(new RoundRobinPool(10, new DefaultResizer(5, 50)));
```

Although pools and routers do share all of the routing logic, and the underlying distribution of messages works in the same way, they do have some interesting underlying differences that can help build scalable architectures. Notably, the ability to use autoresizing pools allows you to build architectures that are not only scalable but also elastic, ensuring that you only ever use and pay for the scale you need, while also maintaining the ability to rapidly scale up and down, as required.

Router performance

Although we've seen that you use routers in the exact same way as actors, and from an external point of view, they appear to be nothing more than an actor, internally they are optimized for message throughput. Because routers are responsible for distributing a potentially large number of messages, it's important that they don't become a single point of bottleneck, which ultimately results in the application slowing down. As such, the core routing logic is not encoded within the actor itself but is instead stored directly within the actor reference returned for the given router. This then ensures that the router's mailbox is bypassed entirely, which decreases the latency associated with the messages and ensures that a router won't end up having a mailbox that overflows due to the high throughput imposed upon it.

7.2.3 Distributing work summary

Routers provide an incredibly simple means of distributing messages between a large collection of potential workers, allowing you to effectively scale the applications you write in order to react quickly to the demands imposed upon them. So far, we've seen how simple it is to write an application that distributes a given message to every one of the routees associated with a router, through the use of a broadcast-based router. This is only one example of the various routers that are available in Akka.NET, and we'll go on to look at the rest later in this chapter.

7.3 Routing strategies within Akka.NET

So far in this chapter, we've seen how, using Akka.NET routers, you're able to distribute a single message to a number of routees associated with the router. But this router only allows you to broadcast a message to all of the intended targets. In certain circumstances, this proves to be beneficial, such as when you want to parallelize workloads. The example we've seen already was in the case of a distributed load testing system. In that case, you want to ensure that you're able to perform as many operations simultaneously as possible.

We've seen with the Reactive Manifesto that you want to be able to build applications where you can remain responsive even if you're experiencing a heavy load on the system. When considering the scaling techniques that are available to us, we saw that the most appropriate choice for your usage to allow for rapid scalability is to scale out. Using this approach, you want to be able to distribute the messages you receive to as many targets as possible, such that you don't encounter a bottleneck on a single actor. Your usage of routers so far has been fairly limited and hasn't allowed you to achieve the overall objective of increasing the throughput of a message queue. Fortunately, Akka.NET provides a number of different implementations of the core routing logic to allow for a variety of techniques of distributing messages through the router to its routees. These routers allow for a wide variety of behaviors, which ultimately allow you to continue to build applications that remain responsive even under intense load. Throughout the rest of this chapter, we'll look at the routers included with Akka.NET and the advantages that they provide. In each case, we'll look at how you can deploy a pool implementation in code by using the classes themselves. We'll also look at how you can configure the router using the HOCON configuration approach. In every case, though, we'll only see the configuration section, along with what the values it needs mean. In all cases, when using the HOCON-based configuration approach for routers, the router itself is deployed in the exact same way as in the broadcast-based actor configuration deployment. Because we're just looking at the pool-based approach, it's important to note that all of the routers also support the group-based approach. But they all follow the same pattern as we saw in the case of the broadcast pool, compared to the broadcast group. This means that by following the same pattern of simply changing the number of routees configuration value to the array of actor addresses, you're able to create a group instead of a pool.

7.3.1 Random routing

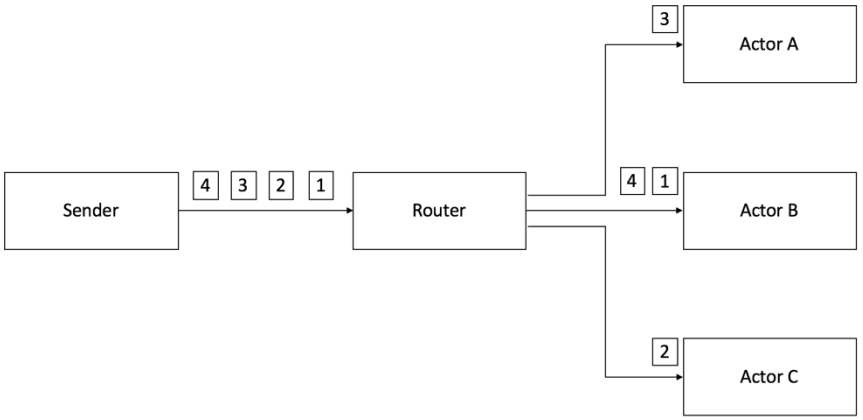


Figure 7.4 - The random router selects a routee randomly for each message.

The simplest possible approach to distributing a message through a router to a single node is to choose one of the routees randomly and then send the message through to it. Assuming the random number generator used on the machine on which the application is currently running is truly random, given enough time, it will generate a random distribution of targets chosen to receive messages.

In order to create a random router, you simply need to provide the number of routees it should create, in much the same way as we saw with the broadcast router. The core difference between the two, though, is that the random router only forwards the message onto a single route, which is chosen at random. To create this type of router in code, you simply need to create an instance of the `RandomPool` class, which you then pass to the router configuration in `Props`:

```
var smsGateway =
    Props.Create<SmsGatewayActor>()
         .WithRouter(new RandomPool(5));
```

The router can also be configured using HOCON, as you might expect, and as you can see, the configuration is mostly the same as the broadcast pool, with the exception that you specify the name for the round robin pool that is to be used. Once again, to configure this, you simply need to supply the number that refers to the number of routees to be created by the router pool:

```
akka.actor.deployment {
  /smsgateway {
    router = random-pool
    nr-of-instances = 5
```

```
  }
}
```

Once this is deployed, you can then see the effect that this has on message distribution. If your random number generator is truly random, then you should see all of the routees receive an equal number of messages between them. In the following diagram, you can see the number of messages each of the five routees received when you sent a total of 10,000 messages. Although all of the routees received a similar number of messages, there was still a difference of several hundred between the lowest number of messages processed and the highest number of messages processed. For small messages, this effect may be negligible, but when dealing with messages that rely on large amounts of processing, you may be left with long time differences between the shortest queue finishing and the longest queue finishing.

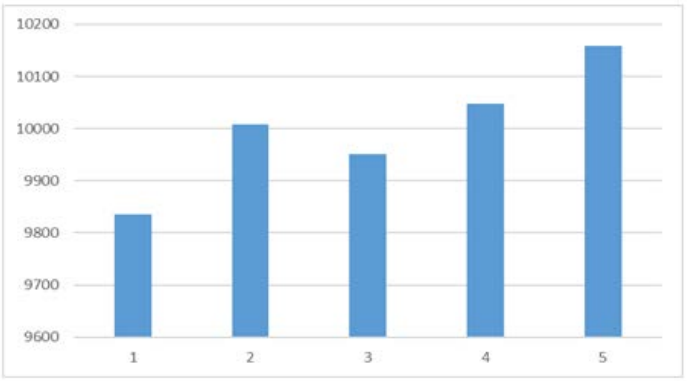


Figure 7.5 - The random router may not evenly distribute messages to every routee.

The random router is an incredibly simple router that can effectively distribute the messages it receives to a number of routees. Despite the simple nature of the router, it still manages to generate a fairly even distribution of messages across all of the routes; but this is ultimately dependent upon the performance of the platform's random number generator. Because the random number generator generates a random sequence of numbers, it may be the case that only one number is generated for an extended period of time, putting a bottleneck on that individual routee, while the other routees have empty queues.

7.3.2 Round robin routing

Because the random router ultimately depends on the underlying random number generator's performance when it comes to generating random numbers, there's the potential for a bottleneck to occur, where only one routee receives the majority of the messages. In order to counter this difficulty, you need a way of ensuring that you see an even distribution of messages to all of the routees of a router. For example, if you have three routees, you want to ensure that, if you have nine messages to distribute, then each individual routee receives three

messages in an even order. In the following diagram, you can see the order of messages in the mailbox for each routee. You can see that the router sends the first message to the first routee, the second message to the second routee, and the third message to the third routee, before it wraps around and sends the fourth message to the first route:

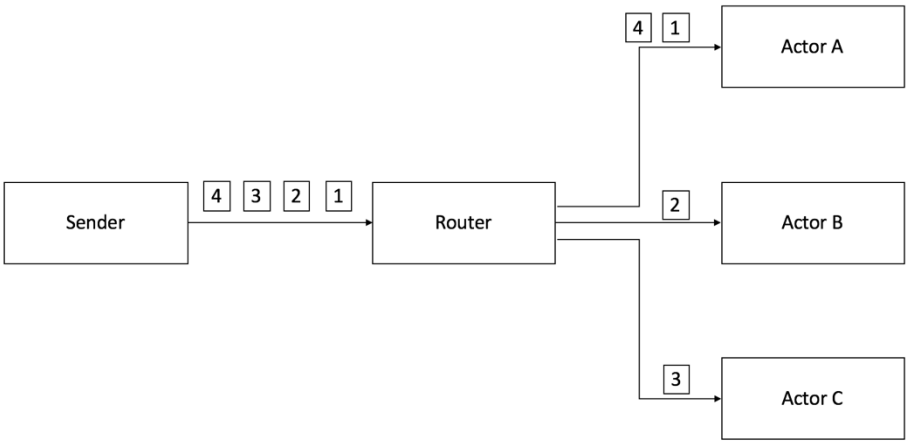


Figure 7.6 - The round robin router chooses the next routee as each message arrives and then starts again from the beginning.

You can create a round robin pool in much the same way as you created the past two routers, namely, by simply specifying the number of routees to be used by the router. When using the code-based approach, you simply need to create an instance of the `RoundRobinPool` class, to which you simply provide the number of routees:

```
var smsGateway =
    Props.Create<SmsGatewayActor>()
         .WithRouter(new RoundRobinPool(5));
```

You can also choose to create the router using HOCON, once again, simply by specifying the number of routees to use within the router, as well as providing the name of the router type to use:

```
akka.actor.deployment {
  /smsgateway {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

The round robin approach to message routing is once again incredibly simple and ensures that you get an even distribution of messages to all of the routees associated with a router. It's

particularly effective where you have a stateless actor, which you might use to communicate with an external service, for example, and you want to scale it out to multiple instances. By using a round robin router, you get a fairly consistent throughput, assuming all of the routees are capable of processing a message at a similar rate.

But this router is still only a best effort router and there are still a number of problems we might encounter when using it, which prevent the round robin router from enabling a truly even distribution. One such example is what happens if one routee is running more slowly than the others. Its queue is likely to grow, while all of the others process their messages quickly, ensuring their mailboxes stay small.

7.3.3 Shortest mailbox queue

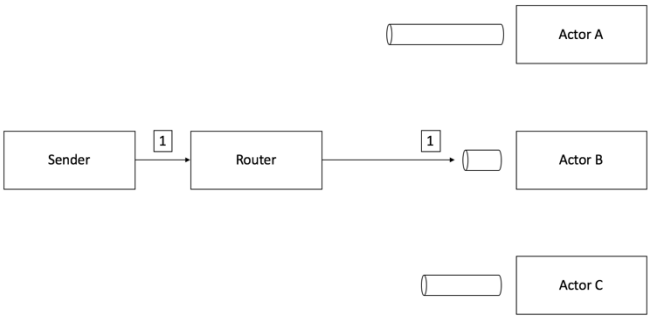


Figure 7.1 - The shortest mailbox router routes messages to the routee with the shortest mailbox.

Within Akka.NET, every actor has a mailbox, as we saw in chapter 3. Once you send a message to an actor, it gets appended to the end of the queue to be picked up at some point in the future by the processing component of an actor. Because an actor can only process a single message at any point in time, this means that, if it is slow to process an individual message, then the message queue can start to grow quickly. If you were to use this actor that is processing messages slowly alongside an actor that is able to process its messages quickly, then the faster actor will quickly get through all of its messages and potentially have no more work to do, while the slower actor might still have a large number of messages left to process.

In order to counter this problem, where you might have one actor idle while another has a long message queue, you can choose to use the shortest mailbox queue-based router within Akka.NET. The concept behind this actor is explained within the name. When a message is sent through the router, it consults the routees in order to see which has the shortest message queue. The message is then automatically forwarded onto the actor with the smallest mailbox. This ensures that actors that are processing messages more quickly are able to receive more messages than other slower running actors.

In order to use a shortest mailbox router, you simply need to once again specify the number of routees to use within the router. It then automatically handles creating these actors and processing based on mailbox size. In order to create the router in code, you just need to create an instance of the `SmallestMailboxPool` class, which is then provided as the router instance to the `Props` for an actor:

```
var smsGateway =
    Props.Create<SmsGatewayActor>()
         .WithRouter(new SmallestMailboxPool(5));
```

The router can equally be created using HOCON configuration, where you simply need to supply the name of the router. In this case, within HOCON, it's `smallest-mailbox-pool` along with the number of instances to use, as we've seen before:

```
akka.actor.deployment {
  /smssender {
    router = smallest-mailbox-pool
    nr-of-instances = 5
  }
}
```

The smallest mailbox router is particularly useful in cases where you don't necessarily know in advance how long it's going to take to process a message. If you have a message with a payload that will lead to a larger amount of work than others, then the smallest mailbox router is an ideal candidate for reducing the impact that the message is likely to have on subsequent messages. But it is far from a silver bullet; you don't know how long it will take to process a message until it is actually processed. As such, if you enqueue a message after a message that ultimately takes a while to process, the enqueued message will encounter a delay before it reaches the processing stage.

7.3.4 Consistent hashing

All of the routers that we've seen so far have relied on selecting a random routee that will be selected to process the message; but sometimes you want to ensure that messages with a common trait always get sent to the same target routee. Let's take an example in which you might want to achieve something like this, building a very simple key-value-based datastore that persists its data to the filesystem upon which it's running. In this case, the common trait between each message will be the key that identifies the item in the database.

We saw a similar scenario in chapter 2, where you had a unique means of identifying the target actor through the use of a sensor identification string that allowed you to directly route any new messages straight through to the actor. This then allowed you to have every sensor running in parallel without worrying whether you'd encounter any bottlenecks. Given these benefits, it might seem to be a natural fit, because it ensures that you're able to perform concurrent operations across large numbers of keys simultaneously; but it also comes with some downsides. The most notable of those is that you need to ensure that you have created an instance of an actor at the provided name before you're able to communicate with it. Every time a request comes in for a given key, you'd have to ensure, first of all, that an actor exists

at that key's address before sending a message to it. If the key didn't exist, you'd then have to create an actor instance and send it a message. This then adds a lot of overhead to every request and introduces a significant amount of latency, which can ultimately lead to applications becoming unresponsive. You also need to have one actor per key and store that actor in memory. Although there is relatively little overhead on an actor, it does start to add up at larger scales. Especially if you're implementing a key value store, you might see millions or even billions of keys and actors in memory, which would then present you with problems. The final difficulty is ensuring the isolation of actors. You saw that every actor should keep its state internal and not share it with any other actors. Because the data you persist to disk for an actor is, by association, part of the actor's internal state, it can't share it with any other actors in the system. This can then impose a lot of resultant pressure on the filesystem, where you end up with one file per key and, therefore, the potential for millions of very small files on the filesystem.

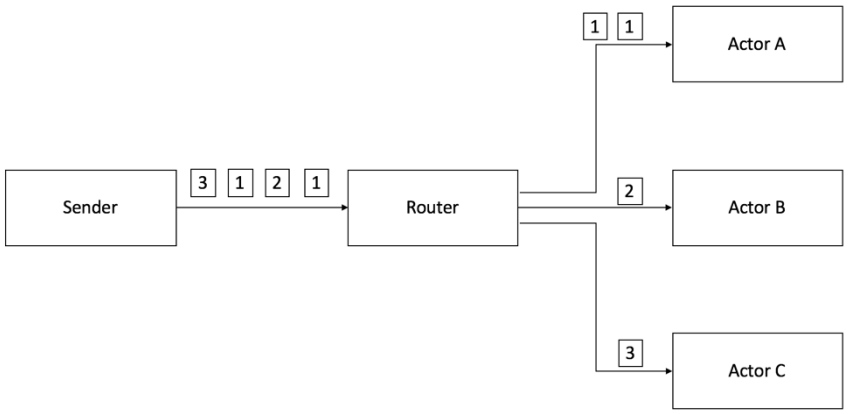


Figure 7.8 - The consistent hashing router directs every message with the same properties to the same actor.

Ultimately, you want to create a means of having a single actor responsible for a select portion of the available keys and simply repeat this, with each other instance being responsible for a different portion of the keyspace. In order to do this, you could store each of the keyspace regions in a specific location, which allows the router to automatically route the message based on what the lookup table says. This approach does, however, lead to a lot of required co-ordination to update the lookup table whenever a new key-value pair is added. Ideally, what you want is a completely stateless router that simply allows you to select a given routee based on the message in a repeatable manner.

A simple approach to this is to simply calculate the hash of the message or a specific property within the message. This hash can then be used to calculate the target routee for the message. A hash is calculated by passing the message through a hash function whose sole

responsibility is to map data down to a fixed size from an arbitrary length. In Akka.NET, the hash function used maps the message property from whatever its data size is, which in the case of a key value datastore will be a string for the key with variable length, down to a fixed length, which is the number of routees available to the router.

This forms the basis of the consistent hashing router within Akka.NET, which computes a hash for the message before using the hash as a means of deciding which routee to send the message to, based on this generated hash. You can create a consistent hashing router in code by simply creating an instance of the `ConsistentHashingPool` class. In order to do this, all you need to do is specify the number of routees to use:

```
var smsGateway =
    Props.Create<SmsGatewayActor>()
         .WithRouter(new ConsistentHashingPool(5));
```

But you also need to do a little more to ensure that the router works in the most effective way possible by choosing the correct property on your message so that all messages with that property in common end up reaching the correct target. You have three possible options for how to manage that within Akka.NET. The least intrusive way for either the routees or the message is to create a simple hash mapping delegate at the point when you create the router. This simply takes in the message and returns the property of it to use as the thing to hash. If you've got a message defined as follows for your key value datastore, then you create the actor instance, you can supply a hash mapping delegate by using the `WithHashMapping` method on the router, which supplies a new router with the mapping applied. In the following code, you have a common interface for all of the database related operations that allows you to retrieve the key out of the key-value pair easily. You use this interface as a means of retrieving the key for the given key-value pair:

```
interface IDatabaseMessage
{
    string Key { get; }
}

public class Get : IDatabaseMessage
{
    readonly string _key;
    public string Key { get { return _key; } }

    public Get(string key)
    {
        _key = key;
    }
}
```

```
var consistentHashingPool = new ConsistentHashingPool(5)
    .WithHashMapping(x =>
    {
        if (x is IDatabaseMessage)
            return ((IDatabaseMessage)x).Key;
        return x;
    });
```

Since all of your messages use a common interface, you could also use the `IConsistentHashable` interface directly, which allows you to specify what the hash key should be. The router then checks to see whether the message implements this interface, and if it does, it uses this interface to retrieve the hash key. Using this approach relies on you editing all of the messages to rely upon an underlying implementation detail, and so may not be a valid option depending upon where the messages originate from.

```
public class Get : IConsistentHashable
{
    readonly string _key;
    public string Key { get { return _key; } }

    public object ConsistentHashKey { get { return _key; } }

    public Get(string key)
    {
        _key = key;
    }
}
```

The final option is to wrap all of the messages you send to the router in an envelope that provides the internal message along with the hash. The router then uses the hash to direct the message before forwarding on the original message stripped out from the envelope. To use the envelope, you just create an instance of `ConsistentHashableEnvelope` with the message and the hash key to use:

```
var message = new Get("Anthony");
var envelope = new ConsistentHashableEnvelope(message, message.Key);
database.Tell(envelope);
```

You're then able to calculate a hash for a given message and choose the routee to which you should send the message. Due to the hash function, you have a known fixed size for the possible output values; for example, it might generate a number in the range between 0 and 255 inclusive. You then create a circle and place these possible values around the edge of it, using the same hash function used for mapping keys to also map the node identifers onto the ring. So, given that you created your router with three routees that act as nodes in the consistent hashing router, you can position them onto the ring. The following diagram shows an example of placing a node on the ring. That node is then responsible for all of the hash values you encounter as you move clockwise around the ring until you reach the next node:

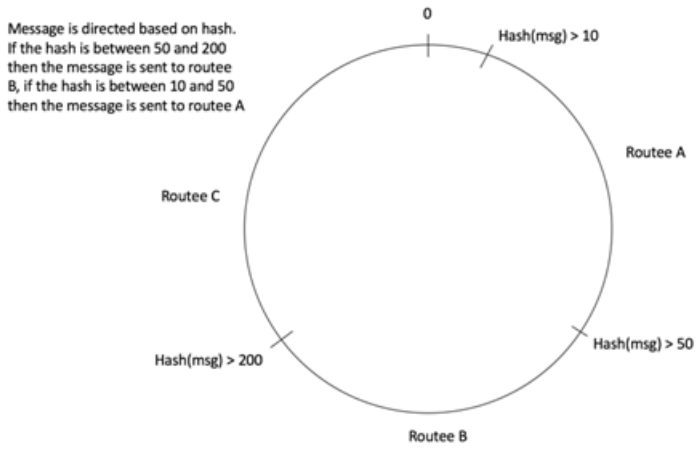


Figure 7.8 - In a consistent hashing router, routees are placed around a ring and each routee is responsible for a portion of the ring.

Although consistent hashing helps you to choose a target for a message in a simplistic way, you can also see some disadvantages with it. The most notable problem is that, due to hashing each node, some nodes will inevitably be responsible for more keys than others, leading to the potential for increased load on one of the routees. In order to counter this problem, the consistent hashing router allows you to specify the number of virtual nodes per routee. This then says that, rather than having five routees with one node each, you can create five nodes with, for example, 10 nodes. This provides you with a total of 50 nodes around the ring, which helps you to ensure you see a more even distribution of keys around the nodes. It's important to note that the new nodes are entirely virtual and are only for the purpose of calculation; you still have only five routees created at any time. By default, Akka.NET uses a value of 10 for the virtual nodes factor; but should you want to, you're able to change this by supplying a different

value in the constructor when you create the router, as shown in the following code, where you use a virtual nodes factor of 20:

```
var databaseProps = Props.Create<DatabaseActor>()
                         .WithRouter(new ConsistentHashingPool(5)
                                     .WithVirtualNodesFactor(20));
```

As has been the case with all routers so far, you're able to create the router using the HOCON configuration rather than relying on storing these values in code. The values you need to set to be able to use it are simply the router name, the number of routees to use, and the virtual nodes factor. You still need to handle how to retrieve the hashing property using code, adopting any of the techniques we saw previously:

```
akka.actor.deployment {
  /services/cache {
    router = consistent-hashing-pool
    nr-of-instances = 5
    virtual-nodes-factor = 20
  }
}
```

Although the router pool also supports automatic resizing in the same way as all of the other routers, you need to be more careful when using it. In the example we saw on a possible use for this router, you stored a state within the actor, the file containing all of the key-value pairs stored within that node. If you were to add another node, then it would affect which node was queried for a given key. This change would then leave you in a situation where historical data is not able to be retrieved from the routees. Auto-resizing with the router is therefore only useful if the routees are completely stateless. If the routees are stateful, then auto-resizing and the consistent hashing router should be avoided.

Consistent hashing routers are a great means of ensuring an even distribution of keys are handled by all of the routees. The consistent hashing-based approach to stateful distribution has proven to be useful in a number of large projects, including a number of distributed NoSQL databases, such as Amazon's DynamoDB and Basho's Riak, as well as the internals of some big data processing tools, such as Hadoop's MapReduce. By using it within Akka.NET, you're able to ensure that the router distributes messages with a common trait to the same actor repeatably with minimal overhead required and no co-ordination within the router.

7.3.5 Scatter gather first completed

None of the routers up to this point have prevented you from returning data from them, but they are designed more for scenarios where you are more likely to be dispatching work to be completed without a result being returned. From time to time though, you do want to use a request-response-based model to ensure that you're able to get data out of a service. In the event that you use this request-response-based approach, your ultimate aim should be that the

latency between sending a message and returning a value in response to this message should be as low as you can possibly make it.

With a single destination choice router such as the round robin router, if you choose a routee with a long queue, then you inevitably suffer from a long delay in reaching the processing stage. But if you choose a short queue that happens to have a number of large messages within it, then you're also left with longer latencies. The easiest way to ensure that you get the shortest possible latency is by sending the request to all of the potential candidates capable of processing the message. Whichever one of those gets to the message first is then able to send the response back to the sender. So, in order to model this, you could simply use the broadcast router to send the message to all routees.

By doing this though, you're left with the problem of all of the routees then replying to the actor awaiting a response. Because you're interested in simply getting a result back, you want to minimize latency. The minimum latency is simply the first response it gets back. You then want to ignore all of the other messages and ensure that it doesn't end up filling the message queue of the original requesting actor. Essentially, you need to ignore every subsequent message that gets returned in response to a given message once you've received the first result.

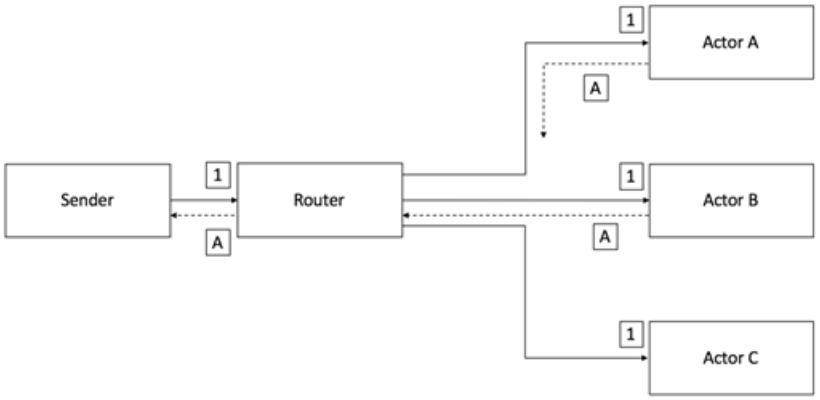


Figure 7.9 - The scatter gather router broadcasts the message to every routee and then returns the first response.

Akka.NET provides a router designed specifically for this purpose, in the form of the scatter gather first completed router. The aim of this router is to distribute a message to all of its routees and wait for the first result. Once it receives that first result, it should send it to the original actor that was asking the actors for a response. Every message it receives from its routees in response to its original request should simply be ignored. This ensures that you get the shortest possible latency from the collection of routees, while also preventing the original requester from being flooded with the same response multiple times. If, however, no reply is

received at all from the routees within a given timespan, then it automatically sends a `Failure` message back to the original sender to notify them of the timeout.

An example of a potential usage of this is in cases where you have a database with a number of replicas. Each of the routees is responsible for communicating with a single replica. When you want to retrieve a value from the database, you query the actor that executes the request. You can then simply get the response back from the first replica that replied and return that to the original actor requesting the data. This then allows you to focus on using a database with the lowest latency.

As you're dealing with actors that each have an independent configuration, you'll create a group router this time, using the `ScatterGatherFirstCompletedGroup` class. You're able to specify the maximum timeout before a timeout failure response is sent back to the original sender. In this case, you specify that if you don't receive a response within 200 milliseconds from one of the database servers, that you should send a timeout failure:

```
var databaseReplicas =
    new ScatterGatherFirstCompletedGroup(
        TimeSpan.FromSeconds(0.2),
        databaseMaster,
        databaseReplica1,
        databaseReplica2);
```

You're also able to create the router in configuration as well. In the next code snippet, you create the same scatter gather router using HOCON. You specify the type of router to create, as well as the standard number of routees. You also specify the timeout period in HOCON. When using times in HOCON, you're able to specify the time using units. In the following example, you specify that you should wait 0.2 seconds. You can also use other suffixes to represent other units of time, such as minutes and milliseconds:

```
akka.actor.deployment {
    /services/database {
        router = scatter-gather-group
        routees.paths =
["/database/master",
        "/database/replica1",
        "/database/replica2"]
        within = 0.2 seconds
    }
}
```

The scatter gather first complete router proves to be an incredibly useful router, particularly in cases where you want to minimize the latency of request-response-based message passing. In this case, despite the fact that you're distributing the work to as many routees as possible, you only need to worry about a single message being returned from the router. This makes it ideal for using with the ask-based approach to receiving a message response we saw in chapter 3.

7.3.6 Tail chopping router

Within asynchronous systems, there's always the potential for a message to be processed at a slower rate by one actor than another, this could be caused by any number of possibilities, ranging from the hardware it is running on, to operating systems, in addition to transient issues any external services may be experiencing. Although these slowdowns are relatively infrequent, they ultimately pass the issue onto the user and they can end up waiting for the result of their request. When you graph out the latency you see across a large number of requests, it typically looks somewhat like the following graph. You have a small number of requests that execute and return almost instantaneously, you then have the majority of the requests around the median latency, before you finally see a long tail of high latency:

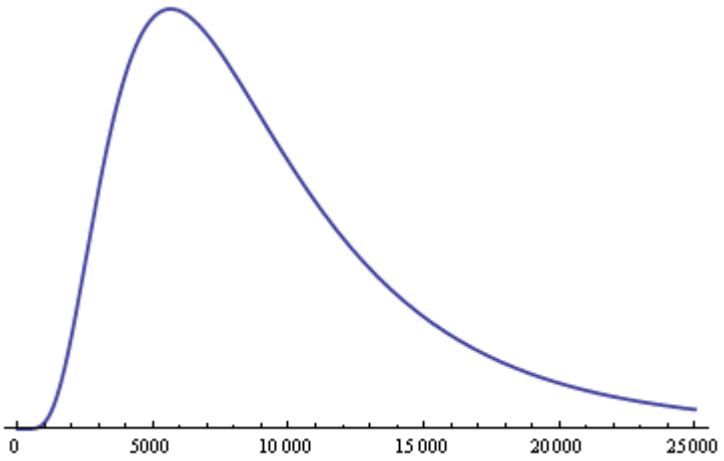


Figure 2.10 - Typically, latencies follow a bell curve with the majority of responses having a common time with some arriving earlier and some arriving later.

In order to ensure that all users enjoy a responsive application, you need to try and minimize the effects of the long tail latencies that you frequently see. In order to achieve this, you need to try and prevent the seemingly random slowdowns that system components might experience. The scatter gather first complete router helped to make this a possibility, thanks to the distribution of messages to all routees. In the event that one of its routees is experiencing a slowdown, it won't cause a significant degradation of service, since one of the other routees will pick up the message and try and process the message. If it manages to process it sooner than the original, then its response is ultimately forwarded on to the original sender. But, using this approach, you end up doing a significant amount of redundant work even if you manage to return a result quickly. This has the potential to lead to problems further down the line, due to the amount of extra work you are forcing yourself to perform regardless of status. Using the scatter gather approach, you're also always assuming the worst possible scenario will occur, which is that the original response will be longer than you can tolerate. But this is not likely to

be the case, since most of the responses complete within a respectable time period. The aim is to shorten the tail of the graph as much as possible, which is frequently much less than 1% of the total requests.

In order to get around this, you should only retry the work if you truly believe that you are going to encounter a scenario that is likely to end up forming part of the graph's tail. This is the aim of the tail chopping router, to significantly shorten the tail of the graph. It does this by combining a number of components. The router first selects a routee at random to send the message to, but if it doesn't receive a response within a certain amount of time, then it automatically sends the message to another routee before it then awaits a response from any of them. Once it receives a response, it automatically forwards the message onto the original sender and ignores all subsequent responses. But if it doesn't receive a message at all before a certain period, then it will automatically send a `Failure` message to the original sender. The tail chopping router works on the basis that there's a high probability that one of the other workers is able to process the message faster than the routees chosen so far.

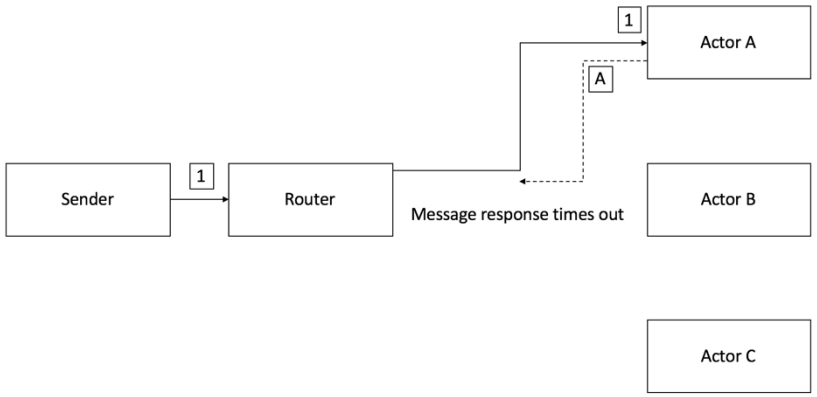


Figure 7.11 - The tail chopping router first sends a request to a random routee and starts a timer, awaiting a response before the timer expires.

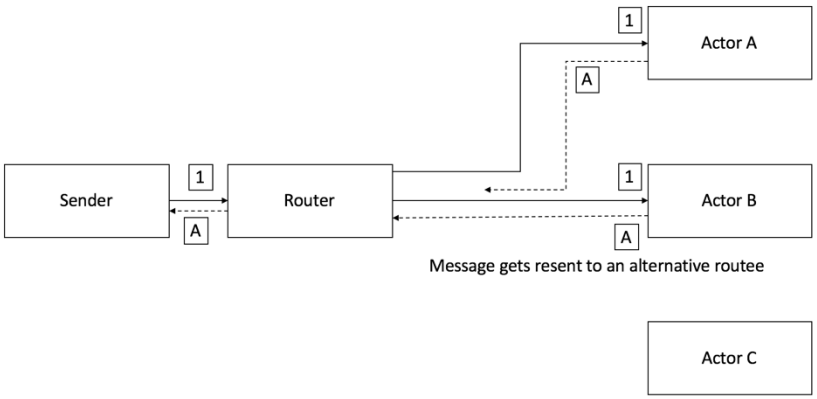


Figure 7.12 - If no response is received then a second routee is chosen at random and the timer is reset; when a message is eventually received, it is sent back to the original asker.

The tail chopping router requires a little more configuration to get working than the other routers we've seen so far in this chapter. You can create a tail chopping router by creating an instance of the `TailChoppingPool`. As with others, you need to specify the number of routees to use, and much like the scatter gather router, you also need to specify the overall timeout before you deem the request to have failed. You also need to specify that you should attempt to contact the next routee after a specific period of time, known as the interval. The following example shows a scenario where you create a tail chopping router with five routees and a timeout of 1.5 seconds. It also says that you should forward the message onto a second routee after 200 milliseconds of no response from the routees that have been contacted so far:

```
var searchApiProps =
    Props.Create<SearchAPIActor>()
         .WithRouter(
            new TailChoppingPool(
                5,
                TimeSpan.FromSeconds(1.5),
                TimeSpan.FromSeconds(0.2)));
```

As has been the case so far, you can also create this router with a HOCON-based configuration, where you follow a similar pattern to the scatter gather router. Many of the configuration variables used are the same, notably the number of routees and the maximum timeout, but you also supply `tail-chopping-router.interval` to specify the time between multiple routee calls:

```
akka.actor.deployment {
    /services/search {
        router = tail-chopping-pool
        nr-of-instances = 5
        within = 1.5 seconds
        tail-chopping-router.interval = 200 milliseconds
```

```
    }
}
```

The tail chopping router-based approach to performing redundant work when you believe you can get a quicker response from another target than the current one has been used to great effect in a number of distributed NoSQL databases to help reduce the tail-end of response latencies. Due to the simplicity of Akka.NET's routers, it ends up being fairly simple to implement in your applications to try and ensure that your users aren't left waiting for a response for too long. There are downsides to it, though, the most important of which is that you're aware of the expected latency of the target routees. Without an understanding of the latency at a point in time, then it's likely that any configuration values supplied for the interval period will not be effective at reducing the latency tail.

7.3.7 Routing strategies summary

Although it may seem like there is an abundance of routers within Akka.NET, many of them are tailored to a specific set of situations and designed to ensure that your applications are able to maintain responsiveness even when faced with an increased load. All of the routers that Akka.NET provides are also able to use the resizer functionality to automatically resize once the application sees a certain sustained level of load upon it.

Broadcast support within routers

Early on in this chapter, we saw the broadcast-based router, which allows you to rapidly send a message to all of the routees. This means of message distribution is also supported in all of the other routers. The consistent hashing router introduced the concept of message envelopes in order to retrieve the hash key for a given message, but there is another envelope supported by all of the routers we've seen here. If you wrap a message in a `BroadcastEnvelope` before sending it to the router, then it will automatically distribute the message to all of the routees. This allows you to easily transmit data to all of the routees of a given router.

7.4 Case study: scaling – throughput – advertising systems

Advertising has become a core component of millions of websites and has also become either the primary or sole source of revenue for a large proportion of those websites. At the heart of web advertising is a vast bidding war being fought between hundreds of advertising agencies, all trying to show their clients' adverts on the most appropriate web pages for the right audiences. For an advert to be shown, a decision first needs to be made on whether it should be, how much the agency should bid, then the advert data should be returned. All of this needs to happen within milliseconds if an agency wants to be competitive in an industry worth billions of dollars every year.

Given the number of webpages that choose to display adverts and the number of people who visit those webpages, advertising agencies have to process a huge number of requests per second. But they also have to cope with the inevitable spikes in traffic from the original webpages. Given this need for low latency, high throughput services, Akka.NET provides an

ideal base upon which to build advertising systems. But it's more than simple scalability that is required; you also need the ability to respond to the traffic spikes on the origin webpages.

In this chapter, we have seen how to use routers as a way of distributing work across a number of actors in the actor system. By using routers, you're able to simplify, and most importantly, automate the process of scaling the number of actors within your application. Given that a higher number of actors all working in parallel are able to process more incoming work, it's no surprise that this allows you to handle more page views and send more bids in the shortest amount of time possible, increasing the likelihood of your advertising agency being chosen as the provider of the advert.

In the following example, you can see how to hide actors behind routers that are able to deploy more actors on demand and shutdown others after a period of low usage. Each actor is responsible for receiving an input containing information related to the visiting user, which includes data such as their IP address, location, anonymized ID, and plenty more. The actor then works out whether a bid should be made. As the original website sends a request for an advert, it's converted into a message, and then it's sent to the router. The router then automatically chooses an actor to process the message. At the same time, the router is computing metrics and calculating the number of messages that are flowing through the router, which it then uses as a means of working out whether more actors are needed to achieve the required throughput.

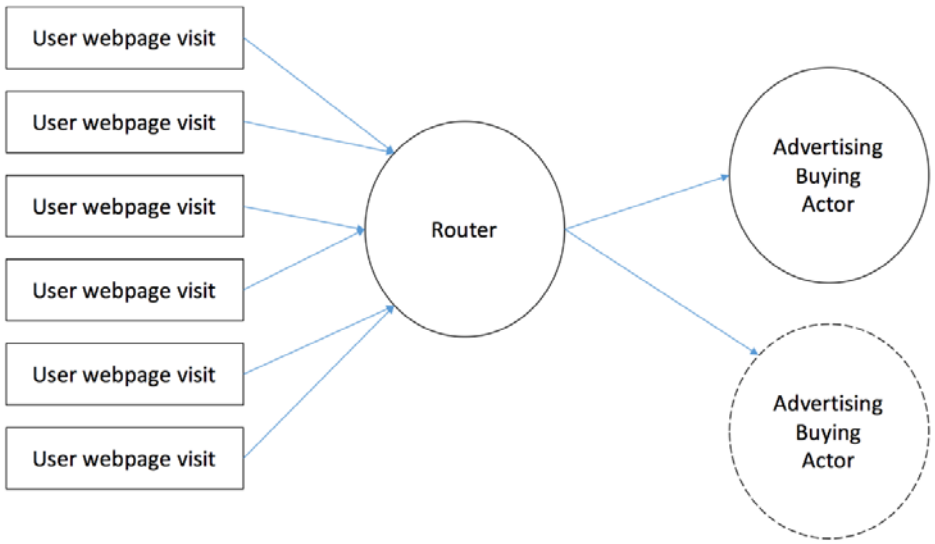


Figure 7.13 - A router allows you to receive a huge number of messages, in this case whenever a user visits a webpage to see an advert, and allows you to split those messages across multiple actors thus increasing throughput.

In cases where you need to react on demand to unpredictable usage, routers allow you to scale the number of actors able to receive and process messages. By parallelising processing, you're able to easily increase throughput while also reducing response times.

7.5 Summary

In this chapter, you learned:

- How you're able to increase throughput through an application by scaling out the number of workers processing messages
- How Akka.NET responds to the load demands of the application by scaling up and down as required when using pools
- How Akka.NET allows you to manage the routees of a router yourself through the use of a group router
- How you can choose different routing methodologies to suit the task at hand

8

Composing actor systems

This chapter covers

- **Linking actor systems**
- **Scaling applications across multiple machines**
- **Creating applications which are able to handle machine level failure**

One of the key points we saw within the Reactive Manifesto is that it is a series of traits shared by many applications to ensure that they present the user with a responsive experience regardless of what is happening within the application. We've seen that this has meant you need to accommodate your designs so that they're able to withstand two key problems, failures internal to the actor system and increased load upon actors within the system. We've seen how Akka.NET is able to handle both of these scenarios thanks to a number of features provided by the framework.

We've seen that failures can be handled by a combination of actor hierarchies and the supervision system. By using the hierarchy, you're able to isolate failures to a single actor and then automatically recover from that failure with a supervision strategy handled by its parent. We've also seen that when it comes to handling an increased load upon the system, you need to be able to increase the level of message throughput through the actors. This is something that you're able to solve with the help of Akka.NET's routers, which are able to evenly distribute messages to a number of actors while appearing as a sole actor. We've even seen that routers are able to scale the number of routees dynamically dependent upon load so that you're able to elastically scale dependent upon load at any given moment in time by providing either more or less compute power.

Although these features go some way to allowing you to create applications that are fault-tolerant and scalable, they all share a common trait. All of the applications so far have only run on a single machine, which ultimately limits their capabilities. When considering fault tolerance,

we've so far only seen a case in which you're able to recover from the failure of small, isolated actors that are running in the context of a much larger application. But, as we saw in chapter 2, there are many cases that might ultimately lead to more catastrophic failure cases for your application. Although there are many failures that you're able to solve at the application level, you're also likely to encounter issues relating to hardware failure which ends up with the entirety of the application no longer being available. Hardware failures are a relatively infrequent occurence, but you must also consider all of the layers of abstraction between the hardware and the application that could result in failures in the application. For example, if you're connecting to the service from a client, there's a strong possibility that due to network connectivity issues, the server may be unreachable; alternatively, a process running on the machine may end up killing the process that is running the actor system. In all of these cases, the problems lie outside of the bounds of the actor system and so need more planning to resolve.

We also explored the notion that actors are cheap in chapters 2 and 3 and that you're able to create millions of them per gigabyte of memory, but you need to consider the downsides of doing this. Notably, the underlying CPU only has a limited number of CPU cores, which, therefore, limits the number of operations that can be considered to be running truly concurrently. Although you're able to easily upgrade to a CPU with more cores or better threading capabilities, you'll still hit a limit on the number of things that can run simultaneously. This then creates a bottleneck where you're waiting for resources on the CPU to be made available for actors to process messages, leaving you in a situation where you need to increase the concurrency capabilities of the CPU.

With these problems, you need to be able to combine multiple machines so that you can either reduce the likelihood of downtime caused by machine-level crashes or distribute workloads more evenly in order to further increase the potential throughput of the actor system.

At its core, Akka.NET has been built with ease of distribution in mind so that it's able to scale out across multiple machines as well as multiple cores. We saw in chapter 1 that if you want to see solutions that are fault-tolerant and scalable, then you need to ensure that your application isn't tied to the physical location of an actor. Instead, you rely on the location transparency that you get as a result of the underlying message-passing-based architecture at the core of Akka.NET. With this location transparency, you only care about a loosely coupled address pointing to the actor's mailbox and let the underlying messaging system calculate how that message should be delivered into the mailbox. This then affords you the freedom to move actors not only between threads or cores on a single machine but also between processors on distinct machines, even if they happen to be located on opposite sides of the world.

The use of actors at the core of the framework also allows further ease of use when it comes to developing applications that are designed to scale across machines. We saw in chapter 3 that actors encapsulate all of their state within the bounds of the actor. This encapsulation prevents other actors within the system directly accessing the data stored at a given location in memory. Instead, you need to communicate with the actor by sending it a

message to retrieve the data stored within it. This ensures that you can safely relocate an actor to any other location without worrying about any potential implicit connections between actors. Because all of the communication is handled through messages, it's simply left down to the messaging system to direct the message to the correct target.

8.1 Introducing Akka.NET remoting

If you want to avoid the problems you are likely to encounter with an application running on just one machine, you need to distribute your application across multiple machines. This will ensure that you're able to provide a consistent level of service in the event that one machine encounters significant difficulties or you reach the concurrency limits of a single machine. If you're looking to join two machines together, you can connect them together through a network and ensure that they're able to communicate with each other. But network programming brings along a number of complexities, which can ultimately lead to complications with the applications that communicate through the network.

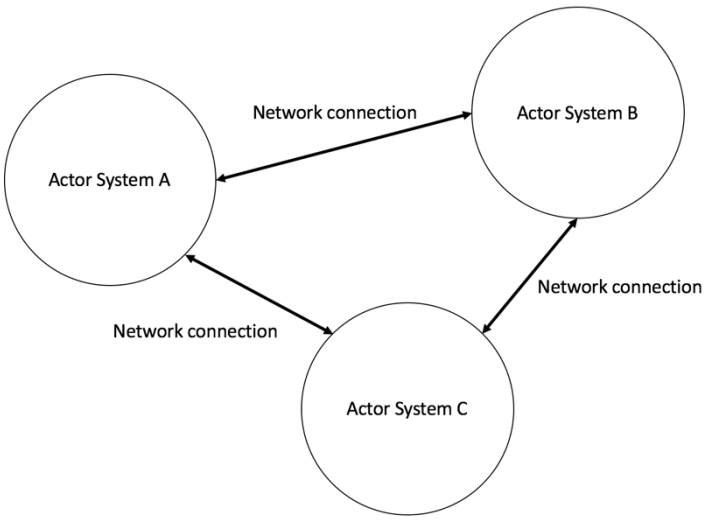


Figure 8.1 - Akka.NET remoting allows multiple independent actor systems to act as a single actor system transparently to the user.

When dealing with networks, there are numerous things to take in consideration. You need to deal with potentially unreliable networks, where there's no guarantee that a message will ever reach its intended target. You also need to consider that messages will only reach their target eventually when sent through the network due to increased latencies caused by the sheer distance between two machines. These complexities then present further difficulties,

such as knowing whether a process on a remote machine is executing correctly or whether it has encountered a faulted state.

In order to cope with these difficulties, Akka.NET provides a number of remoting features designed to allow multiple actor systems to communicate together without the need for in-depth consideration of remote-oriented programming. These systems can exist on either a single machine or multiple machines, with communication happening through a variety of protocols, such as TCP or UDP, with pluggable support for alternative transports tailored to the specific problem at hand.

We've seen in chapter 3 that one of the biggest benefits you get from a message passing architecture is the freedom to relocate actors on a machine as required. You're not tied down to a single reference for an actor, and instead you're free to let the Akka.NET runtime relocate actors to different threads or different locations in memory. Akka.Remote takes this notion of location transparency to its extreme limits by allowing you to develop applications that don't need to concern themselves with knowledge of the underlying network. In fact, Akka.Remote allows you to run your applications without the need for any code changes at all, and instead you can simply modify the configuration of an application and immediately have it scaled out across multiple machines.

Within Akka.Remote, all of the machines end up running the same actor system, and these actor systems are connected together. The design is such that all of the systems are able to run the same code without the need for changes to individual applications within the deployment. This means that there is not a single actor system acting as a server with others connecting to it; instead, all of the actor systems operate as peers. This ensures that you can build truly distributed applications without a potential single point of failure.

This ability to treat actor systems as peers provides the driver to enable you to build applications that are truly indifferent to the environment in which they're running or hosting actors. This allows Akka.NET to truly take location transparency to the limit, to the extent that no code changes are needed to run a single application across multiple machines, and instead the application can be driven entirely by configuration. This is enforced to a point through the limited API that is available for developing networked applications. you only have two key points that could be used directly to influence a remote deployment: the usage of `Props` and the address system of actors, both of which we'll see later in this chapter.

Within the internals of the remoting API, everything that could lead to problems across a network is considered and catered for. All of the features we have seen within Akka.NET that have allowed you to develop applications that are fault-tolerant and scalable while also scaling out the application across the limits of a single machine by allowing the use of a network. Throughout this chapter, we'll see how to use the remoting functionality provided by Akka.NET to distribute your applications across a network with minimal changes needed to the code we've seen so far.

Akka.Remote usage

Akka.Remote is primarily designed for situations in which the two connected machines have the same level of privileges and don't provide support for only running specific roles on a given machine. The design isn't intended for use in cases where a client is connecting to an actor system, for example, a client application communicating directly with an actor system. We'll see examples of how you're able to achieve this in a later chapter where we discuss how you're able to expose the actor system to the outside world.

8.2 Preparing to use remoting

8.2.1 Installing Akka.Remote

The remoting capabilities of Akka.NET are shipped outside of the core distribution and so before you can use it, you first need to install the library. In the same way as the core distribution, the library is shipped through NuGet and so in order to install it, you simply need to add the Akka.Remote project. After installing the library, you need to configure it so that it's able to receive connections from other actor systems.

8.2.2 Configuring remoting for a project

Before you can use the remoting functionality within a project, you first need to provide a number of key parameters to the actor system to configure these features. You do this with the HOCON configuration we saw in chapter 5. At the least, you need to specify the following information within the configuration file.

Within the preceding configuration, you've made two major additions. The first is the change to the actor provider that is responsible for how the underlying Akka.NET library retrieves its `IActorRef` instances and how it then performs all of the routing through to the defined target. Here, you supply the class name for the `RemoteActorRefProvider`, which is able to retrieve references to actors running on remote machines. You also configure the network transport, which is responsible for communication between the actor systems. In this case, you use a TCP socket and configure it to listen on the supplied address and port. It's important that the port specified is not used by anything else. If you want to be supplied with a random unused port, you can instead choose to use port 0 and the operating system will choose one at random.

```
akka {
 actor {
  provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
 }
 remote {
  helios.tcp {
   port = 8080
   hostname = localhost
  }
 }
}
```

This is all that is required to allow other actor systems to connect to this actor system instance, but if you want to use it then you'll probably need multiple actor systems configured to run with remoting enabled. The examples we'll see in this chapter all rely on having two actor systems to communicate with each other. In order to manage this, you simply need to alter the port that you should use to listen on and the address if the actor systems are running on different machines. You can use a number of different approaches to this, as we saw in chapter 5, but for now you'll use two independent files with different configuration, the reasoning for which will become clearer throughout the chapter. In the following example, you set up a second actor system that listens on a different port while running on the same machine. This then allows you to run across multiple actor systems throughout the rest of the chapter.

```
akka {
 actor {
  provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
 }
 remote {
  helios.tcp {
   port = 8081
   hostname = localhost
  }
 }
}
```

Although you now have two actor systems with remoting configured, no connections are made between them until one of the applications needs to communicate with the other. Connections are therefore made lazily on-demand as the actor system needs them. In order to test these remote-based actor systems, you need to run multiple instances of the application, each with a different configuration file. While testing, you can simply supply the configuration file as a parameter to the application and load it on-demand, as in the following example:

```
Chapter8.exe node1
```

With the configuration loaded from the command line, you can simply pass the path to the file as a parameter to the application. This means that if you refer to the configuration we saw earlier as `node1.conf` and `node2.conf`, you can run two instances of the application by running the following commands in the directory containing the executables and configuration files:

```
Chapter8.exe node1
Chapter8.exe node2
```

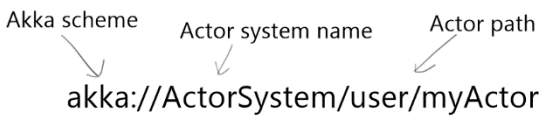
In a production environment, you're unlikely to supply the configuration file path through the command line, and instead use the fallback configuration options we saw in chapter 5 when we first saw HOCON. But because you're running multiple instances in the same environment in the examples in this chapter, you use different configuration files to allow for completely isolated configuration, the reasoning for which we'll see throughout this chapter.

8.3 Communicating with remote actors

Now that you've got a number of actor systems configured to listen on incoming network connections, you can create applications that communicate by sending messages between these instances. We saw that the connections within Akka.NET are created lazily, and so for the basic actor systems you create, despite a network being available, there isn't yet a connection between them. In this section, you'll start to use the many Akka.NET features we've seen throughout the book, but spread across multiple actor systems.

8.3.1 Remote actor system addresses

We saw in chapter 3 that all of the actors in an actor system are identified by an address. The address is then used by the underlying framework to route all messages through to the correct actor instance in memory. So far, all of the actors you've encountered throughout the book have used a simple scheme for representing the address of the actor. You can see in the following example a typical address you might have encountered so far. The address is a simple Uniform Resource Identifier (URI) with three key pieces of information; the scheme, which in this case is the string akka; the actor system name, which is whatever name you provide when you create a new actor system; and the path to the actor, which is used to traverse the actor hierarchy to search for the specific child actor:



When using multiple actor systems together, you'll encounter some additional information that you need to encode within the address to identify the likes of the machine that is hosting the given actor system name. You also need to ensure that the system is able to understand how to communicate with the remote actor system, because we've seen that remoting has support for a number of different underlying networking protocols. An example remote address is coming up; as you can see, there are some changes, but for the most part the address is similar to what we've seen so far.

The first key change is the different scheme used. You now need to supply the akka as before, but you also need to postfix it with the protocol used; in this case, the value supplied is tcp to signify that you'll be connecting to it through TCP. You might see other values in here to represent other communication protocols, such as UDP. You next need to specify the address that is hosting an actor system of the given name. This is composed of two components, the IP address or the hostname of the remote machine and the port that the remote system is listenting on:

If you need to address an individual actor on a running system, this is the simplest way of doing so, but it does go against some of the location transparency aims we've discussed so far in this chapter because you're tightly coupling your machine's location to your codebase. Given the prevalence of the cloud for deploying your code, this may prove to be problematic due to the potential frequency of IP address changes with an autoscaling architecture. In the rest of this section, we'll look at how to abstract locations away from code and drive the logic through configuration as much as you possibly can.

8.3.2 Sending messages to remote actors

So far, we've seen how to send messages to actors in one of two ways, either by using a direct reference to an actor with `IActorRef` or by sending a message to a given address. You can follow the same process of sending messages to an actor even when dealing with remote actor instances.

If you're dealing with a remote actor instance, you have to target it through the address format we saw in the previous section, making sure to include the remote actor system name and target machine. You can then create an actor selection using this address and use it as you would have used at any stage previously. In the following example, you send a message to an actor running on a completely different machine:

```
var remoteActor =
    remoteActorSystem.ActorSelection(
        "akka.tcp://RemoteSystem@localhost:8081/user/remoteActor");
remoteActor.Tell("Hello remote actor");
```

When a message on a remote machine receives a message, it's also able to reply to it in the same way that we've seen so far. Akka.Remote then deals automatically with handling remote senders and serializing them so you can address them in the same way as before. As a result of this, you're able to perform request-response-style queries against remote actors through the use of `Ask`. But because you're dealing with remote targets where you have a network connection linking actors, there's a possibility of message loss leading to a situation where `Ask` never receives a response, causing it to block indefinitely. In order to solve this problem, you can specify a timeout period after which the call should throw an exception to signify that you didn't receive a response. The following example says that you expect a response from a remote actor, but if you don't receive that response within 20 seconds then it needs to throw an exception:

```
var response =
    await remoteActor.Ask<string>("Anthony", TimeSpan.FromSeconds(20.0));
```

Akka.Remote and message ordering

When you dealt with a single in-process actor system, it was guaranteed that all messages would arrive in the correct order, but when you introduce a network connection, you lose that guarantee. In the case where two independent actor systems both send a message to the same actor, there's no guarantee that the two messages will arrive in the order in which they were sent. This is due to the additional latency constraints that a network imposes, which may fluctuate as time progresses. Akka.Remote does, however, provide a guarantee that any messages sent between a pair of actors will be delivered in the correct order. For example, if in a pair of actors, A and B, actor A sends messages to B, and they will always arrive in the correct order, but if C then sends a message to B, it may not arrive in the order at which it was sent.

This setup of accessing known actors on a remote machine is ideal for scenarios in which you are dealing with an entirely external service. For example, in a web application, you may have one actor system running as a payment service responsible for aggregating the charges a customer should be paying and then a second actor system that is responsible for running the core application logic. In the context of the smart home example we saw in chapter 2, the billing system would be responsible for computing the cost based on the number of sensors a customer has running per month, or the amount of data the application is processing, and the core application logic is responsible for running the sensor's processing.

There are, however, times where you want more processing power for a single server with as little overhead as possible. In these scenarios, you should avoid being constrained to a single machine and instead scale out across more machines. In these cases, you typically want to avoid coupling this information directly into your codebase because it then forces you into a situation whereby the infrastructural deployment is locked into the binary of the application. We've seen when discussing scaling that you may need to scale in a really short period in order to cope with a significant load spike. In these situations, you need to be able to provision a new machine and have it usable for more resources.

Akka.Remote allows you to specify that entire sections of the actor deployment hierarchy should be deployed onto a specific machine through configuration. These sections are still addressed in the same way as if it was an entirely local machine, but you can deploy them onto a separate machine. In the following diagram, you can see a typical actor hierarchy that you might encounter, but all of the actors under one of the nodes are running on a different machine:

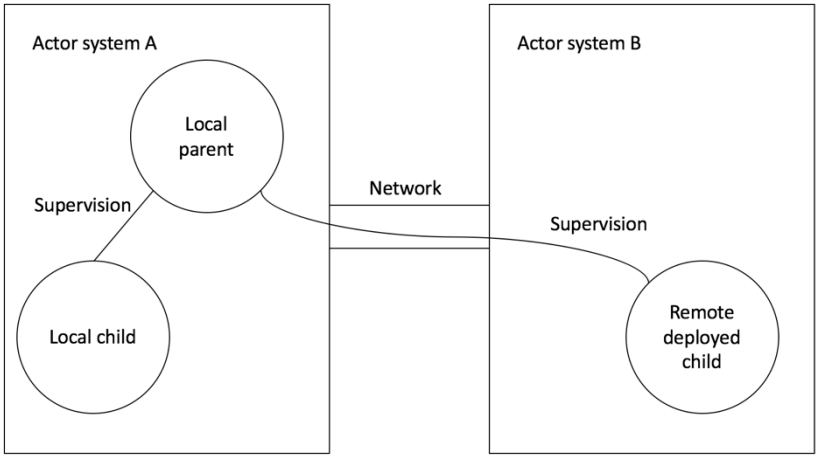


Figure 8.2 Akka.NET remoting allows you to deploy child actors remotely while also allowing for all of the features of local deployment such as supervision.

This allows you to take the notion of location transparency to the limit because you can develop entire applications without having to consider remote deployments but ultimately configure them so that they all run on different machines. Because the remoting capabilities of Akka.NET are driven entirely by configuration rather than an API, this means that you can change the underlying infrastructure of the application with a change to the configuration rather than requiring a full rebuild any time you want to scale out. In the following example, you specify in the configuration that the actor system should use the actor system at the address specified to host the section of the hierarchy stated. In the preceding diagram, we saw that the services branch of the actor system should run on the remote actor system.

```
akka.actor.deployment {
    /services {
        remote = "akka.tcp://RemoteSystem@localhost8081"
    }
}
```

This configuration change then allows you to develop your actor system as though it is all running on a single machine. In the following example, you select an actor below the services actor, which looks identical to any of the previous actor selection examples we saw earlier in chapter 3, but due to the configuration change you made earlier, it's calling over to a remote actor system:

```
var remoteActor = actorSystem.ActorSelection("services/cache");
```

Using this mindset when developing applications allows you to truly embrace location transparency and focus on the core aims of the application without needing to worry about the

underlying details of where actors in the hierarchy exist. This allows you to significantly scale out the actor system so that you're able to overcome the limits you might encounter on a single machine.

8.3.3 Remote deployment of actors

Although being able to access actors on remote machines provides you with some ability to scale out, you need to get those actors running on the remote machines in the first place. In order to cope with this, Akka.Remote allows you to deploy actors on a remote actor system from a completely separate actor system.

In the previous section, we saw that there are two key ways to deal with remoting in Akka.NET. We saw that you can either connect to existing long-running actor systems, or treat other machines as an extension of your existing computing resources. The remote deployment feature works for both ways, but there is one constraint involved with it; you must have the same types deployed across both machines. That is, the remote machine needs to have knowledge of the actor that it is going to deploy before it is able to be deployed. In many cases, this simply means creating an assembly containing the actor definitions and message definitions, which is then shared between the machines.

In the first scenario, where you're dealing with an existing long-running actor system, you may need to deploy an actor into its system when a key event happens. For example, in the case of the web application and billing system, you may sign up a new user to the service, meaning that a new actor relating to their account needs to be created. In order to achieve this, you could either send a message to an actor on the remote system that it should create a new actor, or you're able to automatically create it and deploy it yourself. We'll see throughout this chapter some of the other advantages that this brings along with it.

In order to deploy into a remote actor system, you need to follow the same process that we've seen when you deployed actors into the current actor system. You first need to create the `Props` that define the actor you want to create. As we saw in chapter 5, the `Props` are the way of specifying how an actor should be created. In this case, you need to specify the type of the actor you want to create. You also need to specify any constructor arguments should it take any. You also need to ensure that any constructor arguments you want to pass are also serializable, because they'll be sent over the network to the remote system. The process so far has been no different to what we saw when deploying any other actors, but you also need to specify how the actor should be deployed. You do this by specifying the deploy that the actor should use. Whenever you don't supply a deployer, by default it uses the local deployment, but now that you want to deploy into a remote system, you need to configure it. You do this by creating a new `Props` object by calling `WithDeploy` with the `Deploy` instance you should use. In this case, you need to use a remote deployment, so you need to create a `RemoteDeploy`. This simply takes the address of the remote machine into which you deploy the actor. Then, when you use this in combination with `ActorOf`, you receive an actor reference relating to an actor deployed on a remote actor system that you can communicate with as you would with any local actor.

```
var remoteDeploy =
    Deploy.None
          .WithScope(new RemoteScope
              (new Address("akka.tcp",
                           "RemoteSystem",
                           "localhost",
                           8081)));

var remoteProps =
    Props.Create<Cache>()
         .WithDeploy(remoteDeploy);
```

We also saw in the last section that you can configure the actor system so that any actors existing at a certain address will be automatically routed onto a separate actor system. In this case, any calls to deploy an actor at this address will automatically be routed through to the correct machine as defined in the configuration. As before, you need to ensure that any constructor arguments the actor takes are able to be serialized and sent over the network, but outside of this, nothing else needs to be done, and instead, the underlying Akka.Remote will handle all of the remote deployment.

Remote deployment allows you to rapidly build systems that are capable of scaling out across multiple machines without the need for significant initial design relating to the this. This is especially true in the case of applications in which you make the most of the configuration-driven remoting API. You can write actor-based applications that can automatically make the most of all of the computing resources they need while also getting the location transparency.

8.3.4 Remote actor communication summary

In this section, we've focused on how to communicate with actors across multiple actor systems without having to worry about the underlying network code thanks to Akka.Remote. We've seen how to abstract away the network and focus on the core business logic within your applications without needing to worry about how to add the underlying communication code.

8.4 Elastic scale across machines

We've seen so far that one of the key components of a reactive architecture is one that remains responsive even in the event that the system is under a significantly increased load. We saw one means of solving this problem in chapter 7 when we saw the different routers available to you in Akka.NET. Through the use of routers, you were able to set up your actor system to pool together a number of computational resources so that you were able to treat that pool of resources as a single actor. This pool usage then allowed you to get a significantly higher message throughput than if you had a single actor by running more than one processing stage simultaneously. This simultaneous processing allowed you to process multiple messages concurrently while also maintaining the thread safety that Akka.NET provides. By using this concurrency, you can make the most of the multiple cores and threading capabilities that modern processors provide. But you'll still ultimately hit a limit of what you can do on one machine simultaneously.

For example, many applications will attempt to brute-force a solution to a problem by running the same code with different inputs. In the event that you have millions of potential inputs, you're unlikely to have a CPU capable of running all of these actors simultaneously. You're also likely to encounter other bounds on a single machine, depending on the workload. The work that actors are doing might be heavily reliant on memory usage, and running lots of actors simultaneously might lead to a situation where the machine encounters slowdowns due to a lack of available RAM. Alternatively, actors might be doing a lot of network-related work and downloading large files to the disk. There reaches a point where the network connection becomes a bottleneck.

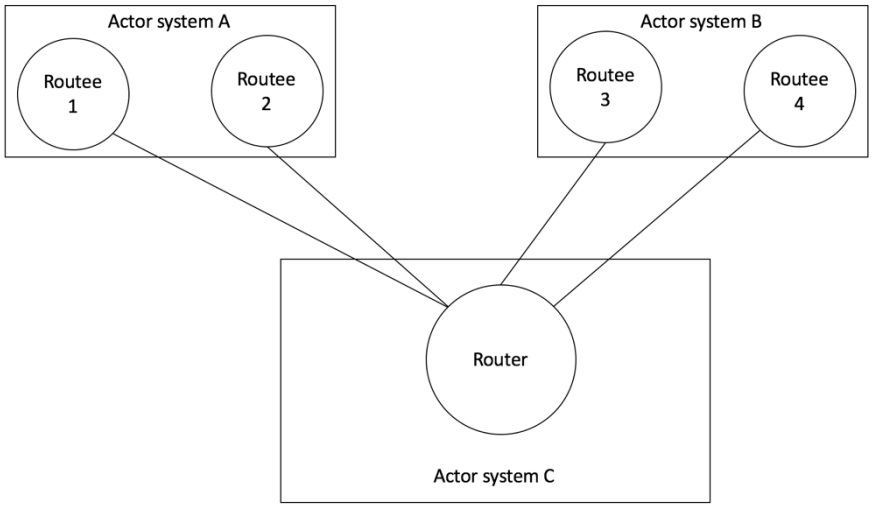


Figure 8.3 - A router can be configured to use other independent actor systems as hosts for routees.

We've seen how easy Akka.NET makes it to communicate with actors running on remote machines, but it's also possible to configure routers so that they use multiple external actor systems as hosts for the routers within a pool or group of actors. This enables you to get all of the benefits of routers, such as different routing strategies, high throughput, and more, while also using multiple actor systems in order to get around the limits imposed by a single machine.

8.4.1 Configuring a router to use multiple machines

When we looked at routers in chapter 7, we saw that there were two key differing types of routers. There was the router group, which dealt with already deployed actor instances; and router pools, which automatically deployed the correct number of routees. You can use either

of these with the remoting capabilities when used in conjunction with any of the available routing strategies.

The simplest routing technique to use with Akka.Remote is the group-based routers. We have seen that these routers use a number of actors that already exist within the actor system and build up the routing capabilities on top of these actors. We saw that in order to create a group-based router, you simply provide a number of addresses that the router should treat as its routees. Because we've seen in this chapter that you can communicate with remote actors using the same address patterns as we've covered previously, this means that you can supply a remote address to a group router and it will automatically route the messages over the network to actors on remote actor systems. You don't need to make any significant changes to the routers you create, and instead you provide the remote addresses. In the following example, you create a new round robin router through the code-based approach to direct messages through to a number of actors both locally and running in a remote actor system:

```
var roundRobinGroup =
    new RoundRobinGroup(
        "akka.tcp://RemoteSystem@localhost:8081/user/greeter",
        "akka://LocalSystem/user/greeter");
```

We've seen so far that hardcoding remote network addresses into the assembly may not be the most practical solution, especially in environments where IP addresses might frequently change on a per-deployment basis. In these situations, it might be better to provide the routees through the HOCON configuration method. This allows you to supply the routee paths at runtime rather than at compile time. In the following example, you configure the same round robin router group but supply it through HOCON in the same way as we saw in chapter 7:

```
akka.actor.deployment {
    /greeterRouter {
        router = round-robin-group
        routees.paths = [
            "akka.tcp://RemoteSystem@localhost:8081/user/greeter",
            "akka://LocalSystem/user/greeter" ]
    }
}
```

Although the group router is ideal for cases where the actors already exist in the system, you may want to be able to automatically deploy the actors as part of the router. When you combine the pool-based routers' ability to automatically deploy routees with the remote deployment capabilities of Akka.Remote, it provides a way to get automatic management of routees across actor systems. When using pool-based routers with the HOCON configuration, in addition to specifying the required information for the given router, you also need to specify the actor system addresses that the router should use to deploy routees to. In the following example, you configure a round robin pool to use two actor systems, one remote and one local:

```
var addresses =
    new List<Address> {
        new Address("akka", "LocalSystem"),
```

```
        Address.Parse("akka.tcp://RemoteSystem:@localhost:8081")
    };

var roundRobinPool =
    new RemoteRouterConfig(
        new RoundRobinPool(5),
        addresses);
```

You can also configure a remote router in code within the actor system as you might expect, but it involves some more significant changes that then impose some underlying knowledge of the fact that remoting is being used, thus breaking location transparency. To use a remote router pool in code, you first need to create a `RemoteRouterConfig` that draws in the addresses of the actor systems onto which you can deploy the routees along with the router pool class. For example, if you wanted to create the same round robin pool as the preceding example but in code, you can write the following, which then automatically handles all of the remote deployment:

```
akka.actor.deployment {
    /greeterRouter {
        router = round-robin-pool
        nr-of-instances = 5
        target.nodes = [
            "akka.tcp://RemoteSystem@localhost:8081/user/greeter",
            "akka://LocalSystem/user/greeter"
        ]
    }
}
```

Remote router drawbacks

Although the majority of routers are able to work well with remoting, there are some that aren't able to make the most of the network scale-out opportunities. One key example of this is the smallest mailbox router. With this router, remote routees take the lowest priority over local routees due to the lack of knowledge relating to remote mailbox sizes. In the event that you'll be scaling out a smallest mailbox router, the round robin router is likely to provide better usage.

This simplicity in creating remote routers allows you to once again continue with the notion of location transparency, and to write applications that can be scaled out onto vast numbers of actor systems with no changes needed to the assemblies themselves. By using these routers, you can further parallelize workloads so that you can increase workloads even further than you might be able to on a single machine.

8.5 Failure handling across machines

Although distributing work through a network provides a wide variety of benefits, it is not without issues. As soon as you introduce a network connection, you also bring along all of the complexities associated with network programming along with it. You encounter problems that you would never encounter when running your applications on a single machine. When you

consider how machines are connected, they typically use a wired Ethernet-based connection over distances ranging from a few centimeters to several kilometers and even in the order of magnitude of thousands of kilometers. This distance may not seem that far, but this drastically limits the speed at which data can travel between these machines. This introduces latency, the amount of time between sending a message from one machine and receiving it on another. Although latency is not a problem on its own, in cases where you need to undertake lots of co-ordination, for example, when dealing with shared mutable state, it can lead to signficant problems. Although this is fortunately not an issue with Akka.NET thanks to its use of the actor model, we still see plenty of other difficulties.

Unfortunately, the distance between machines also poses problems other than simple latency. When multiple processor cores are passing data between themselves, the distance is in the range of several nanometers, leading to little opportunity for data to become corrupted. But when network connections span kilometers, there is the potential for message corruption to occur for any number of reasons. It may be something as simple as somebody cutting through the network connection, causing no transfer to be possible at all, or it may be caused by more complex reasons such as environmental factors influencing the data as it travels through the connection. Environmental influences could include things such as magnetic or electromagnetic fields causing corruption, or even due to the signal becoming attenuated or distorted as it travels over such long distances.

The errors don't stop at the physical network connection; there is also a significant amount of infrastructure that exists between the physical network connection and your application. There are routers and switches designed to ensure the packets reach their intended destination, as well as networking interfaces on the application host and the drivers that interface the hardware with the operating system and then the operating system itself. These are not likely to be perfect, and a bug in any component along the chain could cause significant propagating failures. Even in cases where there are no bugs along the chain, you can still experience packet loss at any stage along the chain with correct operation caused by network congestion. If you try and send too much data through a single connection and it's unable to handle the load, then it has only one option, which is to start dropping packets in order to prevent cascading failures.

We also need to consider the variance associated with packet latency through a network. When you see the vast number of components between the application and the network connection, there's plenty of opportunity for an operating system to choose to perform an operation it deems to be more important than receiving a packet of data through the network, which ultimately causes a delay in the processing time. Once again, network congestion presents a problem in that if there's a significant number of packets in the queue waiting to be processed by the operating system, packets at the back of the queue will encounter delays. All of these then have the potential to skew latencies to the extent that it makes it impossible to effectively calculate the expected latency for a given operation.

The combination of the latency and packet loss makes it incredibly difficult to ensure that the distributed applications you write are able to withstand failures across a network. These

problems make it incredibly difficult to use many of the components you take for granted on a day-to-day basis. When it comes to the variable latencies that you might encounter, it becomes impossible to truly synchronize two independent machines with the same time. The potential for packet loss also makes it impossible to truly know the current state of a remote machine at any one point in time.

We saw when we discussed failure handling earlier in the book how to handle message loss across an unreliable channel; such a system is a significant help to systems that are unable to tolerate message loss. But a core component of Akka.NET is the actor supervision concepts we've seen on numerous occasions, which allow the isolation of errors and automated recovery from these errors. We've seen the difficulties with network connections that prevent you from getting a reliable view of the state of a remote machine. This makes it impossible to know with absolute certainty whether the remote machine has failed or the network connecting the machines has simply lost a packet.

In many typical distributed applications, these are all issues you need to consider, along with how they can potentially affect the safety and stability of the network. Fortunately, when using Akka.Remote, you can mitigate the effects of some of these failures so that you can more easily develop location transparent applications. Thanks to features such as remote deployment and monitoring of actors, you can continue to use all of the Akka.NET fault tolerance patterns, which allow you to continue in your goal to build truly responsive applications.

8.5.1 Supervisor strategies across a network

Throughout the book, we've seen that actors are not deployed independently, but instead as part of a larger hierarchy of actors with an underlying supervision tree. This enables parent actors to watch their children for failures and in the event that an error is raised, you can respond to it in the most appropriate manner. Even when you deploy actors onto a remote node as a child of a local node, you can retain this error handling functionality so that you can build fault-tolerant applications that scale scross a network.

Earlier in this chapter, we saw that you can specify that a certain actor path should be deployed onto a remote actor system. If that path is the child of an actor in the local actor system, then it gets supervised by a special actor on the remote node. In the examples we saw earlier in this chapter where you deployed the actor onto a remote node, it was automatically supervised by a special supervisor for any remote actors on the remote actor system using the provided supervision strategy. You can see in the following diagram how this ends up appearing within the actor system, while being transparent to the developer:

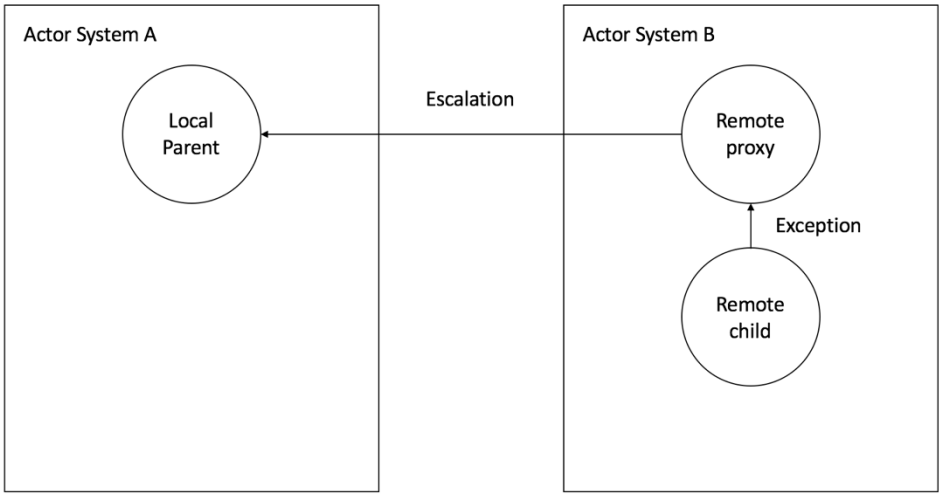


Figure 8.4 Remote children are deployed with a proxy that is responsible for supervision and handling any errors that may occur.

There is further potential to lose location transparency when using remote supervision, caused by the supervising remote actor. When dealing with actors, you use an address as a means of communicating with it. It also maintains some context surrounding the actor's deployment in the hierarchy. When dealing with remote supervision, these values will diverge due to the addressing systems in place. Although its location in the hierarchy is directly below actor A, its contact address has the remote supervision actor instead. Therefore, calling `Context.Path.Parent` and `Context.Parent` will return different results when deployed remote.

8.5.2 Remoting deathwatch

We've seen so far the usefulness of DeathWatch when developing applications, whether it's as a means of watching actors for failures or for detecting completion when using the reaper pattern. As a result of the remote failure detection provided by Akka.Remote, you can also use DeathWatch on remote actors. When using DeathWatch with Akka.remote, you also receive more information about the potential source of failures. Because we've seen in this chapter that there are many potential sources of failure in a networked application, DeathWatch helps to try and resolve the true source of the failure.

You can use DeathWatch in the exact same way if you're targeting a remote actor as when you target a local actor, through the use of `Context.Watch`. You also receive a `Terminated` message, exactly as if the remote actor was running locally, but there is further information encoded within the message that you didn't have a use for when we saw DeathWatch earlier.

In addition to the basic actor reference relating to the watched actor, you also receive two pieces of information that are designed to help diagnose the source of failure, `ExistenceConfirmed` and `AddressTerminated`. By observing the status of these properties, you can determine whether the `Terminated` originated from the remote node or from the Akka.NET remote failure detector. If the message has the `AddressTerminated` property set to `true`, then the message is derived from the fact that the remote actor system is no longer responding to health checks. The `ExistenceConfirmed` property also serves to help determine whether the remote actor itself reported its failure or if it was instead caused due to an inability to resolve a reference to the actor.

In much the same way as you can use supervision strategies across a network, you can use the DeathWatch concepts we saw earlier. You do need to be cautious when using a network connection because there's a higher possibility of failures occurring. But by using the data you receive in the `Terminated` message, you can make better decisions about the status of a remote actor.

8.5.3 Failure handling across machines summary

Using a network presents opportunities for a wide range of problems to occur at various points up the underlying technology stack. But Akka.NET allows you to continue to use the same tooling we saw in earlier chapters, when focusing on a single actor system across multiple actor systems. As failure becomes more likely as the number of running machines grows, it's important that you consider failure early on in the design of network-based applications. We've already seen the benefits of the supervision-based system used within Akka.NET earlier in the book, but thanks to Akka.Remote, you can continue to create applications that are fault-tolerant even when an unreliable network is involved.

8.6 Akka.Remote security

The remoting functionality of Akka.NET is primarily designed to work in a peer-to-peer environment where every actor system connected together has the same level of privileges. In the majority of cases, this typically means that the actor systems will be joined together on a private internal network, only accessible to the machines running the actor system instances. Therefore, it's typical to see an approach similar to the following diagram, where you have the actor systems listening on a specific port that is only accessible to machines within that network, and then providing alternative means of entry into an actor system such as through an HTTP-based API, websockets, or even raw sockets:

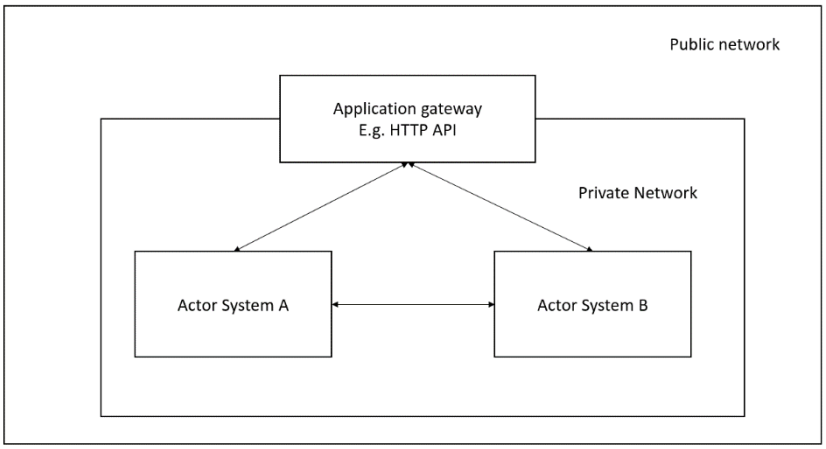


Figure 8.5 A typical Akka.NET actor system won't be exposed to the public internet and will instead stay within a private network and made accessible through a gateway.

But there are still security considerations that need to be taken into account about the applications you build. This is even more important when you open up your application to the whole world. This might be the case if you have an actor system instance running on a client's machine that needs to communicate with your hosted actor system. When you take this approach, you have a means of entry into the actor system that malicious entities could potentially exploit. Because the remoting functionality of Akka.NET is designed primarily for use as a low-level communication layer, you need to ensure that the applications you build take security into account in addition to the

8.6.1 Limiting messages able to be sent over the network

When you send a message over the network, it automatically gets serialized into a binary representation that is then deserialized back into the in-memory message. There is no validation of the original sender of the message, and as such it's possible for an attacker to construct a message payload themselves for a potentially dangerous message, which is then sent to the target actor system. Depending upon the message sent, this could have severe consequences on the stability of the application.

An example of a potentially dangerous message is the `PoisonPill`, which we saw earlier in the book. This is used as a means of telling Akka.NET to shut down an actor. If this was sent over the network to the root actor supervisor at the top of the actor hierarchy, it would shut down every actor in the actor system immediately. This has the potential to cause significant damage to an application.

Within Akka.NET, you can specify that a given message type could be potentially harmful to an actor system by making the message implement an interface. By making a message implement the `IPossiblyHarmful` interface, you're able to prevent dangerous messages from being sent through remoting. When Akka.Remote deserializes a message that implements the `IPossiblyHarmful` interface, it will be automatically discarded. But if the message originates from within the same actor system, then it will be passed through to the target as intended. A number of the core messages you're likely to encounter within Akka.NET implement the IPossiblyHarmful interface, notably the `PoisonPill`, `Kill`, `ReceiveTimeout` and `Terminated` messages. In the following example, you create a new message type that is used to delete a record from a database. This is something you're not likely to want any client to send to the application, and it should instead only be sent on the same actor system. To ensure this happens, you ensure that the message implements the `IPossiblyHarmful` interface:

```
public class DeleteAccount : IPossiblyHarmful
{
    private readonly string _accountId;
    public string AccountId { get { return _accountId; } }

    public DeleteAccount(string accountId)
    {
        _accountId = accountId;
    }
}
```

Sandboxing messages to a single actor system is a good start in terms of security, as it allows you to significantly limit the impact a client can have on a running actor system, whether that client is a regular user with good intentions or a malicious user wanting to compromise the system.

8.6.2 Restricting available remote actor targets

By restricting certain message types from being able to be sent over the network, you can alleviate the potential for significant damage to happen to your actor system. But there are certain messages that you do want to send over the network to other remote actor systems. These messages might be harmful, but in a different context. For example, the actor hierarchy might be laid out so that individual actors are scoped to a single customer. In this case, you don't want an unauthorized entity to be able to send a message to that customer's information, leading to a potential data breach or malicious attacks against that customer's data. This modeling of customer information might be such an actor hierarchy as in the following diagram. Here, each customer has a unique identifier with an actor named based on that customer identifier. All of the customer actors share a common parent in the form of the customers actor:

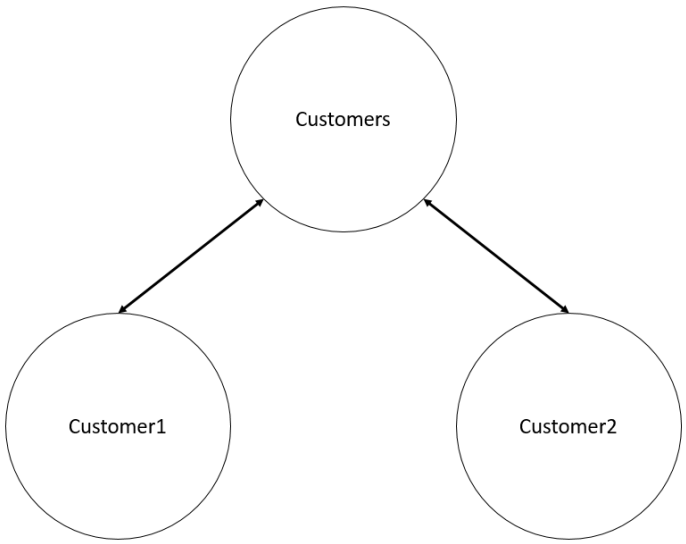


Figure 8.6 An actor hierarchy may contain sensitive customer information, and as such, you need to prevent users from retrieving data from the Customer2 tree if they belong to Customer1.

Typically, once a client has authenticated and you have information relating to the customer account they belong to, you can then communicate with either the scoped customer account actor or one of its children for specific services. But a malicious user may instead change the request to direct it at a different user's account. In this case, there's nothing you can do to prevent it. Instead, within Akka.NET you can specify that only certain paths can be contacted over a network. By using this technique, you can create an actor whose sole responsibility is to act as a receptionist to the remote node. All messages are passed through this actor, which then performs authorization to verify that a user has the correct privileges to communicate with a given target.

You can design such a system of authorization by wrapping all of your messages within a custom authorization message that contains an authentication token[1], an actor target, and a payload. Whenever you send a message across a remote connection, you need to ensure that you wrap it in this message:

```
public class AuthenticatedMessageEnvelope
{
    private readonly object _payload;
    private readonly IActorRef _targetActor;
```

[1] There are many different options available for generating authentication tokens. One example is a JSON Web Token (JWT) that is created by an identity service elsewhere within the system. Such a service is out of the scope of this example, but a common choice for an identity service is the IdentityServer project or Azure Active Directory.

```
    private readonly string _authenticationToken;

    public object Payload { get { return _payload; } }
    public IActorRef TargetActor { get { return _targetActor; } }
    public string AuthenticationToken
        { get { return _authenticationToken; } }

    public AuthenticatedMessageEnvelope(object payload,
                                        IActorRef targetActor,
                                        string authenticationToken)
    {
        _payload = payload;
        _targetActor = targetActor;
        _authenticationToken = authenticationToken;
    }
}
```

You also need to create an actor whose responsibility is to authorize any incoming requests into the actor system before sending the message onto the target. In order to manage this, whenever you receive a message you need to validate that the given token is authorized to communicate with the provided actor address. If you are authorized, then you forward the message onto the intended destination. In the code following example, you achieve this goal by automatically validating any incoming messages and then sending them on to the target specified within the message. You also use one feature here that we've not seen yet for sending messages, the `Forward` method. Typically, when you send a message, you use the `Send` method, which uses the address of the actor which is sending the message, but the `Forward` method sends the message but sets the sender of the message to be the sender of the message currently being processed. This means that the final destination doesn't know that there was an intermediary stage that handled the message:

```
public class SystemReceptionist : ReceiveActor
{
    public SystemReceptionist()
    {
        Receive<AuthenticatedMessageEnvelope>(env =>
        {
            if (IsAuthorised(env.AuthenticationToken))
            {
                env.TargetActor.Forward(env.Payload);
            }
        });
    }

    private bool IsAuthorised(string authToken)
    {
        //This is only sample code
        //A production application should verify the authToken
        //And only return true if the authToken is valid
        return true;
    }
}
```

You also need to add a setting to your HOCON configuration, which specifies that the only path accessible over a remote connection is your authorization actor. Assuming you've deployed that actor at the address "/user/authorisation", you can modify the HOCON to only allow messages to be sent to this path over the remote connection. You simply need to modify the `trusted-selection-paths` element of the `akka.remote` configuration element and supply a list of all of the possible paths. In this case, you create a list containing the path to your authorization actor:

```
akka.remote {
    trusted-selection-paths = ["/user/authorisation"]
}
```

Now, whenever you want to send a message over the network to this actor system, you need to wrap it in the envelope and send it to the authorization actor, which then unwraps it, authorizes it, and sends it on to the correct target within the actor system. By limiting the actor paths accessible to the outside world, you can reduce the effective surface area an attacker might use to try and interfere with your application. But by using the same techniques, you can use it as a means of enabling authorization in order to try and prevent interference between different customers.

8.6.3 Akka.Remote security summary

When you develop any modern application, it's imperative that you consider the security implications of any features you add, and it's no different within Akka.NET. When you have an actor system that contains all of your customer data within it, it's important that you effectively secure it against potential security issues. By using the features we've seen in this section, you can reduce the possibility that an attacker is able to compromise your applications.

8.7 Case study: remoting – network applications – web server and backend server

As application requirements and functionality grow, so do the teams that are responsible for their development. As these teams grow, it's inevitable that teams will ultimately be partitioned into smaller sub-teams, with each team focusing on a specific piece of functionality within the greater application. It ultimately reaches a stage where the independent teams need to work towards integrating the components to complete the product. For example, let's consider the example of a large enterprise project that needs to integrate with a number of different enterprise tools. For example, it's likely that an enterprise software project will need to connect to complex services such as CRM tooling or collaboration software. The level of development effort required to match the integration to the team

A modern enterprise software project is typically a dense sprawl of interdependent software modules, each of which is responsible for performing a small responsibility of a much larger application. But these projects did not simply become a huge mess overnight and simply grew beyond the bounds of what a small team of developers was capable of. All software projects start small with a well-defined scope but continue to grow and grow, incorporating functionality

from a variety of different sources. Ultimately, the work required on the software project exceeds that which is capable of being achieved by the original team. As a result, more developers are assigned to the project until the team becomes too big. This leads to the team being split up into a number of smaller teams, each of which is focused on developing a small piece of functionality within the application. In order to ensure these teams of developers don't frequently get in the way of each other, they'll develop loosely connected systems that are then commonly integrated through the use of an HTTP API or a message queue.

But this instantly increases the complexity of the application, requiring the development of an HTTP API on top of the original application functionality. In addition to this, it requires additional infrastructure to ensure that all components are working correctly, this includes common logging and tracing to track and correct issues across the application boundary. If it's using a message queue, then it requires the maintenance of a reliable message queue; furthermore, it leaves developers needing to understand more technologies on top of the minimum application requirements, thus increasing the minimum level of understanding required before developers are able to contribute effectively to the project.

In this chapter, we saw how you're able to take multiple independent Akka.NET-based actor systems and connect them together while treating the actors on the remote actor system as though it belonged to the local application, driven largely by the location transparency provided by the underlying remoting capabilities of Akka.NET. In the following diagram, we can see how to take two independent components of an enterprise application and connect them, allowing the actors on one actor system to automatically appear in the other and vice versa. This simplifies the integration period between components, as it treats the remote component as an extension of the original component without the need to add additional tooling in between the two.

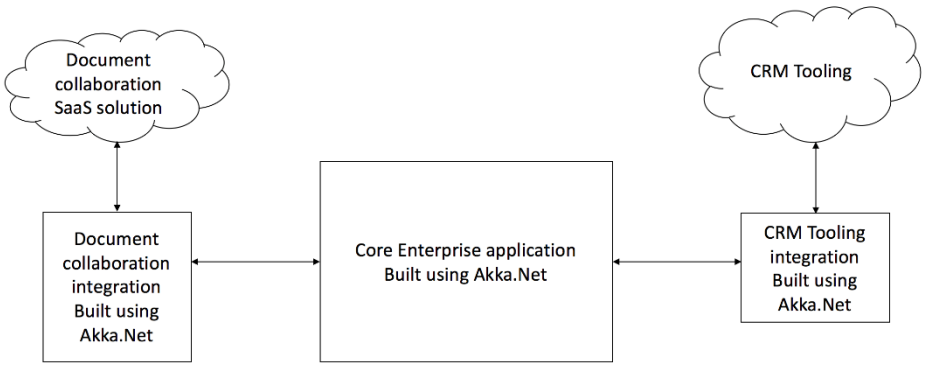


Figure 8.7 Integrations are usually complex, requiring large amounts of custom logic. By developing each integration as an individual application you're able to dedicate resources to it, while Akka.NET remoting allows you to bring the integrations into the core application.

By using the remoting capabilities of Akka.NET, you can simplify the integration of applications created by other teams, allowing you to focus on the development of the core business logic that powers the application rather than tedious efforts to effectively integrate the application.

8.8 Summary

In this chapter, you learned:

- How to join two actor systems in order to start composing larger systems
- How to create secure actor systems using receptionists and authorisation
- How to create scalable systems across a network

9

Testing Akka.NET actors

This chapter covers

- Designing unit tests for verifying the functionality of individual actors
- Creating larger test cases that test the interaction between multiple actors
- Verifying the functionality of distributed actor systems through multinode tests

Throughout the book so far, whenever we've seen the Reactive Manifesto, we've seen that the ultimate outcome is to provide the end user a better experience. We've seen that the services and applications you build should remain responsive even in the face of failure or scalability issues. This is all in an effort to provide the best possible experience to the users of your applications and ensure that they are happy at all times when using your applications. But there is more that is required to create a pleasant experience for users than maintaining responsiveness. You have to try to ensure that whenever users are using the system, they're not faced with an abundance of bugs and issues preventing them from using the application to its full potential.

We saw when considering an e-commerce website that large numbers of users are likely to be visiting the website simultaneously and potentially causing significant amounts of stress on the services, which may lead to failure. We saw how these are the types of scenario you should prepare for in order to try and ensure that users are able to make purchases, especially when more users want to purchase items. But even if you've built a system that is able to automatically scale to the increased load being imposed upon the service, there still may be scenarios that prevent users from being able to make a purchase. For example, as part of the e-commerce website, users are likely to add items to a shopping cart before going through a checkout to complete their purchase, but if there is a bug in the underlying logic powering the shopping cart service, then users won't be able to add new items or complete their purchase, thus leading to frustration and, ultimately, lost sales.

In order to attempt to prevent these issues, we typically write tests that exercise the underlying system with input data and validate that the data output from the system matches the expected value derived from the provided data. Testing plays a key role at all stages of the software development lifecycle, and as such, there are a number of different techniques to use depending upon the functionality that needs to be implemented.

In the world of testing of applications, testing types can be generalized into one of three categories: end to end, behavioral, and unit testing. Each stage of testing relates to the level of depth to which you want to validate the functionality of the system.[1] Each of these three stages of testing goes progressively deeper into the system, starting from an initial black box implementation of the system and simply using the application or service as a user might, for example, through a user interface or a web API, progressing down to the individual components of the system and validating the business logic contained within each of these components.

As you develop systems, you typically encounter areas of complex logic isolated within a component. For example, in the shopping cart example, you may have some logic that applies a discount to the items in the cart when a voucher is added to the purchase. This sort of logic is important from a business perspective in that it acts as a way to attract new customers to the site to make a purchase, but you also need to ensure that you only give the discount when the conditions match entirely, to prevent the company from losing money. In this scenario, you should write a number of tests that cover this logic and validate the core functionality of this small component. Due to the large number of potential inputs for this small component, you'll typically have large numbers of these tests to attempt to validate as many possibilities as you can. These tests are typically referred to as unit tests because they test small, isolated units of functionality that are independent of the rest of the system.

As these units grow toward being a more complete system, they interact with other independent units in order to build up the functionality required for a feature. If you consider the shopping cart example, you may also have other components interacting with the discount calculation, such as a loyalty scheme that rewards the buyer for making purchases with a certain voucher code. In this case, there are two components working together: the discount and voucher application logic and the logic responsible for calculating the number of loyalty points the user is due to receive on a transaction. Although you may have tests in place to verify the functionality of the independent components, it's also important to verify that they are able to operate with each other and the requisite information is being shared between the two components. In many cases, these tests group together multiple components in order to test the integration between them and are frequently known as integration tests.

Having tested the internal functionality of the feature, you'll also need to verify that it works in combination with all of the other components of the system, such as external web services and databases. In order to validate this, you'll write more tests that simply treat the system as

[1] This book does not go in depth into topics relating to software testing and quality assurance. This chapter will provide an overview of the basics of unit testing. For a more in-depth reference on the topic, a number of books are available through Manning Publications, the most notable of which include The Art of Unit Testing, 2nd edition, and Specification by Example.

a black box with no testing of the internals and validate the provided outputs match the expected outputs for a number of known inputs. Because these tests cover all aspects of the application, these are commonly referred to as end-to-end tests.

In this chapter, we'll be focusing on how to effectively test the internals of applications written using Akka.NET by writing unit tests and integration tests that test either single actors or multiple groups of actors that are responsible for communicating with each other. Although end-to-end tests are still needed to completely validate functionality, they don't require anything different from the current approaches to writing these tests.

9.1 Introducing Akka.TestKit

We saw back in chapter 3, in the introduction to the actor model, how different it is from the object-oriented architecture you're used to in C#. Because of this, it requires a different mindset in the approach you take when writing tests. When you write a unit test for a typical C# class, it may look something like the following example. In this example, you will follow a typical three-stage approach to testing: arrange, act, and then assert. You will first arrange all of the data that will be used to validate the functionality; in this case, you will create variables to hold the input data and expected data, and create an instance of the system under test with any dependencies that it might need. Following this, you will act and supply the system under test with the input data and receive output from it, which supplies you with the actual value that the underlying system has computed. Finally, you will assert that the actual data retrieved from the system under test matches the data you expected to receive. Although this test is extremely limited, it follows a pattern that you're likely to see in even more complex examples:

```
[Fact]
public void TheCartContainsAProductAfterAddingIt()
{
    var expectedProduct =
        new Product { Category = "Homeware",
                      Price = 3.52M,
                      SKU = "01222456" };

    var shoppingCart = new ShoppingCart();

    shoppingCart.AddItemToCart(expectedProduct);

    Assert.Contains(expectedProduct, shoppingCart.Products);
}
```

This example focused on the data supplied as a result of calling a method, but there are many other techniques you could use to validate the functionality of the system under test. Other techniques you can use include checking any state that the object is storing and validating that its new value matches the expected outcome. There may also be times where you need to rely on introspection of the object and validate that the internal data stored within it matches some expected value.

But as we saw in chapter 3, there are two key differences between the actor model's approach to componentization and the more typical object-oriented approach. The key

differences are that when using the actor model, you don't have access to the actor's internal state, and every operation within Akka.NET is designed to work asynchronously. Both of these present a challenge when you try and test the components, if these components are actors.

After the simple unit test example, we saw that there may be cases where you need to verify that the object has updated its internal state depending upon a method being called. But when using the actor model, actors completely encapsulate all of their state and don't allow access to it from the outside. In fact, when you create an instance of an actor in Akka.NET, you don't even have a direct reference to the actor itself, and instead you have an address that you are able to send messages to. These messages ultimately get routed through to the actor's location in memory. This means that when using commonly available unit testing tooling, you won't be able to validate that the state within the actor has been updated.

We've considered that the way in which we communicate with actors is through the use of message passing. Internally, all of these operations happen asynchronously and will execute when the scheduler picks up the message. This means that the technique used in the example where you executed a method on the system under test and validate that the response returned from it matches an expected value won't work, and instead you need to rely on having a scheduler in place that is capable of routing the messages through to the relevant processing logic. Finally, you then need to consider the effect that time may have on your tests. In cases where you are simply validating the logic, the environment in which the tests are running may be an important factor in whether tests will pass or fail as a result of timing inconsistencies.

The Akka.TestKit package is available as a means of providing effective tooling to help ease the problems we've discussed. The testing tooling is not designed to completely replace your existing testing tooling, but is designed to work alongside it so that your existing tooling is able to test any actors or compositions of actors that you write in Akka.NET. A number of adapters are available that allow the testkit tooling to work with many of the most common unit testing frameworks, such as NUnit, xUnit, and MSTest. The aim of Akka.TestKit is to provide an implementation of `ActorSystem`, which is responsible for scheduling execution and routing messages and was designed to remove the difficulties we've discussed in this section.

There are several features that we'll explore in this section that can be used to help you more effectively test actor-based implementations. The key changes are the ability to spawn test actors that allow you to introspect an actor reference and see its internal state, and the ability to run tests on a single thread, thus preventing any potential race conditions that you might encounter when running tests. Throughout the rest of this chapter, we'll explore some of the other functionality provided by the Akka.TestKit project and how you can test many of the typical scenarios you're likely to encounter as you develop actor-based applications using Akka.NET.

Installing Akka.TestKit

Akka.TestKit is installed in the same way as you have installed previous packages, through NuGet. But, a number of Akka.TestKit packages are available that contain the adapters required to interoperate with the numerous supported

unit testing frameworks that are available. It's important to include this adapter project, otherwise certain functionality won't be available, such as asserting certain behavior.

9.2 Unit testing actors

We've seen so far in this section that there are two concepts relating to testing the actors you write: you have the opportunity to unit test individual actors in order to verify their functionality when simply sending them a message, and you also have the option of integration testing your actors by verifying the message-passing components work together and messages are sent correctly. In this section, we'll look at the functionality of Akka.TestKit and how to use it to verify the behaviors associated with your actors. Throughout this section, you'll be using the xUnit-based test runner to execute your tests, but the same concepts apply to the other adapters; the only changes you'll need to make will relate to the usage of, for example, any asserts that will depend on the testing framework being used. The reasoning for using the xUnit-based runner will become apparent later in the chapter as we look at other testing methodologies within Akka.NET.

This section will focus on the basic unit testing of actors. This is particularly useful for scenarios where you simply want to test the internal business logic of these actors and validate that they correctly respond to the messages you send them. In order to do this in a more deterministic manner, the Akka.TestKit is able to spawn new actors and run them through a single threaded scheduler that simply executes all of the actors synchronously. This ensures that you no longer need to worry about cases whereby time may affect your tests. Throughout this section, we'll see more about the benefits that these features bring.

9.2.1 Spawning test actors

Throughout the book so far, whenever we've discussed the design of actors, we've consistently seen that actors should be designed to operate on the smallest unit of concurrency possible for that component. This is likely to contain some amount of business logic as part of the application, which needs to be verified as being correct. Thinking back to the shopping cart example in this section, you might have one shopping cart per user's session as they browse the e-commerce section. You might then choose to represent them as an actor instance per cart. This means that you're going to have some logic involved when performing operations such as adding new items to the cart. In this case, you need to validate that the item is in the cart after it's been added. You might also have an option to apply a discount code that leads to an updated price for the items contained within the cart.

As part of your aim to provide users with the best possible experience, you want to ensure that these pieces of functionality work within the actor. In order to prove that they do work as expected, you need to be able to create an actor and send it messages that it might be expected to receive in real-world usage. Akka.TestKit allows you to spawn a new form of an actor reference. So far, we've seen local actor references and remote actor references; you now have the ability to spawn test actor references, which are specialized for use in testing.

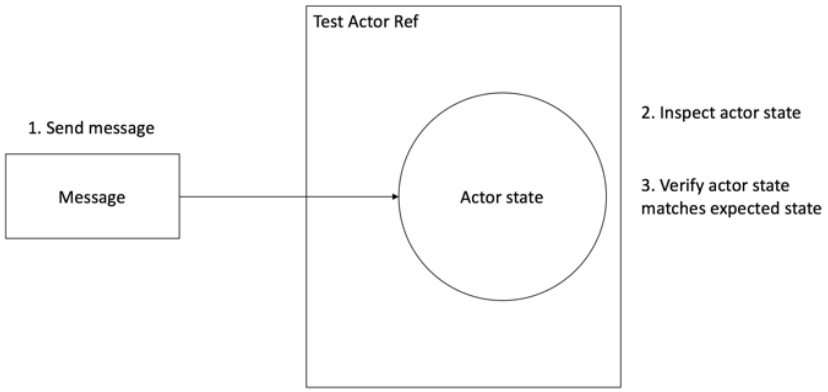


Figure 9.1 - Test actor references allow you to inspect actor state, thus allowing you to verify the actor behaves as expected whenever it receives a message.

For example, if you had an actor for the checkout we've seen in this chapter, you need to write a test for it that validates that whenever it receives a message telling it to add an item with a given quantity, the checkout updates to reflect that. You can retrieve a test actor by simply creating an instance of `TestActorRef`, as in the following example. This `TestActorRef` requires no backing actor system through which messages are routed, and instead, the receive handler method of the actor is called directly whenever you send it a message:

```
var testActorSystem = ActorSystem.Create("test");
var actor =
    new TestActorRef<ShoppingCartActor>(
        testActorSystem,
        Props.Create<ShoppingCartActor>());
```

`TestActorRef` are designed to be lightweight tools for simply testing the internal logic of the likes of receive handlers, and they allow you to write tests that exercise the individual actor's logic and run quickly without the need for the overhead of an actor system. These benefits ensure that you can quickly run tests on actors to validate that the functionality remains consistent even when you may end up refactoring the internal logic. But due to the synchronous nature of `TestActorRef`, you do limit the potential you have for testing. For example, there are certain actor types that rely on asynchronous communication to operate. Although the actors you've seen so far won't cause problems, the actors you'll see in later chapters will prevent you from using synchronous actors.

9.2.2 Validating internal data

We've considered that one of the core examples for unit testing an actor is to assert that the internal state of the actor has been updated to the new state having received an actor. The

actor model, however, provides guarantees that you can't see the internal state to provide a level of safety in order to prevent any concurrency issues. But one of the differences between `TestActorRef` and the other actor references you've seen so far is the ability to retrieve the `actor` instance that the reference is pointing to. This allows you to directly access fields and properties stored in the underlying `actor` instance. Accessing the actor is simply a case of calling the `UnderlyingActor` property on `TestActorRef`. This then returns an instance of the `actor` object and allows you to use it as a regular instance and access data or call methods on it. In the following example, you will retrieve the underlying `actor` for the shopping cart `actor` reference you created earlier:

```
var underlyingActor = actor.UnderlyingActor;
```

By having access to the underlying `actor` instance, you can perform assertions on the internal state and validate that the state has correctly updated given a new state. Here, you can see an example of a test that sends a message to the actor reference and then validates that the actor's state has been updated to the correct number of items as you sent in the message. Thanks to the synchronous nature of `TestActorRef`, you can access the state immediately after sending the message. If you were writing tests in an asynchronous manner, then you would have to implement some means of periodically checking whether the state has been updated, leading to a certain amount of non-determinism, which is not what you want when writing test code.

```
 [Fact]
public void TheCartAcceptsAProduct()
{
    var testProduct =
        new Product
        {
            Category = "Homeware",
            Price = 9.99M,
            SKU = "0122224678"
        };
    var testActorSystem = ActorSystem.Create("test");
    var actor =
        new TestActorRef<ShoppingCartActor>(
            testActorSystem,
            Props.Create<ShoppingCartActor>());
    var underlyingActor = actor.UnderlyingActor;

    actor.Tell(new AddProductToCart(testProduct, 1));

    Assert.Contains(testProduct, underlyingActor.Products);
}
```

As this example showed, you can use the features provided by your preferred unit testing framework of choice, meaning you can essentially write any unit test in a process similar to how you currently write tests. This ensures that you can work in the way that best suits the task at hand, whether that's using tests to validate existing functionality or writing tests first and then developing features.

9.2.3 Testing FSMs

In chapter 4, we saw the importance of finite state machines (FSMs) and how they enable you to more easily develop actors consisting of multiple states. We also saw the specific `FSMActor` type, which allows you to develop more complex FSMs than you might typically manage using the simple switchable behaviors. If you want to test these actors, Akka.TestKit provides a test actor designed to allow the validation the current state of the actor:

```
var testActorSystem = ActorSystem.Create("test");
var actor = new TestFSMRef<TurnstileActor, ITurnstileState, ITurnstileData>(testActorSystem, Props.Create<TurnstileActor>());
```

In a similar way to how you generate the regular `TestActorRef`, you simply need to create `TestFSMRef`, passing in your `FSMActor` to test. Back in chapter 4, you created an example of an FSM, which you used as a means of controlling access through a ticket barrier. Using the FSM testing tooling available within the Akka.TestKit project, you can now create a test for it that is responsible for validating the transitions within that FSM. For example, you might want to ensure that the barrier does not shut if you send it two messages of the same type within a short period of time. In order to do that, you can create a `TestFSMRef` with your `TicketBarrierActor` and start sending it messages.

Having sent the actor a message, you're then able to get access to certain extended properties within the FSM. `TestFSMRef` provides two core properties, `StateName`, which is used to uniquely identify the possible states, and `StateData`, which is where the states are able to pass data between states. By using these properties, you can validate that your internal logic is functioning correctly whenever you send a message to the actor. In the following example, you will send your previously created `TestFSMRef` two of the same message and validate that the state remains `Unlocked`:

```
var expectedState = Unlocked.Instance;
var testActorSystem = ActorSystem.Create("test");
var actor = new TestFSMRef<TurnstileActor, ITurnstileState, ITurnstileData>(testActorSystem, Props.Create<TurnstileActor>());

actor.Tell(new TicketValidated());
actor.Tell(new TicketValidated());

Assert.Equal(expectedState, actor.StateName);
```

By providing a means of seeing the internal state of the FSM within the actor, you are left with a powerful means of testing the potentially complex state machines you may be dealing with when you create larger and more involved actors that may exist in a large number of states. This ability to see the internal state of an FSM provides significant benefits over the simpler receive handler-based approach due to the limited possibilities for seeing the current receive handler in these actors.

9.2.4 Unit testing actors summary

Because many of your actors will feature some internal business logic, the simple `TestActorRef` unit tests allow for a significantly lower overhead. This ensures that you can

write sufficient tests to validate that the internal logic works as expected, and you can ensure the overall quality of the code you have written.

9.3 Integration testing actors

Although it is important to test the logic of each actor within the system independently in order to validate their functionality, you also need to consider how actors interact with each other. For example, an actor is likely to need to communicate with other actors and send messages, this being one of the core operations of an actor, as we saw in earlier chapters. Due to the asynchronous nature of the messages being passed around the system, you can no longer rely on the synchronous testing framework you saw previously, and instead, you now need to work with a dedicated asynchronous testing framework. Akka.TestKit provides such a tool in order to spin up a lightweight test actor system with an increased number of inspection points in an effort to assert that your actors are communicating correctly. In this section, we'll see how to write tests that validate that an actor is sending the correct messages to external actors upon receiving messages internally.

9.3.1 Creating test specifications

In order to write an integration test, you need to create a specification that provides a number of tools that you can use to validate whatever was sent through the actor system. For example, you get access to a test actor system that allows you to spawn an actor just as you would with a normal actor system. You're also provided with the tools that you need to validate that the correct messages have been received. We'll explore these features in more depth throughout the rest of the chapter.

In order to create a specification, you simply need to create a class that inherits from the `TestKit` class within Akka.TestKit. Any tests you write that use the actor system should inherit from this class. Tests are then specified by creating methods that are marked with the `test` attribute for your unit testing framework of choice. For example, in the case of xUnit, which you're using throughout this chapter, you need to mark your test methods with the `Fact` attribute. The following example shows how to define a simple test. Once the project containing the test code is compiled, the test runner for your unit testing tool will detect the tests and run them:

```
public class ShoppingCartIntegrationTesting : TestKit
{
    [Fact]
    public void PricingEngineComputesCorrectPriceForAShoppingCart()
    {

    }
}
```

Once you have the test infrastructure in place, you can write some tests. The first thing you'll need to do is to spawn an actor, which requires the use of an actor system. Within the Akka specification class, you're provided with an actor system through the Sys property on the

base class. This actor system works in the same way as the actor system we've seen throughout the book so far, and allows you to perform common operations, such as spawning actors with `Props`, as we saw in chapter 3.

```
var pricingActor = Sys.ActorOf<PricingActor>();
var shoppingCartActor =
    Sys.ActorOf(Props.Create<ShoppingCartActor>(pricingActor));
```

Having spawned an actor, you can send messages and communicate with that actor. Throughout the rest of this section, we'll see how to verify the information that gets sent out of the actor.

9.3.2 Asserting message responses

Having sent a message to an actor, it will now have its processing logic invoked by the host actor system, and then you can observe its external effects. One of the most common outcomes you're likely to encounter is that you expect to receive a message in response to your original message. In the case of the shopping cart actor, you may want to show the total cost on a webpage. This means that you need to communicate with the shopping cart and retrieve the total. In order to do this, you can send it a message that asks for the total and have the actor respond to this message by sending a message back to the original sender. This is a common pattern for communication with actors that we saw back in chapter 3.

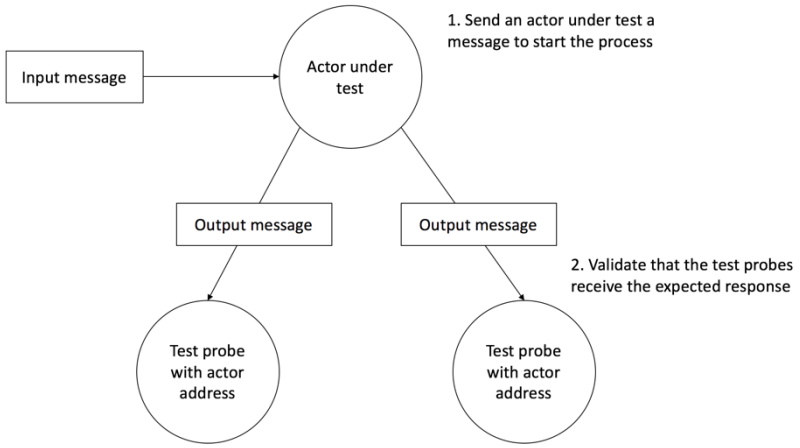


Figure 9.2 - Test probes allow you to verify that actors are receiving the correct messages in response to other messages.

Here, you need to test that the actor is correctly operating within its overall environment by communicating back to the original sender. As part of the integration testing, you don't get to see the internals of the target actor and instead, you're left with a black box system, meaning you can only observe its inputs and outputs. As part of the specification you wrote earlier, in

addition to the test actor system, you also have access to a number of methods that are used to set expectations of what messages should be received within a certain period of time. In the example, if you sent a `GetCartOverview` message, then you would reasonably expect a `CartOverview` message in response to it. In order to achieve this using Akka.TestKit, you can use the `ExpectMsg` method on the object. This will then check the test message queue for an instance of the supplied message. You also need to specify an amount of time within which you should expect to receive the message so that you're not waiting for a message that will never arrive. If the TestKit receives the supplied message, then the test has passed. Otherwise, the test will fail and the test runner will pick it up. In the following example, you can see a case where you assert that when you send a request to calculate the shopping cart total, it returns a response that matches what you expect. In this case, after having sent two messages to store a product in the shopping cart, the total should come to 19.98:

```
var product =
    new Product
    {
        Category = "Homeware",
        Price = 9.99M,
        SKU = "1231214643"
    };

shoppingCartActor.Tell(new AddProductToCart(product, 2));
shoppingCartActor.Tell(new GetCartOverview());

ExpectMsg(new CartOverview(1, 19.98M));
```

As you start to build up more complex actor systems, you're likely to see different usage patterns for how actors communicate with each other. The TestKit provides a number of alternative assertions you can make on the received messages. For example, although the previous example relied on receiving a message that was the same as the one provided, there are occasions where you simply need to validate that the message you have received is of a given type, and you don't care about the internals of it. This is useful, particularly in cases where you can't deterministically compute the value that is set to be contained in the sent message, but you do want to validate the existence of this message. For example, you may not know how to compute the total for your shopping cart and may instead simply want to check that you receive a response at all. To do this, you simply use the generic form of `ExpectMsg`, which takes in the type as a generic parameter.

You also have the ability to use a predicate here to choose to validate certain properties of the message. For example, it may contain a timestamp and a value, and if equality is implemented to include the timestamp, then you can't get the granularity required to ensure that the important properties of the message match the expected values. If the message both matches the supplied type and passes the predicate, then the test passes; otherwise, it fails.

```
ExpectMsg<CartOverview>();

ExpectMsg<CartOverview>(msg =>
```

```
{
    Assert.Equal(1, msg.ItemsCount);
});
```

You might also expect a large number of messages to be received from the actor under test. This may be in cases where it presents a stream of information to the calling actor in the case of a subscription. Alternatively, there may be second actor performing some other action, also causing it to send more than one message. In either of these cases, you can specify that an actor should receive all of the messages supplied. This then allows you to send multiple messages and perform multiple assertions simultaneously:

```
ExpectMsgAllOf<CartOverview>(
    new CartOverview(1, 19.98M),
    new CartOverview(1, 9.99M));
```

There may also be times when you want to ensure that you receive one of a certain set of messages. For example, you may be expecting a pseudorandom response that then opens up the option of receiving one of a range of messages:

```
ExpectMsgAnyOf<CartOverview>(
    new CartOverview(1, 9.99M),
    new CartOverview(2, 14.98M));
```

The final assertion you're likely to make is that you shouldn't receive a message at all within a given time period. For example, we saw in chapter 8 the potential for authorization to certain resources; you may have an application written in such a way that if you fail to provide credentials as you send a request, it won't return anything in response. This is handled through `ExpectNoMsg`, but you also need to supply a timeout within which you should not receive any messages, so that the test does not block indefinitely:

```
ExpectNoMsg(TimeSpan.FromMilliseconds(2000.0));
```

These simple assertions cover the majority of the complex test cases you're likely to encounter as you write integration tests for multiple actors by testing for the message effects that the can see within your application.

9.3.3 Time-based testing

Although the tests so far have simply asserted that you can return a number of messages within a set period of time, you're also likely to encounter scenarios where you have a fixed time within which you need a set number of things to happen. For example, it may be the case that an Akka.NET actor system sits within a much larger API that provides a service-level agreement that all operations will complete within a set period of time. In these cases, you need to ensure that the tests you write are able to validate that these agreements are reachable or will demonstrate that code changes are needed to try and bring the actual times in line with the expected times. In these situations, you're likely to be making more than one call to a number of different actors, each responsible for a small component of the application. For example, in the e-commerce example, you've seen the simple shopping cart, but you'll also

have to deal with product listing services, product recommendation services, search services, and potentially many more, which help to create the overall shopping experience for the customer.

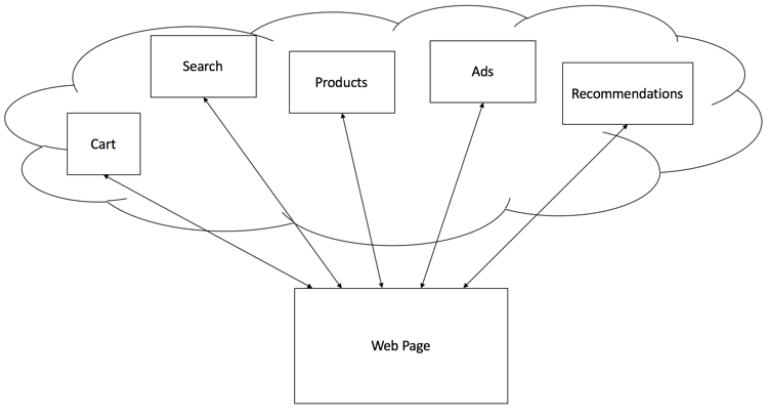


Figure 9.3 - A simple web page on an e-commerce site will communicate with a number of different services before ultimately presenting this information to the user. It's therefore important to verify that all of the requests happen quickly.

As a result of this, it's important to know the impact that a large number of components are going to have on the overall system's responsiveness. In order to cater for this, the TestKit supplies the ability to use a `Within` block. This block provides a timeout within which you should be able to complete a number of steps. If the steps take longer than the timeout, then the test will fail, but if all of the steps complete, then the test was successful. In the following example, you'll integrate with some of the external services you considered earlier in order to build up a response to present to the user. In this case, you'll specify that you should be able to send a message to your shopping cart and get a response back within a couple of seconds:

```
Within(TimeSpan.FromSeconds(2.0), () =>
{
    var product =
        new Product
        {
            Category = "Homeware",
            Price = 9.99M,
            SKU = "1231214643"
        };

    shoppingCartActor.Tell(new AddProductToCart(product, 2));
    shoppingCartActor.Tell(new GetCartOverview());

    ExpectMsg<CartOverview>();
});
```

Many of the tests that we've seen in the previous section have had some degree of reliance on time. Although these tests may succeed when run on a powerful development machine locally, if these tests are run on a build server, then they may ultimately time out due to a lack of available resources. To combat this, you can use dilated timespans instead of a regular time span. By using dilated times, you can provide a scaling factor within the actor system's test configuration. When using the dilated time, the scale factor is pulled in from the configuration and the timeout for each test component is updated as a result. This allows for short timeouts on tests when running locally, as well as a longer timeout for build services that are likely to be a resource shared with many other test runners.

```
Within(Dilated(TimeSpan.FromSeconds(2.0)), () =>
```

Once again, thanks to some of the underlying tooling available within Akka.NET, you can ensure that you're building products and services that are enjoyable for customers to use. In this case, you can accurately test the timings of certain communications between actors and validate that the ultimate experience for the user is responsive, as per the aims of the reactive manifesto.

9.3.4 Test probes

So far, we've seen how to assert that the original sender of the message receives a reply from a given actor, but you're also likely to want to validate that your actor under test sends the correct messages on to other targets. For example, you may need to validate that an actor forwards a message onto another separate actor, or you may want to validate that for a given code path, you send a new message onto a target. As we saw in chapter 3, one of the most common operations actors perform is to send messages onto other actors.

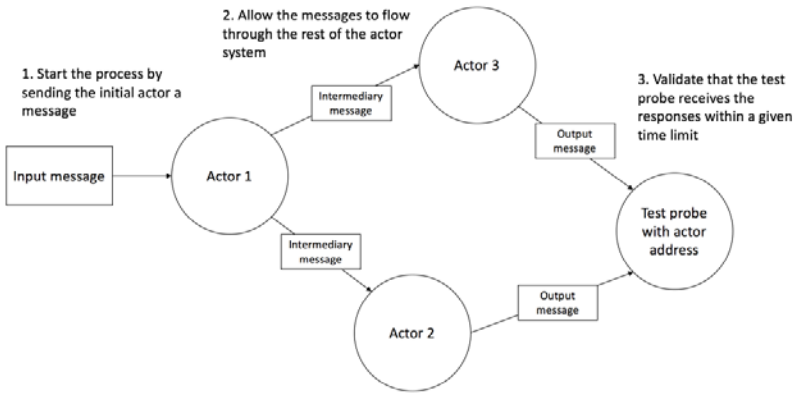


Figure 9.4 - Test probes allow you to validate that even in complex flows of actors, a named target receives the correct messages.

In order to validate that alternative targets are able to receive messages, Akka.NET provides the notion of test probes. Test probes are simply `IActorRef` that you are also able to inspect and verify that it has received certain messages, similar to the previous example, where you were verifying the original sender. In order to use test probes, you first need to create one. This is as simple as calling the `CreateTestProbe` method, which then returns a test probe. If you then want to integrate this test probe with other actors, you need access to the underlying `IActorRef`, which can be accessed using the `Ref` property on the test probe:

```
var testProbe = CreateTestProbe();
```

From here, you can pass it into other actors as a dependency and allow them to communicate with it. The actor under test has no knowledge that it's communicating with a test actor, thus ensuring that the test environment is as close to the production environment as possible:

```
var testProbeRef = testProbe.Ref;
var cartActor =
    Sys.ActorOf(Props.Create<ShoppingCartActor>(testProbeRef));
```

Having created your actor under test with the test probe, you can perform assertions on the test probe to validate that it has received the messages you expect it to. You get access to all of the assertions you saw at the start of this section, allowing you to assert that you've received the correct message, message type, collection of messages, or no messages at all. But you're no longer testing that it's the original sender that's receiving the message and is instead an external actor:

```
cartActor.Tell(new GetCartOverview());
testProbe.ExpectMsg<ComputeCartTotal>();
```

Test probes are a powerful construct, allowing you to create more in-depth tests that start to cover large portions of the actor system hierarchy and ensure that the actors within your system are correctly integrated and communicating with each other. By using test probes, you can continue with black box testing of the system, even when the black box relies on external actors.

9.3.5 Integration testing actors summary

Although many actors can be tested independently and show that they can operate on their own without the need for other actors, in a real-world system, there will be significant interaction between actors. As such, it's important that you test a subset of your system working together to validate that the individual behaviors will work together when faced with the potential difficulties you might see in a production system that may lead to future problems, such as performance problems if messages don't arrive within a specified amount of time.

9.4 Testing distributed applications with the MultiNode Testkit

So far, all the tests we've seen have focused on testing actors that all run within the same actor system, but you should also test the functionality of your applications as you run them across a network. We saw in chapter 8 how to link two existing actor systems together with Akka.Remote and have them communicate over a network. We also saw in chapter 8 how much potential for failure there is once you start to communicate over an unreliable channel such as a network. Given this unreliable channel, there's potential for messages to arrive much later than planned, arrive in a corrupted state, arrive out of the original sent order, or even not arrive at all. These are all potential scenarios that could ultimately lead to a failure of the application or an independent failure of core components within the system. Due to this potential risk, for certain core components it can be beneficial to ensure that these components won't end up in a degraded state as a result of these failures. In this section, we'll look at how to inject failures at the transport level between two applications in order to more effectively test applications that are distributed over a network channel and ensure they maintain resiliency even in the event of uncontrollable failures.

In chapter 6, when discussing potential failure scenarios for your applications, we saw that unreliable channels are one such problem, and we also saw a possible solution for this. Although having a theoretical plan of how to solve these sorts of problems is useful, you should also prove that your plan works in action by testing it, and the MultiNode Testkit provides the ability to achieve this. Testing across the network provides another means of ensuring that the applications and systems work in more possible edge cases. An example of where this level of testing is useful is in distributed database design, where developers need to ensure that data remains consistent and error free even when faced with the presence of unconsidered bugs. Although it may not be the case that you're writing databases yourself, you do need to verify certain aspects of the systems you write, such as their ability to return valid data for 99.9% of requests, a requirement that may typically be included in a service level agreement. In these cases, you need to ensure that your systems focus on high availability as a core tenet. This means having redundant services in place to cope with failure and appropriate routing to deal with failures in the code. These tasks prove to be a core component of any high availability service and should be tested to ensure that they will work and will handle these network failures which we've seen so far.

In addition to the components we've seen so far for testing, Akka.NET also provides an additional test kit known as the MultiNode Testkit. The MultiNode Testkit provides a simple way of starting up multiple connected actor systems through Akka.Remote all running on the same machine, but communicating as though they were all separated across a network. Using similar assertions to those we saw earlier in the chapter when discussing integration tests, you can verify that your actors are able to continue to operate even in the case of network difficulties. The examples we've seen so far on integration testing have focused on the xUnit test kit. This is because the MultiNode Testkit is an extension of the xUnit-based test kit, meaning that you're only able to use xUnit as your testing library.

The MultiNode Testkit consists of two key components that form the basis of the tests you write; the test conductor and the specifications. The test conductor is responsible for orchestrating the work done under the hood by the test runner. This means that it provides support for tasks such as running functions on individual machines, providing barriers to allow tests to operate at the same pace, automating the process of finding the addresses for each of the actor systems, and it also acts as a centralized location where you can inject failures into the network layer deterministically. The specifications are designed to describe the details of the test, providing a means of telling the test conductor what operations it should perform on each independent actor system that is being tested. Throughout the rest of this section, we'll look at how to use these components to test the network resiliency of the actor systems you write and understand how they work under the presence of unreliability.

9.4.1 MultiNode specs

Tests written for the MultiNode Testkit consist of two key components: the test conductor, which is responsible for running the tests, and the test specifications, which describe the behavior that drives the test along with checking for expected behavior. The specification (typically shortened to spec when writing tests) provides a means of describing the overall structure of a test, including the deployment of actors across multiple actor systems, entering barriers, and inducing failures into the network, as well as a means of using the assertions that we saw in the previous sections, such as validating that you do receive messages.

Although the specifications are similar to those we saw when we looked at integration testing, there are some notable differences, which will become clear as you write spec files in this section. A test specification has three main sections: the common configuration shared between the actor systems used in the test, the specification detailing the operations that the test should perform, and specific configuration modifications that may be needed on a per-actor system basis. In this section, we'll look at how to write a simple specification that is designed to demonstrate a simple ping pong between multiple actors on separate actor systems, and what happens when you end up losing a network connection.

When writing a multinode test, the first task is to create a configuration that will be shared by all of the actor systems. In order to do this, you create a subclass of `MultiNodeConfig`. Here, you can create any HOCON-based configuration that your test needs and specify the names of the actor systems to use. In the following example, you tell the test conductor that you will be using two actor systems, referred to as roles within the MultiNode Testkit. If you wanted to add some configuration to be shared by all of the actor systems, then you simply need to assign it to `CommonConfigurationProperty`. For example, in the following example, you use the provided debugging configuration, which provides enhanced logging to aid with debugging across all your actor systems:

```
public class MultiNodeGuaranteedMessagingConfig : MultiNodeConfig
{
    private readonly RoleName _first;
    private readonly RoleName _second;

    public RoleName First { get { return _first; } }
```

```
    public RoleName Second { get { return _second; } }

    public MultiNodeGuaranteedMessagingConfig()
    {
        _first = Role("first");
        _second = Role("second");

        CommonConfig = DebugConfig(true);
    }
}
```

Having created the configuration for your test, the next step is to create the actual test that is going to be executed on each of the actor systems. The test is created as an abstract class that inherits from `MultiNodeSpec`. We'll see why it's abstract soon. You also need to pass your configuration object up the class hierarchy to `MultiNodeSpec`, which you can see here:

```
public class MultiNodeGuaranteedMessagingSpec : MultiNodeSpec
{
    private readonly MultiNodeGuaranteedMessagingConfig _config;

    public MultiNodeGuaranteedMessagingSpec()
        : this(new MultiNodeGuaranteedMessagingConfig())
        { }

    public MultiNodeGuaranteedMessagingSpec(
        MultiNodeGuaranteedMessagingConfig config)
            : base(config)
    {
        _config = config;
    }

    protected override int InitialParticipantsValueFactory
        { get { return Roles.Count; } }
}
```

Having created a test container, next you need to write test methods within the class that will be run on every node. In order to specify that a method is a test, you simply add the `MultiNodeFact` attribute, which the test runner will then pick up. Within your test method, you can perform any of the assertions we saw when looking at integration tests. This includes validating that you do receive the correct message, or no messages at all. You can even create `TestProbe`, as we saw earlier as well. This enables you to complement your already thorough suite of integration tests with larger-scale tests covering a greater surface area of potential problems:

```
[MultiNodeFact]
public void MessageShouldBeResentIfNoAcknowledgement()
{
    var pricingActor = Sys.ActorOf(Props.Create<PricingActor>());
}
```

Having created a simple test method, the final step is to create a subclass of `MultiNodeSpec` for each of the nodes that will be participating in the test. For example, in this case, you've been using two nodes and as such, you need to create two classes that inherit

from your custom abstract `MultiNodeSpec`. When you specify that you should run this test, the test conductor will pick up one class per node in which to run the test:

```
public class MultiNodeGuaranteedMessagingSpec1 :
    MultiNodeGuaranteedMessagingSpec
{ }

public class MultiNodeGuaranteedMessagingSpec2 :
    MultiNodeGuaranteedMessagingSpec
{ }
```

Having created a test project with a number of specs, you can then call the Akka.NET MultiNode Testkit runner, passing in a path to the DLL containing the tests as well as the name of a single spec if you want to only run one test instead of all of them contained within the DLL. The test runner is supplied as part of the NuGet package, but it is copied to the output directory of the project with the tests, meaning it is as simple as running the following example. This will then initialize a number of external processes, each of which is running its own independent deployment of the actor system:

```
Akka.MultiNodeTestRunner.exe Chapter9.dll
```

This is the bare minimum needed to set up a new test that is designed to span multiple actor systems. Throughout the rest of this section, we'll look at the other tools available within the MultiNode Testkit that allow for more powerful tests.

9.4.2 Running code on individual actor systems

When you created your test in the previous section, you specified a test method that is invoked by the test runner whenever the test is run. But there is currently no differentiation between which actor systems run which code. As it stands, every actor system will execute the exact same code. This proves to be a problem, as you don't want all of the actor systems to start doing the same thing. You want them to have different responsibilities. As we saw in chapter 8, there's a number of different architectural styles you can use when developing applications that run across a network.

In order to counter that, you can limit the execution of certain functions to selected machines. This means that you can specify, for example, that the first machine should deploy an actor that the second one is able to communicate with. We saw an example of this back in chapter 8 when we had a billing system on one actor system that would be used by other actor systems. In this case, only one actor system should contain the billing actor.

As part of a multi-node test, you can call the `RunOn` method to limit the scope of the function to a selection of machines. For example, in the following example, you specify that you should only run the actor deployment on the node that you called `First`. When this test is executed by the test runner, it will only execute the function if the role it's currently running on matches the role you've specified it should run on:

```
RunOn(() =>
{
    var pricingActor = Sys.ActorOf(Props.Create<PricingActor>());
```

```
}, _config.First);
```

By limiting the execution of your function to an individual node, you can build up more complex tests that more accurately represent the deployment and release scenarios that you're likely to see in a production environment. So, you can design tests that are capable of catching bugs relating to the overall system infrastructure rather than just the code.

9.4.3 Barriers

Now that you've seen that you can run different test code on different actor systems, you've introduced the possibility of a race condition occurring in your tests. For example, let's consider the scenario where node A deploys an actor into its own actor system and node B makes a call to that actor that's just been deployed. Because node B requires that node A has already spawned an actor, if the deployment takes an extended period of time, then the test may fail because it's unable to find the deployed actor. In this case, you need to ensure that you've had a chance to perform any preparation needed before you proceed with the remainder of the test. As a test is run, it can enter a barrier coordinated by the test conductor. Upon entering the barrier, the current test run will block until every other node participating in the current test has also entered the same barrier.

Entering a barrier is achieved through a simple call to the `EnterBarrier` method, which requires a name for the barrier. The name is then used to ensure that all the nodes within the actor system are at the same stage and have entered the barrier. Once `EnterBarrier` is called, no other test steps will be run and the test execution will pause. In the following example, you can see a case where the second node doesn't need to do any test setup and instead waits for the first node to complete its deployment process to create a new actor. Here, the barrier is called `"DeploymentComplete"`. Assuming every node has also entered the barrier with the same name, you can proceed to the next section of the test, which in this case is for node B to send a message to node A:

```
EnterBarrier("DeploymentComplete");
```

Barriers are a quick and easy way to ensure that the distributed tests you write are able to operate together and work at the same pace in order to ensure that you don't encounter any possible race conditions in your multi-node tests that could ultimately lead to occasional random test failures.

9.4.4 Testing for network failure

We saw in chapter 8 that the underlying network powering Akka.Remote can at times be unreliable. This leads to plenty of opportunities for the systems you write to become unreliable. Testing applications that span a network can often be a difficult or laborious process requiring a number of tools, and can even be dependent upon the operating system you use. But as the test conductor powers all of the underlying network routing for the remoting layer of Akka.NET, you can directly modify the network connection by communicating with the test conductor. This

means that you can inject failures at a completely different layer of the application at a point that sits outside of the application.

The test conductor allows you to perform a number of changes to the underlying network that are designed to simulate some of the networking difficulties you could potentially encounter when your systems are running in a production environment. When using the test conductor, you can simulate a node crash and kill a test runner, add throttling to the network channel to slow messages down, and drop messages completely. These are all similar to issues that you might encounter when you deploy your applications into a multi-tenanted cloud environment, for instance. You may have nodes automatically closed in order to perform operating system updates, messages might start to get throttled if a neighboring VM starts a large downloading operation, or messages might disappear on a given route through a network path.

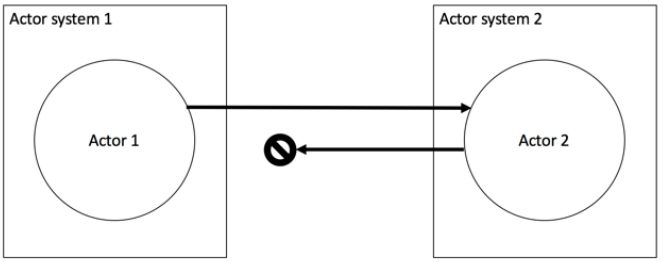


Figure 9.5 - When using the test conductor, you can drop messages sent over a network connection to simulate the effects of a network partition.

Before you can use these features of the test conductor, you first need to configure it and enable them. To enable them, you simply set the `TestTransport` property of `MultiNodeConfig` to `true`. This will then cause the underlying remoting layer to use the fault injecting transports instead of the usual network transports. In this example, you set up the configuration to allow you to test the transport:

```
public MultiNodeGuaranteedMessagingConfig()
{
    TestTransport = true;

    _first = Role("first");
    _second = Role("second");

    CommonConfig = DebugConfig(true);
}
```

Having set up the test transport, you can start communicating with the test conductor and injecting failures into the networking layer. The simplest possibility is to ignore all of the data

that is traveling between two nodes, a feature known as *blackholing* in Akka.NET. This allows you to restrict the messages being sent between a node and drop them either in both directions or in one of receive or send. To start to blackhole messages, you simply call the `Blackhole` method on the `TestConductor` property, specifying the source role, target role, and the direction of message loss. In the following example, you specify that the transport should drop any messages traveling between the source and the target:

```
TestConductor.Blackhole(_config.First,
                        _config.Second,
                        ThrottleTransportAdapter.Direction.Both);
```

Sometimes, you need to allow only a certain number of messages through per second. For example, your actor system may be communicating over a low bandwidth connection, such as a 2G or 3G mobile connection or a slow broadband connection. In these cases, you need to validate that you're not sending so much data that it gives the impression that a remote node is unable to communicate with you. In this case, you can set `TestConductor` to only allow through a certain number of messages per second, calculated by the total size sent over the network. For example, you may have a limitation of 500 Kbps of bandwidth. By using the test conductor, you can limit the communication between two nodes to be 500 Kbps. In the following example, you specify that the communication between node 1 and node 2 should be limited to that of a slow 3G mobile connection, in this case, roughly 0.5 Mbps:

```
TestConductor.Throttle(_config.First,
                       _config.Second,
                       ThrottleTransportAdapter.Direction.Both,
                   0.5f);
```

Finally, you may want to simply make a node disappear entirely. This maps onto cases where machines are shut down immediately without the chance for finalization, such as in the event of a hardware failure, the host operating system restarting, or even the host runtime encountering an unrecoverable error. In order to emulate this, you can simply tell the test conductor to shut down that node. This will then remove the node from all the connected nodes lists, ensuring that any other barriers are then passable:

```
TestConductor.Shutdown(_config.First);
```

These three basic cases cover the majority of the issues that you're likely to encounter that involve the networking layer. By using them, you can ensure that systems are able to endure any difficulties that they're likely to encounter when they run in a production environment. The test conductor proves to be a very valuable tool in gaining assurances that your systems are truly resilient and able to stand up to even the harshest environments.

9.4.5 MultiNode Testkit summary

The MultiNode Testkit provides an incredibly powerful tool that allows you to test your applications in even greater integration cases. Although many test environments replicate production environments, they're unlikely to see the same sorts of issue you might encounter,

but the MultiNode Testkit used as part of Akka.NET allows you to simulate some of the most obscure issues you are likely to encounter and may even overlook during the testing stage of development, but that may have catastrophic outcomes in the event that something does go wrong.

9.5 Case study: testing – test-driven development – unit testing

For the majority of software developers, unit testing and integration testing play a key role in the development process, providing developers with an automatable means of verifying that their application correctly follows the behavioral specification. This use of unit testing has led to development practices such as test-driven development, following the principle of creating a test that probes the new functionality by providing a set of known inputs and matching them to expected outputs. Having created a test, you can then complete the internal logic, which causes the test to subsequently pass.

As part of a typical application in the .NET world, an application will be made up of two projects; a project that contains the actual application logic, and a second project that contains the tests. These tests then reference the application logic and probe it with the known inputs and verify the outputs match the expected values. A normal project will contain enough tests that any significant or complex logic will be covered by a number of tests. Similarly, there will also be tests that verify that multiple components work together correctly and ensure that the integration of subsystems is correct. These are known as integration tests.

By using the Akka.NET test kit outlined in this chapter, you can define tests as part of your development process that work together with any existing tests that you might have. In the following diagram, you can see how to structure a typical solution that uses multiple projects. There's a production project that contains the definitions of your actors. This will be the project that is deployed into the production environment and is responsible for performing the underlying business logic. In addition to the production project is a test project that contains a number of unit and integration tests. When changes are made to the production project, the test project is run through a test runner, and if any tests fail then it can be assumed that the change has broken any application logic that was previously written. This validates that you don't ship broken software to your customers.

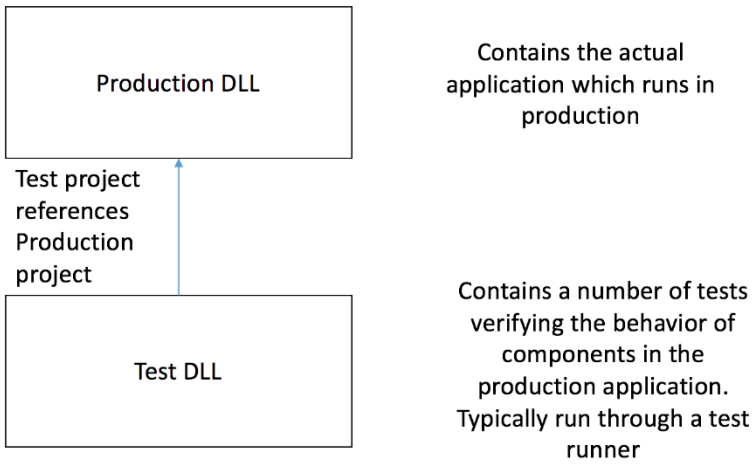


Figure 9.6 - A typical test project structure is very simple but it offers the opportunity to easily reproduce a known test case to check for potential regressions.

By using the Akka.NET test kit, you can use the common test-driven development practices that have become prevalent in other areas of .NET software development. Due to the test kit's integration with other test runners, you can also choose to integrate this as part of an automated system that prevents the deployment of code to a production environment in the event that any changes lead to software regressions, potentially introducing bugs.

9.6 Summary

In this chapter, you learned:

- How to design a unit test that tests the functionality of a single actor
- How integration tests that verify how multiple actors work together can be developed
- How you can test the correctness of distributed algorithms using the MultiNode Testkit

10

Integrating Akka.NET

This chapter covers

- **Designing custom protocols using Akka.NET to receive and send data**
- **Embracing a fully reactive application stack by integrating with real-time connection mechanisms**
- **Adding a web API frontend to allow web applications to communicate with an actor system**

All of the examples we've seen in the book so far have contained the actors within the actor system and not exposed them to the world. As you build applications based on actors, you'll probably need to be able to consume the data stored within them. When it comes to the technologies which are able to consume the data stored within the actor systems, you've got huge amounts of potential data sinks. The data from within these actor systems can end up being forwarded down to individual Internet of Things devices in order to enable command-and-control-style scenarios: you may be sending data down to mobile apps and games to automatically refresh the information on user's devices, you may be sending the data down to video game consoles to provide multiscreen experiences to gamers, you may be sending data to other cloud services that are responsible for other utilities, or you may need to make the data available to websites. Ultimately, in all of these cases, you need to present the information stored within your actor system in the most appropriate manner for each of the individual services you have available.

We saw in chapter 8 when looking at the capabilities of Akka.Remote that you can expose your actor systems over the internet by allowing them to listen on a specific port and accept incoming communication from other actor systems. This certainly has some key benefits, most notably in its simplicity, but it leaves you in a difficult position. In chapter 8, we also saw the recommended practice when it comes to security, which is to only allow access to Akka.NET actor systems from behind a firewall, thus preventing you from exposing it to the whole

internet. But there is another reason why you are unlikely to want to allow communication through actor systems. In all the cases we've seen where you want to integrate with an actor system, you're unlikely to be in a position where you can run an Akka.NET actor system.

Let's consider the example of an Internet of Things device. When you consider the usage of such a device, you're typically looking at low-powered hardware that is designed to be as energy efficient as possible in order to save battery life for as long as possible. Akka.NET is not optimized for these low-powered devices as it needs the likes of a full CLR virtual machine in order to execute the .NET application. This is a lot of overhead needed to execute on something that's low-powered. You also need to consider other issues as well, notably with the transport and integration with other third-party tooling. In the world of the Internet of Things, MQTT has become a standard for simple communication between devices. By using Akka.NET with its remoting capabilities, you're potentially excluding your device from communicating with other manufacturers' devices.

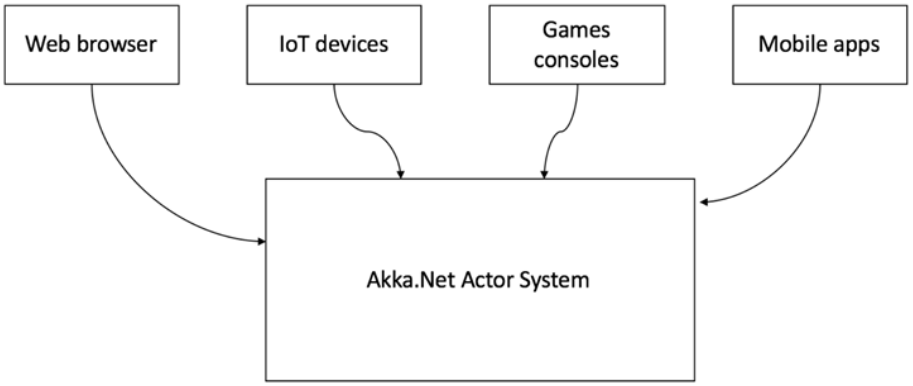


Figure 10.1 - An Akka.NET actor system may need to receive input from a number of different devices all using different technologies.

You also need to consider the supporting libraries available on the client machines. Although some machines may be more powerful and able to support a fully functioning virtual machine, you may be unable to run this sort of codebase due to security permissions on the client's device. For example, an iOS mobile device won't allow you to run a just-in-time compiler, and instead you have to run a full ahead-of-time compiler on the application. This means that you're not able to run a full virtual machine within your application, and this means you need special tools to ensure the abilities of certain tooling.

Another consideration is in the languages used to access the actor system. Akka.NET actor systems are only able to communicate with other Akka.NET actor systems, which forces your client application to be written in C# or F#. There are many cases where this is not possible. One such example is in video game development, which for many involves the use of low-level languages such as C and C++ in order to make the most of the hardware and get the best

performance possible. By using C or C++, these game developers would not be able to communicate with the backend systems.

Finally, you should ensure that the tools you're using to perform interoperation are appropriate given historical and likely future tooling. An example of this is the use of HTTP in web browsers. Although we saw earlier in the book how much the backend of the World Wide Web is changing to move to a more reactive model, communication between the user's web browser and the backend service has become fairly standardized, with communication being based on HTTP. As more and more people focus on using web applications as a means of delivering content to users, this means that you need to support access to the underlying data in a way that is both consumable and manageable.

All of these examples have left you in a situation where you need to be able to expose your actor systems in such a way that they're consumable from the widest range of devices using the most appropriate protocols and technologies. As it stands, although you may have systems that are capable of generating responses in milliseconds and creating enjoyable experiences for your end users, they're not able to access them. As such, another component of an actor system deployment you need to consider is how to design them so that they're able to communicate with other systems, applications and devices.

In this chapter, we'll see how to integrate a new Akka.NET application with a number of different protocols and technologies you're likely to be using in a pre-existing .NET application. This allows you to put incredibly simple integrations in place that are capable of allowing access from as many different devices as possible. We'll see how to create actor systems that can be accessed through HTTP using your preferred web framework; in this case you'll be using ASP.NET, but you'll be able to follow the same guidelines and use any of the alternatives, such as NancyFX, ServiceStack, Suave, or many others. We'll also see how to create more reactive services that run on the web by integrating with web sockets through the use of SignalR. Finally, we'll also see how to integrate low-level TCP socket connections into your application so that you can build protocols up yourself on top of the underlying socket connection.

10.1 Integrating with ASP.NET

Over the past twenty years, the World Wide Web has changed significantly. Originally intended as a means of sharing research materials between academics, it's evolved into a tool that many people rely on as a means of communication, entertainment, and many other applications. As a result of that growth, web browsers have changed and grown in functionality to the extent that it's now possible to create applications with a native feel that are delivered through the internet directly to the user's web browser, all thanks to the power of HTML, CSS, and JavaScript.

Despite all the changes that users are able to see in the frontend of the application, all of the communication with the backend has essentially followed the same pattern of HTTP-based communication. Although the HTTP specification has seen some minor changes, it has for the most part stayed consistent. This level of consistency and standardization presents some clear benefits for those looking to integrate their application over a network. The majority of devices

provide support for either a web browser or, at the very least, a set of tools for performing requests to an HTTP API. This level of availability has led to HTTP becoming the de facto transport of choice for many applications.

Within the .NET ecosystem, there is a wide variety of web servers designed for writing applications designed to communicate over HTTP with clients. In addition to the Microsoft supported ASP.NET web framework, other options have been built by the wider community, including the likes of NancyFX, ServiceStack, and Suave. All of them have their respective merits and all can be used with Akka.NET, but in this section, we'll be looking at Microsoft's ASP.NET framework due to its wide usage. The same techniques used in this section are also applicable to the other frameworks, and as such, the examples can be ported across.

Integration with ASP.NET builds upon one of the concepts we saw back in chapter 3, where we saw that you can send a message to an actor and then wait for a response by Ask-ing an actor in your actor system.

Assuming you have an application already set up[1] and configured to get the basic features required to build an ASP.NET based website, you can add all of the components required to use Akka.NET. As before, you need to add the references to the pre-requisite components of Akka.NET, for the most basic Akka.NET application, this simply relies on installing the Akka package from NuGet exactly as we did in chapter 3.

Before you can use Akka.NET, you first need an actor system to be set up and configured to be run within your application. We saw in chapter 3 that you should only have a single actor system deployed per application which contains all of your actors. In order to make an actor system available across your entire application and ensure that the actor system runs for the entire lifetime of your application, you can use one of two different approaches:

- Global.asax – you can choose to initialize your actor system within the Global.asax file and make it accessible to all parts of the application with a static property. By using the Global.asax file, you can also register to certain events raised by the underlying ASP.NET framework or IIS host. For example, you can listen to events that notify that the application is due to be shut down or restarted. This then allows you to shut down the actor system and save the state of the actors within it in preparation.
- Startup.cs – in addition to an IIS host for your application, you can also choose to use one of the OWIN compatible hosts, which allows you to change the underlying server quickly and easily. This removes the possibility of using the Global.asax file, which is tied to IIS. But as an alternative, you can store data in the OWIN environment, which is then accessible from anywhere in the application.

In this example, you'll be using the Startup.cs option due to its cross-platform capabilities and simple usage. OWIN is a simple contract that .NET web servers can implement that then allows applications to be moved between different hosting technologies with minimal effort. For

[1] This book focuses on features relating to Akka.NET and as such won't go in depth into features related to ASP.NET. For more information on ASP.NET including getting started tutorials, documentation and more then please visit http://asp.net or the ASP.NET in Action series available from Manning.

example, an application written using the OWIN middleware can be targeted at a self-hosted version for local development and then deployed onto an IIS instance when in a production deployment.

OWIN works as a pipeline in which each component has a subsequent operation that is executed, until it gets to the end of the chain, upon which a result is passed back along the chain. This means that if you want to use your actor system in later stages, you need to ensure that you've initialized it at the very start of the queue. In order to add an element to the OWIN pipeline, you call the `Use` method on the `IAppBuilder` provided in the `Configuration` method. `Use` takes a function as a parameter that simply takes the environment and the next element in the chain and returns a task. It's in this function where you'll create your actor system and store it in the environment. Having done this, you can access it again from any other step in the chain that follows it. In the following example, you store the actor system in the environment dictionary under the key `akka.actorsystem`. You follow the exact same process for creating an actor system as if you were creating it in a simple console application, which includes the likes of loading configuration in from a file:

```
public class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        var actorSystem = ActorSystem.Create("webapi");

        appBuilder.Use((ctx, next) =>
        {
            ctx.Environment["akka.actorsystem"] = actorSystem;
            return next();
        });
    }
}
```

If you now have an element that follows the actor system creation, you can access the actor system stored within the environment. If your pipeline looks as follows, then anything within the web API step is able to access the actor system by using the OWIN environment:

```
appBuilder.Use((ctx, next) =>
{
    ctx.Environment["akka.actorsystem"] = actorSystem;
    return next();
}).UseWebApi(config);
```

If you now add a controller to your project, you can retrieve the actor system by using the OWIN extension methods that are available. Now that you have a reference to the actor system, you can interact with it just as you would if it was a console application. Even though it will be accessed by multiple threads, you're still safe to interact with the actor system due to Akka.NET's guarantees that each actor will only process one message at a time, but actors can operate concurrently. This makes it ideal for web applications where you might have multiple users with multiple web browsers or applications trying to modify the data stored on the web server concurrently.

You can now start to work with the actor system in your controller. For example, if you're looking to pull data out of the actor system as part of a get request, you can select an actor path and then send it a message with Ask, awaiting the response. Because Ask returns a task and operates asynchronously, you can use asynchronous controller support alongside it. In the following example, you request data from one of the actors within the actor system by sending it a message and awaiting a response with Ask:

```
public class GreeterController : ApiController
{
    [HttpGet]
    [Route("hello/{name}")]
    public async Task<string> GetGreeting(string name)
    {
        var owinCtx = Request.GetOwinContext();
        var actorSystem = owinCtx.Get<ActorSystem>("akka.actorsystem");
        var greeter = actorSystem.ActorSelection("/user/greeter");
        var greeting = await greeter.Ask<string>(name);
        return greeting;
    }
}
```

As you can see, this integration is incredibly simple, especially in small services, but there are some issues that you might encounter if your web service gets popular. We saw in chapter 7 when looking at scaling that the best option for scaling is to create more instances of something rather than scaling up your existing infrastructure. As it stands, every time you add a new web server, you'll add a new actor system as well, each of which are independent. Fortunately, we saw in chapter 8 how to combine multiple actor systems together thanks to Akka.Remote. This ensures that you can set up your actor system as a service that other servers are then able to connect to. Each of the web servers in your example then has a lightweight actor system containing no actors that you use to communicate with the shared actor system. Your architecture then looks as follows, where each web server shares a centralized actor system. All of this is possible using the features we saw in chapter 8:

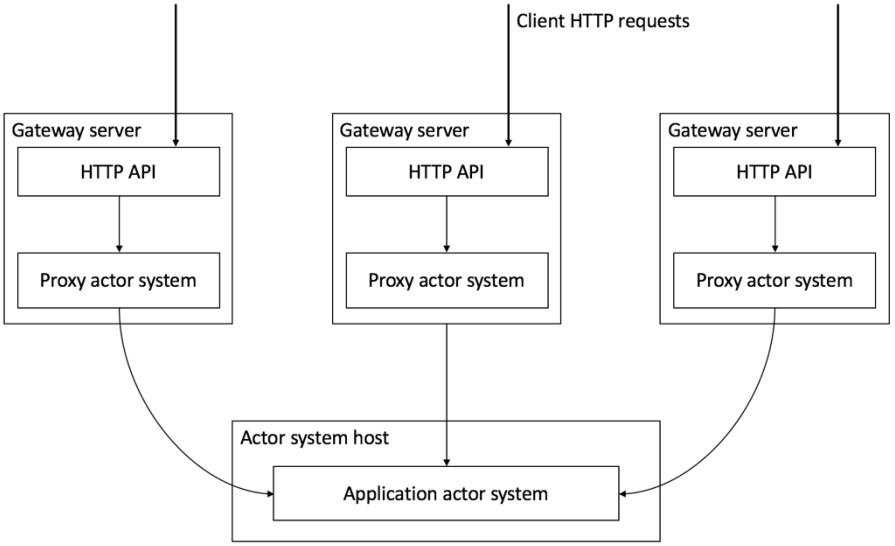


Figure 10.2 - When developing a load balanced web application, you can keep the frontends stateless by having a common actor system that the frontends all communicate with.

By combining your actor system with a web API, you can now access it from the vast majority of devices and systems. Anything that can access your web API or website can communicate with your actor system. You also get the benefit of being able to use the vast array of existing middleware already available for OWIN, ASP.NET, or other web frameworks, thus reducing the possible time spent working on components such as security and using pre-built and thoroughly tested options.

10.2 Integrating with SignalR

Integrating your actor system into your existing applications through a web service affords you some key benefits, such as the ability to access it from almost any device. As a means of data interchange it is relatively static, allowing clients a single-point-in-time snapshot of their data. But in an age where data flows rapidly and users want the most responsive systems possible, you're left having to periodically check whether the service has some updated data for you to use. As part of the Reactive Manifesto, you should ensure that the applications you write are responsive for the users who consume your applications and services. When you defined *responsive* earlier in the book, you saw that it meant that the system was able to react to changes in its environment near instantaneously.

Instead of simply waiting for the user to refresh the webpage, your overall system would be much more responsive if you were able to quickly push changes down to the client without them needing to interact with the application at all. Although this may not have been possible

20 years ago, we've seen that the habits of users have greatly changed, which has pushed further changes through the web browser. One example of a change is the addition of web sockets, which provide a bi-directional persistent channel of communication between the user's web browser and the web server hosting your application. Due to the persistent nature of communication, you can push new messages to the client as soon as they arrive on the server, causing them to potentially update the UI or notify the user of new information becoming available.

One example of this is a situation where you're rendering charts plotting the data being aggregated by your system in real time. In this situation, you can push new readings to the UI, which is then able to append them to the end of the graph providing you with live visualizations of the current state. In this scenario, your actor system may be receiving data from Internet of Things devices that it is then acting on and performing calculations continuously, which ultimately helps you to predict the future direction of readings that you then present to the user.

Within .NET, SignalR is the library most commonly used for providing WebSocket support to web applications and provides a number of abstractions over the top of the low-level WebSocket protocol. The library also provides a number of fallback mechanisms in the event that the user's web browser doesn't support WebSockets. The two abstractions provided by SignalR are persistent connections and hubs, each of which has their own valid use case:

- *Hubs* – Hubs are a means of performing remote procedure calls either from a JavaScript client to a .NET server or from a .NET server to a JavaScript client.
- *Persistent connections* – Persistent connections are essentially a socket over which you can transfer data down to clients by pushing messages through the connection on the server side.

Throughout this section we'll look at how to use SignalR within an Akka.NET application to provide a constant stream of information to clients through the use of persistent connections. Although it's still possible to use hubs, persistent connections more closely match the messaging and usage patterns within an Akka.NET application. In order to use SignalR in your web application, you'll need to add a reference to the Microsoft.AspNet.SignalR NuGet package in the same way you added a reference to the Akka.NET libraries you've added so far.

10.2.1 Communicating through an actor

SignalR provides a fairly advanced level of abstraction over the top of the underlying WebSocket protocol, allowing you to focus on the business logic rather than accepting connections and passing data through these connections. But there's a number of benefits you're likely to see as a result of wrapping your SingalR abstraction within an actor. The biggest advantage is the prevention of concurrency bugs that may occur as clients connect and disconnect and send messages. As the SignalR connection is likely to be used in an ASP.NET application hosted on an IIS server or within a console application, there are a number of threads, all of which can be used to handle incoming connections. This leads to the potential for data races or other concurrency bugs that we saw earlier in the book. By wrapping your

connection within an actor, you can actively prevent these kinds of bug occurring in your codebase. You also gain an advantage in that you can completely integrate your WebSocket connection into your actor system, allowing you to make use of all of the benefits of Akka.NET, notably remoting, routing, and supervision.

You can first define some basic classes that will represent the messages you can process within your persistent connection actor. The messages fall into one of two categories: commands that tell the actor to do something, in this case, you tell the actor to send a message with the `SendMessage` command; and you also have events that inform the actor that something has happened, such as the `UserConnected`, `UserDisconnected`, and `MessageReceived` messages you can see in the following example. This is in line with what we've seen in other chapters where you've created messages to send to actors. You can add a number of properties that reflect the information you'll be looking to store on a per-connection basis. For example, whenever you receive a message, you track the connection that sent the message; or when a new connection is received by the server, you take the username of the connection and the connection identifier.

```
public class ClientDisconnected
{
    private readonly string _connectionId;

    public string ConnectionId { get { return _connectionId; } }

    public ClientDisconnected(string connectionId)
    {
        _connectionId = connectionId;
    }
}

public class ClientConnected
{
    private readonly string _connectionId;

    public string ConnectionId { get { return _connectionId; } }

    public ClientConnected(string connectionId)
    {
        _connectionId = connectionId;
    }
}

public class MessageReceived
{
    private readonly string _connectionId;
    private readonly string _data;

    public string ConnectionId { get { return _connectionId; } }
    public string Data { get { return _data; } }

    public MessageReceived(string connectionId, string data)
    {
        _connectionId = connectionId;
        _data = data;
    }
```

```
}

public class SendMessage
{
    private readonly string _connectionId;
    private readonly string _data;

    public string ConnectionId { get { return _connectionId; } }
    public string Data { get { return _data; } }

    public SendMessage(string connectionId, string data)
    {
        _connectionId = connectionId;
        _data = data;
    }
}
```

Having defined your messages, you can create an actor that is able to respond to these messages and react appropriately. Within the actor, you can maintain any state you want, and it will be safe from any potential race condition-based bugs across all of the threads in the thread pool within which it runs. You may want to store a unique user identifier along with the request so that you can address all of the persistent connections for a given user, thus allowing you to push messages to all of their web browser sessions.

```
public class WebsocketActor : ReceiveActor
{
    public WebsocketActor()
    {
        Receive<MessageReceived>(msg =>
        {
            //Application specific functionality
            //for receiving messages
        });

        Receive<ClientConnected>(client =>
        {
            //Application specific functionality
            //to handle client connects
        });

        Receive<ClientDisconnected>(client =>
        {
            //Application specific functionality
            //to handle client disconnects
        });
    }
}
```

You now have an actor that you can use as the basis for sending messages through a WebSocket connection so that browsers are able to receive direct push-based messages through the web server from the actor system within a web browser.

10.2.2 Connecting to the user's web browser

You currently have an actor that can handle events raised by the SignalR library, as well as handlers that allow you to push data to clients, but you have no means of sending data to clients. In order to counter this, you need to create a SignalR connection that will allow you to interact with the underlying WebSocket.

In order to create this interaction with the underlying socket, you create a class that inherits from SignalR's `PersistentConnection` class. In the following example, you create a simple class that overrides the default behavior for the situations where you receive a message, a client connects, or a client disconnects. Whenever one of these events happens, the method is invoked:

```
public class GraphingConnection : PersistentConnection
{
    protected override Task OnReceived(
        IRequest request,
        string connectionId,
        string data)
    {
        return base.OnReceived(request, connectionId, data);
    }

    protected override Task OnConnected(
        IRequest request,
        string connectionId)
    {
        return base.OnConnected(request, connectionId);
    }

    protected override Task OnDisconnected(
        IRequest request,
        string connectionId,
        bool stopCalled)
    {
        return base.OnDisconnected(request, connectionId, stopCalled);
    }
}
```

Having now created a persistent connection class, you need to be able to create it and host it within the application. This functionality is provided by OWIN as we saw in the previous section. In order to register your persistent connection, you simply add a `MapSignalR` call into your OWIN startup class following the actor system initialization. The following example shows what the OWIN startup looks like in a situation where you have an Akka.NET actor system, SignalR, and MVC within the same project:

```
public class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        var actorSystem = ActorSystem.Create("webapi");

        appBuilder.Use((ctx, next) =>
        {
            ctx.Environment["akka.actorsystem"] = actorSystem;
```

```
            return next();
        }).MapSignalR<GraphingConnection>("graph");
    }
}
```

We briefly mentioned at the start of the previous section that one of the core reasons to wrap your connection within an actor is to ensure that your application is not susceptible to any race conditions it might encounter during its lifetime. SignalR creates a single instance of the persistent connection, but it gets executed as part of a thread pool provided by the application host, which could potentially lead to concurrent invocations of the methods on this class. In order to counter this, you can forward all of the events through to the actor you created earlier. Before you can send a message to the actor, you first need to be able to reference the actor system. By configuring SignalR after you have configured the actor system in the OWIN startup, you can access it from within the OWIN environment, a dictionary of strings and objects relating to the request. This follows the same pattern as we saw in the previous section. You can access the OWIN environment using the `Environment` property on the incoming request on each method you override. Having retrieved the actor system, you can interact with it and send it messages, as we've seen before. In the following example, having received a message, you first wrap the message in an envelope along with the connection identifier and the send it to the actor responsible for the SignalR connection:

```
protected override Task OnReceived(
    IRequest request,
    string connectionId,
    string data)
{
    var actorSystem =
        (ActorSystem)request.Environment["akka.actorsystem"];
    var websocketActor =
        actorSystem.ActorSelection("/user/messagingConnection");
    websocketActor.Tell(new MessageReceived(connectionId, data));
    return base.OnReceived(request, connectionId, data);
}
```

In addition to the scenarios where you receive events from the connection, there are also commands that the actor handles that need to be processed and forwarded on through the connection. In the following example, you can see how to retrieve `PersistentConnection` from within the actor so that you can send messages to a given client identifier in response to a `SendMessage` command:

```
var connection =
    GlobalHost
        .ConnectionManager
        .GetConnectionContext<GraphingConnection>();

Receive<SendMessage>(msg =>
{
    connection.Connection.Send(msg.ConnectionId, msg.Data);
});
```

By wrapping a lot of the behavior you might have seen in the derived `PersistentConnection` class, you can remove the possibility of race conditions occurring within your applications, thus reducing the potential number of bugs.

10.2.3 Integrating with SignalR summary

WebSockets provide a means of creating end-to-end reactive web applications so that users are able to receive updates in their frontend UI immediately after the event has been triggered in the backend systems powering the applications. By using SignalR, you can rapidly build web applications where users are able to directly visualize and manipulate data within your actor system incredibly easily. By combining SignalR with Akka.NET, users can see their application changing and responding to events in real time, allowing them to act upon the information they're presented with as soon as it happens and not as soon as they refresh.

10.3 Custom integrations with Akka.IO

Although the cases we've seen so far have covered a large majority of cases, there are still many other potential systems and devices that are likely to want to connect to your actor systems and send messages. HTTP has clear benefits in its uptake and support across a vast number of devices, but it proves to be a fairly heavyweight protocol due to its reliance on text rather than a simple binary protocol instead. In some environments, this amount of overhead could lead to difficulties. For example, in an Internet of Things scenario where you're sending data through a cellular modem, you need to minimize the amount of data transferred in an effort to bring down costs and also ensure that you minimize the potential amount of data that could be lost. You shouldn't just limit yourself to IoT scenarios where you could see potential benefits, though; if you're interoperating with other custom developed solutions, you may see significantly reduced latencies as a result of using a significantly simpler protocol, which ultimately leads to a more responsive application as per the aims of the Reactive Manifesto.

In these cases, you ideally want a means of providing a low-level connection into your actor system so that you can operate on a socket level rather than having to go through a complex pipeline of operations in an effort to get a request from the user. Instead, you want to receive a message every time the socket receives a message and allow the actor to process the message instead of relying on complex code surrounding sockets, which can be difficult to correctly set up. This allows you to quickly write tooling that relies on more basic or less popular protocols. For example, by using the low-level socket APIs provided within Akka.NET, you can easily create implementations of the likes of DNS servers, monitoring servers, and mail servers, or implement your own protocol tailored to the requirements of your application.

Throughout this section, we'll look at how to use Akka.IO to create a simple socket server designed to communicate with a common metrics collection protocol known as *StatsD*. StatsD is designed to be an incredibly simple API that can receive counters, gauges, and timers sent to it over a network from either TCP or UDP. By bringing this kind of low-level protocol directly within the actor system, you can use all of the key benefits that Akka.NET provides, such as message routing, fault tolerance and on demand scaling. These benefits ensure that you can

react instantly to any load or failure that you might encounter when receiving metrics, an incredibly valuable tool in any operations team for debugging that requires a certain degree of resiliency, even in the face of failure, in order to provide the information you need to try and understand the causes of the failures.

The StatsD protocol is incredibly simple and consists of two core concepts; buckets and values. A bucket is a means of representing a certain metric which you are looking to collect, for example, you may choose to create a bucket called `UserService.Login.Latency` as a way of representing all of the latencies that you've observed by users logging in through the user service. There are a number of valid metrics types you can send, but in this section, we'll focus on three core metrics; counters, gauges, and timers. Counters are used for basic counting tasks; some examples of metrics you may want to count include the number of requests for a web service, the number of times you fail to retrieve a value from a cache and need to talk to a database instead, and the amount of times an exception was thrown. Gauges are for already averaged data. This might include things such as system load or average latency; these are things that are likely to be set once every second and don't fluctuate. The final type we'll look at in this section is a timer, which is used to specify the amount of time an operation took to complete. An example of such an operation might be the latency involved when executing a query against a database or the time it takes between a request being received from the user and a response being sent back to the user.

The underlying StatsD implementation requires a simple line of text to be sent to the socket as part of a single packet sent over the network. The basic structure of the line you need to send to your socket is shown in the following example. As you can see, it's as simple as sending the bucket name, a value for it, and the type of metric it is. For each of these components, you provide a string for each component. These can be anything for the first two parts, and then the final part matches a string representation of each of the metric types. This can then be sent over either a TCP or UDP socket depending upon what the server is configured to listen on.

```
<metricname>:<value>|<type>
UserService.RequestCount:50|c
```

Having created an implementation of the underlying protocol in Akka.NET, you can dispatch the received metrics to anywhere else in your actor system to quickly perform tasks such as data storage, stats aggregation, and notifications. But we'll focus on the actual task of ingestion and how you can map your incoming packets over to messages that can be used within the context of your actor system.

10.3.1 Creating a listen server

Akka.IO provides support for a number of different scenarios based on factors such as whether you should be listening for incoming packets or sending packets over the socket, as well as the underlying socket protocol you should be listening on. Although TCP and UDP support is provided as a pluggable transport within the Akka.NET distribution, you can also create your own underlying transport mechanism to reflect other networking technologies. For

example, you may want to create an Akka.IO transport that can communicate over a pipe rather than a socket, or use a socket but use a different transport protocol than UDP or TCP, such as SPDY. In this example, you'll use a simple TCP socket as the basis of your incoming communications and you'll listen for incoming messages rather than sending outgoing messages over the socket.

Before you can receive messages over a socket, you first need to tell the underlying operating system that you want any messages received on a certain port to be forwarded onto your application. This is achieved by using the TcpManager, which handles all of the underlying bind operations to get the operating system to route any TCP packets through to the application. The TcpManager is made available through an extension method on the actor system itself and is accessed through the TCP extension method on the actor system. In the following example, you get the reference to the TcpManager, which then allows you to bind to a specific port:

```
public class StatsDServer : ReceiveActor
{
    public StatsDServer()
    {
        var tcpManager = Context.System.Tcp();
    }
}
```

The typical approach to building socket servers based on Akka.IO is to have an actor that is responsible for the original port binding and handling the incoming connection requests from the underlying socket before passing the network connection onto another actor, which will deal with the specifics of that one connection. The following diagram demonstrates how to design such a system. There is a single actor that acts as the server and deals with incoming messages from the TcpManager relating to the status of the connection. This includes the likes of binding success or failure messages, incoming connection requests, and socket errors. This server actor then has a number of children that are spawned on a per-connection basis. As we saw in chapter 3, actors are a very cheap abstraction, meaning that you can create millions of them within a single application.

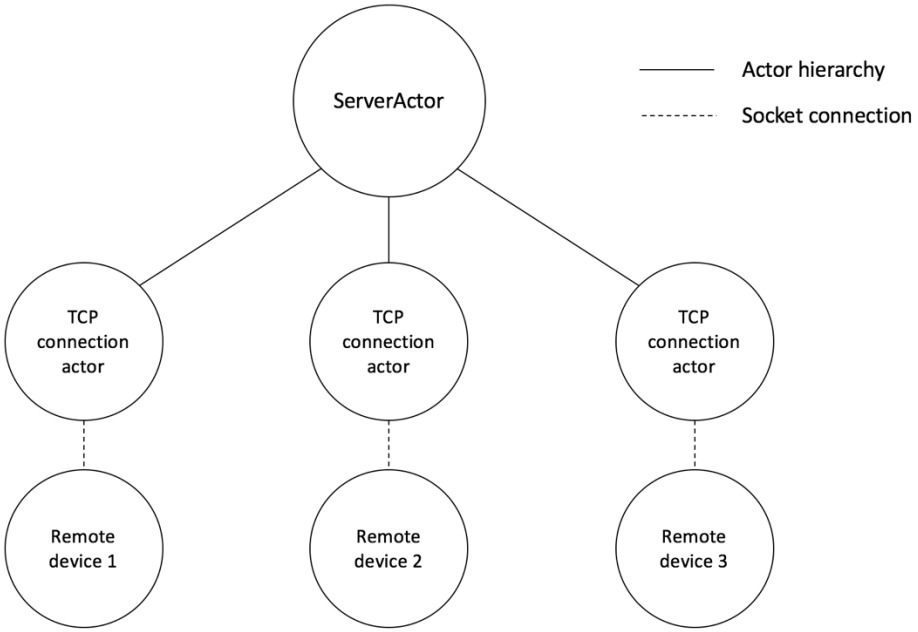


Figure 10.3 - Akka.IO creates an actor per network connection that prevents any potential concurrency problems when receiving messages from multiple senders simultaneously.

The first component of this design is the server actor, which is responsible for the communication with the TcpManager. This has the responsibility of telling the TcpManager that you are using this actor as the target for all new incoming connections. In order to do that, you send it a `Bind` message with a reference to the actor that should be told of new messages, as well as the incoming endpoint on which you should listen for packets. In the following example, you create a new actor that is responsible for any server-related responsibilities and then send a message to the TcpManager telling it that you want it to receive any incoming connection requests:

```
tcpManager.Tell(new Tcp.Bind(Self, new IPEndPoint(IPAddress.Any, 8080)));
```

Having told the TcpManager that you want this actor to receive all incoming connections, you need to then receive the incoming messages containing the incoming connections. You'll typically receive two messages during the initialization of the actor; `Bound` and `Connected`. `Bound` is sent when the underlying socket listener has been established and is able to start receiving incoming connections. The `Bound` message contains the underlying socket address that the operating system is now listening on. Whenever a new connection is established with the TCP listener, a new actor is spawned internally that is responsible for handling all of the low-level socket internals, such as message serialization, buffering, and any other tasks. This

actor sends a message to your server actor, informing it that a new connection is available. Then you can register an actor that should receive the deserialized messages from this connection. You could choose to register them in your server actor, but in the following example, you create a new actor that is responsible for that single connection. This decision allows you to process more packets in parallel while also getting all of the benefits of single actors, including the likes of using finite state machines, which may be useful for more complex protocols that rely on more involved handshake protocols. In the following example, you spawn a new actor that is responsible for each incoming connection and responding to the incoming messages:

```
Receive<Tcp.Bound>(bound =>
{
    Console.WriteLine("The connection was bound to port 8080");
});

Receive<Tcp.Connected>(connected =>
{
    var connectionActor =
        Context.ActorOf(Props.Create(() => new StatsDServerChannel()));
    Sender.Tell(new Tcp.Register(connectionActor));
});
```

Having created your actor hierarchy with an actor per connection, you can now start to process the incoming information from each actor. Whenever a connection receives an incoming packet, Akka.IO first reassembles the complete TCP message, which may have been split across several packets, before sending it to the actor registered to the given connection wrapped in a `Received` message. You can then access the underlying bytes transferred over the network through the `Data` property of the `Received` message. From here, you can convert it into an appropriate format. This may include leaving it in a binary format, converting it into a text format, or deserializing into an object graph with tools such as Google's Protocol Buffers or other binary serialization tooling. In this case, you convert the message to an ASCII string, because this matches the specification as laid out by StatsD. In the following example, you can convert your received ByteString into a string that you can then parse and take the appropriate action on as required:

```
Receive<Tcp.Received>(packet =>
{
    var statsDData = packet.Data.DecodeString();
    HandleStatsDData(statsDData);
});
```

Having now got your metrics data into your actor system, you can use any of the other Akka.NET features at your disposal to better understand the usage. You may want to create an actor for each bucket that will be responsible for aggregating the incoming data and creating alerts based on that aggregated data; you may want to ingest it into other systems or databases for easier analysis or visualization to better understand the metrics you receive.

10.3.2 Sending data through Akka.IO

Having created a socket that is capable of listening, you may want to open a connection to a socket listening at a remote endpoint. You may to create a low-level socket implementation, but because you have Akka.IO, you can bring your client socket connection into the actor system, following many of the same principles we saw when you created a socket capable of listening for incoming data. You can follow many of the same ideas, but as one client is only able to connect to one server, you can simplify it further.

When you created a server, the first step was to bind to a socket so that you'd receive any incoming messages, but when developing a client designed to consume a socket, you need to connect to the remote endpoint before you can communicate with it. In order to connect to a remote socket, you simply need to retrieve a reference to the TcpManager as before, which is responsible for handling all of the low-level socket management, and send it a `Connect` message. In the following example, you create an actor that is responsible for communicating with the server. It does this by sending a message to the TcpManager actor with the endpoint that the server is listening on. Your endpoint consists of two key pieces of information, the IP address of the remote host and the port on which it's listening.

```
var serverEndpoint =
    new IPEndPoint(
        IPAddress.Parse("127.0.0.1"),
        8080);
tcpManager.Tell(new Tcp.Connect(serverEndpoint));
```

Having sent the message to the TcpManager, a new connection is created and the original actor requesting the connection is informed of whether the connection was successful or not. If the connection was unsuccessful, it receives a `CommandFailed` message with a string representation of the issue. But if it was successful, then it receives a message with both the remote endpoint you are communicating with and the local endpoint the connection was opened on. The sender of this message can then be used as a means of communicating with the server. Every message the sender receives, it forwards through the underlying socket. In the following example, you can see how to use the switchable behavior functionality of an actor to be able to communicate with the server by having a different behavior for the connected and unconnected states. The actor can then receive a variety of messages relating to either the connection itself or other actors in the system who want to communicate with the socket.

```
Receive<Tcp.Connected>(msg =>
{
    Sender.Tell(new Tcp.Register(Self));
    Become(Connected(Sender));
});

Receive<Tcp.CommandFailed>(msg =>
{
    Console.WriteLine("Failed to connect to remote endpoint");
});
```

Now that you have a client connection, you can start communicating with the remote server. We've already seen the StatsD protocol, and we'll now see how to directly communicate with a server running the protocol. By having a client within your actor system that deals with communicating with the StatsD server, you can directly ingest metrics into your StatsD server with little effort. This could mean that you can persist metrics such as the time taken to process certain message types or the number of messages an actor has processed. The following example shows how to receive messages from inside your actor system and send them through the socket. In order to send data through the socket, you simply send a `Tcp.Write` message to your coordinating actor, with a ByteString that contains the data you want to send over the network. You can create a ByteString from other ByteStrings, an array, or a string, meaning that you can easily serialize the ASCII strings required by the StatsD protocol. But there may be other situations in which you receive a message from the socket, ranging from an IO error where the network connection was physically cut, a peer reset error where the other party quickly quit the application without first closing the connection, or a situation where the remote party normally disconnects. In these cases, the coordinating actor sends a message relating to the cause of the problem. For example, if a peer reset occurs, a `Tcp.PeerReset` message is sent, or if a network error occurs, a `Tcp.IOError` is sent. You can handle these errors and respond appropriately. For example, you may want to cache any incoming messages until you can reconnect and then send them once the connection has been re-established. In this simple case, you simply write a message to the log that the connection was reset.

```
private Action Connected(IActorRef connection)
{
    return () =>
    {
        Receive<string>(msg =>
        {
            var write = Tcp.Write.Create(ByteString.FromString(msg));
            connection.Tell(write);
        });
    };
}
```

Although the example here focuses on connecting to an Akka.NET-based socket server, the implementation that you've created can connect to any of the available StatsD servers that use TCP, because you're no longer simply constrained to connecting to other actor systems. You're also not limited to using the StatsD protocol, and you can implement clients for any protocol sent over a network. For example, you could just as easily create a client that interoperates with an SMTP server in order to send and receive emails, or a DNS server providing IP addresses for domain names.

10.3.3 Akka.IO Summary

In the example of how to use Akka.IO in this section, you've created a socket connection on both a client actor system and server actor system, allowing you to communicate between the

two. Although you could have easily used the Akka.Remote features we saw in chapter 8, this would have limited your usage to only allowing Akka.NET-based actor systems to connect and record metrics, but by using Akka.IO, you can receive input on a common protocol, in this instance StatsD, allowing other clients to connect and publish metrics. This means that you can use other technologies where appropriate, for example, simple shell scripts when you want to monitor operating system-level metrics or clients available in other languages such as Java or Ruby, allowing you to receive metrics from any system within your overall system architecture.

By using Akka.IO, you quickly and easily set up a high-performance low-level socket connection that enabled you to receive messages sent from applications outside of the actor system with minimal overhead and low latency and immediately have them available within your actor system. This level of abstraction helps to remove many of the complexities you're likely to encounter as you develop network-based applications. By having a well-known, easy-to-use API, you can ensure that you can receive messages from the network in a performant manner without deep technical knowledge of the underlying operating system kernel and how it handles packets received from the network.

10.4 Case study: IO – integration – IoT applications

As you see an increase in the uptake of Internet of Things devices, you see devices used in potentially more hostile environments where you don't necessarily have many of the same conveniences that you might expect when deploying a traditional software project. As an example, when designing systems, you typically expect a fast internet connection between the client and the server. As an example, high speed broadband connections are found in the majority of homes, but as you seek data from more remote environments, you're unlikely to have access to a home broadband connection. Instead, you're more likely to encounter low speed, low bandwidth, and high latency connections operating over older cellular connections. This puts pressure on you to choose the correct protocol for transferring data over the network. In typical scenarios, you might choose to use HTTP, but in these hostile environments, you'll use a much lower-level protocol, specialized for the task at hand, focusing on minimizing the overall packet size.

You can therefore use the IO components of Akka.NET to use actors as a means of simplifying the acquisition of data from a network socket and immediately processing it within your actor system. From here, you can decompress or deserialize the contents of the network packet and push it through to other actors in the system. This allows you to then perform more complex logic on actors that process the messages as .NET objects.

As an example, one of the frequent tasks relating to Internet of Things workflows is to receive time series data and perform complex event processing on this time series data, helping to understand historical data and predict future trends. You can see an example architecture in the following diagram, where you have a number of Internet of Things devices deployed in a field on a farm. These devices monitor a number of factors, including the moisture of soil and weather data over time, which they periodically upload to an Akka.NET based system in the cloud. This system then aggregates the data from multiple devices and

calculates predictions based on the data and historic weather data to ascertain whether irrigation should be switched on or natural water is used. Given that these devices will be deployed in rural locations outside of major population centers, it's likely that the only internet connection available will be through a cellular connection, probably 2G or EDGE. As such, you'll have to use a protocol that prioritizes small packet sizes because of the high cost of cellular data and the low bandwidth available. In the following diagram, you can see how you can create an actor that receives data from a network socket and then passes it onto an actor within the system dedicated to processing the event data from each of the individual devices. Further actors are then able to aggregate data from clusters of devices if necessary.

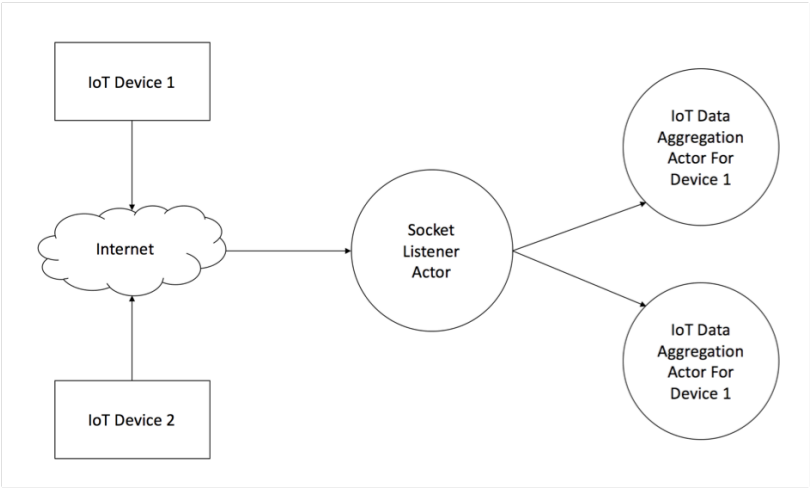


Figure 10.3 - Internet of Things applications operating in hostile environments may be subject to bandwidth constraints. By opening a direct socket into your actor system, you have the opportunity to create your own custom protocol relevant to the circumstances.

By using the IO components of Akka.NET, you can simplify the ingestion of data from a network source that might not have a pre-existing network client. Given the complexities typically associated with low-level socket programming, the use of IO components within Akka.NET allows you to simply treat a socket as another actor.

10.5 Summary

In this chapter, you learned:

- How Akka.NET can be combined with other tools, such as Web API and SignalR, to build completely reactive applications
- How to use Akka.IO to treat sockets as first-class components of an actor system

11

Storing actor state with Akka.Persistence

This chapter covers

- **Adding a persistent backing datastore to an actor to save its state**
- **The concepts behind event sourcing**
- **Creating evolvable applications using Akka.Persistence and event sourcing**

Throughout the book so far, we've developed a wide variety of actors designed to operate in a number of different scenarios, whether they've been small isolated actors or larger systems such as e-commerce applications. But although all of these actors have had significantly different use cases, they have all shared a common trait, which is that they exist as a simple abstraction over the top of application memory. Although we've seen how actors embody a number of key traits, such as message queues, processing, and state, the ultimate aim of an actor is to more easily support concurrent workloads and reduce the potential surface area on which bugs relating to concurrency may occur.

For every actor you've created so far, the actor's state has been ephemeral, and if the actor is shut down, then the actor's state gets reset back to what it was when the actor was first created. But you sometimes need an actor to be more resilient and be able to return its state to what it was before it shut down. Actors in Akka.NET can be shut down for a wide range of reasons: you might need to relocate the actor onto a different actor system because you're running low on resources, you might also take the approach of failing fast and letting Akka.NET restart the actor in the event of an error, or you might need to shut the application down when you want to upgrade it, which will cause all of the actors within the actor system to shut down.

In all of these instances, you need to be able to have systems in place so that you can recreate an actor's state as and when it is required. Let's consider the example of the shopping

cart we've seen in previous chapters. If you have an actor that represents the user's shopping cart and stores references to items in the actor's state, then you want to ensure that your user continues to see those items in their cart as they move around the website. If you were to encounter an application failure that ultimately ended up crashing the entire application, after the application restarts, you want to ensure that the user has all of their previous items in their shopping cart. From a business perspective, if a user had spent an extended period of time browsing the website and adding a number of items to their cart, then they may end up feeling irritated and may take their business elsewhere, losing the company a potentially large sale.

This presents a challenge, in that you need to be able to perform a number of tasks related to saving and recovering an actor's internal state. You need to be able to take your actor's internal state and move it into a persistent storage system such as a database or a filesystem. You also need to be able to recreate the actor's internal state as a result of reading in the state you persisted to an external file store. Fortunately, as part of the Akka ecosystem, there is the Akka.Persistence tool, which enables you to quickly write actors that can persist and recover state from a variety of filestores, ranging from a variety of SQL and NoSQL databases through to flat files in a filesystem, as well as the ability to plug in alternative data stores. In this chapter, we'll look at how to write applications whereby actors are able to store and recover state thanks to Akka.Persistence.

11.1 Understanding event sourcing

Before you start using Akka.Persistence, it's important to understand how the underlying state is persisted to the data store. When you look at an actor, you have the notion of an actor's state, which is an abstraction over some location in the computer's RAM. In many applications, you simply persist this location in memory to a database table through the use of an object relational mapper (ORM). This pattern, known as the *active record pattern*, has become a common approach due to its overall simplicity, but it presents a number of problems, notably the possibility for object-relational impedence mismatch, whereby objects in memory don't map directly onto database tables.

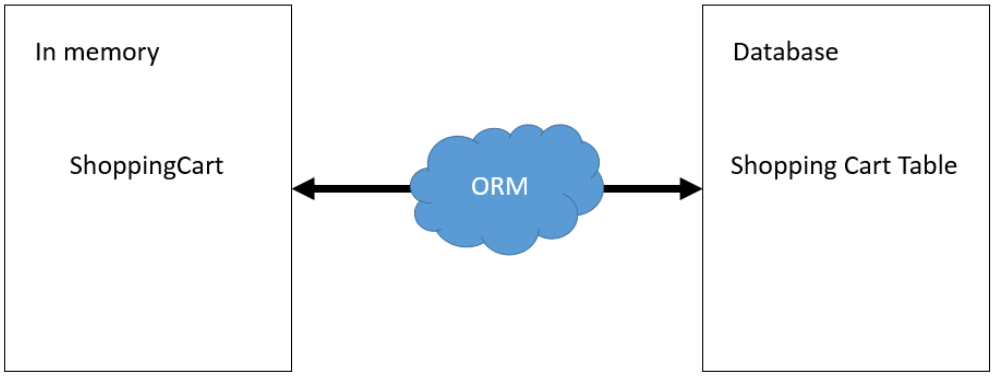


Figure 11.1 The active record pattern relies on mapping an in-memory object directly into an object in a datastore that any mutations are then saved to.

We saw earlier in the book, in chapter 3, how there are three components that comprise an actor: the message queue, which presents events to the processing stage, which depends upon the content of the message it receives, which it then uses to modify the internal state. You can therefore consider the internal state to be nothing more than an initial state along with a number of changes in the form of events. Although this might sound relatively obscure, this idea forms the basis for systems that are hundreds of years old. For example, when you consider accounting records, rather than simply record the total money in the account at any point in time, you instead maintain a log of all the transactions that have taken place against that account. This includes all debits from the account to their destination, and the credits into the account from their origin. This provides vastly improved traceability and auditing capabilities than what you would typically achieve with the active record approach, in which you simply have a line representing the amount of money in the account. You can also rebuild the state, in this case representing the total money in the account, by simply reapplying every event onto the starting state.

This conceptual model is not simply limited to the accounting world, and a number of systems in software development also rely on these ideas. For example, a Git source control repository also follows this model. You have an initial state, which is an empty directory, along with a number of changes, which represent the changes to files within that directory. You might add files to the directory as you develop an application, make changes to files to add new functionality, or remove files entirely. Every time you commit a change to the repository, that is a point-in-time snapshot of the state. This model of persisting changes ends up being significantly more lightweight for source control systems, as you may need to rebuild the state at a given snapshot. Although you could store the whole repository's contents at that point in time, it would rapidly expand to a potentially huge file.

This model is not just limited to source control technology, though; you can apply the model to the applications you write. For example, let's reconsider the shopping cart example

that we've seen several times now. You have an internal state of the shopping cart, which is the items contained within it. But for a customer to get there, they have to either add items to their cart or remove them. By persisting these events, you can then rebuild them into the current list of items. In the following diagram, you can see how to represent your shopping cart as a process of time where you apply events in order to the state and build up a new state. As you move from left to right, you can see how the events are applied. You initially start with an empty shopping cart and then add a television to the cart; you then add a laptop and remove the television. At each stage along the way, you can see exactly what the contents of the shopping cart are. This provides significantly enhanced observability of the changes to your system, as well as the ability to audit any changes that have been made. Finally, you also gain the ability to fix any historical errors that might exist within your application. As you have an ordered list of all the changes that exist within your application, by replaying the events with a different actor processing stage and changing the actor's behavior, you arrive at a new state. When dealing with active record-based systems, you don't get this level of historical detail and so you aren't able to correct historical application errors as easily, and frequently you can end up in a situation where you have invalid data as a result of a lack of depth.

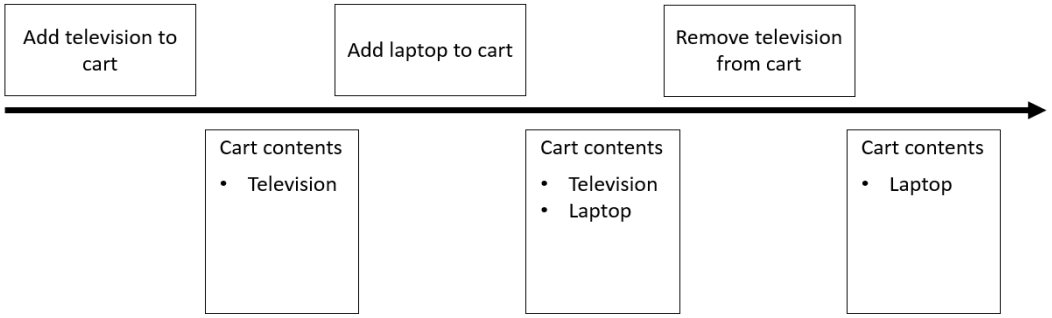


Figure 11.2 You can think of a shopping cart's state as the result of applying the events to it from an initial starting state.

This model, known as event sourcing, sits at the heart of Akka.Persistence and provides the basis for how to persist and recover actor state as part of a larger actor system. Throughout the rest of this chapter, we'll look at how to use Akka.Persistence to create actors that follow this model of event sourcing.

11.2 Using Akka.Persistence

Now that you have an understanding of the underlying concepts, you can build actors that use these concepts as a means of persistence. In order to use Akka.Persistence, you first need to install the library files; once again, the library is provided through NuGet and can be

installed from the Akka.Persistence package. Having installed the library, you're then able to start writing actors based on the actor types provided by Akka.Persistence.

11.2.1 Writing persistent actors

At the core of an Akka.Persistence-based actor is `PersistentActor`. `PersistentActor` is a further variation of the actors we've seen so far. For example, we've seen how to write actors either as simple handlers matched to a message type with `ReceiveActor`, or you might also have a more specialized actor in the form of a finite state machine, which you can write with the help of `FSMActor`. `PersistentActor` follows this same concept of an actor specialized for a specific usage, in this case, persistency. In order to create an actor that has a backing store, you simply inherit from the `PersistentActor` class and implement the required methods on it. In the following code example, you can see how to define a shopping cart actor that would be responsible for storing the items a user has added to their shopping cart as they browse around an e-commerce site. As you can see, the actor has a number of additional methods that you need to implement as part of the contract:

```
using Akka.Persistence;

public class ShoppingCartActor : PersistentActor
{
    public override string PersistenceId { get { throw new NotImplementedException(); } }

    protected override bool ReceiveCommand(object message)
    {
        throw new NotImplementedException();
    }

    protected override bool ReceiveRecover(object message)
    {
        throw new NotImplementedException();
    }
}
```

The first requirement is that you must create a persistence ID. This is the value that Akka.Persistence uses as its means of identification when it persists the state into a database. It's important that this is unique on a per-actor basis, otherwise you may be in a situation where you are potentially muddling state and events between two distinct actors, leading to potentially unexpected behavior. One example of how to name them is based on the actor's location in the actor hierarchy, or the actor's name if the name is unique. In the following example, you simply give the actor a persistence ID of `"shoppingcart"` because you will only have one shopping cart across the application. In a production environment, you are likely to have a larger number of shopping carts, and as such, you'll need some means of uniquely identifying actors. This may be either a generated GUID or an integer representing it. The persistence ID is also used to recover the state for the actor, and as such, it's important to use a deterministically generated identifier so that it can recover the state. As such, this might mean having a parent actor responsible for persisting all of the actors that exist below it.

```
public override string PersistenceId { get { return "shoppingcart"; } }
```

You also have a method called `ReceiveCommand` that you need to implement. `ReceiveCommand` is analagous to the `Receive` method that you've had to implement in certain other actor instances. The first noticeable point about this method is that it is called `ReceiveCommand` and not `ReceiveEvent`, despite the fact that you've already seen that Akka.Persistence operates based on the concept of event sourcing. The key reasoning for this is the separation between commands and events in event sourcing. Although the two seem fairly similar, there is a subtle difference between them:

- *Commands*—Commands are you telling an actor that it needs to do something. For example, in the case of the shopping cart, you are likely to pass it a command called `AddItem`. Commands are typically named in an imperative manner as they are informing the target to do something.
- *Events*—Events are the outcome of a command. For example, if the shopping cart gets told to add an item, then the direct outcome of that will be an `ItemAdded` event. Events are typically named in the past tense, as they are the result of a previously completed action.

Although this might seem unnecessary, you need to consider what actions are to be undertaken if an actor has to perform some side-effects while it is being used. In this simple example, you don't need to do anything beyond adding the item to your internal state, but in a more complex example, you will probably need to perform some more complex operations. For example, you may need to post a message onto a message queue for an external system to process. If you were to post that message onto the queue every time you process an event, then you might end up enqueuing the same message multiple times, due to the actor reprocessing the event whenever it recovers its state. As such, by splitting events into commands and events, you can more easily handle the outcome of side-effects.

You can see in the following example how to write an actor that receives commands and then emits events as a result. You receive a command called `AddItem` that contains an item identifier that represents a stock number as well as the number of items to add. You are then able to match on the incoming message and perform the appropriate action as a result. In this case, you create an event called `ItemAdded`, which once again contains the stock number as well as the total number of items to add. After creating the event, you then persist it by calling the `Persist` method with an event and a handler. Upon calling `Persist`, the message is written into what Akka.NET calls a *journal*. The journal is essentially an ordered log of all events for a given persistence ID. The handler is executed when the internal storage of the event succeeds, and is where you'll modify any internal state of the actor. In this case, you add the items to a dictionary of all of the stored items.

```
class AddItem
{
    public string ItemId { get; }
    public int ItemCount { get; }

    public AddItem(string itemId, int itemCount)
    {
        ItemId = itemId;
```

```
        ItemCount = itemCount;
    }
}

class ItemAdded
{
    public string ItemId { get; }
    public int ItemCount { get; }

    public ItemAdded(string itemId, int itemCount)
    {
        ItemId = itemId;
        ItemCount = itemCount;
    }
}

protected override bool ReceiveCommand(object message)
{
    if (message is AddItem)
    {
        var msg = (AddItem)message;
        var itemAddedEvent = new ItemAdded(msg.ItemId, msg.ItemCount);
        Persist(itemAddedEvent, HandleEvent);
        return true;
    }
    return false;
}

private void HandleEvent(object @event)
{
    if (@event is ItemAdded)
    {
        var evt = (ItemAdded)@event;
        if (_items.ContainsKey(evt.ItemId))
        {
            var currentCount = _items[evt.ItemId];
            var newCount = currentCount + evt.ItemCount;
            _items[evt.ItemId] = newCount;
        }
        Else
        {
            _items[evt.ItemId] = evt.ItemCount;
        }
    }
}
```

A note on stashing

We saw earlier in chapter 3 how to create a message stash for any actors you create. When dealing with Akka.Persistence, the actor already has a stash defined, which it uses to store any incoming messages when a persist operation is already in progress. This can potentially interfere with some of the methods we've seen of how to shut down an actor. For example, we saw how to shut down an actor by sending it a `PoisonPill` message, but this message is automatically handled by the actor system, meaning that the `PoisonPill` will be handled even if there are other messages currently in the stash awaiting processing. The way to handle this problem is to create a dedicated message that is used for handling shutdown within persistent actors, for which the command simply calls `Context.Stop`. You can see an example of how you might handle that in the following code:

```
class Shutdown { }

protected override bool ReceiveCommand(object message)
{
    if(message is Shutdown)
    {
        Context.Stop(Self);
        return true;
    }
}
```

Now that you have a system in place for handling commands, generating events, and persisting those events, you need to be able to handle those events if your application is currently recovering from a failure. In order to do this, you need to implement the `ReceiveRecover` method, which receives events from an event journal after an actor is restarted. During recovery, any commands received by the actor are stashed so that they're handled after the state has fully recovered. This then gives you the ability to modify the state in line with what we saw previously when modifying the state in a command handler.

Now, whenever you create a new instance of this actor type, `PersistentActor` checks the event journal to see whether any events have already been persisted. If they have then they are sent to the actor in order to be handled through the `ReceiveRecover` method. In this method, you can match on the event and update the state depending upon the type of event received. In the following example, you can see how you're matching on the event type and updating the state exactly as you did when you were writing command handlers:

```
protected override bool ReceiveRecover(object message)
{
    if(message is ItemAdded)
    {
        HandleEvent(message);
        return true;
    }
    return false;
}
```

During the actor's lifecycle, you can check what state the actor is in, that is, whether it is in recovery mode and currently receiving persisted events, or whether it is in regular operation and is currently receiving commands from other actors. The `IsRecovering` property tells you whether you are still recovering other events, and the `IsRecoveryComplete` property is used to let you know whether you are now able to process commands received through general usage.

```
if(this.IsRecovering)
{
    //Perform specific logic for when the actor is in the recovering state
}
else if(this.IsRecoveryComplete)
{
    //Perform specific logic for when the actor has finished recovering it's state
}
```

You also have a proactive means of determining the status of recovery through the use of the `RecoveryCompleted` event, which is sent to the `ReceiveRecover` method whenever the system has successfully

processed all of the incoming events. In order to use this, you simply need to create a further match case that handles the `RecoveryCompleted` event.

```
protected override bool ReceiveRecover(object message)
{
    if(message is RecoveryCompleted)
    {
        //Perform specific actions once the actor state has been recovered
    }

    if(message is ItemAdded)
    {
        HandleEvent(message);
        return true;
    }
    return false;
}
```

As you can see, although there are some additional concepts to understand when using Akka.Persistence persistent actors, there are not a huge number of changes that need to be made to be able to effectively create an actor that is able to persist all of its changes to a database and then recover that state upon a restart.

11.2.2 Configuring a journal

In the previous section, we saw how events get persisted to a data store, or in Akka.Persistence parlance, a journal. This journal can be any of a wide variety of options, including a large number of SQL and NoSQL databases, but by default, Akka.NET will persist its events to a data structure in memory. It is, by default, not long-term storage, and will be deleted if the application is closed. If you want to change this setting then you will need to modify the HOCON configuration in a similar manner to what we saw in chapter 5.

A number of different plugins are provided for persisting journal-based events, but in this example, you'll look at how to store them into a Microsoft SQL Server-backed journal. In order to use the SQL Server-backed journal, you first need to install the actors so that you can communicate with it. This is done by installing the Akka.Persistence.SQLServer package from NuGet in the project alongside Akka.Persistence.

Having installed the package, you then need to configure it so that you can communicate with it. In order to do that, you need to add to your HOCON configuration file. You can follow the same process of creating a configuration file and using it as we did back in chapter 5 when we saw some of the other configuration options, but in this case, you need to modify it to add settings for Akka.Persistence. In the following example, you can see how to specify the journal to use as being SQL Server, and you can also see how to connect to it by specifying the connection string. You can see in the following example how to create a dedicated configuration section in the Akka.Persistence settings for the journal. Within that configuration section, you specify the journal to use, which is simply a location in the configuration file used to identify it. The specified location in the configuration file then contains two core pieces of information: the

class to use as the persistence journal (in this case it's the `SqlServerJournal` class) and the connection string to the database you are going to be using to persist state.

```
akka {
    persistence {
        journal {
            # SQL Server journal plugin.
            sql-server {

                # Class name of the plugin.
                class = "Akka.Persistence.SqlServer.Journal.SqlServerJournal,
       Akka.Persistence.SqlServer"

                # Dispatcher for the plugin actor.
                plugin-dispatcher = "akka.actor.default-dispatcher"

                # The connection to the SQL server database which will store the persisted
       events.
                connection-string = "<SQL Server Connection string>"
            }
        }

        snapshot-store {
            sql-server {

                # Class name of the plugin.
                class = "Akka.Persistence.SqlServer.Snapshot.SqlServerSnapshotStore,
       Akka.Persistence.SqlServer"

                # Dispatcher for the plugin actor.
                plugin-dispatcher = ""akka.actor.default-dispatcher""

                # The connection to the SQL server database which will store the persisted
       snapshots.
                connection-string = ""
            }
        }
    }
}
```

In this example, you simply used SQL Server as the journal of choice, but there is a wide variety of plugins available, including Postgres, Azure Table Storage, Azure Blob Storage, and others. In addition, members of the Akka.NET community have created a wide range of other plugins for various databases. It is also possible to create custom journals in the event that one doesn't already exist for your datastore of choice, but that is out of the scope of this book.

11.2.3 Using Akka.Persistence summary

In this section, we've seen how to build simple actors that are backed by a journal that stores events. We've also seen how to change the journal you use for events. In the rest of this chapter, we'll dive deeper into some of the more advanced features of Akka.NET that allow you to get better performance or different overviews of data.

11.3 Akka.Persistence performance tuning

Although the performance of Akka.Persistence is just as good as the active record approach in most situations, there are potential cases whereby you may need to look into the options you have for increasing the message throughput of persistent actors. In this section, you'll look at a couple of approaches you can use to ensure that you can make the most of the library and continue to use it, even in cases where you might otherwise be strained.

11.3.1 Snapshot stores

Although event source-based applications provide plenty of advantages, there are also downsides, one of which is the amount of time it can take to recover the internal state. As the state is computed by applying events in the order they were persisted, this means that in a situation where you huge number of events are persisted, it will take longer and longer to recover the state.

Let's consider the example of an indecisive customer using their shopping cart. Because you're storing every change to the actor's state, if a customer was to keep adding and removing an item from their cart, you could quickly build up a large number of events to represent the state of the shopping cart. If this is being done across a large number of items as well, as might be the case if this was for a grocery store, this could lead to a lot of events. As you add more events, it takes time to retrieve them all from the datastore, and it takes time to apply the events because they need to be applied one after the other.

This presents you with a problem when you consider the aims of a reactive application when combined with the design of Akka.Persistence. We saw in the previous section how the actor stashes any incoming messages while it is in the process of recovering state. This presents a problem if an actor needs to recover state frequently, for example, if it fails frequently. If an actor is taking seconds to recover its state as a result of a large number of events, then it's unable to respond to messages from other actors until it has completed its recovery. If you're building applications that require low latency in order to ensure the application remains responsive to users, then having operations block for several seconds isn't ideal.

In order to counter this, Akka.Persistence provides the option of using a snapshot store, which is used for storing snapshots of actor state at specific points in time. This then allows you to replay states from a position at a point in time. You can see an example of this in the following diagram. Rather than needing to recover every event from the beginning of the actor's lifecycle, you can choose the latest snapshot and replay events from that point onwards.

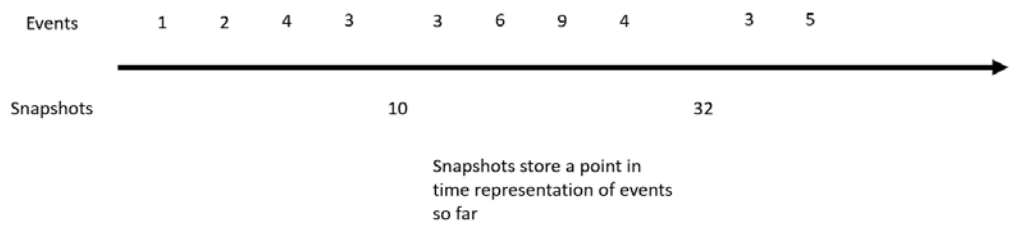


Figure 11.3 - Snapshots allow you to limit the number of events you need to replay any time a persistent actor needs to recover its internal state.

In order to store a snapshot within a persistent actor, you simply call the SaveSnapshot message with whatever you want as the state. In the shopping cart example, you would choose to store your dictionary of items and item counts. But you don't want to store a snapshot after every event as there is no garbage collection of historical snapshots, and instead they simply accumulate in the database. Instead, you should save a snapshot either after you've emitted a certain number of events or after certain period of time since the last message. In the following example, you keep track of how many messages it has been since it last stored a snapshot, and if you exceed a maximum then you persist a new snapshot. In comparison to the persistence of events, snapshots are persisted asynchronously, and therefore other messages can be processed while snapshots are persisted. This is due to the fact that snapshots are designed primarily as an optimization technique rather than a primary storage mechanism.

```
private int _eventsProcessed = 0;

private void HandleEvent(object @event)
{
    if (@event is ItemAdded)
    {
        //Event logic
        _eventsProcessed++;
    }
    if(_eventsProcessed > 10)
    {
        SaveSnapshot(_items);
        _eventsProcessed = 0;
    }
    return false;
}
```

Having now saved a snapshot, you now need to be able to handle recovering the internal state from the snapshot itself instead of replaying events from the beginning of the actor's lifecycle. For this, you need to handle a new type of message during the recovery stage. If a snapshot exists for the persistence ID of a given actor, then it first retrieves the latest snapshot and the event associated with the snapshot. After this, it is presented to the actor as part of a

SnapshotOffer message. You can access the state you persisted through the Snapshot property on the message. In the following example, you can see how to set the internal state as a result of the SnapshotOffer message being received:

```
protected override bool ReceiveRecover(object message)
{
    if(message is SnapshotOffer)
    {
        var snapshot = (SnapshotOffer)message;
        _items = (Dictionary<string, int>)snapshot.Snapshot;
        return true;
    }

    if(message is ItemAdded)
    {
        HandleEvent(message);
        return true;
    }
    return false;
}
```

After the snapshot offer has been received, the persistent actor is then sent any events that follow the sequence number of the snapshot, if any. This means that you can continue receiving events to recover to the latest state.

Snapshots as a technique allow you to effectively reduce the time it takes to recover to the latest state by maintaining history at a specific point in time. But it's important to understand that they exist as an optimization technique rather than as a basis of storage.

11.3.2 Async write journals

We saw in the previous section how, by using the Persist method on a persistent actor, you can supply an event and a callback that then persists an event and executes the callback on completion. We also saw how the actor stashes any incoming messages until the write has completed, ensuring the operation remains consistent. This technique is ideal for most situations, because you typically get good enough performance for the usage, but ultimately there are situations where any form of blocking will cause the response time to grow rapidly. One such example of this is in applications in which there are a large number of writes going through the system at any one time. In these cases, the potential blocking is capable of causing the queue of incoming messages to rapidly grow, potentially causing the system to fail due to too many messages.

In order to counter this situation, Akka.NET allows you to use the PersistAsync method, which will send a request to the write journal containing the event to write. You once again supply an action for when the write request completes, but the main difference is that while the write request is in flight, the actor is capable of processing the next message in its mailbox. This does, however, come at the cost of a relaxed consistency guarantee. When dealing with the synchronous persistence, the application will only handle the next message if it was able to successfully write the event to the journal. If it failed for any reason, such as a database being unavailable, then the next messages would not be processed. But when using the

asynchronous persistence, you might be in a situation where you're continuing to process messages without having any guarantees that the state has been written. This makes it ideal for use cases where you're receiving vast amounts of data, some of which you can afford to lose. One such example of this might be writing and aggregating log or metrics data. In these cases, you ideally want to record all the data for later use, but if you lose several minutes of data then it's acceptable if it means the system does not fail.

You can see an example of using `PersistAsync` as follows, where you write an event asynchronously to the event journal. The rest of the actor stays the same, and you simply need to change your call to `Persist` to `PersistAsync`:

```
protected override bool ReceiveCommand(object message)
{
    if (message is AddItem)
    {
        var msg = (AddItem)message;
        var itemAddedEvent = new ItemAdded(msg.ItemId, msg.ItemCount);
        PersistAsync(itemAddedEvent, HandleEvent);
        return true;
    }
    return false;
}
```

Using `PersistAsync` allows you to rapidly increase message throughput for a given actor by allowing the actor to process other messages while waiting on writes to complete. You do, however, need to be mindful of the potential effect that this could have on the actor, and if consistency is a core requirement of the actor, then other alternatives should be investigated before asynchronous write journals are chosen.

11.4 Akka.Persistence performance tuning

Akka.Persistence is on its own incredibly fast, but there are certain circumstances where the default configuration is unfortunately not as fast as an active record-based approach. In order to counter this, you can solve the problems with a number of provided features. We've seen how you can speed up the time to recover from a restarted application using snapshot stores. We've also seen how to relax the consistency of your actors by using asynchronous write journals, which then ultimately allows you to handle more write-based operations on events in a smaller amount of time.

11.4.1 At-least-once delivery

We saw earlier in the book, in chapter 6, how there are a number of different methods through which an actor can encounter problems. We saw cases of how to restore actors following errors through the use of supervisors and supervision strategies. We also saw how asynchronous message-based applications could potentially have problems whereby you ultimately fail to receive a message due to it getting dropped by a network, for example. In this scenario, you built up a small actor that was responsible for receiving acknowledgements so that you were able to guarantee that you are correctly processing the messages sent to the

target actor. If you failed to receive an acknowledgement in response then you would resend the message until you did get an acknowledgement in response.

You saw how this was known as at-least-once messaging, because the message would be delivered more than once until you were certain it had been processed. This is contrary to the typical Akka.NET approach to messaging whereby you send at most one message and it essentially acts in a fire and forget manner. The actor you developed was satisfactory for small cases, but there are significant edge cases that can occur in asynchronous distributed systems and as such, such a protocol needs guarantees that it will continue to operate even in the face of significant adversity in the system. Fortunately, as part of the Akka.Persistence library, Akka.NET provides a thoroughly tested and developed solution for sending messages at least once.

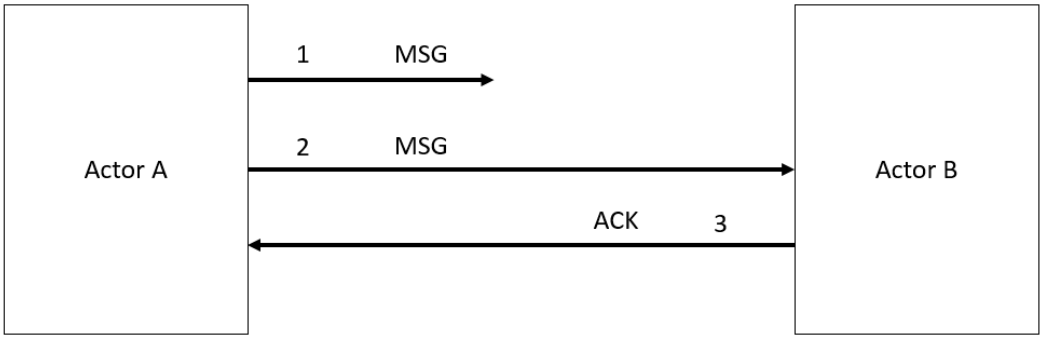


Figure 11.4 - The at-least-once delivery semantic tries to continuously resend a message to a target actor until it eventually receives an acknowledgement in reply.

This at-least-once delivery system within Akka.Persistence permits other features, such as persisting whether the target has correctly received the message in the event that the application fails and the entire actor system is shut down. This provides even stronger fault tolerance guarantees than those you saw when you developed a simple solution that simply focused on blue-sky scenarios. In this section, we'll look at how to use the at-least-once delivery actor contained in Akka.Persistence to develop actors that are able to automatically resend messages until they are successfully processed.

In order to create an actor that will attempt to deliver multiple times, you need to inherit from `AtLeastOnceDeliveryActor`. `AtLeastOnceDeliveryActor` provides a number of additional features on top of what a typical `PersistentActor` provides, which makes it easier for you to develop actors with different delivery semantics. In this section, we'll consider a trivial example where the actor is simply sending a string to a remote target and waiting for a response. This might be applied into real-life scenarios where you can't afford to lose an update

such as when deducting money from a user's account or when performing an update that must succeed. In the following code, you can see the overall shell of the actor you will be creating, you simply inherit from the `AtLeastOnceDeliveryActor` class:

```
class CustomDeliveryActor : AtLeastOnceDeliveryActor
{
    public override string PersistenceId { get { throw new NotImplementedException(); } }

    protected override bool ReceiveCommand(object message)
    {
        throw new NotImplementedException();
    }

    protected override bool ReceiveRecover(object message)
    {
        throw new NotImplementedException();
    }
}
```

In the same way as the persistent actor definition we saw before, you need to supply a `PersistenceId` which is used to persist and recover the state. You need to follow the same restrictions as we saw when dealing with persistent actors, and you need to ensure that the `persistenceId` is deterministic and is recoverable between actor restarts. In this case, you simply hardcode the `persistenceId`, but in production environments, you need to ensure that it is unique across multiple actor instances:

```
public override string PersistenceId { get { return "GuaranteedSender"; } }
```

Once again, in a similar manner to the persistent actor, you need to ensure that you handle the commands within your application. In this case, you handle two possible commands: the request to send a message to a target and the confirmation message received from the target. In the simple example, you will use a fixed target that will receive all messages, and you will operate on the basis that any strings the actor receives will be treated as data to be sent to the target actor. Coming up is an example of this in which you're matching on the incoming data. If it's a string, then you create a new `MessageSent` event, and if you receive a `Confirm` command then you create a `MessageConfirmed` event. You also need to handle the recovery of events in the same manner as with a persistent actor in order to restore state if the actor fails. In this case, the operations you perform on events are the same when you're recovering state as well as when you're emitting events. You can create a common method that both the command handler and recovery use for each of the events. In the following example, you can see the method definitions alongside the command handler and recovery code that simply calls the `UpdateState` method:

```
protected override bool ReceiveCommand(object message)
{
    if(message is string)
    {
        Persist(new MessageSent((string)message), Handler);
        return true;
    }
```

```
    else if(message is Confirm)
    {
        Persist(new MessageConfirmed(((Confirm)message).DeliveryId), Handler);
        return false;
    }
    return false;
}

protected override bool ReceiveRecover(object message)
{
    if(message is MessageConfirmed)
    {
        Handler((MessageConfirmed)message);
        return true;
    }
    else if(message is MessageSent)
    {
        Handler((MessageSent)message);
        return true;
    }
    return false;
}
```

Having now got your `UpdateState` shell in place, you need to implement the logic within it. This is where you will either deliver the message or mark it as being received depending upon the event you are dealing with. If you have a `MessageSent` event then you need to call the `Deliver` method, which is provided by the `AtLeastOnceDeliveryActor`. `Deliver` takes in the path to the destination actor and a callback that is used to create the message. `AtLeastOnceDeliveryActor` maintains a unique sequence number that represents the next message to be sent, which it uses to mark determine which messages have been confirmed. You need to send the sequence number to the target so that it can tell you in response which messages it has processed. As such, in the following example, you send a `Message` wrapper that contains your string to send as well as the sequence number provided by `AtLeastOnceDeliveryActor`. You also need to handle the `MessageConfirmed` event which you create whenever you receive a `Confirm` command from the target actor. In this case, you call `ConfirmDelivery` with the sequence number you received in response from the target actor. In the following example, you can see how these components fit together in the `UpdateState` method you defined the shell of earlier.

```
private void Handler(MessageSent message)
{
    Deliver(_destinationActor.Path, deliveryId => new MessageEnvelope(deliveryId,
       message.Message));
}

private void Handler(MessageConfirmed confirmed)
{
    ConfirmDelivery(confirmed.DeliveryId);
}
```

Having now got the infrastructure in place on the sending side, you also need to send a confirmation message from the target back to the original sender with the sequence number of

the message you are currently processing. Here is a small example that receives your wrapper event, prints the string to the console, and then sends an acknowledgement message to the sender in response:

```
class GuaranteedPrinterActor : ReceiveActor
{
    public GuaranteedPrinterActor()
    {
        Receive<MessageEnvelope>(msg =>
        {
            Console.WriteLine("Received a message: {0}", msg.Message);
            Sender.Tell(new Confirm(msg.DeliveryId));
        });
    }
}
```

Within `AtLeastOnceDeliveryActor` there are a number of properties that allow you to configure certain semantics of the delivery protocol, all of which are configurable from the HOCON configuration for the at-least-once delivery actor functionality. For example, by using the `MaxUnconfirmedMessages` property, you can specify how many unconfirmed messages you keep in memory while awaiting responses. If you then try to have too many messages in flight, then any time you call `Deliver`, it will fail and throw a `MaxUnconfirmedMessages-ExceededException` error, allowing you to then back off and try again later. By default, it is 100,000, but it may be necessary to lower this number if you have a large number of `AtLeastOnceDeliveryActor` instances to prevent errors caused by a lack of memory. In the following example, you reduce the number to 1,000:

```
akka.persistence.at-least-once-delivery.max-unconfirmed-messages = 1000
```

`AtLeastOnceDeliveryActor` also sends warnings if it sends a message a certain number of times without a confirmation in response. By default, that number is five times, but it is possible to increase or decrease that number by setting `warn-after-number-of-unconfirmed-attempts` in the HOCON configuration. Then you can handle the `UnconfirmedWarning` messages in the command handler in order to choose the appropriate course of action.

```
akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts = 5
```

`AtLeastOnceDeliveryActor` will periodically resend any unconfirmed messages that it still has in memory. It will batch these messages up and send them in one burst every time period. By default, this property is set to 10,000, but, depending upon things such as the network, it may be useful to modify this. In order to do this, you simply provide a value for the `redelivery-burst-limit` property in the HOCON. In the following example, you can see how to change the limit to 1,000 instead of 10,000:

```
akka.persistence.at-least-once-delivery.redelivery-burst-limit
```

`AtLeastOnceDeliveryActor` is a powerful construct provided by Akka.Persistence that allows you to create actors that can overcome transient issues caused by network difficulties in a completely fault-tolerant manner. Due to the journal backing store, you can even resend

unconfirmed messages to target actors in the event that the application completely shuts down and leaves you in a position where you might not have the full picture of what happened before the shutdown.

11.4.2 Upgrade strategies for applications using event sourcing

As you develop applications over time, you are likely to encounter situations that have arisen solely due to the evolution of your application. For example, you might have multiple versions of messages you might encounter over the lifetime of your application. Let's consider a concrete example in the shopping cart that we saw earlier in the chapter. We saw that you represented the items in your shopping cart with a simple stock identification number and the number of that item in the cart. This was then stored in the event journal as one of the core events, which tells you what happened. But although this might be acceptable as an approach today, in several months time, you may encounter problems with the way you are persisting stock identifiers. For example, you may have chosen one format if you only had a few products available for sale, but due to increased growth, you may find yourself in a situation whereby you now need to use a new stock identification methodology. In this case, once you change your processing logic to use the new event type, you won't be able to recover your old events. This contradicts the reason you wanted to persist state in the first place, so that you can recover the same state in the future.

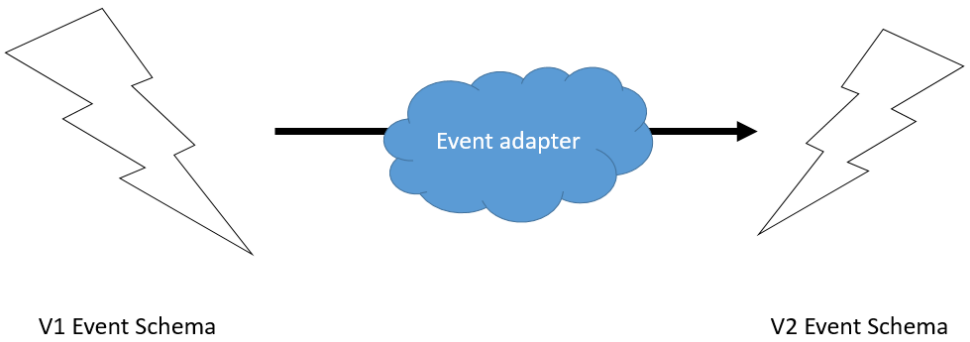


Figure 11.5 - Event adapters allow you to modify events from a journal to an actor and vice versa, allowing you to make changes to event sourced applications while also maintaining the full event history.

In order to handle these situations, Akka.Persistence provides the ability to use an event adapter, which allows mappings of the data types used in your code to data stored within event journals. This separation provides a number of distinct advantages:

- *Data migrations*—This is one scenario we've already considered. When an event model changes between two distinct versions of the application, you need to be able to handle transitions between the data types so that you can continue to read in old data types.
- *Separation of domain and journal data models*—When you develop your application

domain, you might create special classes to represent cases that should never happen in an effort to make illegal states unrepresentable. One example of this is in cases where you might create a custom email class that is designed to only be created if it follows a certain pattern or will otherwise fail. Ultimately, when you then persist this data to the journal, you don't necessarily need the wrapper class and it will simply inflate the amount of time and disk space taken to persist the individual event. Another situation is if you wanted to use a custom serialization technology as a means of persisting the data. For example, in the case of large messages, you may want to use a technology such as Google's Protocol Buffers as a means of compact storage of events. By creating an event adapter, you can specify that you use one representation for the data when it's in memory and another for when it's stored in the journal.

- *Journal-specific features*—Depending upon the database being used, it may be that it supports additional features, thus opening up additional benefits. One such example of this might be if the database supports the persistence of JSON documents, in which case, it may be beneficial to persist the journal events as JSON documents, thus getting either better performance or better future query features. By using an event adapter, you can take the in-memory representation and convert it into the shape required by the database, and vice versa.

In this section, we'll look at how you might write an event adapter for the shopping cart we saw earlier. We'll assume that in version 1 of your events, you simply persisted the stock identification number as a string, but in the upgraded version 2, you need to map the events into a new format. In this case, the new format will simply be a new class that holds the stock identifier. You can see in the following example how to create an event adapter by implementing the `IEventAdapter` interface. The interface provides three methods that you need to implement in order to use it.

```
public class ShoppingCartEventAdapter : IEventAdapter
{
    public IEventSequence FromJournal(object evt, string manifest)
    {
        throw new NotImplementedException();
    }

    public string Manifest(object evt)
    {
        throw new NotImplementedException();
    }

    public object ToJournal(object evt)
    {
        throw new NotImplementedException();
    }
}
```

The first method is `Manifest`, which allows you to link a specific manifest to an event. This manifest is essentially a type hint that allows you to match on the string and use it for future deserialization of the event by using the appropriate serialization mechanism for the given

event. This can be set to any specific value for your purposes, but in this case, you just return an empty string:

```
public string Manifest(object evt)
{
    return String.Empty;
}
```

You also need to implement the `ToJournal` method. This method is called when an event is being persisted to the event journal, that is, after the `Persist` method has been called. Because you are handling a case where you have old events persisted, you don't need to do anything special and you can simply return the event that was passed to you. This event will then be serialized and persisted into the datastore. If you wanted to store an event in the datastore as a Procol Buffers-encoded byte array, then this is where you would convert it into that format with the Protocol Buffers serializer.

```
public object ToJournal(object evt)
{
    if(evt is Events.V2.ItemAdded)
    {
        var newEvt = (Events.V2.ItemAdded)evt;
        return new Events.V1.ItemAdded(newEvt.ItemIdentifier.Identifier);
    }
    return evt;
}
```

The final method you need to implement is the most important one, the `FromJournal` method, which allows you to map an event into a new event type. In this case, you need to create a new event from the event that you were provided that contains the important properties that you are interested in. You start by defining the new event that you'll be using; in this case, it's a simple class containing a `StockIdentifier` property and a count. You can see the class definition in the following example. You also then define your mapping function by creating an instance of the new event from the old event. In the following example, you create a new instance and map across all of the individual properties into the new event. One point to note about the `FromJournal` method is that it returns an `IEventSequence` interface. This means that you can return one event, more than one event, or no events at all. In your case, you return a single event, which you store in a new event sequence.

```
public IEventSequence FromJournal(object evt, string manifest)
{
    if(evt is Events.V1.ItemAdded)
    {
        var oldEvt = (Events.V1.ItemAdded)evt;
        var newEvt = new Events.V2.ItemAdded(new
       Events.V2.StockIdentifier(oldEvt.ItemIdentifier));
        return EventSequence.Single(newEvt);
    }
    return EventSequence.Single(evt);
}
```

Finally, you also need to configure the event journal to use the event adapter you have written when it passes the events to the journal. In order to achieve this, you use the HOCON configuration for the event journal you need to use, as we saw earlier in the chapter. You need to specify the event adapter bindings to use first by creating a mapping of the event adapter type to a name for it. In the following example, you can see how the fully qualified type name, including the namespace and assembly, is given a name, in this case, v1. You then also set up an event binding, which maps an event type onto an adapter. In the following example, you take your version 1 event type, once again with namespace and assembly, and provide an accompanying event adapter name. In this case, you supply the v1 string, which is what you called the previous adapter. Because you're also converting from V2 into V1 format for storage into the journal, you also need to add a type binding for the V2 event.

```
akka.persistence.journal {
    sql-server {
        event-adapters {
            v1 = "Chapter11.ShoppingCartEventAdapter, Chapter11"
        }

        event-adapter-bindings {
            "Chapter11.Events.V1.ItemAdded, Chapter11" = v1
            "Chapter11.Events.V2.ItemAdded, Chapter11" = v1
        }
    }
}
```

Event adapters allow you to easily solve one of the key evolvability issues you are likely to encounter when developing event sourcing-based applications over an extended period of time. They also allow you to make use of some of the advanced features of datastores you're using for your event journal, while also continuing to make the most of all the advanced features provided by Akka.Persistence.

11.5 Case study: persistence – storage – staged upgrades

For many industries, compliance is a core aspect of any system which is designed. This is especially true in the world of finance, where detailed records need to be kept to maintain an accurate historical representation of every action that has been taken. An example of this is in the world of automated trading, where hundreds of stock transactions happen every second. It's imperative that these companies can demonstrate to the relevant authorities, such as the SEC in the USA or the FCA in the UK, that any profits that have been made as a direct result of securities trading were legally valid. For example, the relevant securities commission will want to ensure that no trades were made with non-public knowledge, which could lead to accusations of insider trading. Similarly, securities trading offers opportunities for nefarious parties to launder money, hiding the source of income in order to either evade law enforcement or taxation.

As such, in the finance industry, businesses will frequently face regulatory audits where trades are assessed to ascertain whether any were in breach of these regulations. In order to

pass these regulatory audits, businesses need to maintain a history of every single trade that has been made, rather than simply storing the stocks currently held in a portfolio. In this chapter, we saw how the persistence components of Akka.NET maintain a persistent ordered history of every event that was received by an actor. This event history can then be replayed to see the events that lead to a state at any point in time. You can model this by creating an actor per stock portfolio, where a stock portfolio is a collection of stocks in different listed companies along with the count of stock owned in each case. Now, every time you send the actor a message to either buy or sell stock from the portfolio, you persist the message, meaning that you can maintain a log of every transaction that was made per portfolio. You can see this in the following diagram, where you have an actor, which is your stock portfolio, that receives messages indicating whether to buy or sell a specific stock. These messages could either originate from other automated systems detecting potential trades or human involvement, but regardless of the origin of the message, as soon as they are received by the portfolio actor, they are persisted to a data store. From this data store, the individual trades can then be replayed, allowing the portfolio to be rebuilt to any point in the history of the portfolio.

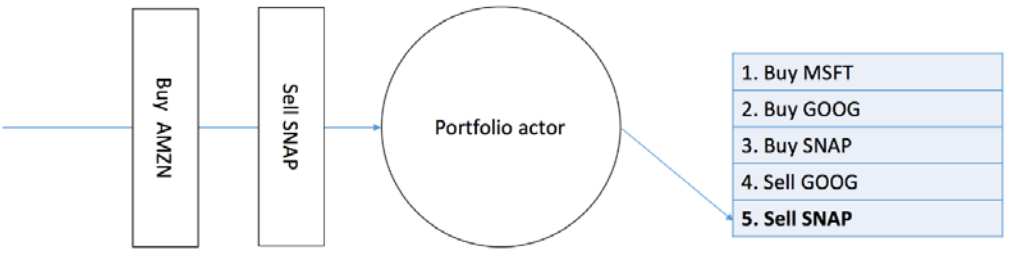


Figure 11.6 - Finance applications need full audit trails in the event of investigation. By using Akka.NET persistence, you can simplify the process of maintaining an audit trail by persisting every event that the actor processes, in this case, a list of trades.

By using Akka.NET persistence, you can maintain a log-based representation of every event that was created by an actor. This allows you to simplify the process of audit trail creation, a feature that is a requirement in many heavily regulated industries, such as finance, insurance, and healthcare.

11.6 Summary

In this chapter, you learned:

- How to add a backing data store to an actor to persist changes to its state
- How to apply concepts relating to event sourcing to actors
- How to upgrade the logic contained within an event sourcing-based application

12

Building clustered applications with Akka.Cluster

This chapter covers

- Creating elastically scalable actor systems that span multiple machines
- Interacting with the underlying Akka.NET cluster infrastructure within an actor system-based application
- Applying the concepts covered throughout the book so far to clustered applications

We've seen throughout the book the numerous tools and design methodologies that are available and that help you build reactive applications. We've seen that for an application to be considered reactive, at its core you need systems that are responsive, regardless of what is happening within the system. This means that you need systems that can handle any failures that might be occurring within the system itself or in any external dependencies, while also surviving any increased load that might occur on the system. We've seen how to get around many of the problems relating to fault tolerance and scalability when running on a single machine through the use of Akka.Remote.

But when you create applications that use Akka.Remote, you architect with the explicit intention of having to potentially perform manual configuration changes. For example, when you had a subset of the actor deployment tree hosted on another machine, you needed to specify the exact network address for the machine that was hosting it. You also had to do the same thing when you assigned routees on separate machines in order to develop applications that were able to scale out across multiple machines. There was also no process in place for what should happen if the machine hosting the actor system failed. In this case, you'd be left in a situation where you had no actor system running for your application. This meant that you'd

need to be constantly monitoring your Akka.Remote-based application and be ready to update the configuration at any point during the day or night so that you can get applications that stay resilient even in the face of failure. Even if you managed to script the process of updating the actor system to point to new instances as they became available, you'd still be faced with inevitable downtime as you'd need to reprocess configuration by restarting the actor system.

Although the Akka.Remote approach certainly presented some key advantages over a single machine, it doesn't yet match all the core traits of a reactive application. We've seen throughout the book how reactive applications can react to their hosting environment, which enables them to remain fault-tolerant in the face of failure or automatically scale when faced with increased load or even decreased load. The key part here is the automatic reaction to their environment. With Akka.Remote, this is something that wasn't automatic and ultimately required some degree of human intervention to manage. In an ideal situation, you'd be able to simply add more machines and have them automatically become part of a single actor system distributed across multiple machines.

Akka.NET provides this functionality through the use of the Akka.Cluster module, which builds on top of the low-level functionality provided by Akka.Remote to create a high-level overview of the networked actor system by creating a cluster of all of the individual actor systems and allowing you to address all of the machines as though they were a single actor system. In this chapter, we'll see how to take a number of independent actor systems and create a cluster out of them. We'll see how to deploy actors into the cluster, and let Akka.Cluster transparently handle the deployment into the system on any machine in the cluster. We'll also see some of the more advanced features available in Akka.Cluster that allow you to more easily build applications that can make the most of the resources available in these vast clusters of machines.

12.1 Introducing Akka.Cluster

Akka.Cluster builds on top of the networking abstraction provided by Akka.Remote to create a scalable, fault-tolerant cluster of machines capable of representing a cluster of actor systems as a single actor system. Akka.Cluster is therefore incredibly useful for scenarios in which you need high availability guarantees that you would not typically be able to achieve when using a single machine to host an actor system.

Some of the concepts contained in Akka.Cluster can be confusing, especially considering it presents a potentially new area of computing, distributed computing. Therefore, it's useful to understand some of the key terms that will be used throughout this chapter:

- *Node*—A node is an individual actor system instance that runs as part of a larger cluster. A node runs an Akka.NET application that connects to and communicates with other nodes in the cluster.
- *Seed node*—A seed node is a contact point for new nodes to join the cluster. It has all of the same responsibilities as a regular node, but it simply has a well-known address that new nodes can contact.
- *Cluster*—A cluster is a set of individual nodes that are joined together through the

membership component of Akka.Cluster. The cluster of nodes is then addressable as though it was a single node rather than a collection of independent nodes.

- *Gossip*—The gossip protocol is an internal component of Akka.Cluster that is responsible for communicating changes in the cluster membership to all of the other nodes in the cluster in an effort to create a uniform overview of the cluster state across all nodes in the cluster.
- *Leader*—The leader is the node within the cluster that is chosen to be in charge of accepting changes to the cluster state by adding or removing new nodes that are then able to be disseminated through the gossip protocol to other nodes in the cluster. There is no leader election process; instead, the leader is chosen deterministically when the cluster is in a stable state.
- *Failure detector*—The failure detector is the Akka.Cluster component that is responsible for detecting whether a given node has become unavailable due to the lack of heartbeat messages being received within a given time period.

Although many of these terms are components that are run internally within Akka.Cluster without the need for intervention from you, they provide the low-level tooling that directly powers many of the high-level features that we'll see throughout this chapter and will be referenced in each section. When all of these components are combined together, you are left with an actor system representation that spans a number of machines. This then presents you with the following benefits, in addition to the other benefits you get when building applications with Akka.NET running on a single node:

- *Scalable*—In order to add a new machine to the actor system cluster, you simply need to provide the address of one of the nodes that is already running within the cluster, which allows you to instantly scale and add significantly more resources on demand.
- *Fault tolerant*—All of the nodes within the cluster periodically send heartbeat messages in order to demonstrate their liveness to other nodes in the cluster. If no heartbeats are received in response from a single node, then the failure detector is able to detect that a node is unresponsive and communicate this with the cluster automatically. If sufficient nodes agree that the machine is no longer responsive, then the node can be automatically removed from the cluster and the actors hosted on that machine can be automatically deployed onto another node in the cluster.
- *Peer to peer*—All of the peers within an Akka.NET cluster have the same responsibilites, allowing you to instantly create new nodes that can talk to any peer within the actor system and then receive events relating to other peers in the cluster.
- *No single bottleneck point* —Because Akka.Cluster creates a peer-to-peer cluster of machines, there is no single machine that coordinates all of the other actor systems. This ensures that you are not in a situation in which the overall performance of the cluster degrades as a result of a single node in the cluster being under heavy load.
- *No single point of failure*—All nodes in the cluster are responsible for detecting failures of other nodes and then disseminating this information to the other nodes in the cluster through the use of gossip messages. This ensures that there isn't a single node in the

actor system that is responsible for handling membership to the cluster, guaranteeing that if one node in the cluster becomes unavailable, then the cluster is able to autonomously fix itself and continuing to run the application.

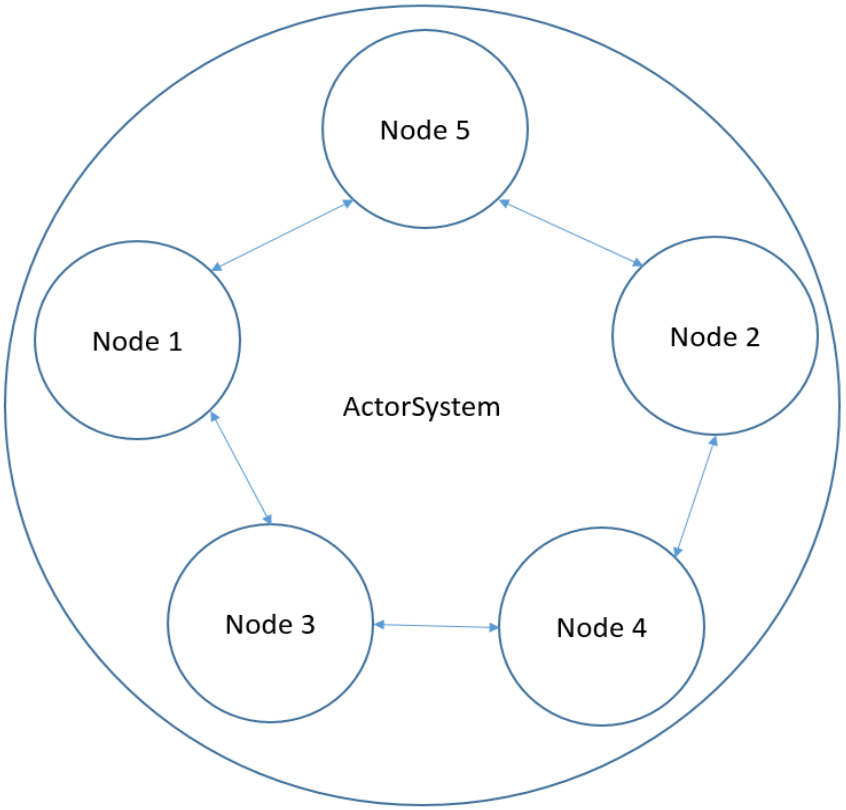


Figure 12.1 By using the clustering capabilities of Akka.NET, you can treat a cluster of individual machines as a single actor system.

Before you can use Akka.Cluster, you first need to install the Akka.Cluster package. As with other Akka.NET packages, it's distributed through NuGet and can be installed by running the following command in the package manager window, or by searching for Akka.Cluster in the NuGet UI:

```
Install-Package Akka.Cluster
```

Having now installed the Akka.Cluster package, you need to configure your application so that it can use the clustering capabilities. All of the configuration for Akka.Cluster is handled

through the HOCON configuration in a similar way to how you configured Akka.Remote earlier in the book. You can see an example HOCON configuration for Akka.Cluster in the following example. There are three key components contained within it; configuring the actor reference provider, specifying an endpoint for Akka.Remote to listen on, and providing one or more seed nodes that you can join to when initialising the cluster. You can see how to configured the actor reference provider to use `ClusterActorRefProvider` rather than the single node actor reference provider. This is similar to when using Akka.Remote you used `RemoteActor-RefProvider`. You also specify a listen endpoint; in this case, port 8080, and you listen on localhost. Because this is just using Akka.Remote under the hood, you can configure this exactly as you did with Akka.Remote, allowing you to specify other properties, such as public hostnames if you're running the application through a load balancer. Finally, you can also see how to add a seed node for Akka.Cluster. In this case, you specify the seed node to be the node's own address. This means that it will join itself and simply form a cluster of one node.

```
akka {
    actor.provider = "Akka.Cluster.ClusterActorRefProvider, Akka.Cluster"
    remote {
        helios.tcp {
            port = 8080
            hostname = localhost
        }
    }
    cluster {
        seed-nodes = ["akka.tcp://Chapter12Cluster@localhost:8080"] # address of seed node
    }
}
```

Now that you have a single node running, you can create a configuration for any subsequent node that uses the address configured for the first node as the seed node. Because a seed node is simply an address of one node that is running within the cluster, once you have more than one node in the cluster, you can specify any of the nodes as a seed node.

```
akka {
    actor.provider = "Akka.Cluster.ClusterActorRefProvider, Akka.Cluster"
    remote {
        helios.tcp {
            port = 8081
            hostname = localhost
        }
    }
    cluster {
        seed-nodes = ["akka.tcp://Chapter12Cluster@localhost:8080"] # address of seed node
    }
}
```

One of the other most useful confiuration changes you can make on a per-node basis is to specify a role for the node. A role is essentially a way of limiting the scope of the work that a node is responsible for. For example, let's consider a case in which you have an application that runs some intense computational work, such as financial modeling, and some IO-intensive workloads such as web page scraping. You can get better performance out of your application if

you tailor the underlying hardware to the usage. This might be by using a machine with a faster network for web scraping and machines with lots of CPU cores for financial modeling. In this case, you don't want IO work happening on machines with lots of cores. In order to counter this, you can use roles, which allow you to scope the work you deploy onto them, as we'll see throughout this chapter. To add node to a role, you modify the cluster configuration to specify the roles that the current node belongs to:

```
akka.cluster.roles = ["network"]
```

You can then create an actor system on each node with the configuration exactly as we had done in previous chapters. It's important to note, however, that you need to ensure that every actor system that joins the cluster has the same name. If the names differ then the node wanting to join will be blocked from joining. This means that every node in the actor system differs only by the listen address. In the following example, you create a cluster with the name `Chapter12Cluster`:

```
var configFile = File.ReadAllText("chapter12.conf");
var config = ConfigurationFactory.ParseString(configFile);
var actorSystem = ActorSystem.Create("Chapter12Cluster", config);
```

Now that you have a number of machines communicating together through Akka.Cluster, you can start to develop applications that use the functionality provided by Akka.Cluster to create more advanced distributed applications. In the rest of this chapter, we'll see some of these features and how to use them in the Akka.NET applications that you build.

12.2 Cluster aware routers

We've seen that one of the biggest benefits of using Akka.Cluster is that you can automatically scale out your actor system onto more nodes as load on the system increases. But you need to be able to make the most of this level of scalability somehow. When we looked at scalability in chapter 7, we saw that the most commonly used Akka.NET feature that allows you to scale the application easily is the selection of routers, all of which are capable of routing messages automatically to actors that have either already been created, in the case of group routers, or are created automatically by the Akka.NET router, in the case of pool routers. You can even automatically add new routees to the pool when demand is too high if you are using pool routers.

In chapter 8, when we saw Akka.Remote, we saw how to integrate routers directly with Akka.Remote so that you can specify that routees should be automatically deployed onto other nodes if you're using pools, or you can specify full addresses for group routers, allowing you to reference actors already deployed onto remote actor systems. Because Akka.Cluster uses Akka.Remote as its networking layer, you can still use this technique to create routers that span nodes in the cluster, but you ultimately end up losing some of the benefits of using Akka.Cluster, namely fault tolerance, as you'd have a configuration reliant on a single node in the cluster, potentially leading to a single point of failure.

To counter this, Akka.Cluster provides the notion of *cluster-aware routers*. These are the same routers that we saw in chapter 7, but they integrate directly with the underlying gossip protocol used by Akka.Cluster to receive notifications of when certain events happen within the cluster. For example, you might choose to be notified when new nodes are joined to the cluster or when existing nodes are suspected to no longer be responsive. In these cases, when used with cluster-aware routers, you can automatically remove routees if you suspect the node hosting those routees has become unavailable, or add new routees if you receive notifications that new nodes have been added to the cluster. Therefore, by using cluster-aware routers, you can use the routers that greatly simplify the development effort required to create scalable applications while also making the most of the scalability benefits provided by Akka.Cluster. All of the routers that are available to you in the standard Akka.NET package are available to use with Akka.Cluster, but the same caveats apply as when using these routers with Akka.Remote. Although many of the routers work well across a network, the smallest mailbox router is left acting effectively as a random router due to the lack of knowledge of mailbox sizes across a network.

12.2.1 Creating cluster aware router groups

As we saw earlier in the book, router groups allow you to specify the paths to a selection of actors and automatically direct messages to those actors. You can do the same thing with Akka.Cluster, but you can send the messages across a cluster of machines instead. The cluster-aware router groups use the underlying cluster membership to determine which of the routees specified exist on machines that are determined to be live within the cluster. If the routee specified exists on a node that is believed to be unresponsive, then messages won't be directed to that target and will instead be sent to other potential routees. It also uses gossip messages to react to changes in the cluster and add or remove routees to the group as their hosts are thought to be live or unresponsive.

You can see an example of this in the following diagram. You see the router and how the routee nodes join the cluster. You can see that first node joins the cluster, which alerts the router that a new potential routee is available. It then verifies that the new node has routees on it, and if it does, it starts to route messages on to it. Later, another node joins the cluster and the process repeats again, allowing the router group to add that routee to its list of available routees.

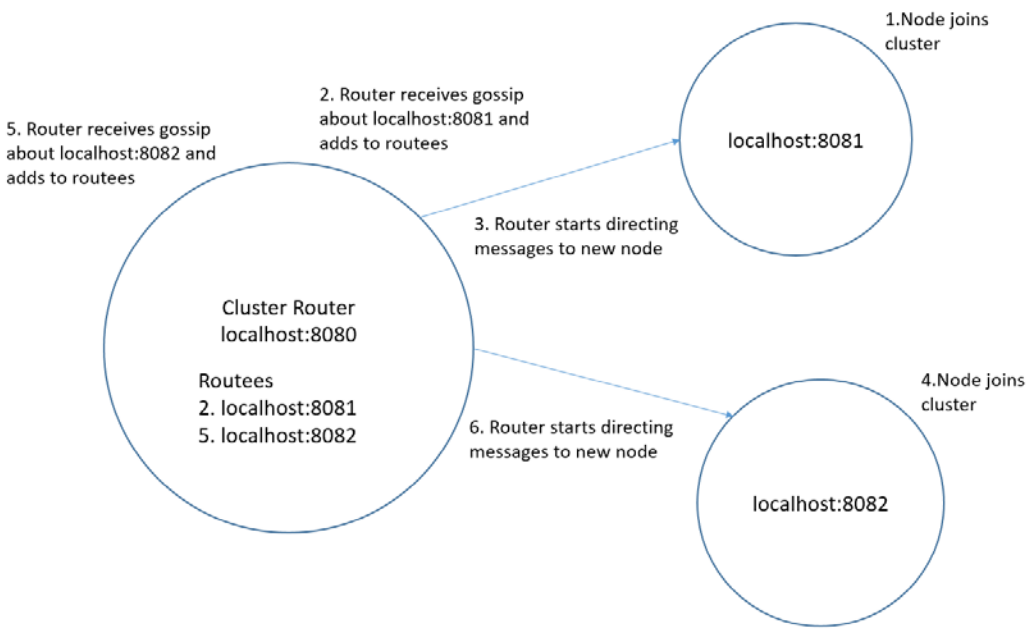


Figure 12.2 - You can see the sequence of events that occur within the router as you add new nodes to the cluster.

When you use groups of actors with cluster-aware routers, it's important that the actors exist at the paths specified as soon as possible after startup. The other nodes within the cluster will all start to send messages to the provided path on any node as soon as it's marked as being up within the cluster. This means that if a routee is delayed at startup, it may lead to race conditions where messages are sent to actors that don't yet exist.

We saw earlier in the book that you can create routers using either HOCON configuration or directly using code. In the following example, you can see how to use HOCON configuration for a router. Within the deployment section of the HOCON configuration, you can specify the router for a given path. The example also shows how to specify a cluster-aware router for a given path. You follow the same format as developing a local router, but you also specify a number of specific properties for creating cluster-based versions. By adding a `cluster` section to the actor deployment, the router is automatically configured to use clustering capabilities. You need to first enable the clustering by setting the enabled property to on as well as specifying whether the routees should only be sent to those nodes that have the specified role. You can also specify that you shouldn't use local routees and instead automatically direct all of the messages to actors on other nodes. This is particularly beneficial in situations where you want to create a master and workers environment, where a number of masters submit work into a cluster of worker nodes. By default, Akka.Cluster limits the total number of routees to 10,000, but you

can configure that number through the use of the `nr-of-instances` property. This means that you can reduce the number if using a number that is too large could overload an external system.

```
akka {
   actor{
      provider = "Akka.Cluster.ClusterActorRefProvider, Akka.Cluster"
      deployment {
        /workdispatcher {
          router = consistent-hashing-group # routing strategy
          routees.paths = ["/user/worker"] # path of routee on each node
          nr-of-instances = 3 # max number of total routees
          cluster {
             enabled = on
             allow-local-routees = on
          }
        }
      }
   }
}
```

Having created the HOCON configuration for an actor, you can deploy it by specifying that the router definition should be taken from the config when you create the `Props` for the router. By using this definition, you can switch between systems that run locally on one node without clustering to creating a multinode clustered environment by simply changing the configuration and restarting the application.

```
var worker = system.ActorOf<Worker>("worker");
var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "workdispatcher");
```

You can also choose to create routers using code instead of relying on the HOCON configuration. In this case, you simply create an instance of the `ClusterRouterGroup` class and supply it a router group that provides the underlying routing logic as well as the settings that define how the router gets deployed across the cluster. In the following example, you create the cluster-aware router using the `ConsistentHashingGroup` router group type, but you don't provide the internal router with any specific paths. These will be provided by the cluster-aware router. The cluster router group also needs a settings object that contains all of the configuration for the cluster router, specifying many of the settings we saw when creating a router using HOCON. You need to specify the maximum number of routees that should be used within the router, you also supply a list of the routee paths that should be used while also including the root user actor in the path. You also specify whether you should use local routes, as we saw with the HOCON configuration, and whether the router should only target paths on nodes belonging to a given node.

```
var routeePaths = new List<string> { "/user/worker" };
var clusterRouterSettings = new ClusterRouterGroupSettings(3, routeePaths, true);
var clusterGroupProps =
    Props.Empty.WithRouter(new ClusterRouterGroup(new
       Akka.Routing.ConsistentHashingGroup("/user/worker"), clusterRouterSettings));
```

By using cluster-aware group routers, you can route messages across to any node in the cluster automatically while also maintaining ownership of the deployment lifecycle of those actors. Because the router has deep integration with the cluster membership, this ensures that you don't need to manage which routees are available within the cluster at any one time.

12.2.2 Creating cluster aware router pools

In chapter 7, we saw how to create actor routers that can automatically deploy all of their routees and scale them on demand by using router pools. Once again, you can create the same type of router across a cluster of nodes with Akka.Cluster. In this case, whenever a new node joins the cluster, the actor is deployed onto the new node and it then gets added to the internal list of available routees. If a node becomes unresponsive, then it gets removed from the list of available routees.

You can see this in the following diagram, where you see the new nodes that join the cluster and how these get added to the router as potential routees. The Node2 node joins the cluster, which triggers the notification to the router. The routee then gets deployed and messages are routed to the new routee. The process then repeats again when Node3 now joins the cluster. The message is sent to the router notifying it that the new node has joined, the routee is deployed to the new remote node and the node is added to the list of routees.

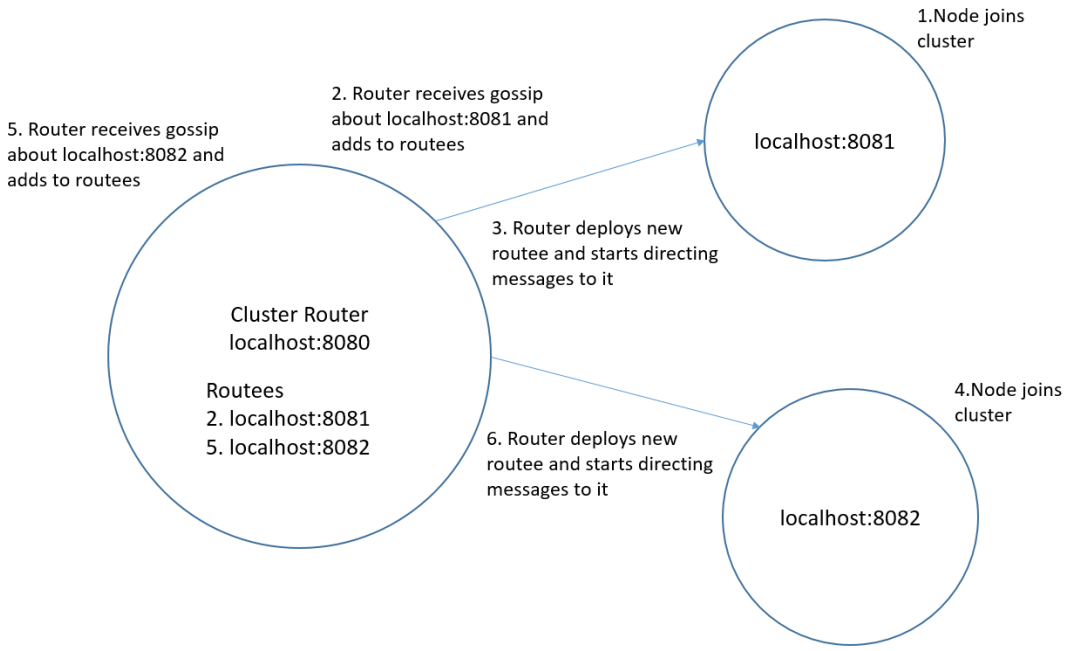


Figure 12.3 - The sequence of events that occurs when a new node is added to the cluster when using a pool router.

Because the only real difference between group routers and pool routers is which actor maintains ownership of the lifecycle of the routees, you can use the same principles to create cluster-aware pool routers as you can create cluster-aware group routers. This means that you can use the HOCON configuration to create a router instance, or you can create the router instance entirely in code. In the following example, you can see how to create the pool router using the HOCON configuration. The only main difference is that you specify the internal router type to be a pool instead of a group, and the removal of the paths that should be used as routees. But there is also the addition of `max-nr-of-instances-per-node`, which specifies how many routees should be created by the pool on each individual node.

```
akka {
   actor{
      provider = "Akka.Cluster.ClusterActorRefProvider, Akka.Cluster"
      deployment {
        /workdispatcher {
          router = round-robin-pool # routing strategy
          max-nr-of-instances-per-node = 5
          cluster {
             enabled = on
             allow-local-routees = on
          }
        }
      }
   }
}
```

Having created the HOCON configuration, you can deploy it using the same approach and specifying that the router definition should be taken from configuration instead of creating it directly in code:

```
var routerProps = Props.Empty.WithRouter(FromConfig.Instance);
```

But should you wish to, you can specify the creation of your router pool directly in code without having to rely on the configuration. When creating a pool, though, you need to specify that you're to create a `ClusterRouterPool`. This works in a similar way to the cluster router group in that it takes in an underlying router to specify the internal logic as well as a settings object that contains any configuration. In the following example, you create a round robin pool that operates across a cluster by passing `RoundRobinPool` to the cluster-aware router that does not have any routees. You also pass in the settings, supplying the values as they are needed. In this case, you need the maximum number of routees per node as well as in total across the cluster, and whether to use the local node to deploy routees or which roles should host the routees.

```
var clusterPoolSettings = new ClusterRouterPoolSettings(1000, 5, true);
var clusterPoolProps =
    Props.Create<Worker>().WithRouter(new ClusterRouterPool(new RoundRobinPool(5),
       clusterPoolSettings));
```

Cluster-aware pool-based routers allow you to incredibly easily deploy actors right across a cluster that are automatically able to handle messages on any node with minimal intervention

from you, allowing you once again to create cluster-based applications without needing to worry about the intricacies of cluster-based development, instead pushing the focus onto the underlying business logic that powers the applications you build.

12.2.3 Cluster-aware routers summary

Cluster-aware routers allow you to really quickly and easily create applications that can scale across a cluster even as you add more and more instances. If you're using the HOCON configuration approach to developing routers, then you can go from a solution that runs locally on a single actor system to running on a cluster of thousands of machines without the need for a code change, instead focusing on the configuration.

12.3 Working with cluster gossip

We saw in the last section how to use cluster-aware routers in order to scale out across the cluster. Although routers provide with a large amount of functionality to quickly develop scalable applications, you sometimes need to drop down to a lower level and interact directly with the cluster itself. You can use the same Akka.NET gossip protocol notifications used in the cluster-aware routers within any actors that you create yourself. In this section, we'll see how to get an overview of the cluster at a point in time, and we'll also see how to react to cluster changes automatically by registering actors for notifications from the gossip service within Akka.Cluster.

12.3.1 Retrieving the cluster state

As you develop cluster-based applications, you may sometimes need to take an overview of the cluster at the current point in time as seen by one of the nodes. For example, you might be building monitoring tooling that needs to see what the cluster is doing at any point in time. In order to do this, you can retrieve it directly from the actor system cluster extension. In the following example, you can see how to directly retrieve the actor system extension:

```
Cluster cluster = Cluster.Get(Context.System);
```

Once you've retrieved the cluster extension, you can then get an overview of the cluster by accessing the `State` property. This allows you to see which nodes are currently members of the cluster through the `Members` property. You can also see which of them are unreachable by inspecting the `Unreachable` set of members. If the cluster is in a stable state, you can also see which member is currently the leader by accessing the `Leader` property, and finally, you can get the set of all roles that currently exist across all nodes by using the `AllRoles` property.

12.3.2 Handling cluster gossip messages

In chapter 6, when we discussed failures, we saw that there are many different sources of failure for an application running on a single machine, and even more once you have a network connecting multiple applications. This means that when you're running an application on Akka.Cluster, you'll encounter situations where you believe a node has become unreachable.

Fortunately, you don't have to worry about these details yourself because the Akka.Cluster failure detector handles periodic heartbeats automatically without the need for any intervention. But you may still want to receive notifications of changes in the cluster state caused by these potential issues. We've already seen one example of how this information is used in this chapter alone when we saw how to create routers that are influenced directly by the cluster status. There are plenty of other scenarios in which you would want to receive this information. For example, if you're storing state within actors and a node hosting one of those actors becomes unreachable, then you might need to switch to using a different node hosting a replica of the data.

Within Akka.Cluster, you can register an actor you develop with the cluster extension to automatically receive notifications of any changes that occur within the cluster. In order to do this, you first retrieve the cluster extension as shown in the following example:

```
Cluster cluster = Cluster.Get(actorSystem);
```

Having retrieved a reference to the cluster extension, you're then able to use the `Subscribe` method to retrieve events within the cluster. When you call the `Subscribe` method, you supply a reference to the actor that is due to receive cluster messages. You also supply a value indicating how you want to receive the initial state of the cluster: either as a single snapshot with a picture of the current state of the cluster, or as a sequence of ordered events that represent the transitions the cluster has taken from its initial state to its current state. In the following example, you specify that you want the current actor to receive the current state in the form of the events that represent the current state, and then you want to receive the member events whenever there's any unreachable nodes:

```
cluster.Subscribe(Self, ClusterEvent.SubscriptionInitialStateMode.InitialStateAsEvents,
       typeof(ClusterEvent.UnreachableMember));
```

You can now handle these messages like any others in order to build up a picture of the current actor system state automatically as soon as any changes are detected in the cluster state by the gossip service. In the following example, you can see how to write out to the actor system log whenever a change is detected in the actor system:

```
Receive<ClusterEvent.UnreachableMember>(msg =>
{
    Context.System.Log.Info("A node was detected as being unreachable: {0}",
       msg.Member.Address);
});
```

By handling cluster events as they happen, you can proactively adjust your application so that you can minimize latency or maximize the availabillity of the application, ensuring that users get the most responsive experience possible.

12.3.3 Working with cluster gossip messages summary

In many instances, you'll rarely need to work with the cluster status APIs directly, and instead you'll build up your applications on top of the abstractions that Akka.Cluster provides,

one of which we've seen already in this chapter in the form of cluster-aware routers, and there are several others that we'll see in the rest of the chapter. But in certain circumstances, by using the cluster status APIs you can build applications that are more responsive, either because they can redeploy actors if the cluster size grows to handle an increased load or you can redirect messages if it becomes apparent that a node failure could cause increased latency.

12.4 Cluster singleton

There are times when you don't necessarily want an instance of an actor on multiple nodes as you would get if you were to use a router. Instead, you might want to have a single instance of an actor that is responsible for performing a certain task. An example of this is if you need an actor that is responsible for handling the coordination of resources across the cluster. In this case, you don't want a large number of actors potentially competing with each other and overloading a given resource. For example, if you were using an actor to decide which machines to deploy a virtual machine or container image onto, you don't want to deploy multiple images that ultimately use up more resources than are available on the machine by having multiple actors handling the deployment.

In this case, you could instead choose to deploy a single actor that is responsible for this task. The Akka.Cluster cluster singleton allows you to deploy an actor onto the cluster on a single node and automatically migrate it whenever the node hosting it becomes unavailable. It's important to note, however, that using this approach and creating a single actor responsible for performing a certain task will lead to the creation of a single point of bottleneck. A single point of bottleneck is a component of the application that may cause the application to slow down and degrade performance, as all traffic must travel through this single component. As such, it's important to consider whether alternative solutions exist that could potentially spread the load and distribute it across a number of actors.

In order to create a cluster singleton, you need to use `ClusterSingletonManager`, which will handle the creation and deployment of the actor into the actor system on the most appropriate node. The cluster single will always get deployed onto the oldest node in the cluster as it is most likely that this node will not be subject to the node churn caused by adding and removing new nodes. In the event that this oldest node is then removed from the cluster, `ClusterSingletonManager` is then responsible for redeploying the singleton onto the new oldest actor within the actor system. It is therefore also possible that during handover while the singleton is deployed, there is no running instance of the cluster singleton.

Another consideration is what happens during a network partition if the cluster thinks it has split and has then automatically marked a node as down. The two halves of the cluster might then deploy their own cluster singleton, leading to a situation where there are two cluster singletons deployed in independent sections of the same cluster. This might lead to problems when dealing with situations where you need to guarantee only one actor is ever accessing an external resource at once, which might cause unexpected behavior. In these cases, it's likely that the best case is to perform manual removal of nodes from the cluster rather than relying on the failure detector to automatically mark unreachable nodes as down.

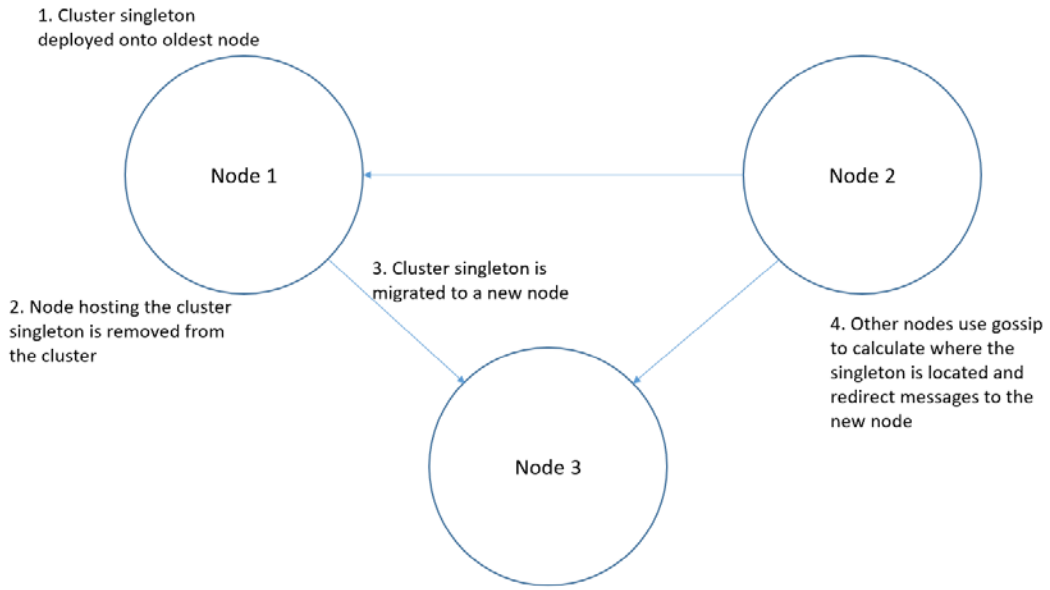


Figure 12.4 - The cluster singleton is a powerful tool that permits only one instance of an actor to be deployed in a cluster.

You create the cluster singleton by using specialized `Props` generated by `ClusterSingletonManager`. In the following example, you can see how you create `Props` for an actor called `Coordinator`. In a typical application, it's likely that this actor is responsible for coordinating certain actions across all nodes in the cluster. As you can see, you create the regular `Props` that you need to create the actor, in which you can specify how to create the actor and which constructor parameters are required. These are then passed to `ClusterSingletonManager`. You also specify the message used for termination, should the singleton need to be relocated in the event that the oldest node becomes available again after a failure. In this case, you just use `PoisonPill`, but if you have more complex work before shutting down, then you may want to use a custom termination message. Finally, you also supply the settings used for handling the cluster singleton. For example, in this case, you have specified that the singleton should only be deployed onto a node that belongs to the `orchestration` role.

```
//Create the Props for the actor which will be deployed on only one node in the cluster
var coordinatorProps = Props.Create<CoordinatorActor>();

//Create the settings for the cluster singleton manager, retrieving them from the actor
      system
//configuration and then overriding the name of the singleton
var settings =
    ClusterSingletonManagerSettings.Create(actorSystem)
```

```
    .WithSingletonName("coordinator");

//Create the Props for the cluster singleton manager by providing the props of the singleton
       and the
//settings to use for the singleton
var clusterSingletonProps =
    ClusterSingletonManager.Props(coordinatorProps, settings);
```

Having created these specialized `Props`, you can deploy the actor in the same way as you would deploy any other actor in our system. In the following example, you can see how to use the `ActorOf` method on `actorSystem` to deploy the actor:

```
var coordinatorSingleton = actorSystem.ActorOf(clusterSingletonProps, "coordinatorManager");
```

Having created a singleton, you now need to create a means of accessing the actor that has been deployed automatically in the cluster. To aid in this, Akka.Cluster provides `ClusterSingletonProxy`, which acts as a message target and allows you to easily forward any received messages onto the singleton's correct location in the cluster. The proxy is also registered to track the cluster state, allowing it to automatically reroute the messages to whichever node is now hosting the singleton. In the following example, you create the `Props` required for the proxy and then deploy it. When creating the `Props`, you need to specify the actor path to the singleton that you previously created. As you deployed it at the top of the actor hierarchy, the path will be `/user/coordinator`. You also need to supply any additional configuration. In this case, you need to specify that you deployed the singleton onto nodes with a specific role. If the roles failed to match, then the proxy would not be able to accurately route the messages to the correct target.

```
//Create the settings for the proxy using the configuration supplied in the actor system
//configuration and override the name of the singleton to match the name provided above when
//creating the actor for the cluster singleton manager
var coordinatorProxySettings =
    ClusterSingletonProxySettings.Create(actorSystem)
    .WithSingletonName("coordinator");

//Create the Props for the coordinator, ensuring the path matches that of the cluster
       singleton manager
//created above when deploying the cluster singleton manager
var coordinatorProxyProps =
    ClusterSingletonProxy.Props("/user/coordinatorManager", coordinatorProxySettings);

//Deploy the coordinator proxy to the path /user/coordinatorProxy other actors will then be
       able to
//send messages to this address and have them automatically forwarded onto the singleton
       wherever it is
//deployed within the cluster
var coordinatorProxy =
    actorSystem.ActorOf(coordinatorProxyProps, "coordinatorProxy");
```

Having created the proxy, you can now send messages to it as you would any other actor in our actor system, and they then get automatically forwarded onto the singleton regardless of where it exists within the cluster.

```
coordinatorProxy.Tell("Hello coordinator!");
```

Cluster singletons should not be the first choice when developing cluster-based applications when using Akka.Cluster, but there are certain tasks that rely upon only having a single instance available in the cluster at any point in time and the Akka.Cluster cluster singleton makes the task significantly easier. By automatically handling migration in the event of failure, it ensures that you always have an instance of the actor available somewhere. When this is then combined with the Akka.Persistence features we saw in the previous section, it allows you to easily create reliable actors that are backed by a datastore.

12.5 Cluster sharding

As you develop applications using Akka.NET, you naturally begin to develop a hierarchy of actors where logical children exist of parents driven by the overall domain design. For example, you may logically group all actors related to a given user of the application so that there is no possibility for cross pollination between users of the application. This, however, presents something of a problem. When dealing with these hierarchies, unless you explicitly state that part of an actor hierarchy should be deployed onto a given remote target, then the actor will be deployed locally directly underneath the actor spawning the child. You may, however, be in a situation in which you have too many actors deployed under a single parent on a given node within the cluster. In this case, actors may end up processing messages slower as not enough CPU resources are available, or actors may even encounter issues when modifying their internal state due to a lack of available memory on the node for all actors involved.

In order to counter this problem, you need to be able to shard these actors across a number of nodes in the cluster so that you can balance the load and prevent any single node in the cluster becoming a single point of failure or single point of bottleneck. In order to prevent this from happening, you need a way to deterministically deploy certain actors within that hierarchy onto alternative nodes within the cluster and then redirect all messages to the node on which the actor is deployed. Akka.NET provides the Akka.Cluster.Sharding module, which provides a means through which you can partition child actors across a cluster into shards without the need for significant work on our part. A shard in this case is a horizontal partition of the actors within our actor system. In the following diagram, you can see how a number of actors with the same parent might be partitioned across a number of nodes within our cluster. By sharding actors, you can deploy a larger number of actors with a common parent than you might typically be able to achieve on a single node.

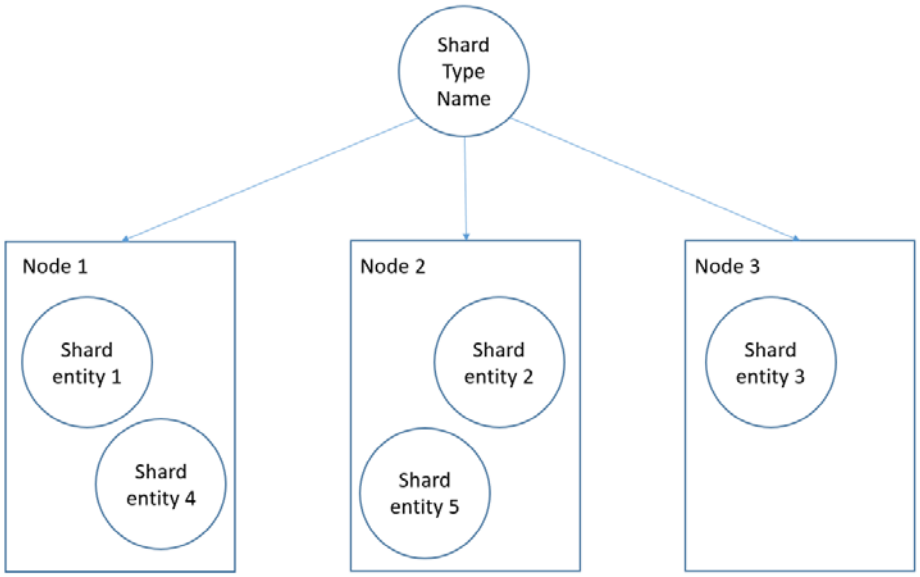


Figure 12.5 - Akka cluster sharding permits the automatic partitioning of actors across multiple nodes within a cluster.

12.5.1 Creating a new shard

Let's consider an example in which you might need such sharding capabilities. We've seen how to create an actor designed for storing the state of a visitor's shopping cart when building an e-commerce website. In this scenario, you may have hundreds of thousands of shopping carts because there may be signed in users as well as guests who may just be browsing. This presents a problem if your application is deployed on a large number of nodes running on commodity hardware. Although you could use machines with more RAM to host all of these shopping carts, it means you'd still be paying for the large amount of RAM even if you're only hosting a small number of shopping carts. This is an ideal scenario for Akka.Cluster.Sharding in that you can easily shard these shopping carts across multiple nodes in the cluster.

You'll define your shopping cart actor exactly as you have in the past. For this example, you'll simply use `ReceiveActor`, meaning that in the event of the cluster shuffling the actors, you'll lose any state stored within the actor. When dealing with Akka.Cluster, the recommended approach is to use persistent actors so that the state can be restored if the actor gets moved to another node. In order to keep this example focused on cluster sharding, you'll simply see the use of `ReceiveActor`. The definition is similar to examples we've seen before, where the actor receives a request to add a new item to the cart or retrieve the cart status.

```
public class ShoppingCartActor : ReceiveActor
{
    private List<string> _items = new List<string>();
```

```
    public ShoppingCartActor()
    {
        Receive<AddItemToCart>(msg =>
        {
            _items.Add(msg.ItemId);
        });
    }
}
```

You will, however, add an extra property onto any messages you send to the actor, which contains the target identifier of the actor you want to send the message to. The reasoning for this will become apparent later in the section, when you see how to direct messages to the correct shard containing the target actor. This addition of the shopping cart identifier means that your messages now look like the following. You still have the same properties that we've seen before, but in addition you have a string containing the shopping cart identifier:

```
class AddItemToCart
{
    public string ItemId { get; }
    public string ShoppingCartId { get; }

    public AddItemToCart(string itemId, string shoppingCartId)
    {
        ItemId = itemId;
        ShoppingCartId = shoppingCartId;
    }
}
```

Before you can create a shard, you first need to create a message extractor. This class will be responsible for retrieving the identifier of the actor from a message as well as the shard on which the actor is located. This message extractor will then be used by the underlying Akka.Cluster.Sharding tooling to handle redirection of messages to the correct actor instance. In order to create a message extractor, you simply create a class that implements the `IMessageExtractor` interface. In the following example, you create a message extractor for your shopping cart actor messages. You need to implement the three methods for retrieving the entity ID, the entity shard, and the actual message. In the case of retrieving the entity ID, you simply check the type of message and if it's one of the messages you defined above then you extract the shopping cart identifier. When dealing with the shard handling, you need to choose an appropriate number of shards to make the most of the clustering capabilities. If you choose too few shards, then you end up in a situation whereby you're unable to make the most of all available resources when you scale out the number of instances. If you choose too many shards, then you end up with increased latency caused by the underlying routing struggling to find the correct node on which the shard is located. A good rule of thumb is to use ten times more shards than the number of nodes expected in the cluster. For example, if there's a 10-node cluster, then use 100 shards. You calculate the shard for our message by using the hash code of the node identifier and then calculating the remainder when dividing by 100. Assuming the hash function is uniform, this then creates a uniform distribution of actors across the

cluster. The final method you need to implement is retrieving the message, because your message contains the shopping cart identifier itself, you just return the message directly. This method is useful if you're storing the message as a property within an envelope message.

```
class ShoppingCartMessageExtractor : IMessageExtractor
{
    //The EntityId method retrieves the identifier for the shopping cart to which
    //the message should be directed
    public string EntityId(object message)
    {
        return (message as AddItemToCart)?.ShoppingCartId;
    }

    //The EntityMessage retrieves the overall content of the message which should be sent
    //to the target actor
    public object EntityMessage(object message)
    {
        return (message as AddItemToCart);
    }

    //The ShardId method is used to work out which shard the message should be directed to
    public string ShardId(object message)
    {
        var hash = (message as AddItemToCart)?.ShoppingCartId.GetHashCode();
        var shardId = hash % 100;
        return shardId.ToString();
    }
}
```

Having defined the actor and the messages it can receive, as well as the message extractor used by the cluster sharding tooling, you need to specify how you tell Akka.NET to deploy the actors across a number of shards instead of onto a single node. In order to create a shard, you retrieve the cluster sharding extension and then call the `Start` method, supplying the shard name to use, the `Props` for how to create the actor that you'll be sharding, as well as an instance of the message extractor you created previously. In the following example, you can see how to define a new cluster shard. Call this on every node that you want to act as a host for cluster sharding. This may be every node in the cluster or only selected nodes, such as those with a specific role.

```
//Create the Props for the actor which resides within a shard
var shoppingCartProps =
    Props.Create<ShoppingCartActor>();

//Retrieve the shard extension from the actor system
var shardingExtension =
    ClusterSharding.Get(actorSystem);

//Create the settings for the cluster sharding, retrieving them from the
//actor system config
var clusterShardingSettings =
    ClusterShardingSettings.Create(actorSystem);

//Build up a shard region which will be responsible for creating and shutting
//down actors on each individual node within the cluster
```

```
var region =
    shardingExtension.Start("shoppingCart",
                            shoppingCartProps,
                            clusterShardingSettings,
                            new ShoppingCartMessageExtractor());
```

Having now created a shard of actors, you can send messages to it and have the underlying cluster sharding infrastructure direct the messages to the correct shard. You need to start the cluster sharding on every node that you want to host actors in an individual shard, and as such, the preceding code to start the cluster sharding will be run on the startup of every node that is part of the cluster. In the next section, we'll see how to communicate directly with the shard, as well as how to create a proxy to communicate with the shard.

12.5.2 Communicating with actors in a shard

Having now created a shard for hosting our actors, you need to be able to send messages to the correct actors within that shard. If you have a direct handle to the shard, for example, if you just created it as in the previous example, then you can simply give it a message. In the following example, you create a message to add a new item to the shopping cart called `1`. When you send this message to the shard co-ordinator, it gets directed to the node that is hosting the shard. If that actor doesn't already exist, then the shard co-ordinator automatically creates a new instance of the actor before sending it a message. If it does exist, however, then it simply sends it a message.

```
region.Tell(new AddItemToCart("REF2201", "1"));
```

There are, however, times when you don't want a node to participate in cluster sharding but do want it to be able to communicate with actors that exist within the shard. In order to handle this, you have to create a proxy that will handle the routing of messages through to the correct shard. In order to create a proxy, you get the actor system extension and call the `StartProxy` method, supplying the name of the shard, which role it's deployed on, and the message extractor you created previously. Having created a proxy, you're then able to send messages to it, and they will automatically get directed to the correct node within the actor system.

```
//Create a shard proxy which will automatically forward messages onto the correct shard which
//is hosting the actor
var shardProxy =
    shardingExtension.StartProxy("shoppingCart", null, new ShoppingCartMessageExtractor());
```

Now that you can communicate with an actor within a shard, you can build up actor system-based applications that scale out across entire nodes with minimal effort. Akka.Cluster.Sharding also provides some other techniques that allow you to build applications that continue to react to their environment, which we'll see in the next section.

12.5.3 Handling passivation within shards

Sometimes, actors within a shard might only be required for a small amount of time before they can be shut down again. They might only receive messages for a five-minute period every 24 hours, for example. In these cases, it's beneficial to shut the actor down because it's not needed. By using Akka.Cluster.Sharding, you can use actor passivation, which allows you to shut an actor down after a period of inactivity and then allow the shard co-ordinator to recreate it when it next receives a message. Passivation is a property of Akka.Clustering that allows you to shut down actors that have not been recently used. For example, if you had a chatroom application used by businesses, then the rooms will only be used within typical working hours, namely 8 A.M. to 6 P.M. Outside of these times, you can shut the actor down so as not to unnecessarily use computing resources that could better be used elsewhere.

In order to create an actor that supports passivation, you simply need to tell the parent that you're shutting down. It's important that you don't use `Context.Stop`, as this may lead to losing messages; instead, you send the parent a `Passivate` message, which then automatically shuts down the actor after it has finished processing any messages. If it then receives another message intended for that actor then it automatically recreates it and starts sending it the new messages. In the following example, you can see how to create a receive timeout of 1 hour, after which the actor receives a `ReceiveTimeout` message. From there, you can send the parent a `Passivate` message, supplying the message that should be sent to the child to shut it down.

```
public ShoppingCartActor()
{
    SetReceiveTimeout(TimeSpan.FromHours(1.0));

    Receive<ReceiveTimeout>(msg =>
    {
        Context.Parent.Tell(new Passivate(PoisonPill.Instance));
    });
}
```

Passivation is a useful concept when dealing with actors that you may only need once every few hours. It allows you to shut down the actors, preserving resources that other actors can use.

12.5.4 Akka.Cluster.Sharding summary

Akka.Cluster.Sharding is an incredibly powerful tool that allows you to rapidly scale out actors across an entire cluster without needing to worry about the underlying details relating to state. This means you can focus on the underlying application logic without needing to worry about what should happen in the event of a node failure or other potential edge cases.

12.6 Distributed publish and subscribe

When we looked at Akka.Remote, we saw how to retrieve a path to an actor on a specific actor system and send a message to that path. Although this approach also works in

Akka.Cluster, you need to consider how Akka.NET handles the movement of actors around the cluster. When we saw the cluster sharding functionality, we saw how the shards hosting actors can be moved around the cluster and onto different nodes. This poses a problem, particularly when dealing with specific paths to actors, because you may be in a situation whereby an actor sends a message to a path on which an actor no longer exists because it has been moved elsewhere in the cluster.

Akka.Cluster provides the distributed publish/subscribe feature, which allows you to automatically send messages to actors that have registered to receive messages on a specific path or on a specific topic. By using `DistributedPubSubMediator`, you can send messages around the cluster to any actors that are listening on a specific path or on a specific topic without having to worry about the node on which the actor is deployed.

Internally, the mediator maintains a registry of actor locations along with subscriptions, which is disseminated automatically around the cluster in an eventually consistent manner. This means that subscriptions may not appear automatically across all nodes, but after a few seconds the gossip protocol used by the mediator will have disseminated any changes to the registry to all of the other nodes in the cluster.

In this section, we'll see the two ways in which you can use the distributed publish/subscribe feature of Akka.Cluster to send messages to all actors interested in a particular topic or only selected actors listening on a specific path.

12.6.1 Topic-based messaging

Topic-based messaging allows you to publish a message onto a named topic, and all of the actors that have subscribed to that topic will receive that message. An example of this is a chatroom where you want to publish a message and send it to all members of the chatroom. In this case, you might have one topic per chatroom; for example, if you had a chatroom called general and another called random you'd create two topics named after their respective chatrooms.

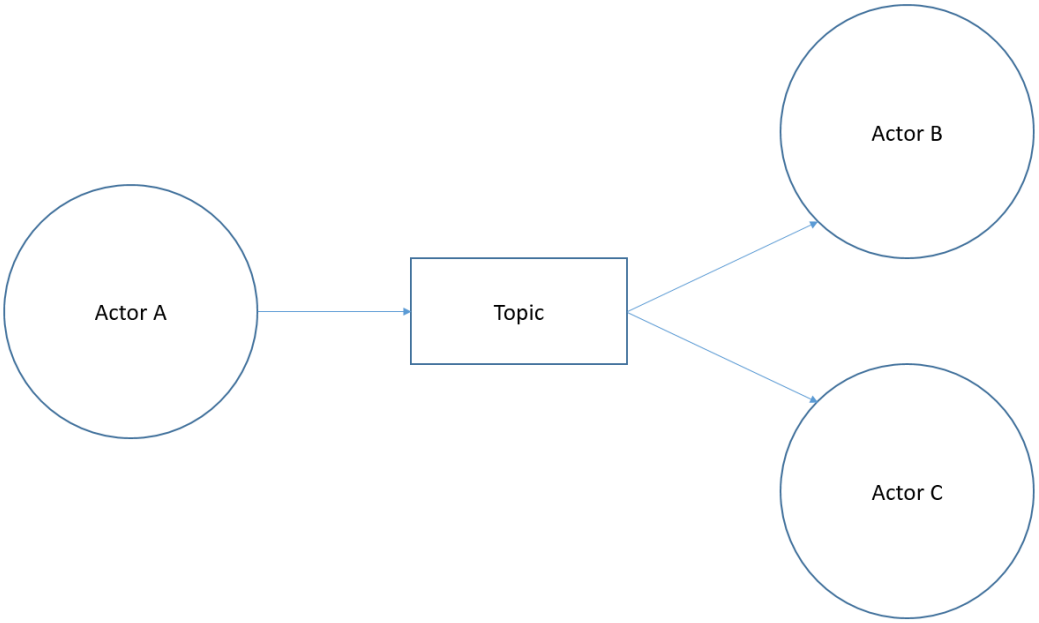


Figure 12.6 - Topic-based messaging allows you to broadcast a message to all actors that are subscribed to a certain topic regardless of their location in the cluster.

You can model each of the users of the chatroom with an actor, which then registers with the mediator for each of the chatrooms the user belongs to. In the following example, you can see a simple actor that receives a message to join a chatroom and then sends a message to the mediator that it should start to receive messages sent to that chatroom. As you can see, you first retrieve the mediator from the distributed publish/subscribe actor system extension. This is the entry point for all applications that need access to the mediator. Having retrieved the mediator from the extension, you're then able to send a message to the mediator requesting that you subscribe to the given topic name when you receive a `JoinChatroom` message within your actor. You send a `Subscribe` message to the mediator that contains the name of the topic that you want to subscribe to, in this case the chatroom name, and also the `IActorRef`, which will receive all of the messages sent to that topic. In this case, you use the `Self` identifier.

```
//Create a class which holds the message which will be sent to the given chatroom
class ChatMessage
{
    public string Sender { get; }
    public string MessageContent { get; }

    public ChatMessage(string sender, string messageContent)
    {
        Sender = sender;
```

```
        MessageContent = messageContent;
    }
}

//Create a class which will allow an actor to join a given chatroom
class JoinChatroom
{
    public string ChatroomName { get; }
    public JoinChatroom(string chatroomName)
    {
        ChatroomName = chatroomName;
    }
}

//Create a class which represents a user which is able to join a chatroom and receive
//messages from that chatroom
class UserActor : ReceiveActor
{
    public UserActor()
    {
        Receive<JoinChatroom>(msg =>
        {
            //Retrieve the pub/sub extension and then subscribe to updates to a given topic
            var mediator = DistributedPubSub.Get(Context.System).Mediator;
            mediator.Tell(new Subscribe(msg.ChatroomName, Self));
        });

        //The pub/sub extension informs us if our subscription to the topic was successful
        Receive<SubscribeAck>(_ => Become(Subscribed));
    }

    public void Subscribed()
    {
        //Now that we're subscribed, you can listen out for messages which get published
        //to the topic
        Receive<ChatMessage>(msg =>
        {
            Console.WriteLine("Received a message from {0}: {1}", msg.Sender,
       msg.MessageContent);
        });
    }
}
```

You can then create instances of this actor on any node within the cluster and regardless of where the actor has been spawned, it will receive any messages that have been posted to the chatroom. You can now expand your user actor to handle a message that tells it to post a message to the chatroom, in this case publishing a message onto a topic. In the following example, you can see how to send the mediator a `Publish` message. Within the message, you specify the topic you want to send the message to, in this case, the name of the chatroom and the message you want to distribute to all of the subscribers of the topic.

```
var mediator = DistributedPubSub.Get(actorSystem).Mediator;
mediator.Tell(new Publish("chatroom", new ChatMessage("user1", "Hello from user1")));
```

The topic-based messaging option within the distributed publish/subscribe functionality in Akka.NET is the true definition of publish and subscribe, allowing you to easily post a message out to hundreds of listeners simultaneously, regardless of their location within the cluster.

12.6.2 Point-to-point messaging

In addition to topic-based messaging, the distributed publish/subscribe functionality allows you to send a message to an individual target within the actor system. Any messages you send using this technique will be delivered to only one target somewhere in the cluster, but once again you don't need to concern yourself with where that target is actually located within the cluster. Continuing with the example of a chatroom that we saw in the previous section, by using point-to-point messaging, you can create a private messaging functionality for your chat application where only one user is sent the message.

In this approach, actors register with their path in the actor hierarchy. For example, you might have an actor that exists in the hierarchy with the path `/user/customers/-customer1/users/user1`. In the following example, you can see how to register your individual actor to receive messages directed to it. Instead of using `Subscribe`, which you used when dealing with topic-based subscriptions, you now use the `Put` message, which simply takes in the `IActorRef` of the actor that is subscribing. The actor reference you send to the mediator must exist on the same local actor system as the mediator, otherwise the subcsription request will be ignored and an error will be generated.

```
public UserActor()
{
    //Register the actor to receive any messages published on the path of the actor
    var mediator = DistributedPubSub.Get(Context.System).Mediator;
    mediator.Tell(new Put(Self));

    Receive<SubscribeAck>(_ => Become(Subscribed));
}

//Spawn the actor at the address /user/anthony
var user =
    actorSystem.ActorOf<UserActor>("anthony");
```

Having registered an actor to listen on its path, you can then wrap your message in a `Send` envelope along with the path to which you want to send the message. In this case, if you send a message with the path `/user/customers/customer1/users/user1` then the individual actor registered with that path will receive the message. In the following example, you can see how to format the actor path based on the username you receive in the `PrivateMessageUser` message. You then send the `Send` message to the mediator, which automatically routes it to the correct node in the actor system.

```
mediator.Tell(new Send("/user/anthony", new ChatMessage("user2", "Hello user1!")));
```

It's important to note that if you have multiple actors registered with the same path then the mediator will choose which target receives the message. By default, it uses random routing logic and so chooses one of the registered routes at random. You can choose to specify that the

mediator should use the local actor system, if a registration exists for the path on the local actor system, by using the `LocalAffinity` flag within the `Send` message. In the following example, you can see how you mark the `LocalAffinity` flag on the `Send` message when you send it to the mediator:

```
mediator.Tell(new Send("/user/anthony", new ChatMessage("user2", "Hello user1!"),
       localAffinity: true));
```

Alternatively, you could choose to modify the routing logic that the mediator should use by editing the HOCON configuration to specify it. In the following example, you can see how to modify the HOCON configuration to specify a round robin approach to distributing the message to multiple subscribers instead of the default random approach:

```
akka.cluster.pub-sub {
  # The routing logic to use for 'Send'
  # Possible values: random, round-robin, broadcast
  routing-logic = round-robin
}
```

You might also need to broadcast a message to all the individual subscribers within the entire cluster with the registered path. For example, you might have three actors registered around the cluster with the same path and all perform the same task for redundancy purposes. In this situation, you might need to broadcast a message to all subscribers to update the configuration, for example. By sending a `SendToAll` message to the mediator, you can broadcast the message to all subscribers.

```
mediator.Tell(new SendToAll("/user/downloader", new ConfigurationUpdate(usePersistentStorage:
       true)));
```

Using point-to-point messaging allows you to easily send messages across a cluster to an individual target without having to worry about where exactly that target is located within the cluster. This vastly reduces the amount of state our application needs to maintain and instead allows you to focus on the core logic of the application.

12.6.3 Distributed publish/subscribe summary

By using the distributed publish/subscribe functionality within Akka.NET, you can solve many of the problems you're likely to encounter when building applications across a cluster. Rather than needing to keep track of lots of state within your application, you can delegate it off to the mediator instead, which handles all of the problems you're likely to encounter had you tried to develop something similar yourself.

12.7 Cluster client

Having now created an Akka.NET application that uses clustering, you've let the underlying Akka.Cluster package deal with many of the tasks related to deployment of actors within the cluster at nodes driven by computation and the reactive nature of the framework. This presents you with a potential difficulty when it comes to accessing the actors stored within it from

outside of the cluster. One example of this is if you have multiple independent Akka.NET clusters running separate applications. You may at some stage need to integrate the two applications together and allow communication between the two clusters. But with no known fixed hosts for actors in the cluster, this means that you can no longer follow the Akka.Remote approach to communicating with remote actors by specifying fixed well-known paths.

To cater for this scenario, Akka.Cluster provides the ClusterClient, which allows you to connect to the cluster without joining it. It's important to note that the ClusterClient should only be used for communicating between separate actor systems and should not be used for communication within an actor system. Instead, the functionality provided by the distributed publish/subscribe component should be used.

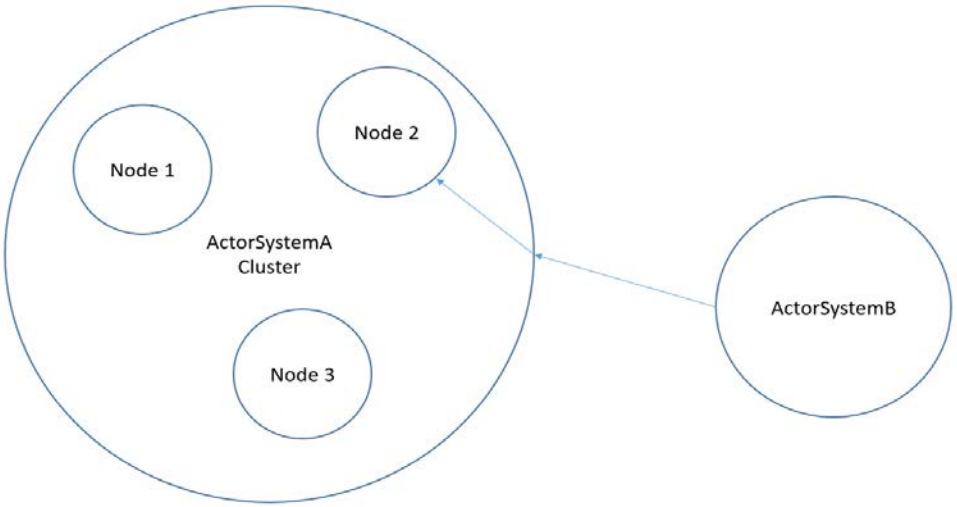


Figure 12.7 - The cluster client allows you to communicate with actors hosted in a cluster from other actors that are located on actor systems outside of the cluster.

The ClusterClient works by specifying one or more known contact points with the cluster that you can use to connect to it, much like the seed nodes we saw earlier when discussing how to get new nodes to join an existing cluster. In order to connect to a cluster, you first need to run the cluster receptionist component on all of the nodes within an actor system or only on selected nodes that belong to a specific role. The first step to using the ClusterClient is to specify which services are accessible to actors that reside outside of the cluster. This increases the security of the cluster by ensuring that external clients can't interact with actors that may be storing sensitive data as their internal state. In the following example, you can see how to retrieve the cluster receptionist extension and then register an actor to be accessible to external actors by calling the `RegisterService` method on the cluster receptionist. In the example, you create an actor that simply acts as an echo service, replying to the original

sender with whatever message it received. You could also choose to use `RegisterSubscriber`, which allows you to listen to a specific topic, similar to the distributed publish/subscribe functionality we saw earlier.

```
//Create an actor which replies to the original sender with the message
//it received
class EchoActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        Sender.Tell(message);
    }
}

//Deploy the echo service to the path /user/echo
var echoService = actorSystem.ActorOf<EchoActor>("echo");

//Retrieve the cluster receptionist extension
var receptionist = ClusterClientReceptionist.Get(actorSystem);

//Register the echo service with the cluster client
receptionist.RegisterService(echoService);
```

Outside of the cluster, you then need to configure your client application to be able to communicate with the cluster. You first need to ensure that you have a port open on your actor system in order to allow messages to be sent in response. You also need to change `actorRefProvider` to use either `RemoteActorRefProvider` or `ClusterActorRefProvider`. In the example client HOCON configuration here, you create a port to listen on and set `actorRefProvider` to use `RemoteActorRefProvider`:

```
Client HOCON configuration
```

Now, on the client, you need to create an actor that acts as a proxy to the cluster. You first specify the initial contact points, which are the nodes within the cluster that are running the cluster receptionist and will allow you to communicate with them. In this case, you've created two receptionist paths that the cluster client will use to try and communicate with. If there are other receptionists running within the cluster on different paths to those that the client already knows about, then they will be sent to the client. This ensures that even in the event of a node becoming unresponsive, the ClusterClient is still able to try and communicate with the cluster through another distinct node. On the client, you can create a ClusterClient proxy that uses the contact points as the initial cluster receptionist points.

```
//The set of nodes which the cluster client will first try to connect to within the cluster
var initialContacts =
    ImmutableHashSet<ActorPath>.Empty
    .Add(ActorPath.Parse("akka.tcp://Chapter12Cluster@localhost:8080/system/receptionist"))
    .Add(ActorPath.Parse("akka.tcp://Chapter12Cluster@localhost:8081/system/receptionist"));

//The settings used to configure the cluster client
var clusterClientSettings =
    ClusterClientSettings.Create(actorSystem)
    .WithInitialContacts(initialContacts);
```

```
//Creaste the Props used to deploy the cluster client on the local actor system
var clusterClientProps =
    ClusterClient.Props(clusterClientSettings);

//Deploy the cluster client into the local actor system
var clusterClient = actorSystem.ActorOf(clusterClientProps, "clusterClient");
```

You can now send messages to the cluster through the proxy and communicate in the same way as you communicated through the distributed publish/subscribe functionality. In the following example, you can see how to send a message to services listening on a certain path within the actor system. Here, you send a message to the `/user/echo` path, which has the echo service listening on it that you created on the cluster:

```
var response = await clusterClient.Ask("Hello echo service");
```

Once a connection is established between a cluster receptionist and a ClusterClient, the client becomes an extension of the cluster. During usage, heartbeats are sent between the receptionist and the client to ensure that both are still available. If the receptionist currently being used for communication fails to respond to heartbeats then it will automatically switch to using a different receptionist hosted within the cluster. By extension, because this information is being sent and received by the client, you can register an actor to receive messages from the cluster client informing it whether more contact points have been added. In the following example, you can see how by sending the `SubscribeContactPoints` message to the client from within an actor, you will be notified whenever the receptionist informs you of more contact points being made available:

```
clusterClient.Tell(SubscribeContactPoints.Instance);
```

Similarly, within the cluster, you can register an actor to receive notifications of when new clients connect to the actor system. In this case, sending a `SubscribeClusterClients` message to the receptionist notifies you when new clients connect to the cluster.

```
clusterReceptionist.Tell(SubscribeClusterClients.Instance);
```

The ClusterClient and cluster receptionist provide a safe means of communicating with an Akka.NET cluster without the need to proxy through an external network protocol such as HTTP. By using the ClusterClient and the cluster receptionist, you can also get an experience that is almost equivalent to our client being a node within the cluster.

12.8 Case study: clustering – scaling – cluster management

For many modern applications, big data processing has become a core component, providing valuable insight and analytics to either the business or the consumer. A big data processing pipeline needs to ingest data from tens or hundreds of disparate data sources, joining rows quickly and efficiently across these datasets before finally unlocking the insight contained within this dataset. In many instances, this is achievable using a single server capable of processing all the data sources. But as the datasets grow in size and count, there's a

lot of difficulties that you might encounter. You may start to be limited by the amount of memory on the machine. If there's not enough RAM to hold all of the datasets, then you end up needing to use techniques that cache files to the hard disk, possibly increasing the time taken. You may equally be limited by the computation that you're performing on your dataset; if the operations you're performing are computationally intensive then it may start to take longer and longer to complete. In an industry where users and businesses are increasingly looking to get results as soon as possible, it's imperative that businesses can match these expectations in order to remain competitive.

In this chapter, we saw how to take an existing actor system and scale it out across multiple nodes thanks to the clustering components of Akka.NET. You can think of a big data pipeline as a sequence of actors. Each actor takes in the results from the previous stage and other datasets before performing any joining, filtering, or mapping, and then it finally passes any results onto the next actor in the chain. An example simple big data pipeline is shown in the following diagram. You can see how you take two datasets and map them into an internal format. After that, the two datasets are joined together based on a common key before a final filter stage removes any unwanted lines. This maps onto four distinct actors, two for performing the map stage on the initial two datasets, one for performing the join, and the final one for filtering the data.

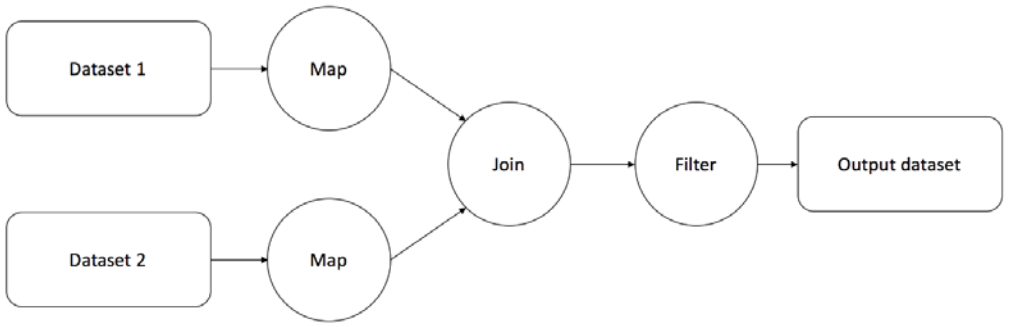


Figure 12.8 - Big data jobs are made up of several distinct stages. By using Akka.NET clustering, you can partition each of these stages across a cluster of machines without needing to understand the underlying network topology.

But as the datasets grow in size, a single server might start to struggle to process all of the data. Because actors are simple to scale, though, you can transparently move the actors around a cluster of machines. The underlying decision on where to host actors is made internally by the Akka.NET runtime, meaning that you can instead focus on the data processing pipeline itself rather than the difficult task of coordinating work across multiple machines.

12.9 Summary

In this chapter, you learned:

- How to take many of the concepts we saw when using Akka.Remote further to allow you to automatically and transparently scale actor systems across multiple nodes
- Patterns used within Akka.Cluster that allow you to design applications that can scale across machines
- Features of Akka.NET that make it easier for you to write distributed applications

13

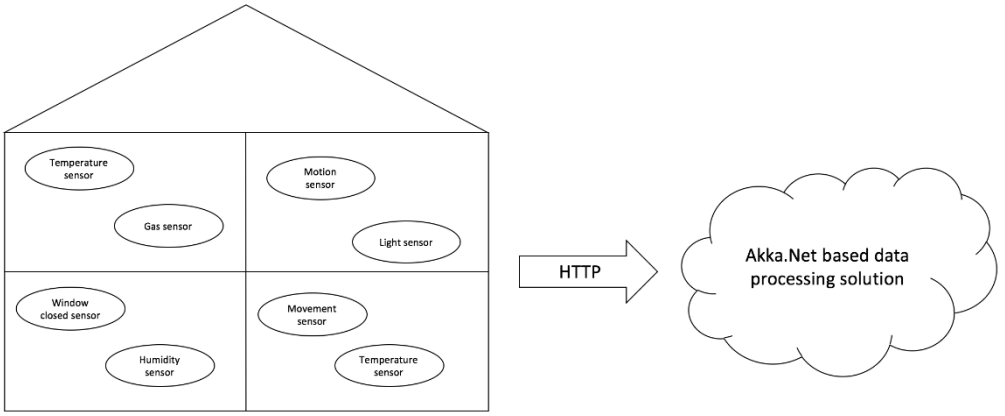
Applying Akka.NET and reactive programming to a production problem

Throughout the book so far, we've looked at several smaller case studies, each chapter detailing how the Akka.NET feature is applicable in a real-world scenario. In chapter 2, we also saw how you're able to design a well-known application using the traits of the Reactive Manifesto. In this chapter, we'll see how you're able to combine the two by designing a reactive system from scratch using features within Akka.NET.

This chapter will focus on the design and development of an Internet of Things application using Akka.NET. Sensors and consumer electronics have continued to grow in usage over the past few decades and it's now possible to create ultra-low-cost and ultra-low-power sensor devices capable of recording data and sending it through an internet connection to be collected elsewhere. This is the type of system we'll look at in this chapter, notably in the context of a smart home, whereby you might have several sensors per room collecting data to give us insights about the overall state of your house. For example, a typical smart home might have sensors such as movement sensors to detect motion, temperature sensors to better control heating within the home, and carbon monoxide detectors to detect the presence of gas within the home. These sensors will collect data at a fixed frequency and send it to a backend system, so that it can be accessed from anywhere in the world, allowing you to see the status of your home when you're at home, at work, or even on holiday. Assuming that this is a product being developed for consumers, you'll also have to deal with multiple people using the application to collect data from their smart home. Given the potentially large number of users with an even larger number of sensors, this will lead to vast amounts of data being sent through your system. You're looking to provide feedback to users as soon as possible after the data has been

collected, meaning that you need a system that is able to scale with the load you're expecting, while also continuing to be responsive in the presence of ever-changing data. Based on the concepts we saw in chapter 1, this makes a reactive architecture an ideal solution for your problem.

The following example illustrates how a smart home may look. There are multiple rooms, with each room having multiple sensors collecting data. This data is then sent to your backend system, where it is processed. In this chapter, we'll only focus on building the backend system from the point of data collection onwards; we won't focus on how to build the physical devices for the smart home.



Once data has been collected within the system, you need to aggregate it in real time, so that it's understandable to users of the application and they're able to gain insight from the collected data. You can do this by storing the time series aggregated data in a database for later presentation in graphs, but you also want to dynamically update the graphs whenever new data arrives to ensure that your application is completely responsive.

13.1 Designing with actors

So far, we've seen at a high level that you want to be able to receive data readings from IoT devices at a given frequency and aggregate those readings into a format that can be presented to users in the form of graphs and warnings. We want to follow the traits of the Reactive Manifesto, so that you're able to build a larger responsive system by being able to scale it out as the load grows; we also want to be able to handle errors by isolating them as much as possible. This will prevent any knock-on effects on other system components.

When designing systems with Akka.NET, it is often beneficial to think of the smallest possible component you can use that can exist independently of other components. For example, in this case, it would be beneficial for us to create an actor per sensor within each smart home. By doing so, we'll be able to add more and more sensors to your system more

easily, without developing a backlog of messages to be processed. It will also be possible to isolate errors should an individual sensor start to send faulty data through the system.

But your sensors exist as part of a broader context, which contains a number of other linked components. Notably, sensors are deployed in rooms, and although you want to be able to see data from individual sensors, you also want to see a broad overview of the state of a single room. Broadening the context further still, individual rooms are constituent parts of a house and you want to be able to see an overview of the status of the house as a whole. Although you could create a single actor, responsible for an entire house and receiving readings for every sensor deployed within the house, that would leave your application open to a potential bottleneck. An alternative is to create a hierarchy representing your overall application structure. This would allow us to be able to perform aggregations on a sensor level, then push aggregated data up to the room level, where further aggregations can be done before pushing those aggregations up to the house level. We'll look at further benefits of the hierarchical approach in the next section, where we consider the failure handling capabilities you get as part of this design.

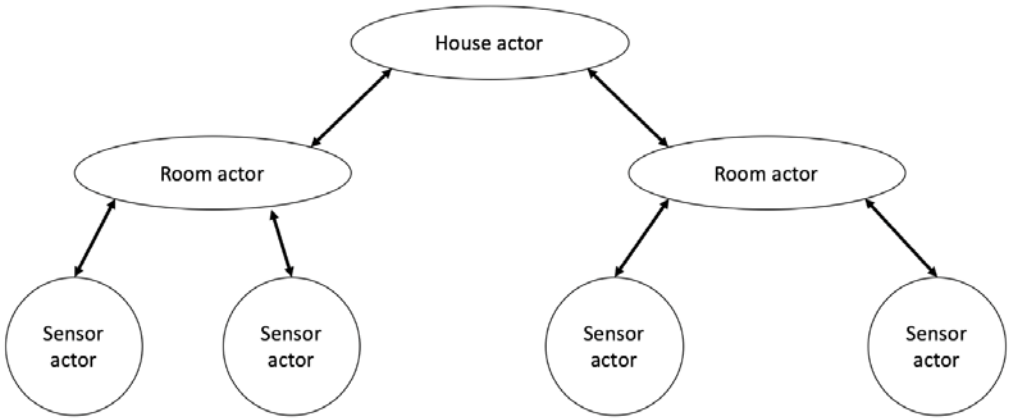
In addition to the ability to perform aggregations on the data, you also want to be able to create alerts. As you collect data, you will be on the lookout for anomalies indicating that something has happened within a sensor, which could indicate an event of interest happening in the smart home. For example, when using a motion sensor, several constant readings indicating movement would suggest consistent movement within the home and is something that should be investigated. But, at the same time, you don't want to send a notification every time movement is detected, or the user may end up receiving hundreds of notifications every day. There should be some logic in place that continues to perform aggregations but does not send data. Although you could set a flag within the actor, this could increase the complexity of your code, especially as the number of potential states grows. For example, you should also consider what would happen if you wanted to reset the actor after a certain period of time, or allow for other functionality, such as a cooldown timer. In such cases, we can create state machines using the functionality we saw in chapter 4. This allows us to change the actor's behavior at runtime to an entirely new function for incoming messages, rather than trying to combine everything in a single handler function.

13.2 Handling failure

When working on any software project, failure will be an inevitable part of it. The likelihood of failure is further increased when you factor in the hardware component of your IoT application. In addition to the possibility of software failure, you need to consider what would happen in the event that one of your sensors developed a fault. In that scenario, you would need to consider the possibility that invalid data, such as null or empty readings, might be sent by a sensor. If such readings were sent to the processing backend, there is a strong likelihood that it would lead to failures for a given sensor. If a sensor does start to fail, we need to ensure that it doesn't affect other system components. In chapter 7, we looked at how you're able to

supervise each actor that gets spawned as a child. This means that you need to build up a hierarchy of actors within your system.

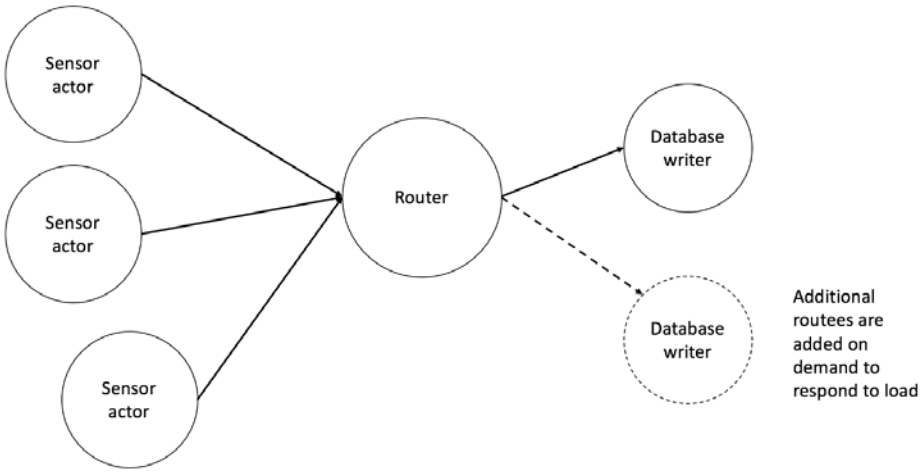
Fortunately, you have a very hierarchical data model in place as it stands. Every sensor is deployed within a room, and each room exists as part of a house. This means that, at the very top level, you have an actor that represents the house. Deployed under the house, there are a number of actors for each room, and then each room has a number of actors deployed for each sensor. This means that if a sensor starts to fail, it will propagate the error up to the room, which will isolate that sensor's errors from the other sensors. If the sensor fails on a frequent basis, you can then choose to escalate the error up to the level of the house so that you can display an error more prominently within the application. This results in the hierarchy shown in the following diagram. By modeling your application as a hierarchy of components where the children are responsible for the most dangerous work, you're able to build systems that isolate failure in the smallest individual component.



13.3 Designing for scale

Given the rise in popularity of IoT devices in recent years, it's possible that your system will see continuous growth over the next few years. This continuous growth will lead to more data being sent through the system and this is something you need to consider. We've already seen that another thing to consider in order to ensure that your application is reactive is the persistence of data in a database, in a format optimised for reading. To do that, you'll create an actor whose sole responsibility is to receive a record, which is then inserted into a database. You'll create an actor dedicated to this task primarily as an optimisation. Although you could create a database connection per sensor, that has the potential to lead to performance problems caused by connections not being reused. It also prevents us creating batches of writes against the database, as you have to send each row for each sensor, rather than writing a number of rows at once for a selection of sensors.

This creates a system that looks like the following diagram. In this case, you have several sensors, all sending data to a single actor that is responsible for writing data into the database. But it's feasible that the number of sensors will grow, forcing more data through the database writer actor. As more data is forced through this actor, the message queue will grow, reaching the point where there is a significant delay in writing the data into the database. This will reduce the overall responsiveness of the application, which ultimately goes against the aims of a reactive architecture. To ensure that your system can stay responsive, you need to eschew bottlenecks, which limit the overall throughput of your system. Fortunately, in your case, you're able to deploy multiple database writers, and thanks to Akka.NET's routers, you're able to treat a collection of independent writers as a single actor. This then provides us with the benefit of being able to add more writers as the load increases, ensuring that your application can scale with the expected load.



13.4 Handling configuration

As you design your system, it's important to make sure it continues to be maintainable and testable. This includes the ability to run the application and its dependencies in multiple environments to verify that all functionality is working as expected. You'll also want to ensure that the same application is running in each of these environments. For example, you want to be able to run the application locally on a developer's machine, as well as in a production environment. In these two cases, it's important to use different databases so that we don't include test data in your production database or risk customer data by using it in a test environment.

In the last section, when we were looking at designing your application for scale, you saw how you will have a dedicated database writer that will be responsible for pushing aggregated data into the database in an efficient manner. By parameterising the connection string that this

actor uses, you're able to run the same application both in production and locally on a developer's machine by simply modifying the connection string. But within your application, you'll also have a certain configuration for the likes of routers, which are common across both environments. In chapter 5, we saw how we can create configuration files that can then be automatically accessed by actors within the actor system.

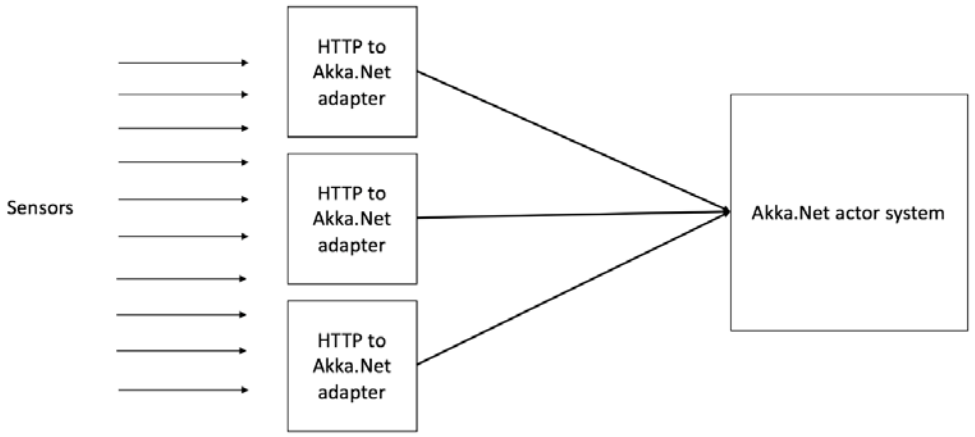
In your application, you'll create three configuration files, one being the common configuration responsible for holding the overall actor system configuration, and the other two providing environment specific settings, notably connection strings for the database. By splitting this into two files with different responsibilities, you're able to simply override the connection strings and merge the two types of file.

13.5 Ingesting data

Now that you have an internal infrastructure in place, you need to start to receive data from the IoT devices that have been deployed in people's homes. The way you'll do this is by creating a publicly accessible API, through which the devices are able to send messages. This API will be responsible for the security of any data sent to the system, as well as converting it from an efficient protocol into the internal format of the actor system.

Although in chapter 10 we saw that you're able to write a custom protocol that interacts directly with the actor system thanks to Akka.IO, you're going to use a HTTP collection endpoint instead. This offers a number of benefits, including the existence of a number of pre-existing security solutions, which will help to protect users' data as it is being sent to backend systems. As these devices are being deployed in users' homes, you'll also have a stable internet connection with sufficient bandwidth to send data through HTTP, rather than needing to heavily optimise the protocol.

Although your actor system will hold all of your state, you should ensure that your ingestion nodes are stateless, so that you're able to quickly provision more of them on demand as more data flows into the system. You can achieve this by separating your core actor system (which holds your state) from your edge nodes (responsible for data ingestion), and then connecting the two together using Akka.NET Remoting, as we saw in chapter 8. Because the data only flows from devices to the system in its current state, the ingestion nodes simply act as a pipeline stage, converting the messages from one format to another. This then leaves us with an architecture that starts to resemble the following diagram, in which messages are sent from sensor devices to the ingestion nodes, where they are immediately forwarded on to the actor system.



13.6 Testing

Having now got the underlying actors designed and in place, it's important to verify that they function as expected. We can do this by creating a test project that exercises the expected behavior of the actors by sending a number of messages when the application runs in production. In chapter 9, we saw how we can test actors using a number of different methods, either testing them in isolation or testing how they integrate with other actors. These tests should be designed to test as much of the functionality of the actors as possible, to eliminate any surprises when the system is running in production and new messages are delivered.

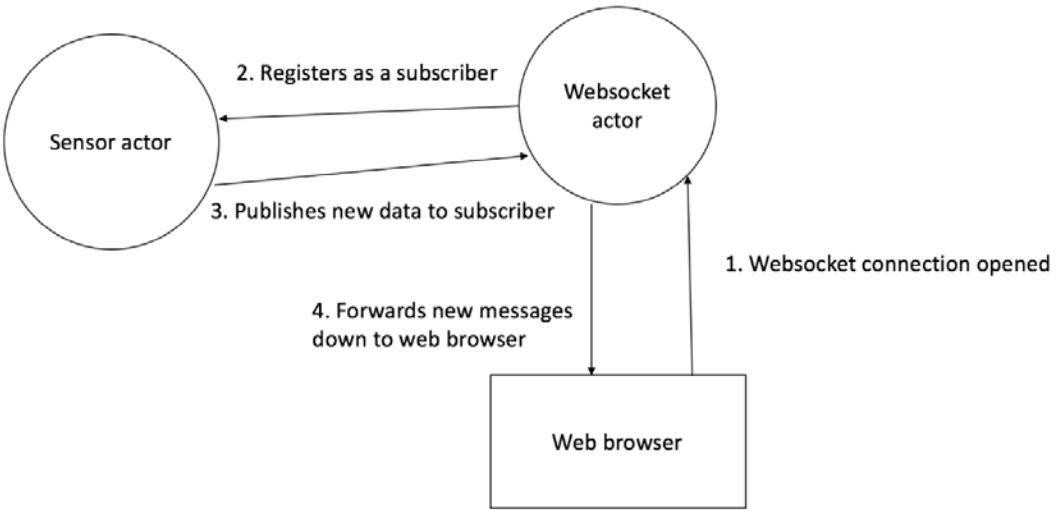
By ensuring that a suite of tests is written, you're able to verify that we won't break the functionality of the system every time you make a change to one of your actors, allowing us to catch errors before the application is put into production. This ensures that you're able to fix any errors that may arise before users discover them, which might negatively influence users' opinions of the system.

You're also able to further integrate the tests that you write with a continuous integration server. By constantly checking the overall quality of the build as it is checked in source control, fewer bugs will slip into the production environment than if you were using a manual testing process.

13.7 Real-time integration

One of the key features that you want within your application is the ability to push new data down to clients as it becomes available. This will ensure that users can see the most recent data processed by the system and then respond to it appropriately. For example, if a user has a motion detector installed as part of their smart home and they have the web page open, they will want to be notified if movement is detected as soon as it happens so that they're able to take appropriate action.

You can use one of the techniques we looked at in chapter 10 to achieve this and add a websocket connection outside of the actor system. Whenever a user navigates to a web page that has got a websocket-based protocol on it, the actor system will be notified that the user is connected. The actor system is then able to register the websocket as a subscriber to any events published by, for example, a room or an individual sensor. In the following diagram, you can see the process for a user navigating to a web page with a websocket connection and how the actor system is then responsible for managing any internal subscriptions to what happens when a new event is generated.



13.8 Data persistence

We have seen so far how you're able to write data in a database in an optimised time series schema, but there are also a number of other pieces of static data that you'll want to store alongside a sensor as metadata. This data includes the time period you should use when performing aggregations for a given sensor. For example, some sensors may need a finer level of granularity than others. In the case of a motion sensor, you want to receive an alert as soon as a number of events have occurred, whereas with a temperature sensor, you might be happy to use a broader timescale. This is all data that will be configured by users, so you need to be able to persist it in the event that you need to restart your sensors or redistribute your sensor actor onto a different node.

Although you could choose to persist the data in a similar fashion to the database writer we saw earlier, a better option in this scenario is to use the persistence layer included as part of Akka.NET, which we looked at in chapter 11. Using this, we can get a simplified data access layer included as part of an actor's definition. By using the persistence components, we can persist specific events that flow through an actor. In this case, you'll choose not to persist the

individual sensor readings; instead, you'll store the events that change the actor's behavior. This includes events that will manipulate the overall output of messages, as defined by your application.

13.9 Cluster scale out

As more customers start to use the service, it's inevitable that you'll need to use more computing resources to handle the increased amount of data flowing through the system. As you've designed the actors within your system to be as independent as possible, you can choose to locate them on any machine you want, because there are no direct dependencies between two distinct actors. This means that we can use the cluster sharding capabilities of the Akka.NET clustering tools to distribute the sensors around a number of different machines, as we saw in Chapter 12. Given the hierarchy that you've developed so far (where you have a top-level actor for each individual home, which has several rooms, and each room has several sensors), by partitioning the actor system at the level of the home, we can maintain the locality of all of the sensors and rooms, which are children of the home. This ensures that components that may need to communicate with each other are located on the same machine, meaning that all communication between those actors will be faster than if the actors were located on different machines. This opens up potential opportunities later in the application's lifetime to allow for increased intelligence across sensors.

We'd then be able to add more machines to the cluster, either manually or using the auto scaling capabilities provided by many cloud providers. In chapter 12, we saw that we can connect more nodes to the cluster at any point in the application's life, and they will automatically become part of the cluster without the need for manual intervention. This allows us to respond to any increase in demand the application may face in the future and automatically scale to handle an increased load. In addition to this, the underlying Akka.NET clustering implementation can detect individual node failures and reallocate resources from the cluster to other healthy nodes. This means that you're able to build systems that are elastically scalable and resilient even to failures more significant than those affecting only small components of the system.

13.10 Conclusion

We can start to see how components fit together when designing systems using Akka.NET. Although there is no single one-size-fits-all solution to every problem, this chapter has given you an idea of the sorts of considerations we need to make when architecting broader solutions. When this understanding of system design with Akka.NET is combined with a solid understanding of the principles of the Reactive Manifesto, it presents an opportunity to create systems that are robust and scalable. Having seen this broader case study, along with the smaller case studies included in each chapter, you should now have an understanding of how the individual components of Akka.NET fit into a larger reactive system architecture, as well as how to develop those individual components specifically using Akka.NET.