O'REILLY®

Compliments of Lightbend

Designing Reactive Systems

The Role of Actors in Distributed Architecture

Hugh McKee

Designing Reactive Systems

The Role of Actors in Distributed Architecture

Hugh McKee

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Designing Reactive Systems
by Hugh McKee

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian Foster
Production Editor: Nicholas Adams
Copyeditor: Kim Cofer
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

September 2016: First Edition

Revision History for the First Edition
2016-09-06: First Release

978-1-491-97088-1

[LSI]

Table of Contents

CHAPTER 1
Introduction

We are in the midst of a rapid evolution in how we build computer systems. Applications must be highly responsive to hold the interest of users with ever-decreasing attention spans, as well as evolve quickly to remain relevant to meet the ever-changing needs and expectations of the audience.

At the same time, the technologies available for building applications continue to evolve at a rapid pace (see Figure 1-1). It is now possible to effectively utilize clusters of cores on individual servers and clusters of servers that work together as a single application platform. Memory and disk storage costs have dropped. Network speeds have grown significantly, encouraging huge increases in online user activity. As a result, there has been explosive growth in the volume of data to be accumulated, analyzed, and put to good use.

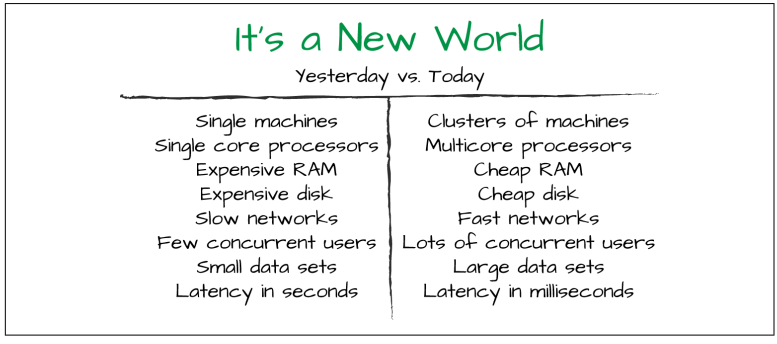

Figure 1-1. It's a New World

Put simply, science has evolved, and the requirements to serve the applications built nowadays cannot rely on the approaches used over the past 10–15 years. One concept that has emerged as an effective tool for building systems that can take advantage of the processing power harnessed by multicore, in-memory, clustered environments is the Actor model.

Created in 1973 by noted computer scientist Carl Hewitt, the Actor model was designed to be "unlike previous models of computation... inspired by physics, including general relativity and quantum mechanics."

The Actor model defines a relatively simple but powerful way for designing and implementing applications that can distribute and share work across all system resources—from threads and cores to clusters of servers and data centers. The Actor model is used to provide an effective way for building applications that perform tasks with a high level of concurrency and increasing levels of resource efficiency. Importantly, the Actor model also has well-defined ways for handling errors and failures gracefully, ensuring a level of resilience that isolates issues and prevents cascading failures and massive downtime.

One of the most powerful aspects of the Actor model is that, in many ways, actors behave and interact very much like we humans do. Of course, how a software actor behaves in the Actor model is much simpler than how we interact as humans, but these similar behavioral patterns do provide some basic intuition when designing actor-based systems.

This simplicity and intuitive behavior of the actor as a building block allows for designing and implementing very elegant, highly efficient applications that natively know how to heal themselves when failures occur.

Building systems with actors also has a profound impact on the overall software engineering process. The system design and implementation processes with actors allows architects, designers, and developers to focus more on the core functionality of the system and focus less on the lower-level technical details needed to successfully build and maintain distributed systems.

> "In general, application developers simply do not implement large scalable applications assuming distributed transactions."
>
> —Pat Helland

In the past, building systems to support high levels of concurrency typically involved a great deal of low-level wiring and very technical programming techniques that are difficult to master. These technical challenges drew much of the attention away from the core business functionality of the system because much of the effort had to be focused on the functional details.

The end result was that a considerable amount of time and effort was spent on the plumbing and wiring, when that time could be better spent on implementing the important functionality of the system itself. When building systems with actors, things are done at a higher level of abstraction because the plumbing and wiring is already built into the Actor model. Not only does this liberate us from the gory details of traditional system implementations, it also allows for more focus on core system functionality and innovation.

Summary

Technology adoption is rarely cyclical; however, in case of the Actor model (created in the early 1970s) the spotlight is swinging back to this unique approach to distributed, concurrent computation. As Forrester Research points out in "How To Capture The Benefits Of Microservice Design" (2016), the Actor model is receiving "renewed interest as cloud concurrency challenges grow" in enterprises building microservices architectures.

This report is targeted toward decision makers in the enterprise and provides some high-level insight into how actors and actor systems can be used to create lightweight business systems that evolve quickly, that can scale, and that can run without stopping. Inside, you'll read how the Actor model's proven approach to concurrent computation is the best way to build distributed systems correctly from the start, allowing your teams to focus on the business logic of your applications instead of wiring together low-level protocols, in turn helping you accelerate time-to-market while keeping infrastructure costs low.

CHAPTER 2

Actors, Humans, and How We Live

Imagine a world where most people are glued to small, hand-held devices that let them send messages to other humans across oceans and continents... wait, we already live in this world!

With actors, it is much the same. The only way to contact a software actor is to send it a message, much like how we exchange text messages on mobile devices. As an example, consider a typical text message exchange between you and a friend. While commuting to work you text your friend and say "Good morning" (see Figure 2-1).

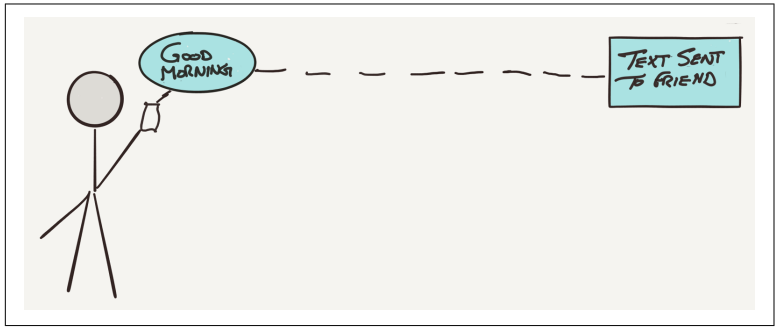

Figure 2-1. Actor messages are like text messages

After you send your friend a message and before she responds, you are free to do other things, such as sending text messages to other friends. It's conceivable that you would also receive requests via text messages to perform other tasks, which may send you off to do other things and interact with other people.

Your friend may quickly see the message, and responds "Hello, how R U today?" (see Figure 2-2).

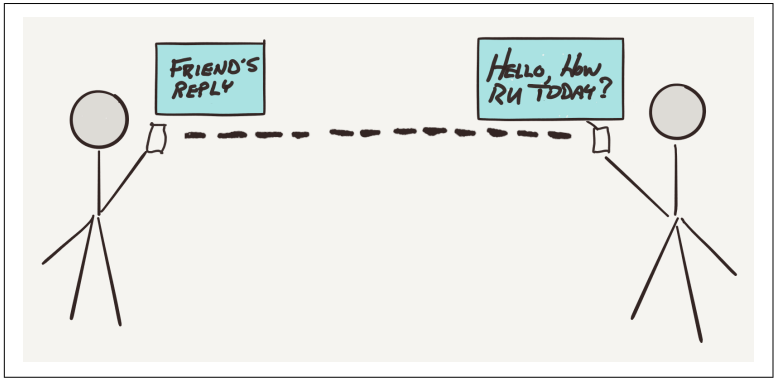

Figure 2-2. Actors behave like humans exchanging text messages

This is basically how messages between software actors behave. When an actor sends a message to another actor, it does not wait for a response; it is free to do other things, such as send messages to other actors.

When you send a text message to a friend or colleague there are a number of possible outcomes. The typical expected outcome is that a short time later you get a response message from your friend. Another possible outcome is that you never get a response to your message.

If you expect a response and never get one (see Figure 2-3), you might wait for a while and then try to send her another text message (see Figure 2-4).

In between getting a quick response and no response is the possibility of getting a response after you are unable to or uninterested in continuing the conversation, because your attention is focused elsewhere. In this example texting conversation scenario, you still do not get a response.

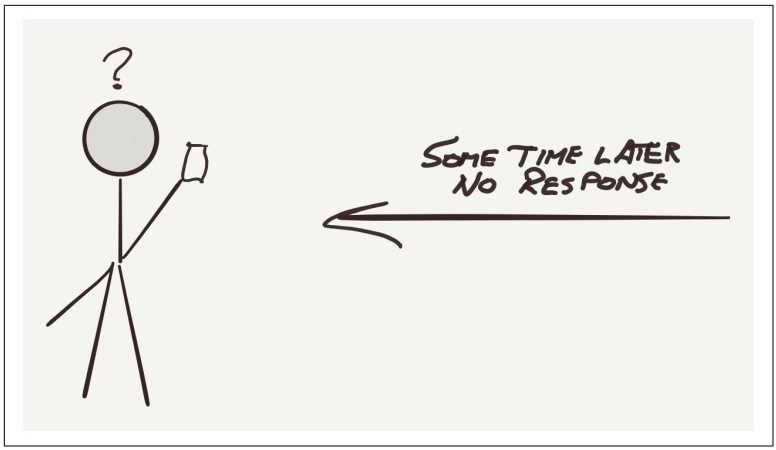

Figure 2-3. No response to your text, what do you do?

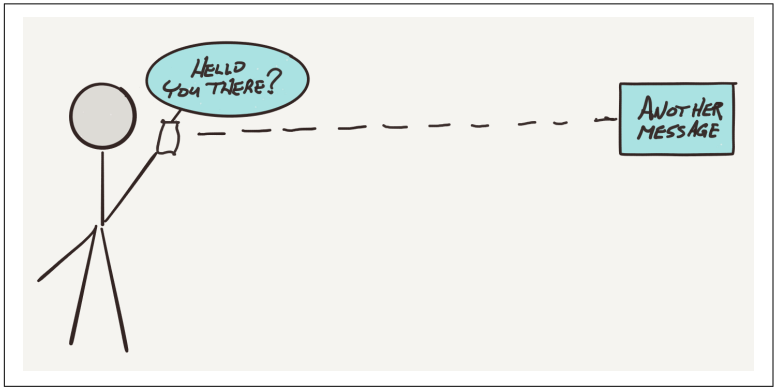

Figure 2-4. No response to a text, send another text

At this point you may be getting somewhat concerned (see Figure 2-5). You expected your friend to respond, and now you are unable to get in touch with her. So what do you do next? This is where you may have to consider your options, such as:

- Wait some more and try to message her again
- Call your friend by phone
- Get in touch with someone else that may know the status of your friend

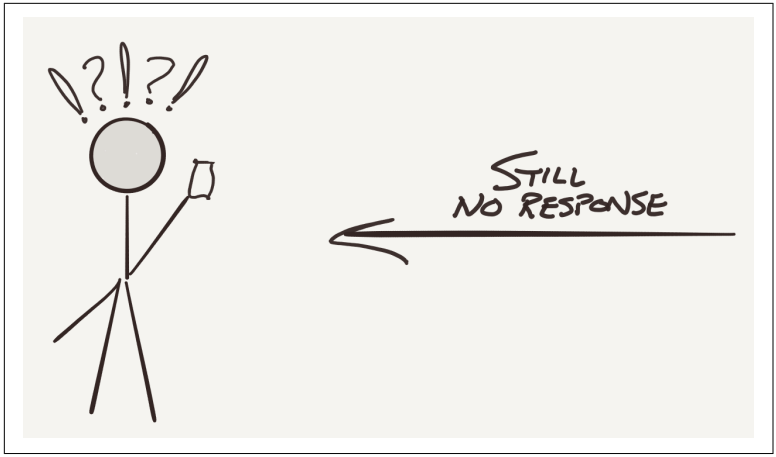

Figure 2-5. Still no response

To be fair, there is a fourth option: stand still and do nothing until you hear from your friend. Blocking all activities until a response is received is probably not considered a feasible thing for most humans, and would make for a pretty inefficient world; yet in many traditional systems, this is precisely what happens, and why the Actor model was created.

Just as with humans sending text messages, actors that can send and receive messages are prepared for multiple possible outcomes: a response may be received quickly, there may be no response at all, or there may be a response that is too late to be useful. The point here is that the actor may be implemented to handle one or more of these possible outcomes.

A well-defined actor can not only handle a typical expected message exchange, it may be capable of handling the other possible outcomes. For example, Actor A sends a message to Actor B. The message is a request to B to perform a specific task.

The desired outcome is for B to perform the task and then send a response message back to A (see Figure 2-6). There is a possibility that B is unable to perform the request task and, as a result, A never gets a response (see Figure 2-7). A is prepared for this and has a plan to handle not getting a response.

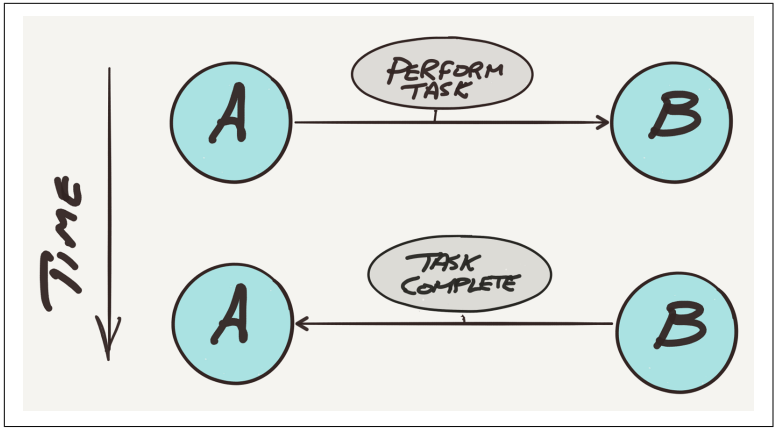

Figure 2-6. Actor A sends a message to B and later gets the expected response

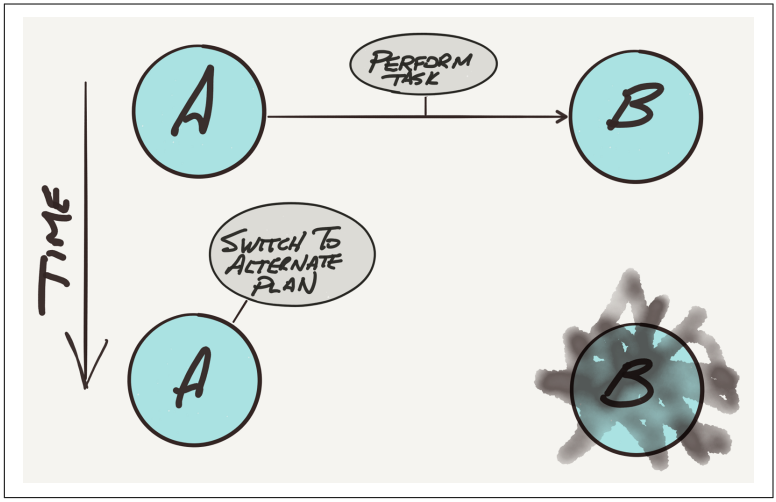

Figure 2-7. Actor B never responds to message from A

A common approach for handling a no-response scenario is for the actor to send a request to another actor to perform a task, setting up a timeout message to be sent to itself at some point in the future. If A asks B to perform a task and B responds back to A before the timeout message arrives, then all is well and A cancels the timeout message (see Figure 2-8).

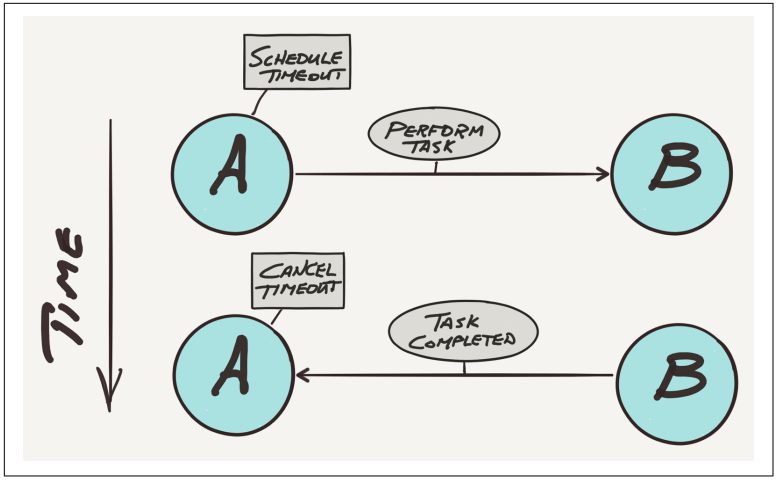

Figure 2-8. Actor A sends message to B and it responds before the timeout

On the other hand, if the timeout message arrives, this triggers A to switch to an alternate plan. How it handles this alternate scenario is very specific to each actor. In some cases, Actor A may simply resend the message to Actor B. In other cases, Actor A may give up and report an error back upstream to the sender of the message that triggered A in the first place (see Figure 2-9).

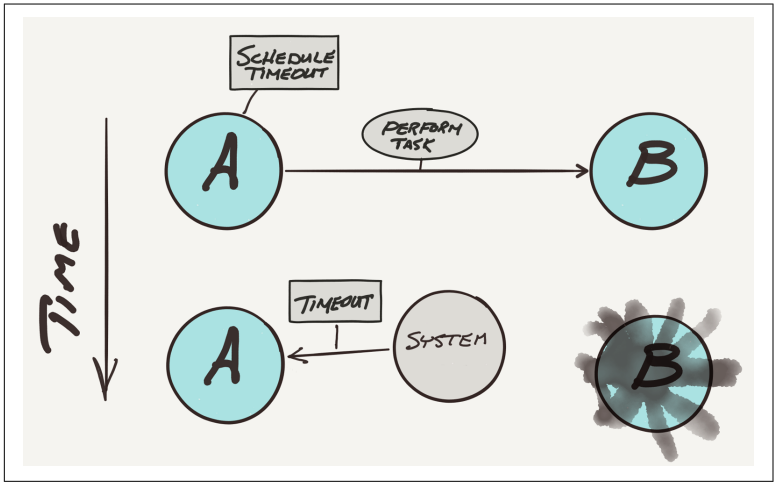

Figure 2-9. No response from Actor B and A gets timeout message

The key point here is that actors exchange messages with each other to communicate. These messages are sent asynchronously, very much in the way humans exchange text messages. Due to this asynchronous behavior, actors are designed not only for the expected happy path, they are also designed and implemented to handle the unhappy path. In the Actor model, failure handling and recovery is an architectural feature, and not treated as an afterthought.

Actor Supervisors and Workers

Actors may create other actors. When one actor creates another actor, the creator is known as the *supervisor* and the created actor is known as the *worker*. Worker actors may be created for many reasons, but among the most common reasons is for delegating work. The supervisor creates one or more worker actors and delegates work to them (see Figure 2-10).

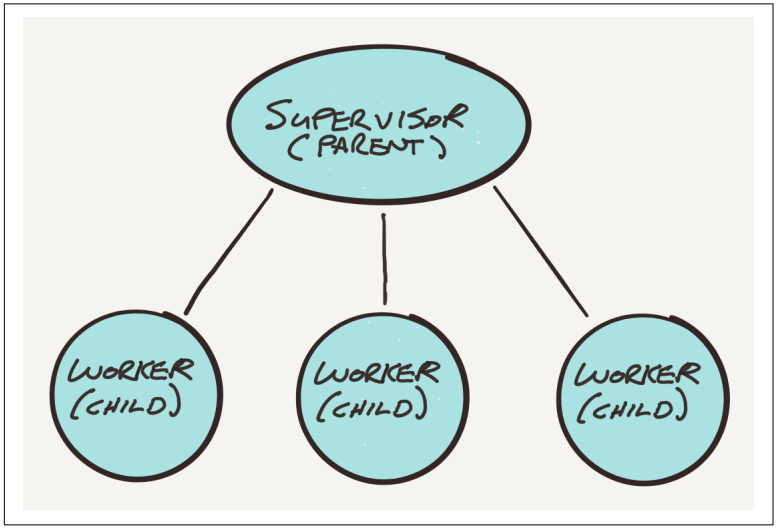

Figure 2-10. Supervisor actor creates worker actors

The supervisor also becomes a caretaker of the workers. Just as with a parent watching over the well-being of their children, the supervisor watches out for the well-being of its workers. If a worker runs into a problem, it suspends itself and notifies its supervisor of the failure (Figure 2-11).

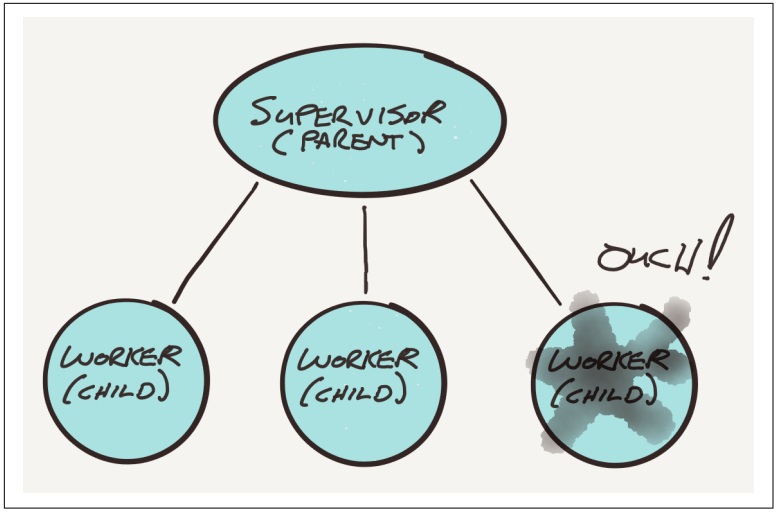

Figure 2-11. Worker actor has problem and notifies its supervisor

When the supervisor is notified it determines what should be done to handle the problem and how to get the worker actor back into a healthy state (see Figure 2-12).

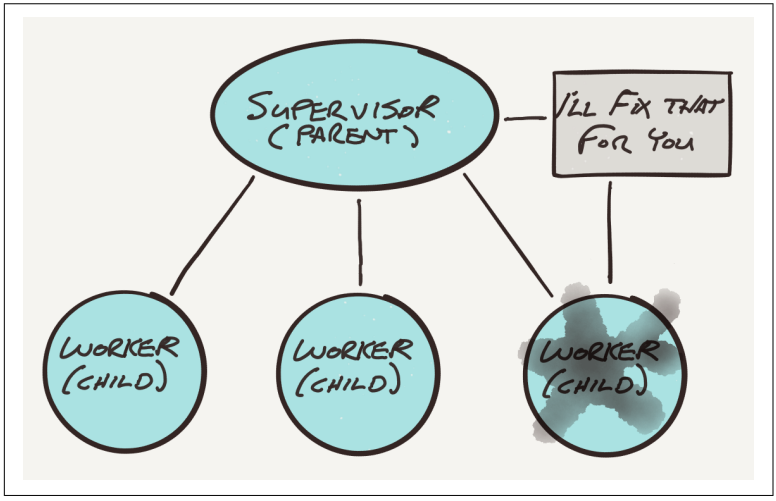

Figure 2-12. Supervisor fixes worker that has experienced a problem

Because the supervisor can send messages asynchronously, without having to wait for a response, the supervisor may send messages to each of its workers, which run independently and concurrently. This is in contrast to the traditional sequential programming model.

Let's consider a scenario where a supervisor actor receives a message to perform a task. In order for the supervisor to complete this request, it must complete three subtasks (see Figure 2-13):

- Obtain the customer profile
- Pull recent customer activity
- Calculate customer-specific sales opportunities

The supervisor delegates this work to three worker actors. The individual actors used are specifically designed to handle the individual tasks, so that one actor knows how to get a customer's profile and the other two actors are each designed to pull recent activity and calculate sales opportunities.

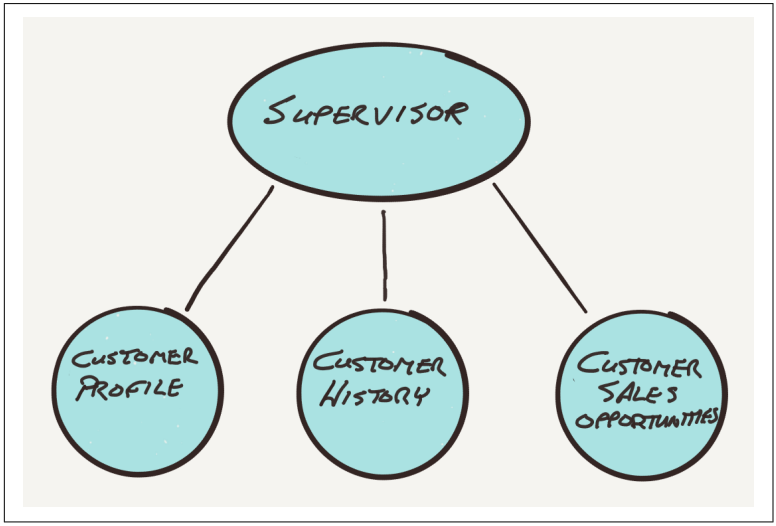

Figure 2-13. Supervisor actor delegates tasks to worker actors

The supervisor sends messages to each of the three worker actors asking them to perform their specific tasks. When each worker completes its task, it sends a message back to the supervisor with the results. Once all three workers have responded, the supervisor combines the results into a single response to the actor that sent it the initial request.

There are a number of benefits to this approach of delegating the work out to worker actors. One key benefit is *performance*. Because the workers run concurrently, the performance is based on the over-

all time it takes for the supervisor to respond and reduced to the response time of the slowest worker.

Contrast this to the traditional synchronous approach: when these three tasks are processed sequentially, the overall response time = task 1 time + task 2 time + task 3 time (see Figure 2-14). In the Actor model, these three tasks are processed asynchronously, so that the overall response time = maximum(task 1 time, task 2 time, task 3 time).

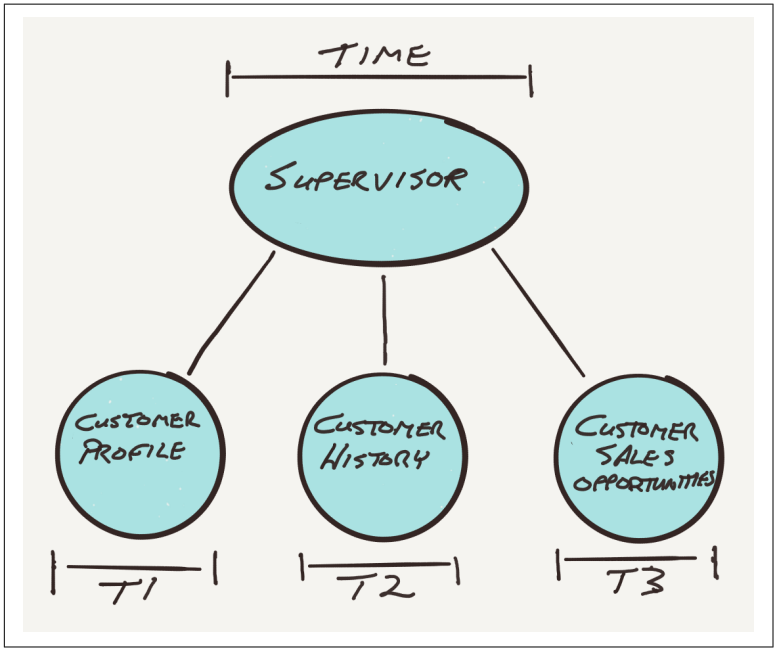

Figure 2-14. Worker actors perform tasks asynchronously

The asynchronous approach also *scales more efficiently* (Figure 2-15). The cost of adding more workers is much less than in a synchronous implementation.

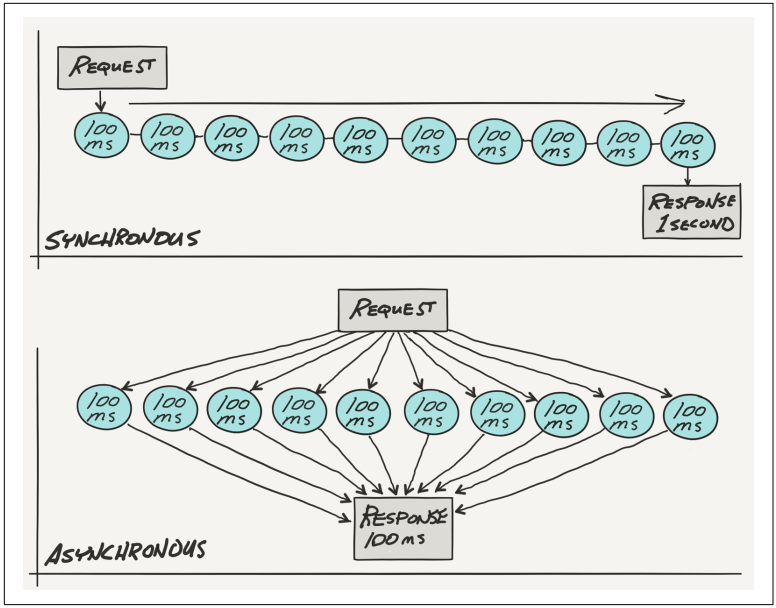

Figure 2-15. Ten 100 ms tasks performed synchronously take 1 second while the same ten tasks performed asynchronously take about 100 ms to complete

Say there is a need to add more functionality to this process, and you want to do more things for the customer to make the application more useful and attractive. With the asynchronous approach, the overall response time is still the response time of the slowest worker, while the response time for the synchronous approach increases with each added worker. This is exactly how highly efficient and feature-rich sites like Netflix and LinkedIn architect their systems.

Another benefit of the asynchronous delegated approach is related to *failure resilience* and *recovery*.

Recall the timeout approach previously discussed. In this scenario, the supervisor schedules a timeout message to be sent to itself when it sends tasks to each of the workers. This provides a way for the supervisor to handle workers that are unable or are slow to respond. The supervisor will only wait so long for the workers to respond. If a worker does not respond in time, then the supervisor decides how to proceed. How this is handled is application specific, but some typical approaches used are to simply omit the missing information,

or to submit some default information in its place. The main point here is that a slow service or a down service is not going to break the system.

The main takeaways of this chapter are:

- Actors exchange messages asynchronously.
- Actors typically are set up to handle all contingencies, both when things work as expected and how to handle the unexpected.
- Actors can create other actors in a supervisor-worker relationship.
- Supervisors can delegate tasks to workers, and also take care of workers that run into problems.

CHAPTER 3

Actors and Scaling Large Systems

> One actor is no actor. Actors come in systems.
>
> —Carl Hewitt

As discussed previously, actors can create other actors in a supervisor-worker relationship, where the supervisor delegates tasks to the workers. In the previous example, the supervisor delegated specific subtasks to the workers. Another common pattern provides a level of elasticity as more work requests arrive, and then the supervisor delegates the tasks to idle workers. If there are no idle workers, then the supervisor adds more workers (see Figure 3-1).

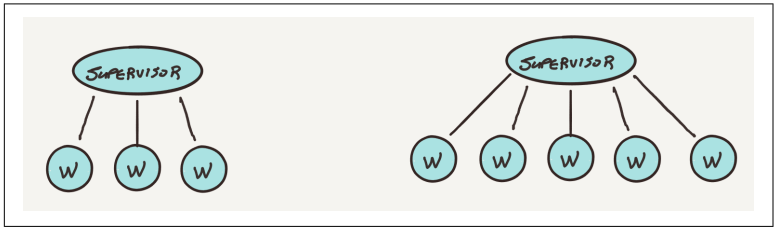

Figure 3-1. Supervisor adds more worker actors as the processing load increases

How the supervisor decides to add workers is up to the supervisor. Supervisors typically have some limit as to how many workers they will add, as well as a built-in way for shedding excess idle workers (see Figure 3-2).

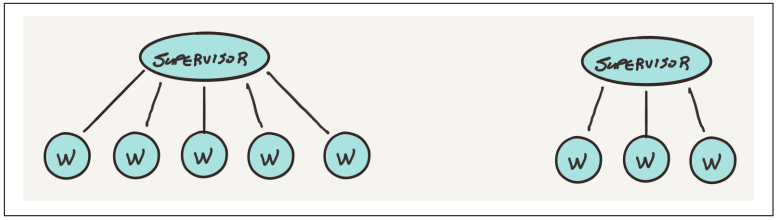

Figure 3-2. The supervisor sheds workers as the load decreases

Here again the scheduled timeout mechanism comes into play. Each of the workers could schedule an idle timeout message to be sent at some point in the future. Every time a worker gets a request to perform a task, it resets the timeout. If no tasks are sent to a worker, and it receives the timeout message that tells the worker that it has been idle for too long, it triggers a shutdown of itself.

Using this relatively simple pattern provides for elastic scalability on the actor level, where a supervisor is free to expand out when the workload increases and contract in when the workload decreases. Other strategies and technologies may be used to support various forms of elasticity across the complete system, but the basic functionality of actors allows for a lot of creativity in how application systems may be designed and implemented.

Another point to be made here is that the supervisor is also an actor. From the perspective of other actors that send it messages, the fact that the supervisor is delegating tasks out to workers is completely transparent to the senders. Imagine again sending a friend a text message asking him to perform some task. You have no idea how he may actually carry out that task. Your only concern is that the task gets completed, and it gets completed in the time frame that you expect. If it happens that he is delegating these tasks to a team of workers, it is none of your concern. It is also transparent to you that while he's working on your task, he may also be concurrently working on tasks for others.

A Look at the Broader Actor System

Actors exist within an *actor system*, and the process of actors sending asynchronous messages to each other is handled by the actor system. Yet an actor system provides much more than just the protocol for messaging between actors; it is also designed to manage

actor activities by allocating the system resources needed to support the actor environment.

The following section covers how the actor system coordinates the activities of the actors and how the actor system allocates system resources at a conceptual level. We use an analogy of comparing actors working in the actor system to that of office workers working in office spaces. The goal here is to provide some insight and intuition of the mechanics of how the actor system runs without going too deeply into the technical details.

Virtually every computer system is composed of a set of processing cores, some memory (RAM), and other resources such as disk space, network interfaces, and other devices. In this discussion the focus is on memory and processing cores.

When programs run in a computer system, the actual work is handled by what is called a *thread* or *thread of execution*. Just as each computer system has a limited number of cores and a limited amount of memory, there is also a limited number of threads. A typical computer system has a small number of cores, and common small or medium-sized servers may have 4 to 16 processing cores. (Just looking at Amazon Web Services instances, you can see it's possible to provision cloud instances with 128 processor cores and 2 terabytes of in-memory RAM.) For memory (RAM), this varies from a few gigabytes to terabytes. A typical server has 8GB to 64GB, but this number keeps going up as the cost of memory decreases in light of technological improvements. For threads, the count typically falls into the hundreds range.

The number of cores and the amount of memory dictates how much work a given computer system is capable of handling. This is roughly analogous to a room. Depending on the size of an office space, more workers will fit in the room if it is larger, which means it can handle more work (threads).

How Actors Manage Requests

In this example scenario, we have a room that can fit three desks at which workers sit and do work. Requests to do work come into the office, with each task delegated to an available worker. Workers can only work on their assigned tasks when the desk has power, which in this example scenario is a limited resource (see Figure 3-3).

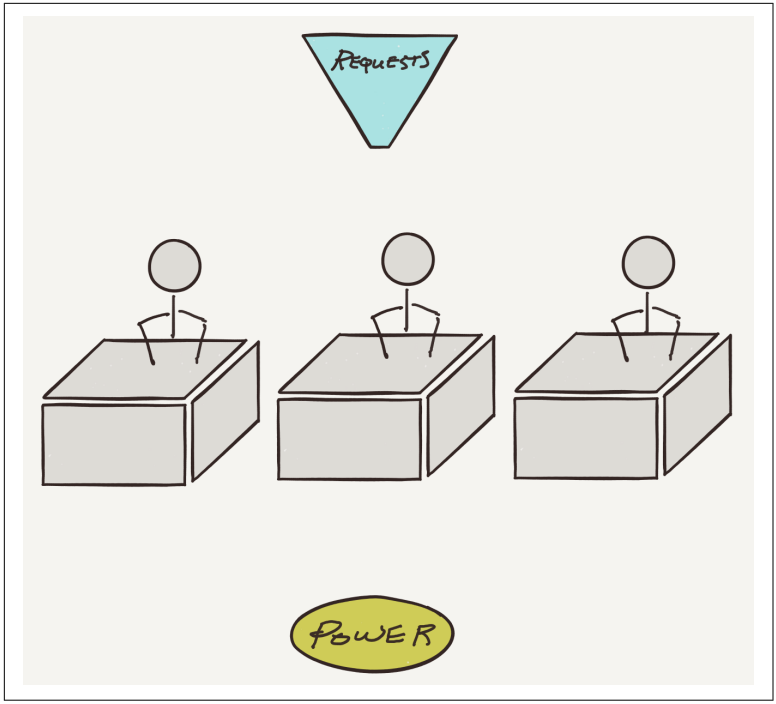

Figure 3-3. An office example where workers sit at desks to perform tasks only when the desk has power

We are going to use this office space analogy to discuss how the actor system coordinates actor activity in a computer system. The idea is that the room size determines how many desks will fit into a room. This is an attempt to model how the memory size of a given computer system determines how many threads can be supported. So room size is analogous to memory size, and the number of desks is analogous to the number of threads. Workers sit at desks to get work done.

To model the processing cores, the idea is that a worker sitting at a desk can only do work when the power is turned on (see Figure 3-4). This demonstrates how a given thread is only active when it is given a processing core to run on. Compared to the number of available cores on which to do work, there are many more threads representing the requests for work to be done. In this office analogy, there are a limited number of desks that can be powered on at any point in time.

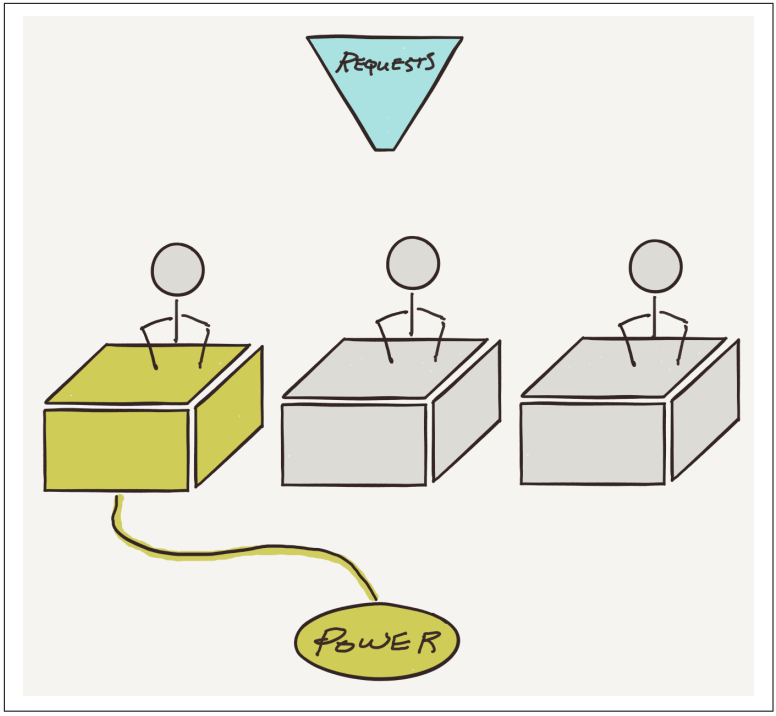

Figure 3-4. One desk has power, so the worker at that desk may work on its assigned task

Now imagine that in this hypothetical office the workers perform tasks that are coming in from the outside. For example, customers are sending in text messages to the office. These text messages are delegated to available workers sitting at the desks (see Figure 3-5).

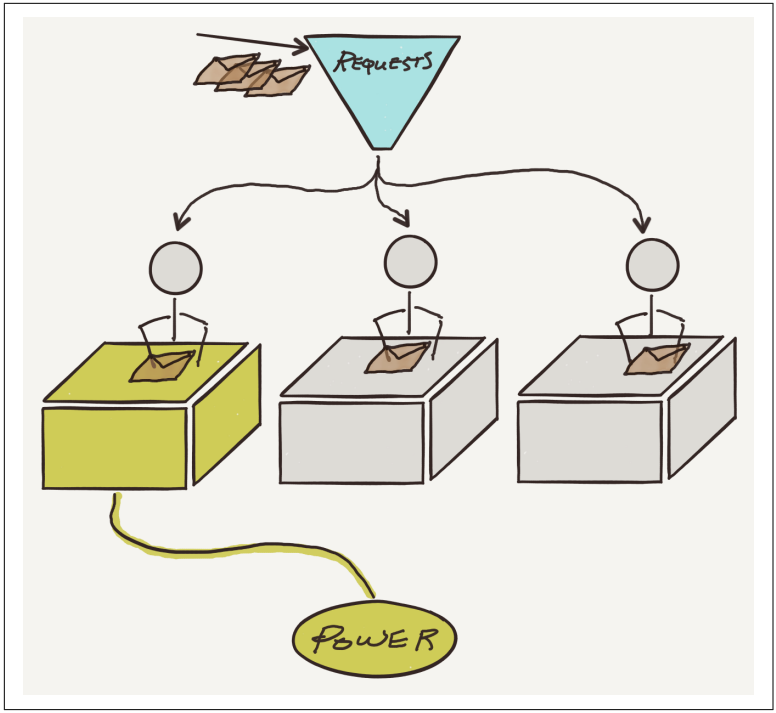

Figure 3-5. Messages are sent to the office and then routed to each worker

The power is constantly being shifted between desks to the workers that are ready to process these text messages. It is the responsibility of the system to make sure that no single worker can hog one of the limited power connections (see Figure 3-6).

This works the same in a computer system, which must constantly shift from thread to thread, allocating limited processing cores while trying to maintain a fair utilization of cores with all of the threads that are ready to run. So in the same way that our office example has more desks than available power connections, computer systems also have many more threads than available processing cores.

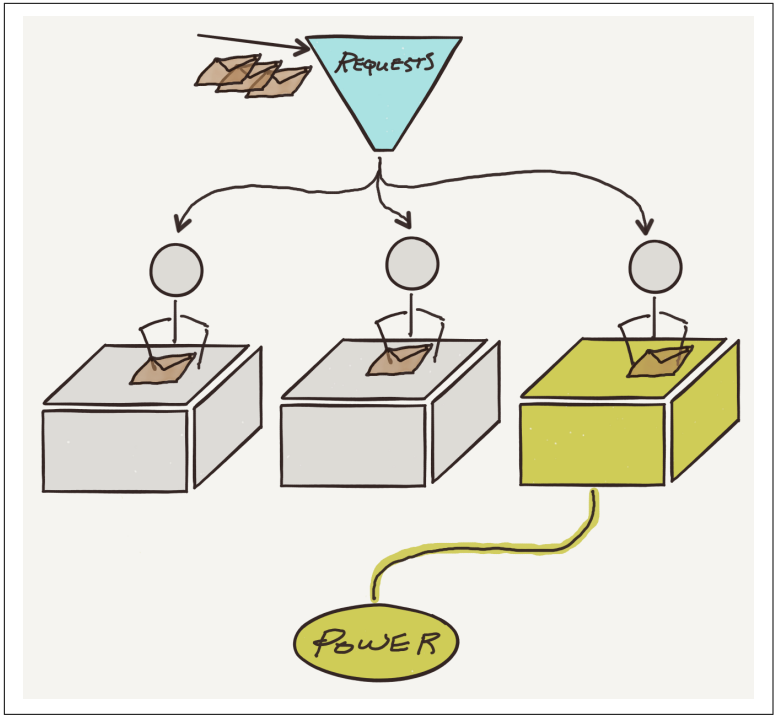

Figure 3-6. The system swaps the power between desks to allow an even distribution of work

Traditional Systems Versus Actor-based Systems

So how does this work in traditional, synchronous systems versus those built on the Actor model?

This constant shifting of the processing cores from thread to thread happens in all computer systems. When running traditional synchronous systems versus actor-based asynchronous systems, the difference is how the threads themselves are managed.

With synchronous systems, a thread is assigned to a request and it handles it until the processing of the request is completed with a response. This also includes holding the thread while waiting for I/O to complete. With asynchronous systems, threads are managed more dynamically—threads are only allocated to actors when they are ready to run and they are ready to use an available processing

core. When an actor must stop and wait for an I/O to complete, *the thread is released* and made available to be allocated to another actor.

In a synchronous software system, the common approach is that when a request is received it is delegated to a thread and that thread is responsible for handling that request until it is completed and a response is returned. In our office analogy, this is modeled with text messages sent to the office, where the office delegates the text message to a worker sitting at a desk, and that worker handles that request until it is completed. Once the office worker is assigned to a desk, she will stay at that desk until the given task is completed.

With the synchronous processing approach, the system limit for how many requests can be processed concurrently is limited by the number of threads. With the text messages example, the limit of the concurrently handled text messages is the number of desks. If the office space allows for 100 desks then that office can concurrently handle 100 text messages. When all of the workers at the desks are busy handling text messages, if more text messages arrive they will not be processed until some of the workers complete their current task and are ready to do more work. When text messages arrive faster than they can be processed the system needs to either put the pending requests into a queue or reject the excess messages (see Figure 3-7).

In addition, with the synchronous processing approach, when the processing must be suspended while waiting for various I/O operations to complete, the thread that is handling the request is idle, *yet still in use and consuming memory*. This is like the office worker sitting at a desk doing nothing while he waits for some operation to complete.

The idea is that the office worker received the incoming text message, which triggers the office worker to send another text message to someone else to perform some subtask, and the office worker is sitting there idle waiting for a response text message. The result is that the limited threads are in use but not doing any useful work, much like an office worker sitting at a desk doing no useful work while waiting for someone else to finish a subtask. In the office example, the number of desks limits how many text messages can be handled at one time.

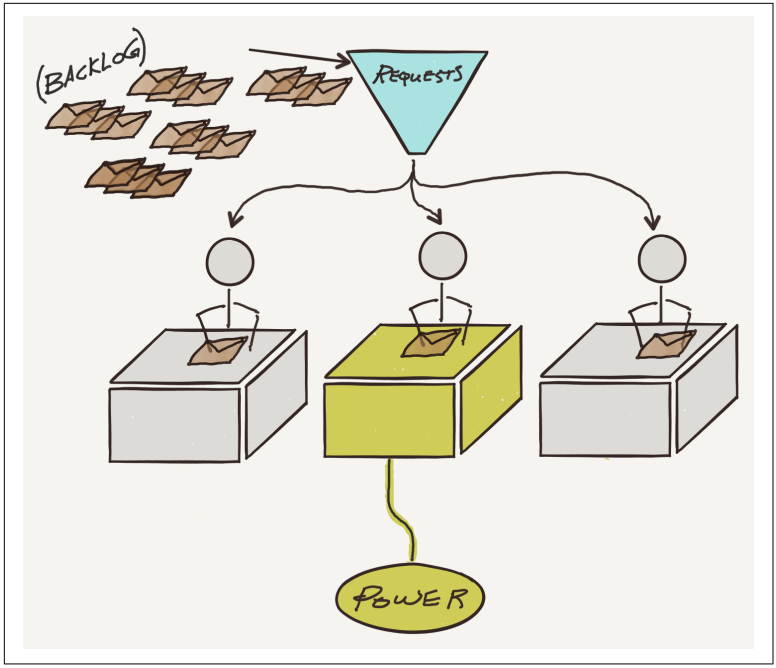

Figure 3-7. When messages arrive faster than the workers can perform each task, then there is a backlog of pending messages that are either queued or discarded

With the Actor model, the dynamics of the system are very different. The actor system processes things asynchronously.

In our office model, workers only occupy a desk when they have something to do. When a worker is idle, she goes to the side of the room and the desk is freed up for other workers to use. In an actor system, threads are allocated to actors that have messages to process. When the actor has no messages to process, the thread is allocated to other actors that have messages to process and that have something to do, so that they are not sitting idle while waiting for something like an I/O to complete (see Figure 3-8).

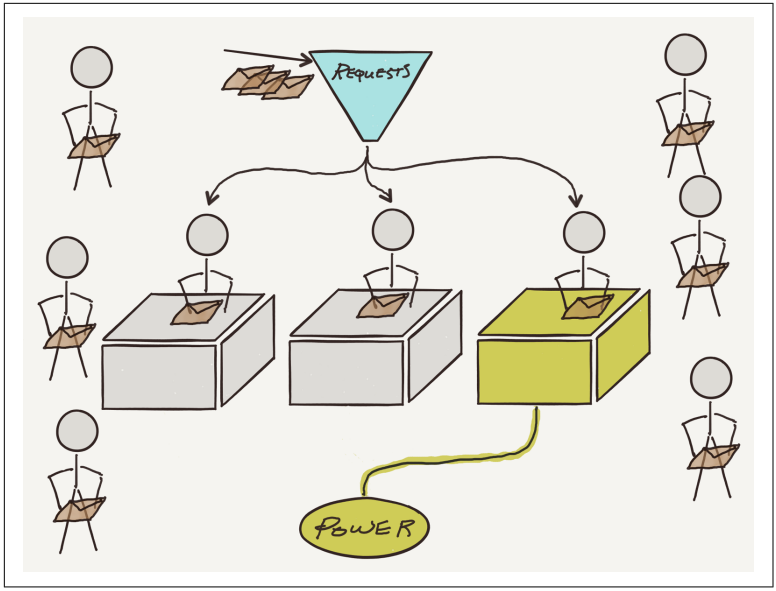

Figure 3-8. In an asynchronous system, workers only occupy desks when they have something to do

At the same time, no single actor can squat on a thread doing nothing. Even when an actor still has more messages to process, it is given a limited amount of time with a thread. The system is constantly moving the threads around to various actors trying to maintain a fair distribution of thread usage without letting some actors dominate their time with threads or, conversely, let some actors be starved for attention.

The end result is that asynchronous actor systems can handle many more concurrent requests with the same amount of resources *since the limited number of threads never sit idle* while waiting for I/O operations to complete.

This is analogous to the office model. In the synchronous processing flow, there is the possibility for many workers sitting at the limited number of desks to be idle at any point in time. With the asynchronous processing flow, a worker that is idle or has nothing to do goes to the side of the room, which frees up the desk for another worker that is ready to do some work.

Expanding into Clusters of Actors

The story does not end there. Say things get to the point where the rate of work to be done has outgrown the maximum message processing rate of the current office. Even with the use of asynchronous actors and their efficient use of a single system, the level of activity may still get to the point that the flow of work to be done exceeds the capacity of the system to process it. To address this the actor system can form clusters. A cluster is where two or more systems, each running an actor system, are configured to work as a collaborative group (see Figure 3-9).

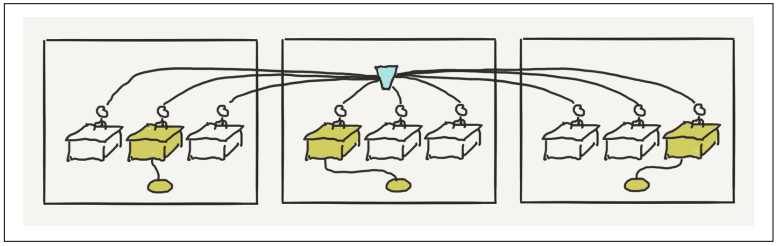

Figure 3-9. Actor systems may run in a cluster

From the perspective of the actors running in a cluster, there is no difference between sending messages to actors in the same system, or between actors on different systems because the actor system still handles all the actual messaging between actors. Of course, messaging between actors on different systems takes more time (i.e., in the milliseconds range) than messaging on the same system. However, the actor system is optimized to handle both the local and remote forms of messaging. The typical actor system is capable of handling millions of messages per second, and this applies from a single system cluster to many system clusters.

The actor system cluster capability provides the building blocks for elastically expanding the processing capacity as the load goes up and contracting the capacity as the load goes down. This level of elasticity cannot be patched on after the fact, but must be designed as a part of the architecture of each individual application implementation.

In some implementations, there are manual processes used to expand and contract the cluster as needed. In other implementations, elasticity is handled automatically by the application. The key

point here is that the actor system provides all the building blocks needed for creating clusters, which can then be used to architect, design, and implement distributed systems that can elastically expand and contract as needed.

This is in stark contrast to the traditional ways we have been building systems for the past decade and longer. With older system architectures, we are typically forced to scale vertically, using larger and larger systems and an ever-expanding infrastructure footprint. In these systems, there were limited options to distribute the work horizontally across multiple systems. In addition, the scale of these traditional synchronous systems is static, in that these systems were sized to reasonably handle peak loads. This results in low resource utilization because during slow times you are paying for wasted capacity that is sitting idle and, to make matters worse, it is extremely difficult to predict what the peak loads will be. The typical results are when there are peak periods of activity and capacity is insufficient to handle the load. When this happens, the system users suffer annoying response times. Annoyed users are more often former users because in many cases they can take their business elsewhere in an instant.

Ultimately, asynchronous processing provides a way of doing more work with less processing capacity. Clustering provides the building blocks for building application systems that grow and contract as the processing load requires. These two features of the actor system directly impact the operational costs of your application system: you use the processing capacity that you have more efficiently and you use only the capacity that is needed at a given point in time.

The main takeaways in this chapter are:

- Delegation of work through supervised workers allows for higher levels of concurrency and fault tolerance.
- Workers are asynchronous and run concurrently, never sitting idle as in synchronous systems.
- Efficient utilization of system resources (CPU, memory, and threads) results in reduced infrastructure costs.
- It's simple to scale elastically at the actor level by increasing or decreasing workers as needed.
- Using clusters gives the ability to scale at the system level.

CHAPTER 4

Actor Failure Detection, Recovery, and Self-Healing

In the previous chapters, we covered some of the features of actors and how they relate to handling errors and failure recovery. Let's dig a little deeper into this, shall we?

There are a number of strategies available for handling errors and recovering from failures both at the actor level and at the actor system level.

At the actor level, failure handling and recovery starts with the supervisor-worker relationship. Actors that create other actors are direct supervisors, and for error handling this means that supervisors are notified when a worker runs into a problem. In the supervisor role, there are four well-known recovery steps that may be performed when they are notified of a problem with one of their workers:

1. Ignore the error and let the worker resume processing
2. Restart the worker and perform a worker reset
3. Stop the worker
4. Escalate the problem to the supervising actor of the supervisor

How a supervisor handles problems with a worker is not limited to these four recovery options, but other custom strategies may be used when necessary.

All actors have a supervisor. Actors will form themselves into a hierarchy of worker to supervisor to grand-supervisor and so on (see Figure 4-1). At the top of the hierarchy is the actor system. If a problem is escalated to the actor system, its default recovery process is to restart the worker (or terminate the worker when more serious problems occur). This supervisory approach frees up the worker from handling its own errors, which means that it is focused completely on performing its tasks. This allows for creating actors with much less error handling code that clutters and hides the main business logic.

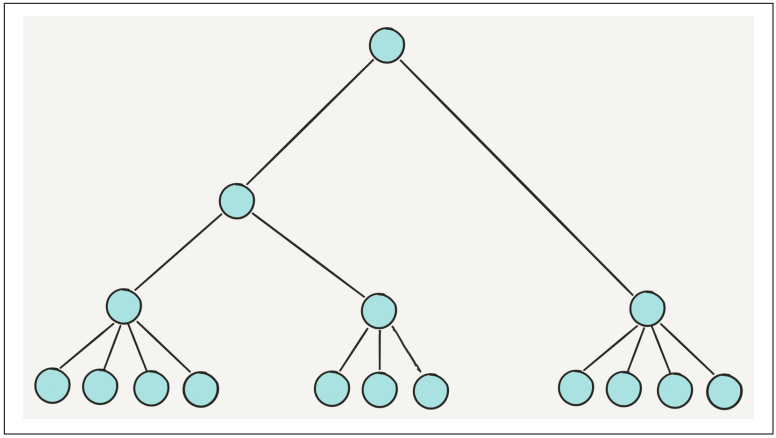

Figure 4-1. Actors form hierarchies

Actors Watching Actors, Watching Actors...

In addition to this supervision strategy, the actor system provides a way for *one actor to monitor another actor*. If the watched actor is terminated, the watcher actor is sent an "actor terminated" message. How the watcher reacts to these terminated messages is up to the design of the watcher actor. This sentinel pattern allows for building some very innovative application features. This pattern is often used to implement forms of self healing into a system (see Figure 4-2).

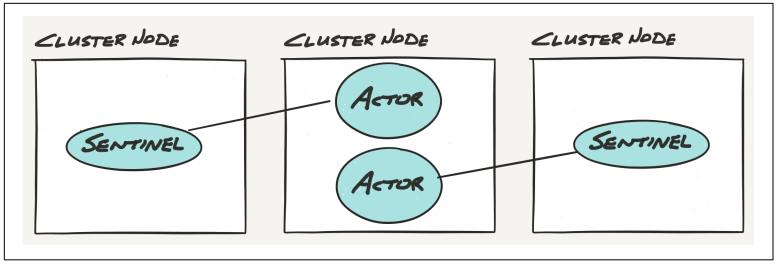

Figure 4-2. Sentinel actors watch actors on other nodes in the cluster

In this example, critical actors may be monitored across nodes in a cluster. If the node where a critical actor is running fails, the *sentinel actors* are notified (see Figure 4-3). This can trigger some form of recovery and self-healing process by the sentinel actor.

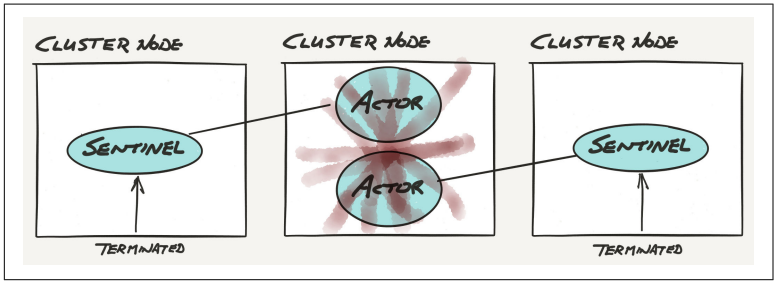

Figure 4-3. When a node fails, the sentinel actors are notified via an actor terminated message

It is common for a set of actors to perform some type of dangerous operation outside of the actor system. By "dangerous operations" we mean one that is more likely to fail from time to time—for example, among a set of actors that perform database operations.

In order to successfully perform these database operations, a lot of things need to be up and running. The backend database server needs to be running and healthy. The network between the actors and the database server needs to be working. When something fails, all of the actors that are trying to do database operations fail to complete their tasks. To exacerbate the problem, in many cases this triggers retries, where either the systems automatically retry failed operations or users seeing errors retry their unsuccessful actions. The end result is that the downed service may be hammered with requests and this increased load may actually hinder the recovery process.

To deal with these types of problems, there is an option to protect vulnerable actors with *circuit breakers* (see Figure 4-4). Here, a circuit breaker encapsulates actors so that messages must first pass through the circuit breaker, which are generally configured to be in a closed or open state. Normally, the circuit breaker is in a closed state, meaning that the connection allows messages to pass through to the actor. If the actor runs into a problem, the circuit breaker opens the connection and all messages to the wrapped actor are rejected. This stops the flow of requests to the backend service. The idea is to avoid hammering a failed service, such as a down database, when you know that all the requests are going to fail.

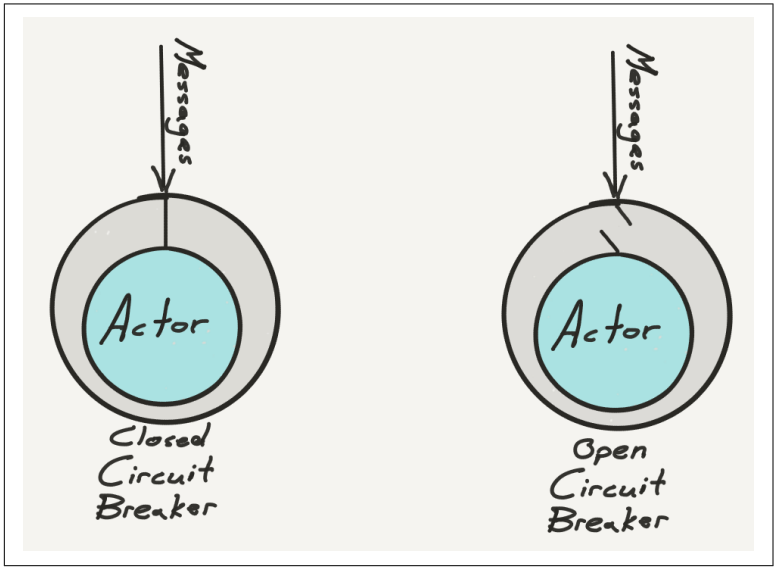

Figure 4-4. Circuit breakers can be used to stop the flow of messages to an actor when something unusual happens

Circuit breakers are configured to periodically allow a single messages to pass to the actor, which is done to allow checks to see if the error has been resolved. If the message fails, the circuit breaker remains open. However, when a message completes successfully the circuit breaker will close, which allows for resuming normal operations. This provides for a straightforward way to quickly ascertain a failure and begin the self-heal process once the problem is resolved. This also comes with the added benefit of providing a way for the system to back off from a failed service.

Another added benefit of the use of circuit breakers is that they provide a way for *avoiding cascading failures*. A common problem that may happen when these types of service failures occur is that the client system may experience a log jam of failing requests. The failed request may generate more retry requests. When the service is down it may take some time before the error is detected due to network request timeouts. This may result in a significant buildup of service requests, which then may result in running out of systems resources, e.g., memory.

On a larger scale, when running a cluster of two or more server nodes, each of the nodes in the cluster monitors the other nodes in the cluster. The cluster nodes are constantly gossiping behind the scenes in order to keep track of each other, so that when a node in the cluster runs into a problem and fails or is cut off from the other nodes due to a network issue, the remaining nodes in the cluster quickly detect the problem.

Actor flexibility extends even into being notified when there are node changes to the cluster. This not only includes nodes leaving the cluster, but also nodes joining the cluster. This feature allows for the creation of actors that are capable of reacting to cluster changes. Actors that want to be notified of cluster changes register their interest with the actor system. When cluster node changes occur, the registered interested actors are sent a message that indicates what happened. What these actors do when notified is application specific. As an example, actors that monitor state changes to the cluster may be implemented to coordinate the distribution of other actors across the cluster. When a node is added to the cluster, the actors that are notified of the change react by triggering the migration of existing actors to the new node. Conversely, when nodes leave the cluster, these actors react to the failure by recovering the actors that were running on the failed node on the remaining nodes in the cluster.

The main takeaways of this chapter are:

- Actor supervision handles workers that run into trouble, handling error recovery that frees workers to focus on the task at hand.
- Actors may watch for the termination of other actors and react appropriately when this happens.

- Actors may be wrapped in a circuit breaker that can stop the flow of messages to an actor that is unable to perform tasks due to some other, possibly external, problem. Circuit breakers allow for graceful recovery and self-healing, stemming the flow of traffic to a failed service to accelerate the service recovery process.
- Actors may be cluster aware and designed to be notified when nodes join or leave the cluster. This can be used to react to the cluster changes.

CHAPTER 5
Actors in an IoT Application

In this final chapter, let's work through a more realistic example of using actors to implement features in a real-life system. In this example, we are responsible for building an Internet of Things (IoT) application, in which we currently have hundreds of thousands of devices that are monitored continuously (with the expectation of this to grow over time into the millions).

Each device periodically feeds status data back to the application over the Internet. We decide that we want to represent each device with an actor that maintains the state of the device in our system. When a message arrives over the Internet to our application the message somehow needs to be routed to the specific actor.

Our system then will have to support millions of these device actors. The good news is that actors are fairly lightweight (a default actor is only 500 bytes in size, compared to 1 million bytes for a thread), so they do not consume a lot of memory; however, in this case one node cannot handle the entire load. In fact, we do not want to run this application on a single node, we want to distribute the load across many nodes so as to avoid any bottlenecks or performance issues with our IoT application. Also, we want an architecture that can scale elastically as more devices come online, so the application must be able to scale horizontally across many servers as well as scale vertically on a single server.

As a result of these requirements, we decide to go with an actor system that runs on a cluster on multiple nodes. When messages from devices are sent to the system, a given message may be sent to any one of the nodes in the cluster. This brings some questions to the table:

- What specific set of actors could support this system?
- How can the system handle scaling up when adding new nodes?
- What happens when a given node in the cluster fails?
- Finally, how do we route device messages to the right device actor across all of the nodes in the cluster?

Of course, for most software problems there may be many possible solutions, so to meet these requirements we offer the following possible solution. Recall that an actor may register itself with the actor system to be notified when a node joins or leaves the cluster. We implement an actor that runs on each node in the cluster. This actor handles incoming device messages that are sent to the node that it is running on. It also receives messages from the actor system when nodes join or leave the cluster. In this way, each actor that is resident on a node in the cluster is always aware of the current state of the entire cluster.

Let's call this the *Device Message Router* actor (shown as DMR in the diagram). Every message in this example contains the device's unique identifier. The DMR actor is a supervisor that has to find the specific device actor (shown as D in the diagram) using the device identifier so that it can forward the message to it (see Figure 5-1).

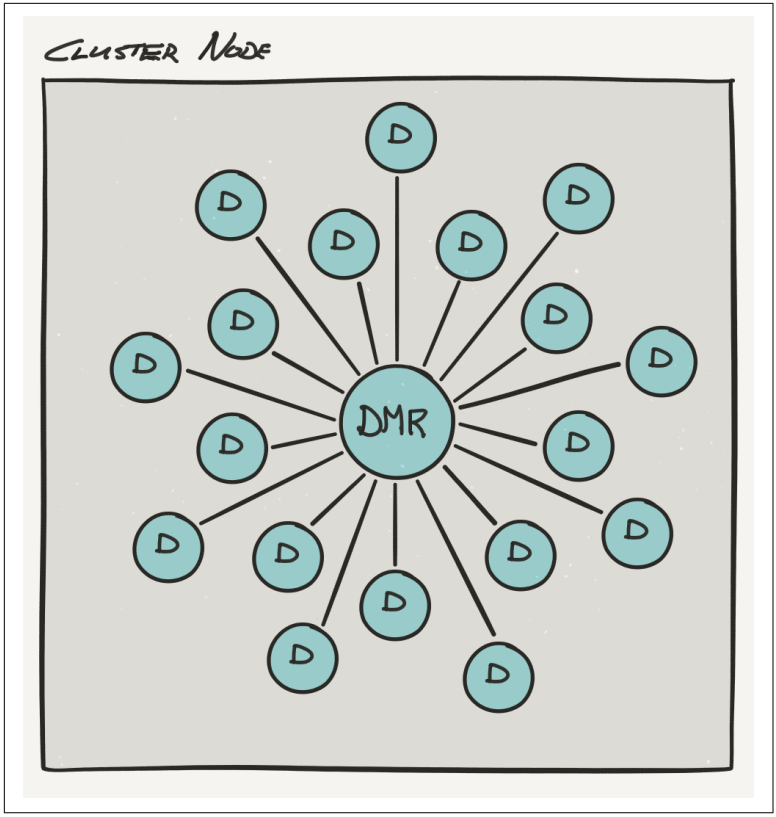

Figure 5-1. Device Message Router actor manages device actors

But wait, how do we know what node in the cluster contains the specific device actor? We are running in a cluster of many nodes and a given device actor is located somewhere out on one of those nodes.

The solution for locating specific device actors is to use a well-known algorithm called the *consistent hashing algorithm*. Without going into too much detail, consistent hashing provides for a very efficient way to distribute a collection of items, such as a collection of our device actors, across a number of dynamically changing nodes. We use this algorithm to determine which node currently contains a given device actor (see Figure 5-2). When a request is randomly sent to one of the DMR actors, it uses the consistent hashing algorithm to determine which node actually contains that device actor.

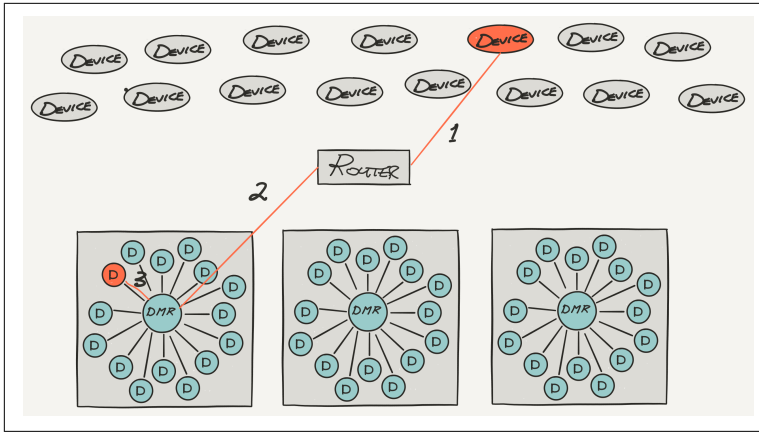

Figure 5-2. Device message routing across the cluster using the consistent hashing algorithm

If the device actor happens to be on the same node, then the DMR actor simply forwards the message to this local device actor. However, if the device actor is located on another node, the DMR actor forwards the message to the DMR actors on the other nodes (see Figure 5-3). When the DMR actors on the other nodes receive the forwarded message, they perform the consistent hashing algorithm to determine if the device actor is on the same node and forwards the message.

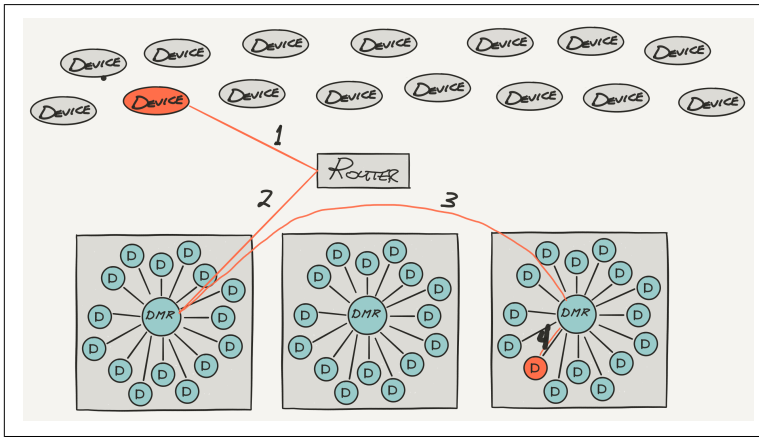

Figure 5-3. Routing device messages across the cluster using DMRs

Location Transparency Made Simple

What we have so far is pretty good but we are not done yet. What happens when a new node joins the cluster? How do we handle the migration of nodes to the new node? The beauty of the consistent hashing algorithm is that when the number of nodes changes, the index of some of the devices that were located on other nodes will now point to a new node. Say a device was on Node 2 of a cluster of three nodes. When a fourth node is added to the cluster, the device actor that was on Node 2 is now located on Node 4. When the request for that device comes into the system, the message will now be routed to the DMR actors resident on Node 4.

There is one thing that we have not addressed yet. How are device actors created on the nodes in the cluster in the first place?

The answer is that the DMR actors create device actors for each device. When a DMR actor first receives a message from a device and it determines that the device actor is resident on the same node, it checks to see if that actor exists or not. This can be done simply by attempting to forward the message to the device actor. If there is no acknowledgment message back from the device actor, this triggers the DMR actor to create the device actor. When a device actor first starts up, it does a *database lookup to retrieve information about itself* and then it is ready to receive messages.

But let's not forget about the old actor... is it still on the previous node after is has migrated to a new node? And how do we handle device actors that have migrated to another node when the topology of the cluster changes?

A simple solution for this problem is to use an *idle timeout message*. Recall that actors can tell the actor system to send itself a message at some time in the future. We set up each device actor to always schedule an idle timeout message. Whenever a device actor receives a device message, it cancels the previously scheduled idle timeout message and schedules a new one. If the device actor receives an idle timeout message, then it knows to terminate itself.

Because the device status messages are no longer routed to the old device actor, the idle timeout will eventually expire and the timeout message will be sent to the device actor by default. Using these fairly simple mechanisms, such as self-scheduled messages, we have

designed a fairly simple way to clean up device actors that have migrated to new nodes in the cluster.

There is an added bonus to this solution. What happens when we lose a node in the cluster? This is handled in the same way that we handle new nodes when they are added to the cluster. Just as when new nodes are added, any nodes leaving the cluster impacts the consistent hashing algorithm. In this case, the device actors that were on the node that failed are now automatically migrated on the remaining nodes in the cluster. We already have the code in place to handle this.

One last major detail. How do the DMR actors know how many nodes are in the cluster at any point in time?

The answer is that there is a way for actors to retrieve these details from the actor system, recalling the sentinel actor concept where actors ask the actor system to send them a message when nodes join or leave the cluster. In our system, the DMR actors are set up to receive messages when nodes join or leave the cluster. When one of these node-joining-the-cluster or node-leaving-the-cluster messages is received, this triggers the DMR actor to ask the actor system for the details of the current state of the cluster. Using that cluster status information it is possible to determine exactly how many nodes are currently in the cluster. This node count is then used when performing the consistent hashing algorithm.

Of course, this design is not complete; there are still more details that need to be worked out, but we have worked out some of the most important features of the system. Now that we have worked out this design, consider how you would design this system without the use of actors and an actor system.

Ultimately, the solution here handles scaling up when it is necessary to expand the capacity of the system to handle increased activity, as well as recovering failures without stopping and without any significant interruption to the normal processing flow.

The actual implementation of the two actors here is fairly trivial once we have the design details worked out. The design process itself was also fairly straightforward and did not get bogged down in excessive technical details. This is a small example of the power and elegance of the Actor model, where the abstraction layer provided by actors and the actor system lifts us above the technical

plumbing details that we have to deal with in traditional, synchronous architectures.

The main takeaways of this chapter are:

- Clustered actor systems can be designed for resiliency and elasticity.
- Actors can be implemented to react to nodes leaving and joining a cluster.
- Work can be distributed across a cluster.
- Actors and actor systems provide an abstraction layer that allows for higher levels of concurrency.

CHAPTER 6
Conclusion

Today, it is now possible to create distributed, microservices-based systems that were impossible to even dream of just a few short years ago. Enterprises across all industries now desire the ability to create systems that can evolve at the speed of the business and cater to the whims of users. We can now elastically scale systems that support massive numbers of users and process huge volumes of data. It is now possible to harden systems with a level of resilience that enables them to run with such low downtime that it's measured in seconds, rather than hours.

One of the foundational technologies that enables us to create microservices architectures that evolve quickly, that can scale, and that can run without stopping, is systems based on the Actor model. It's the Actor model that provides the core functionality of Reactive systems, defined in the Reactive Manifesto as responsive, resilient, elastic, and message driven (see Figure 6-1).

In this report, we have reviewed some of the features and characteristics of how actors are used in actor systems, but we have only scratched the surface of how actor systems are being used today.

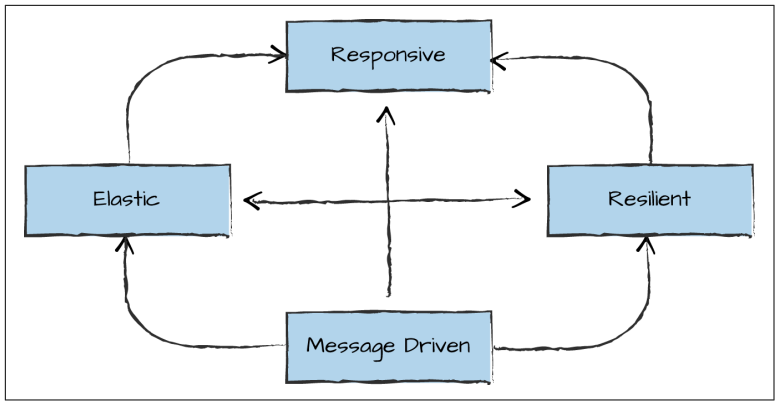

Figure 6-1. The four tenets of reactive systems

The fact that actor systems can scale horizontally, from a single node to clusters with many nodes, provides us with the flexibility to size our systems as needed. In addition, it is also possible to implement systems with the capability to scale elastically, that is, scale the capacity of systems, either manually or automatically, to adequately support the peaks and valleys of system activity.

With actors and actor systems, failure detection and recovery is an architectural feature, not something that can be patched on later. Out of the box you get actor supervision strategies for handling problems with subordinate worker actors, up to the actor system level, with clusters of nodes that actively monitor the state of the cluster, where dealing with failures is baked into the DNA of actors and actor systems. This starts at the most basic level with the asynchronous exchange of messages between actors: if you send me a message, you have to consider the possible outcomes. What do you do when you get the reply you expect and also what do you do if you don't get a reply? This goes all the way up to providing ways for implementing strategies for handling nodes leaving and joining a cluster.

Thinking in terms of actors is, in many ways, much more intuitive for us to think about when designing systems. The way actors interact is more natural to us since it has, on a simplistic level, more in common with how we as humans interact. This enables us to design and implement systems in ways that allow us to focus more on the core functionality of the systems and less on the plumbing.

About the Author

Hugh McKee is a solutions architect at Lightbend. He has had a long career building applications that evolved slowly, that inefficiently utilized their infrastructure, and that were brittle and prone to failure. That all changed when he started building reactive, asynchronous, actor-based systems. This radically new way of building applications rocked his world. As an added benefit, building application systems became way more fun that it had ever been. Now he is focused on helping others to discover the significant advantages and joys of building responsive, resilient, elastic, message-based applications.