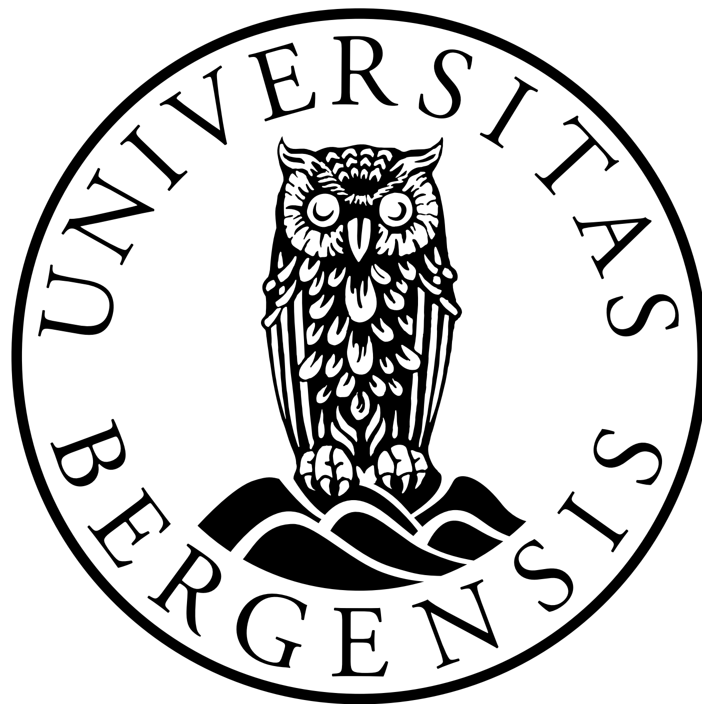
Master thesis in Secure and Reliable communication at Simula@UiB

Reliable Asynchronous Communication in Distributed Systems



Author :
Espen Johnsen

February 15, 2018

Acknowledgement

I would like to thank my thesis advisor Kjell Jørgen Hole at Simula@UiB. Hole has been a wonderful advisor who has given me valuable guidance. Thank you.

I would also like to thank Roy Hidle, Kavita Bhamra, Alexander Polden, and Markus Karlsen for their support and encouragement throughout my years of studying. This accomplishment would not have been possible without them.

Contents

Acknowledgement

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Goals . . . 1

1.3 Targeted Audience . . . 2

1.4 Thesis Overview . . . 2

2 Basics **3**

2.1 What is a state? . . . 3

2.2 What is immutability? . . . 4

2.3 What is a monolith? . . . 4

2.4 What is a microservice? . . . 4

2.5 What is an actor? . . . 5

2.6 What is Akka? . . . 6

2.6.1 Actor operations . . . 6

2.6.2 Akka structures . . . 6

2.6.3 Code example . . . 10

2.7 Summary . . . 11

3 Architecture **13**

3.1 Service Discovery . . . 13

3.1.1 Service registry . . . 13

3.1.2 API gateway . . . 15

3.2 Communication between services . . . 16

3.2.1 Synchronous and asynchronous communication 16
3.2.2 Service interaction 16
3.2.3 Events, commands, and reactions 18
3.2.4 Backpressure 19
3.2.5 Circuit breaker 19
3.2.6 Failed delivery, dead letter queue 20
3.3 Failure reduction and handling 20
3.3.1 Bulkheading and supervisor hierarchy in the actor model 21
3.3.2 Event store 21
3.3.3 Containerisation and orchestration 24
3.4 Summary 26

4 Design 27
4.1 The CAP theorem 27
4.2 Strategic design 29
4.2.1 Bounded context 29
4.2.2 Ubiquitous language 30
4.2.3 Core domain 30
4.2.4 Subdomain 31
4.2.5 Context mapping 31
4.2.6 At-least-once delivery and idempotent receiver 32
4.3 Tactical design 32
4.3.1 Entity 32
4.3.2 Value object 32
4.3.3 Aggregate 32
4.3.4 Consistency 35
4.3.5 Domain event 40
4.4 Summary 42

5 Building an Infrastructure with Akka 43
5.1 Achieving asynchronous communication in Akka 43

5.1.1 Futures . . . 43
5.1.2 Promises . . . 44
5.1.3 Usage . . . 46
5.2 Load balancing in Akka . . . 46
5.2.1 Router functionality . . . 47
5.2.2 Task scheduling through router types . . . 49
5.3 Clustering . . . 51
5.3.1 Joining . . . 52
5.3.2 Leaving . . . 53
5.3.3 Membership states . . . 54
5.3.4 Gossip protocol . . . 55
5.3.5 Failure detection . . . 56
5.3.6 Working with a cluster . . . 57
5.3.7 Distributed publish subscribe . . . 58
5.4 Persistence . . . 62
5.4.1 Persistent actor . . . 62
5.4.2 Enabling at-least-once delivery . . . 63
5.4.3 Snapshots . . . 64
5.4.4 Cluster persistence . . . 64
5.5 Summary . . . 68

6 Analysis and Discussion 71
6.1 Summary and discussion . . . 71
6.1.1 Summary . . . 71
6.1.2 Discussion . . . 71
6.2 Future work . . . 72
6.2.1 Distributed systems in the cloud . . . 72
6.2.2 Cloud security . . . 73
6.2.3 Distributed Data and Chaos Engineering . . . 73
6.2.4 Different approaches . . . 73

List of Figures

2.1 State illustration 4
2.2 Supervision tree 6
2.3 Mailbox 7
2.4 Dispatcher 8
2.5 Poison pill 9
2.6 Stop command 9
2.7 The different parts of a communication string 9

3.1 Client-side service discovery 14
3.2 Server-side discovery 15
3.3 API gateway 16
3.4 Synchronous and asynchronous communication 17
3.5 Events, commands, and reactions 18
3.6 Circuit breaker 19
3.7 Bulkheading 22
3.8 Event sourcing 23
3.9 Deletion and resizing of an event log 24
3.10 Command query responsibility segregation 25
3.11 Containers and virtual machines 25
3.12 Simple orchestration setup 26

4.1 Views of store 27
4.2 Store communication 28
4.3 Bounded context 29

4.4 Core domain 30
4.5 Context mapping 33
4.6 Aggregate 34
4.7 Aggregate root, entity, and value object 35
4.8 Processes communication with a storage system 36
4.9 Casual Consistency 37
4.10 Server-side consistency example with different read and write values 38
4.11 Consistency failure on the left with a total of 3 operations and 3 nodes, and on the right two nodes and two operations 39
4.12 Conflicting write with write operation below the number of nodes 39
4.13 Partition consistency 40
4.14 Domain events with a subscribing and publishing bounded context 41

5.1 Future as a placeholder for the results of the function 44
5.2 Code example from Listing 5.2 illustrated 45
5.3 Router pattern 47
5.4 Router types 47
5.5 Router escalating problem to its supervisor 48
5.6 Task being processed too slowly on routee 1. 50
5.7 A cluster with four nodes, each containing an actor system. 52
5.8 Cluster initialisation timelaps 53
5.9 The leader leaves the cluster 54
5.10 All transitions and states of a node 55
5.11 By receiving an older version of the state the node with the newer state sends the newer version to the gossiper. 56
5.12 When the gossiper has a newer version of the state than the receiving node, the receiver sends back its version and obtains the new state. 57
5.13 Job processing with a cluster 59
5.14 Cluster supervision hierarchy 60
5.15 A cluster setup where one publisher sends to two subscribers 60
5.16 Message sent to either all or one subscriber, or only to the ones that do not have SOMTEG set to true 61

5.17 Persistent actor with recovery method and command method 63
5.18 An persistent actor persisting the received command 63
5.19 At-least-once delivery with persistent actors 65
5.20 The cluster singleton hierarchy 66
5.21 One singleton actor between three nodes, managed by the ClusterSingleton-Manager 67
5.22 Cluster sharding hierarchy 67
5.23 Shards in a cluster 68
5.24 The correct shard region is communicated with, and the shard region creates a Shard for its entity 69
5.25 The incorrect shard region is communicated with, and the shard region sends the message to the correct shard region where the same as the previous example happens again. 70

Listings

2.1 Simple ping pong example 10
5.1 Future example 44
5.2 Promise and future example 45
5.3 Resizer configuration example for a pool router 48
5.4 Gossip state representation 55
5.5 Movie event example 64

Chapter 1

Introduction

1.1 Motivation

When I started the research for this thesis, my goal was to make a distributed software system and make algorithms that would handle data safely and effectively. What I figured out fast was that there is much literature that gives many different explanations for how to achieve this goal, but the ideas are scattered all over the literature and not always readily connectable. For someone who had no prior knowledge of how to build a distributed system, the literature was overwhelming. Hence, I decided to write a thorough introduction to the design and implementation of distributed software systems. The motivation is to guide the reader through the aspects of creating a distributed system. This thesis will by no means cover everything within the area, but it will be a good starting point to obtain an overview of the current trends, and a list of sourced reading material that can be of service to anyone wanting to delve into distributed systems.

1.2 Goals

The main objectives of the thesis are to introduce the different aspects of the software development of distributed systems. Important terminology is explained to facilitate the reading of the literature and figures are used to explain important points. The thesis aims to create a foundation that readers can use in future projects. In particular, the thesis should allow the reader to determine when to build a distributed system.

1.3 Targeted Audience

This thesis will be targeted especially at programmers that have a functioning programming knowledge. It could also be read by people who are curious about the area since the amount of code in the thesis is minimal. The coding examples are more illustrative than necessary to understand the whole thesis.

1.4 Thesis Overview

Chapter 2 will introduce the foundational concepts that are needed not to become confused later in the thesis. It will also explain important topics that have no good placements in the other chapters but are important none the less. In Chapter 3, we will try to boil down the main architectural decisions to consider and understand if one were to create a distributed system. In Chapter 4, we will delve into some of the more important definitions and concepts used within domain-driven design because it is the design method that has gotten the most attention when creating a large distributed systems. Chapter 5, will be an introduction to Akka, a toolkit used with Scala and Java, to demonstrate one of the many ways to create a distributed system. Lastly, Chapter 6 summarise the thesis, discuss the most important insights, and suggests possible future work.

Chapter 2

Basics

It is hard to divide a large deployment monolith into a well-functioning system of autonomous processes called microservices. It is challenging both to choose a proper microservice design and select the right technologies to realise it. This chapter provides the fundament that is used in later chapters to describe how to create a distributed microservice systems.

2.1 What is a state?

In computer science, we talk about the state of a process all the time. To have a meaningful discussion in this thesis, we need to have a good understanding of what should be understood by the term. The abstract, non-informatics way of describing a state is the present condition of a system, as in the coffee is hot or not. The temperature itself is a number, but hot has a different reaction than cold (both taste and vapour). Something is stateless when it does not change, such as the colour of the coffee. Variables have a state; values do not.

Within a microservice, also denoted an actor, a state can be thought more of as a combination of available actions or a position on a path. A concrete example is an actor that has three states (see Figure 2.1). State one is where the actor can get messages, state two is when the actor does something with the data internally, and the third state is when the actor is sending messages. If we also imagine that the states can just change to its immediate neighbour, meaning state one can only go to state two, two to three, and the opposite direction. This state changing scheme might be a bit of an exaggeration, but illustrates how we will use states in this thesis [1].

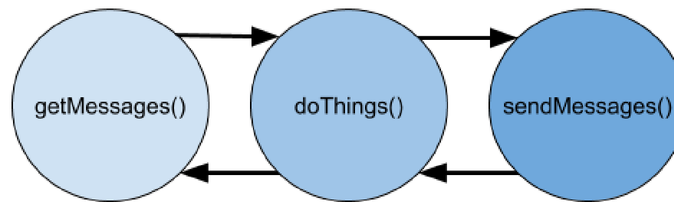


Figure 2.1: State illustration

2.2 What is immutability?

To mutate is to change. A *mutable* object can be changed after it is created. An *immutable* object cannot be modified after it is initiated. The simplest construct that illustrates immutability is the *final* keyword. If any variable has the final keyword before it, the variable cannot change after a value is assigned. An immutable actor will carry out the same operations every time it gets the same inputs. Immutability increases the predictability of an object [2, pp. 59–61].

2.3 What is a monolith?

A monolith is a self-contained software application. A monolithic application performs every step that is needed to complete a task. It is a software program of strongly connected components, meaning that there are many necessary dependencies between the components. These dependencies make it hard to change one component without also changing other dependent components. As a consequence, it is hard to scale a monolith to handle more users.

2.4 What is a microservice?

There is no generally agreed upon definition of the term "microservice." I will provide a definition suited to the purpose of this thesis, and define some useful terminology.

The *micro* in microservice is not its size. It is the scope of what it does [3, pp. 27–29]. The single responsibility principle is one of the fundamental rules of a microservice. This has been a successful principle used by Unix for a long time.

Assume that the two microservices A and B communicate. The microservice A is *strongly dependent* on the microservice B when A's functionality is significantly degraded when B malfunctions or crashes. Strong dependencies between microservices lead to the propagation of single service failures into systemic failures affecting a large number of services. To avoid strong dependencies and failure propagation, microservices must be

isolatable [3, pp. 25–27], that is, it must be possible to isolate a malfunctioning service by taking down its connections to other services without seriously affecting the functionality of the other services [4].

For a service to be completely isolated, the service needs to own its mutable state, exclusively [3, pp. 29–37]. The services can ask for information about another service's state, but not own it. No shared state makes it possible to scale an application to handle more users.

Isolatable microservices can act autonomously [3, pp. 25–27]. This means that a service can make its own decisions, cooperate, and coordinate actions as it wants. Looking at a service from the outside, it promises what it will do by publishing its application programming interface (API) or protocol.

When services communicate, it is important that the communication happens asynchronously [3, pp. 37–47]. If processes communicate synchronously, it will block the effective progress of the processes, and slow down the system as a whole.

Furthermore, a system will terminate and restart processes all the time. It may also move processes around while they are active. Hence, processes must have addresses that support mobility [3, pp. 47–51]. Techniques to achieve the described properties will be discussed at length later in this thesis.

2.5 What is an actor?

Since microservices is such a diluted term, we will use the term *actor* instead. Actors were introduced by Carl Hewitt in his paper *Actor Model of Computation: Scalable Robust Information Systems* [5], where he lays out the fundamentals of what an actor can do:

- send messages to (unforgeable) addresses of other actors;
- create new actors;
- designate how to handle the next message it receives.

These tenants will help build a robust actor system.

There are some programming languages based on the actor model and its underlying theory, including Erlang [6] and Elixir [7]. Erlang was developed by Ericsson to handle large amounts of traffic in switches with high uptime. An Erlang program with more than a million lines of code controlled the AXD301 switch, which reached an uptime of nine nines. The nine nines is a measurement for how long a system is up and running throughout a year, where nine nines means it is up 99,999999999 percent of a year or having a downtime of 31,5569 milliseconds per year.

2.6 What is Akka?

Akka [8] is a toolbox that programmers can use with Java or Scala to implement the actor model. It has gained a lot of interest lately since developers can use a programming language they are familiar with to implement robust and scalable distributed systems. We will go through the basic structures of Akka here.

You can see a supervision tree in Figure 2.2, where the application has a connection to the first supervisor, who has created 3 actors. These actors are in turn the supervisors of the actors created by them. The supervision tree is used to detect and handle errors in the different actors.

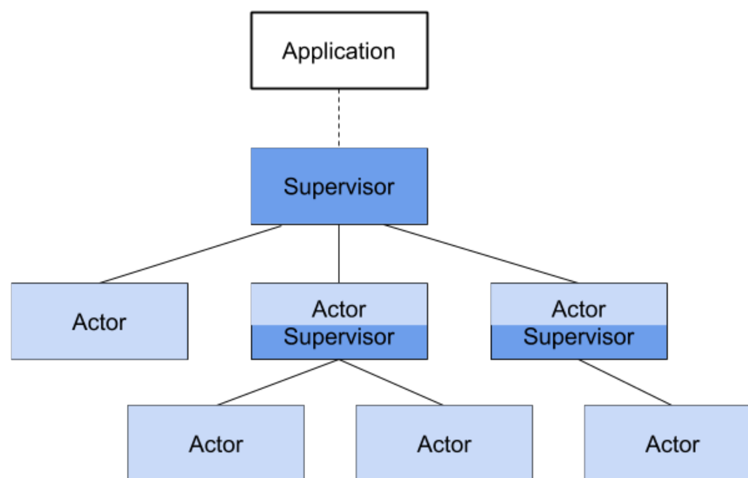


Figure 2.2: Supervision tree

2.6.1 Actor operations

When an actor is created, it will be supervised by its creator. This supervisor is responsible for the health of the actor, called a *worker*. A supervisor will handle a worker that has developed a problem or has crashed. The only actor in this system that does not have a supervisor is the first actor called the *actor system*. It will create the first actor and will be the last actor if every other actor fail. It acts as a supervisor to the actors it creates. After an actor is created, it can send messages to other actors and receive messages from actors [9, pp. 20–22][10, pp. 93–97].

2.6.2 Akka structures

An *ActorRef* is associated with every actor. Actors communicate with each other via their ActorRefs. If an actor with a particular ActorRef goes down and is restarted, or just

replaced with a new version, other actors can still communicate with the new instance via the original ActorRef. The use of ActorRefs makes it possible to create a flexible system where actors come and go and where actors can be updated without affecting other actors.

An actor has a *mailbox* in case the actor cannot handle the requests fast enough. If there are more messages than can be processed, the mailbox fills up. Figure 2.3 illustrates messages being sent to the ActorRef who sends it to the mailbox.

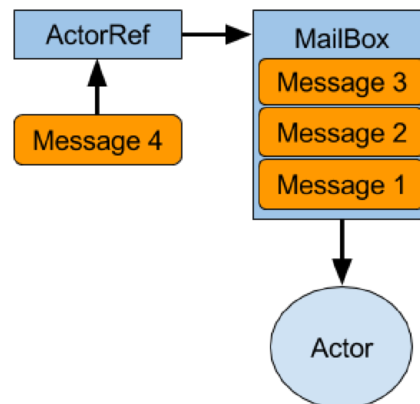


Figure 2.3: Mailbox

A *dispatcher* is connected to the mailbox. The dispatcher pushes the messages from the mailbox to the actor for processing. A dispatcher can push messages to many actors by filtering the packages [9, pp. 20–22].

In Figure 2.4, we can see two dispatchers having different setups. The dispatcher on the left has a single mailbox it manages, while the one on the right has control over two mailboxes. The different colours on the messages illustrate that the messages are different in some measurable way so that it becomes natural to separate them into two different mailboxes.

To stop an actor, you can either use the stop method or send a *poison pill* message to the actor. The difference between the two is that the poison pill message is just like another message, so it will terminate the actor after the messages in the mailbox are processed. The stop method stops the actor right after the current message is finished processing. In Figure 2.5, you can see the poison pill delivered to the mailbox. After the messages have been processed, the actor is terminated. In Figure 2.6, you can see the stop command in action. It terminates the actor immediately, and moves the messages to the dead-letter queue. The *dead-letter queue* is a queue where the messages that are not delivered are placed. We will discuss the dead-letter queue in more depth later in this thesis.

In the actor system, there are two actors that have different paths, with one root actor that supervises the other two. The first is the user actor called *The Guardian Actor*. This actor is the parent of all the user-created actors. The second actor is the system actor

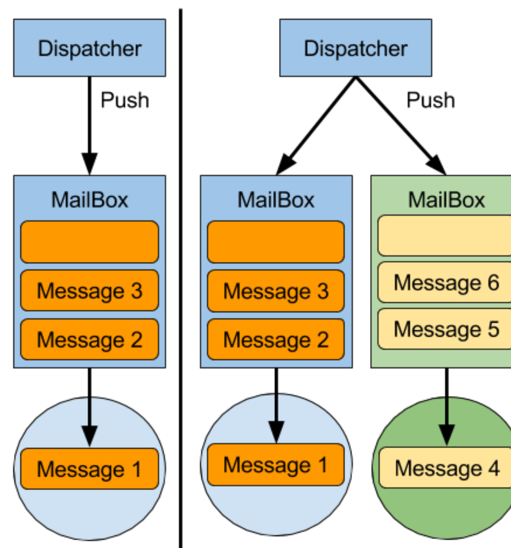


Figure 2.4: Dispatcher

called *The System Guardian*. This actor is logging while the actors are shut-down in an orderly fashion. The actor type we will discuss in this thesis is an actor that is under the user actor.

To communicate with actors in different actor systems, one can do this by giving the full path from the actor system to the actor. The string that can be used to enter an actor system would contain the parts illustrated in Figure 2.7.

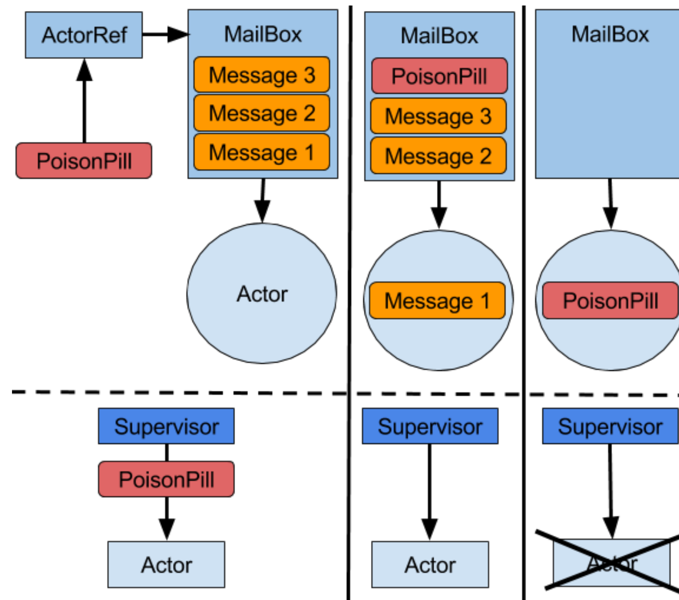


Figure 2.5: Poison pill

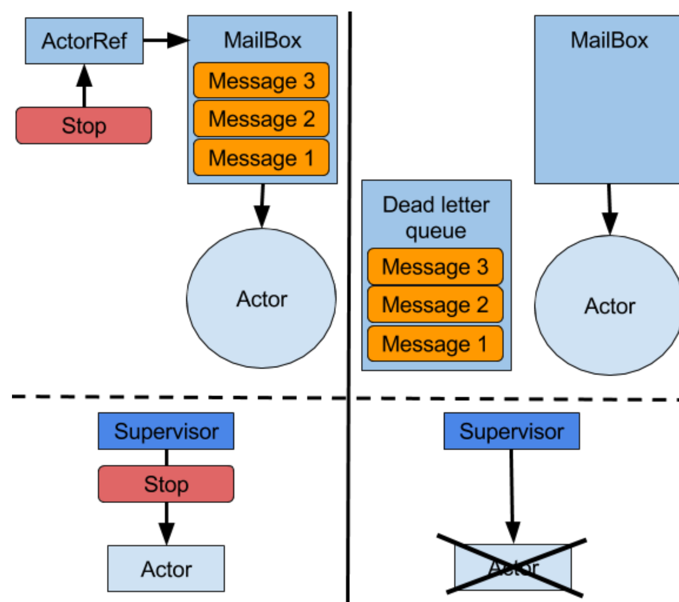


Figure 2.6: Stop command

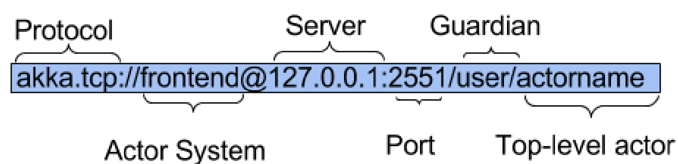


Figure 2.7: The different parts of a communication string

2.6.3 Code example

Here is a simple code example in Scala that demonstrates the use of the ActorSystem, ActorRef, dispatcher, PoisonPill and *Props*. Props is a configuration class that is immutable. It specifies options about the creation of an actor [11].

```
import akka.actor.{ ActorSystem, Actor, ActorRef, Props, PoisonPill }
import language.postfixOps
import scala.concurrent.duration._

case object Ping
case object Pong

class Pinger extends Actor {
  var countDown = 100

  def receive = {
    case Pong =>
      println(s"${self.path} received pong, count down $countDown")

      if (countDown > 0) {
        countDown -= 1
        sender() ! Ping
      } else {
        sender() ! PoisonPill
        self ! PoisonPill
      }
  }
}

class Ponger(pinger: ActorRef) extends Actor {
  def receive = {
    case Ping =>
      println(s"${self.path} received ping")
      pinger ! Pong
  }
}

    val system = ActorSystem("pingpong")

    val pinger = system.actorOf(Props[Pinger], "pinger")
    val ponger = system.actorOf(Props(classOf[Ponger], pinger), "ponger")

    import system.dispatcher
    system.scheduler.scheduleOnce(500 millis) {
      ponger ! Ping
    }
    // $FiddleDependency org.akka-js %%% akkajsactor % 1.2.5.1
```

Listing 2.1: Simple ping pong example

2.7 Summary

This chapter has introduced many of the fundamental concepts that will be used later in this thesis. Firstly, it introduced states. Then, it explained immutability and some of the areas where it can be used. The difference between a monolith and a microservice was explored. Lastly, the actor model was introduced, where Akka is one of the toolboxes one can use to create an actor system.

Chapter 3

Architecture

This chapter will cover the service discoverability, communication, and structure of a microservice system. While every method described here is contained in the Actor model, a few of the methods are not realised by Akka. We will mainly use the term “service” instead of actors because the literature on Akka refers to services.

3.1 Service Discovery

Here, we will describe concepts and methods needed to understand how services are introduced into a microservice system.

3.1.1 Service registry

The *service registry* [12] provides access to the services that are in a system. It is often done by keeping the static IP addresses and ports of the services. The information is used to separate services from each other and to get an overview of the services that exist in the system. The technique used by the services to connect to each other is different from the normal techniques used on the Internet [2, pp. 91–94][13][14, pp. 34–43].

Inversion of control

Instead of the sender knowing where to send the data, it is the receiver’s job to make itself available. The two most popular techniques a service use to announce its existence will be discussed below. But the common practice is always that the sender uses a repository to get the address of the intended receiver, while the receiver updates this repository with its location, by providing its IP address [15][2, pp. 91–94].

Client-side service discoverability

Here, a single service, called the Service Registry in Figure 3.1, provides the contact information needed to communicate with all the services in a system. The downside is that the selection of what service to use is done by the service that is asking. The service is also responsible for load-balancing [16][14, pp. 35–36]. In Figure 3.1, you can see the Service Registry is given the IP addresses of Service 1, 2, and 3. The Service Registry then gives the IP of Service 2 to Service A.

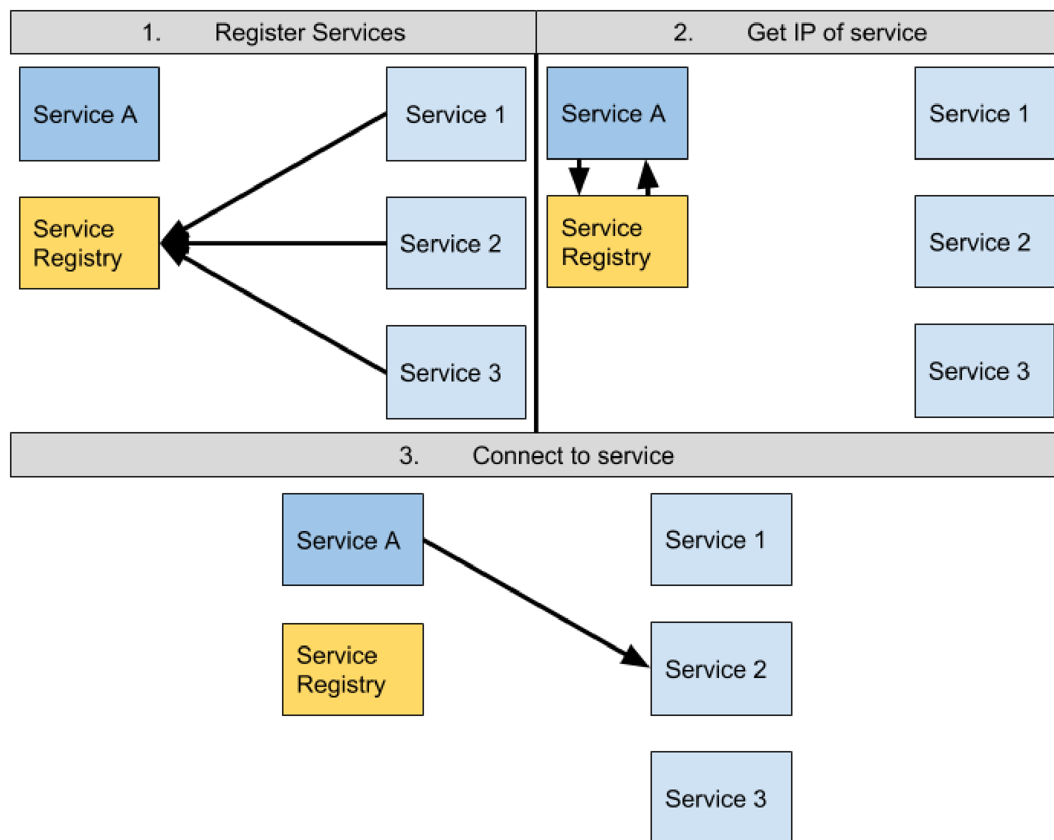


Figure 3.1: Client-side service discovery

Server-side service discoverability

Using a separate *load balancer* gives the discoverability another layer of abstraction. A service communicates through a router that also functions as a load balancer [17]. In Figure 3.2, we see a server-side service discovery setup that contains a load balancer. We can see that services 1-3 give their IP to the service registry. Then Service A asks the load balancer for a service. The load balancer asks the service registry for the IPs of the

instances of a particular services, and then the load balancer decides what instance is best to send to based on the information available [2, pp. 62–63].

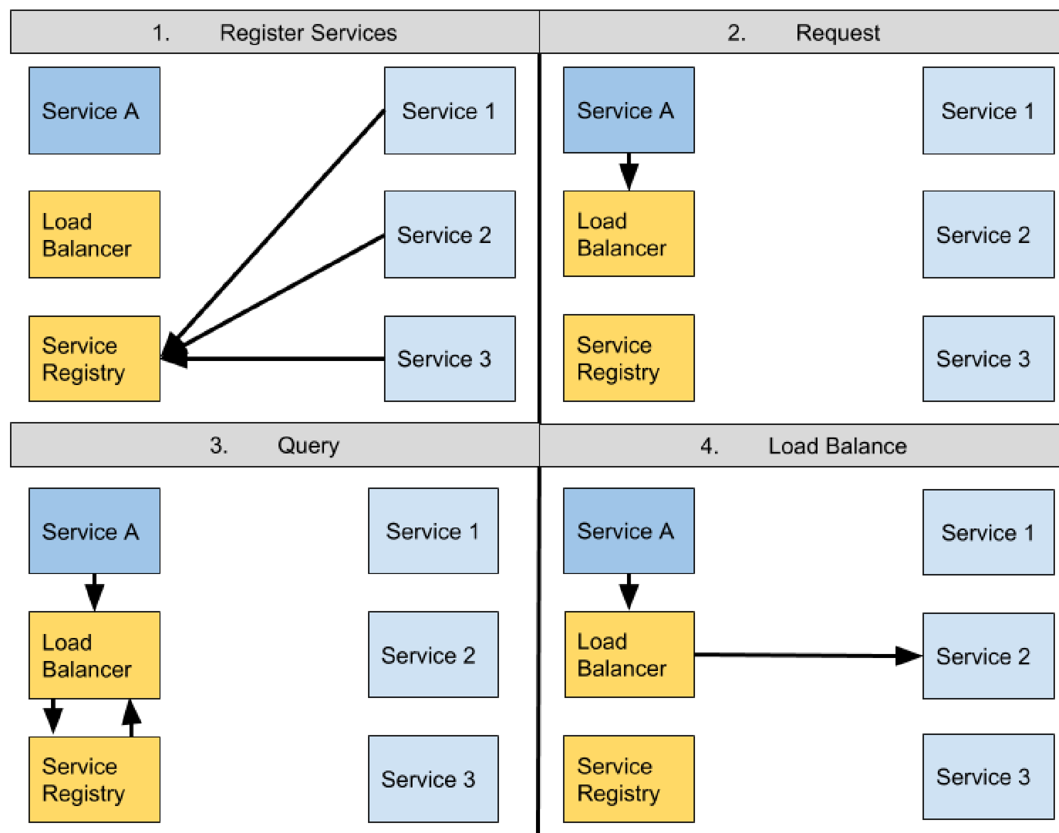


Figure 3.2: Server-side discovery

3.1.2 API gateway

When you communicate with a system that uses a microservice architecture, there will be many services that are called to complete a task. This means that it may be necessary to keep track of hundreds of different protocols, service versions, and APIs. If the caller was to keep track of the many implementations, then the services would be strongly linked, making it hard to refactor the services. Therefore, it is a good idea to keep an API gateway that tailors the services needed for the different requests and keeps track of the method you need to communicate with a particular service. An API gateway encapsulates the services, and makes sure that only the services with external APIs are contacted by other parts of the system [18][14, pp. 12–20]. A simple API gateway is illustrated in Figure 3.3.

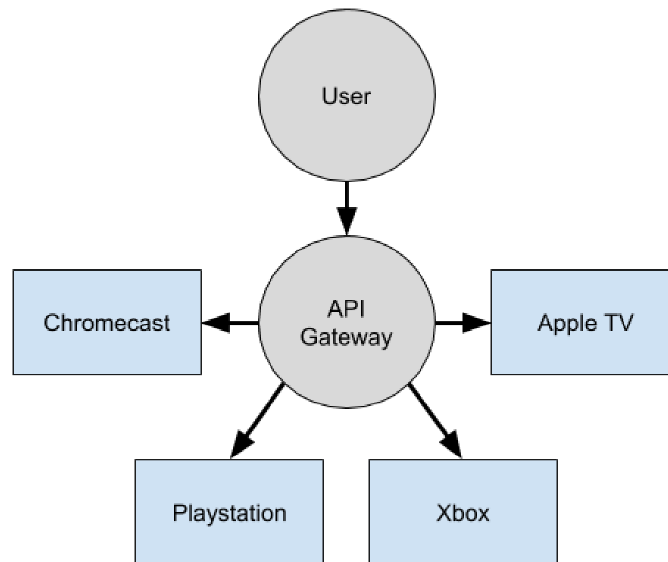


Figure 3.3: API gateway

3.2 Communication between services

Here, we will discuss the distinctions between different types of messages and protocols, and how to communicate within a microservice system.

3.2.1 Synchronous and asynchronous communication

To have a fast-responding program, it is important that the communication within the program is asynchronous and non-blocking. This form of communication will free up a massive amount of time for both the sender and receiver. If a program sends a package asynchronously, the program will only use the time it takes to send the package. If it sends the package synchronously, the sender will have to wait for the confirmation from the receiving service to be able to move on, and the waiting time can build up fast when it sends many packages. One popular synchronous protocol is REST. It works fine for external communication between systems, but for internal communication, it is recommended to avoid REST [2, pp. 68–74]. In Figure 3.4, we can see the time difference of sending ten packages that take 100 milliseconds to process.

3.2.2 Service interaction

There are two main categories of communication [14, pp. 22–29]:

- *One-to-one*, where one service communicates with exactly one other service.

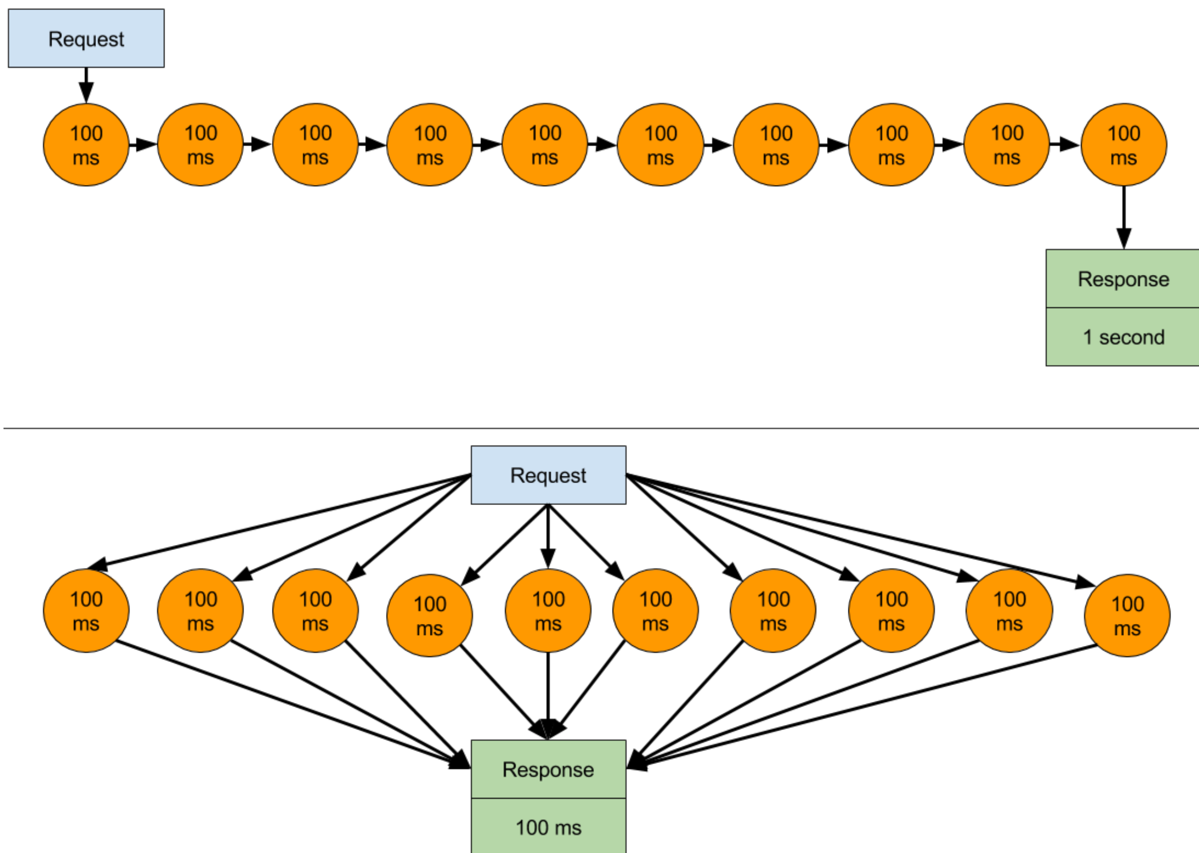


Figure 3.4: Synchronous and asynchronous communication

- *One-to-many* where one service communicates with more than one service.

With these two categories, we have different methods to communicate. For one-to-one we have:

- *Request/response*, where a service sends a request and wait for a response within a set timeframe. This method can be both synchronous and asynchronous, meaning blocking and non-blocking messaging.
- *Notification*, where a one-way message is sent with no expectation of response.
- *Request with an async response* is when the service asks another service for information and the responding service answers when it wants.

For one-to-many communication there are two main methods:

- *Publish/subscribe* means that the service subscribe to certain events. This helps with the decoupling of services and makes the communication more efficient. Instead of

pinging the service every 10 seconds if a change has happened, the service tells you what has happened. This is the primary method for the actor model [3, pp. 68–71].

- *Publish with async response* is when the service requests some information and expects responses within a time limit.

3.2.3 Events, commands, and reactions

It is essential to distinguish between an *event*, a *command*, and a *reaction*. A command represents an intent to perform some action [2, p. 42], and most often is made by the person using the application. A command can also have side effects. An event is what has happened. It is immutable and will be demonstrated to be a fundamental concept later in the thesis where we will use a technique called Event Sourcing. An event is also singular, meaning that two changes cannot be one event. One event can only change one thing. A reaction is something that needs to happen after something else happens. Depending on your preferred modelling technique, you can view a reaction as a separate modelling event, or model it as a normal event. We will delve into domain driven design in the next chapter, and there we will only use commands and events [19]. A reaction is most of the time created by the system [2, pp. 44–50]. In Figure 3.5, we can see a command in blue, multiple events in orange, and one reaction in yellow. The command is to submit a credit application, and the event is when the credit application is submitted. Then the reaction is to evaluate whether or not to accept the application. The evaluation will result in one of two events, namely an approval or denial of the application.

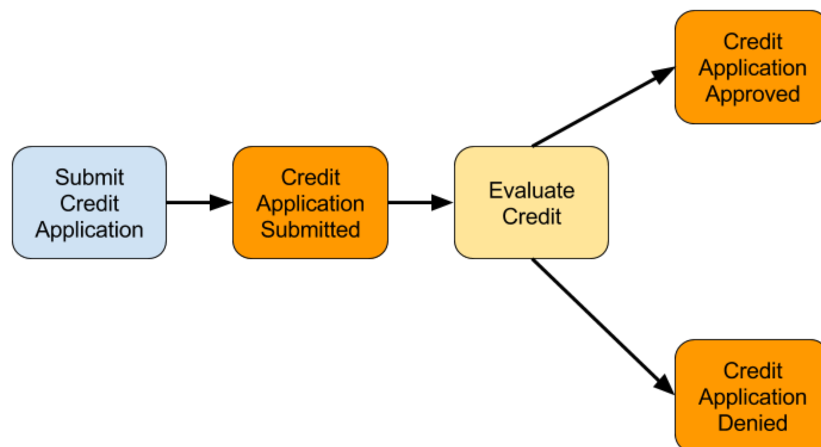


Figure 3.5: Events, commands, and reactions

3.2.4 Backpressure

Backpressure is the signal that is sent by a receiver of messages to get the sender to slow down the transmission of packets. Backpressure will prevent the receiver from being overwhelmed, and end up filling the queue and eventually losing packages. To realise backpressure, a system uses a dedicated channel to signal the status of the flow [2, pp. 74–75].

3.2.5 Circuit breaker

When communicating with an outside system that has not implemented the proper protocols, using synchronous communication, no backpressure, or any other problematic setup, it is smart to use a mechanism that guards the entry-point to one's system to avoid problems. One possibility is to use a circuit breaker. A circuit breaker is a three state finite-state machine with the states open, closed and half-open [2, pp. 75–79]. This is visualised in Figure 3.6.

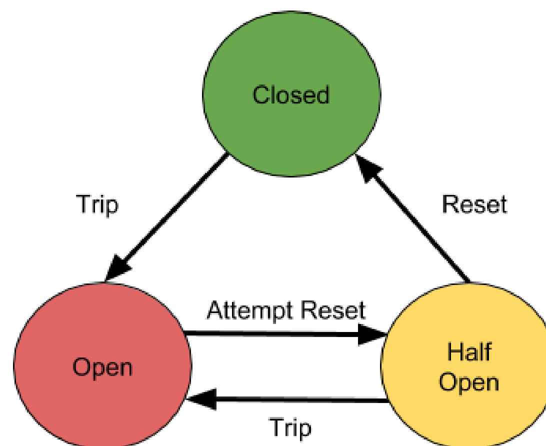


Figure 3.6: Circuit breaker

Closed

Closed means that the circuit breaker is not activated. This state implies that the communication works as it should. The circuit breaker is checking that there is no problem with the communication.

Open

Open is when the circuit breaker is activated. This means that all communication with a problematic service is blocked. There is often the possibility to program in a *graceful degradation* [20]. Instead of just closing down the communication completely, the circuit breaker provides a standard reply until the problematic service is replaced.

Half-open

Half-open is when the system tries to connect again. After a timeout, it will check if the other system is still acting poorly, and if it is, then the circuit will remain open, else it will be closed.

3.2.6 Failed delivery, dead letter queue

A message ends up in the dead letter queue when the queue it was originally sent to does not exist, the queue is full, or some transmission error occurred. A message in the dead letter queue can be picked up and processed [9, pp. 224–226][2, pp. 87–91].

3.3 Failure reduction and handling

The main advantage of a microservice system is the ability to scale its functionality to handle varying amounts of traffic in real time. Furthermore, unlike in a monolith, failures need not take down the whole system because it is possible to make microservices that can be isolated when they misbehave. In the following, we discuss how to make a microservice system robust to errors.

Before we specify how Akka enables a robust system, it is important to understand why we need to handle failures differently. In the traditional monoliths, the way we have dealt with failures is that we try to predict them. The simplest way is to build a try{ method() } catch(); clause around the code and hope that the error lands within the predicted parameters. This works in a well defined and controlled environment. Therefore, it is still the norm, and most likely will continue to be the norm when it comes to app development. The problem arises when an app connects to a system that is supposed to handle many processes at the same time. If the system fails while handling data, it would be bad practice and bad business to let the whole system fail, including the section that did not initiate the failure. You might be able to control the failures with a system with small amount of processes, but when you have millions of users and thousands of microservices, failures will happen.

Although computing is deterministic, a computer programmer is not able to program with every possible side effect in mind. Therefore, there will always be some unknown small probability that there will be an unforeseen consequence of a failure. If the failure has a one in a million chance of happening every year, and if there are a million services with this code in it, the unforeseen effect will, most likely, happen once a year. Failure can also happen on the server your code is running on, to the internet providers that deliver internet connectivity to the server, or the power plant delivering power to the server farm. Failure can happen everywhere.

3.3.1 Bulkheading and supervisor hierarchy in the actor model

The word bulkheading comes from ship design, where the product shipped is placed in different segments instead of one big compartment. The rule of an actor is that it can stand on its own, even if all of the other actors fail in some sense (except for its parent). This is very well structured within Akka, where you can build a supervision hierarchy.

Before, we have discussed that an actor becomes the supervisor of the actor it creates. The responsibilities of a supervisor are to react to a failed actor. There are a few options:

- *Resume*, meaning that the exception is ignored and the actor continues to do its job.
- *Restart*, removing the failed actor and replacing it with a new instance.
- *Stop*, the default strategy is to stop the actor and kill it.
- *Escalate*, to decide that the problem needs to be handled by the actor above itself.

It follows that there can be a whole tree of responsibilities, and reducing the instances that will escalate by handling them at the level they occur [2, pp. 94–99][9, pp. 73–76]. Figure 3.7 illustrates a simple failure in actor E. E does not know what to do, so it escalates the problem until the exception is recognised within actor A. A decides to restart actor E and sends the message down, and E restarts.

3.3.2 Event store

Event Sourcing occurs when events take place and are recorded in a journal, also called an *event store*. The usefulness is that you can trace what has happened through the system at any time since it is events, and not commands, that are logged. As said before, commands have side effects, events are isolated facts. It should be mentioned that there can be more than one event store in a system, but in this thesis, we will only use one [21][22][2, pp. 79–81][14, pp. 52–53]. In Figure 3.8, we see an order made with a message on the left, and an

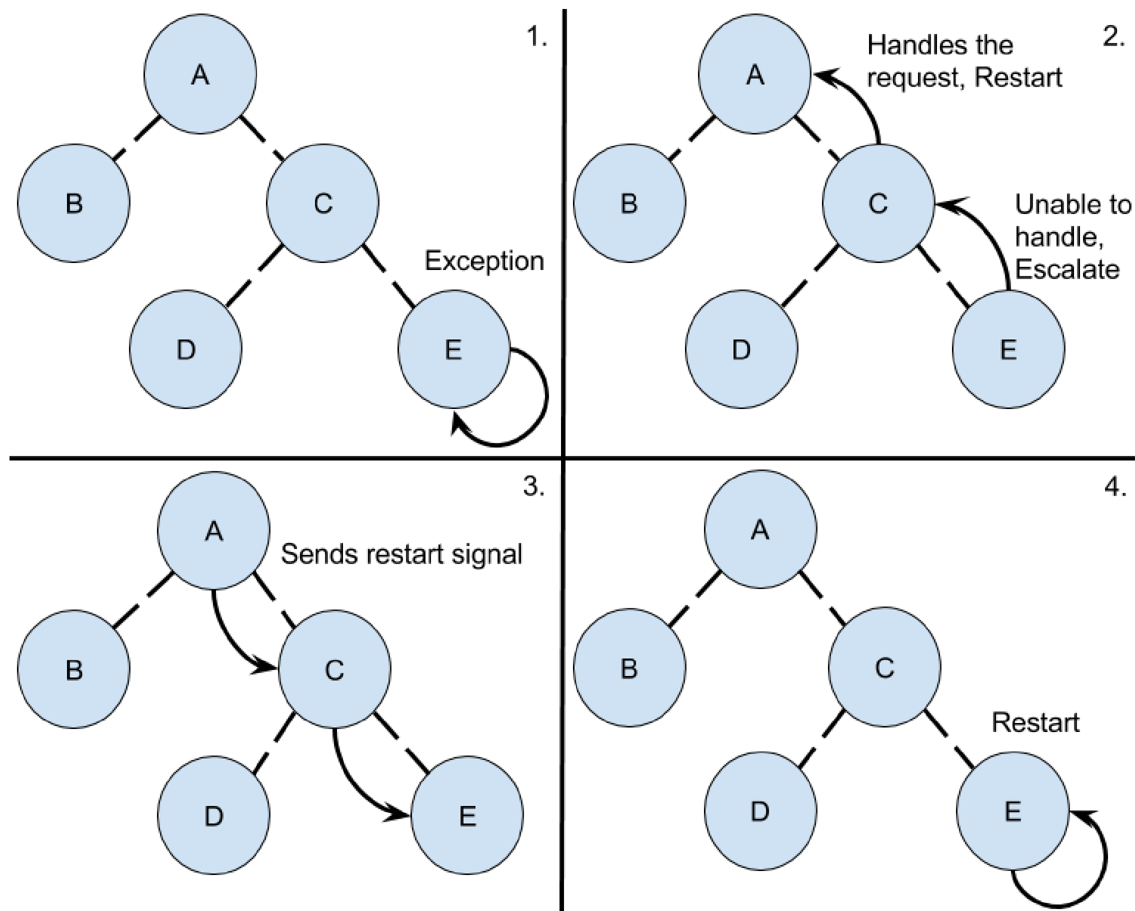


Figure 3.7: Bulkheading

order within the event store on the right. The event store registers the state of the event. Connected are the services that published events and subscribes to the events.

Event log

The nice thing about an event store that records all events in a system is the ease with which it is possible to study what the system has done. The event store can help with debugging, auditing, and similar useful actions. With a full log of all the events in the system, this becomes much easier. It is important that a saved event is not edited after it is stored. If something wrong has happened, then it can be corrected.

As the event log allows append-only, it reduces the complexity and speed penalties of concurrent access. Two examples of problems that might occur if one to delete an event in an event log can be seen in Figure 3.9, where the above illustrates the problem if the log resizes, and the bottom illustrates the problems of the nonresizable log. There are methods to create a system that might avoid these problems, but it will not be discussed in this

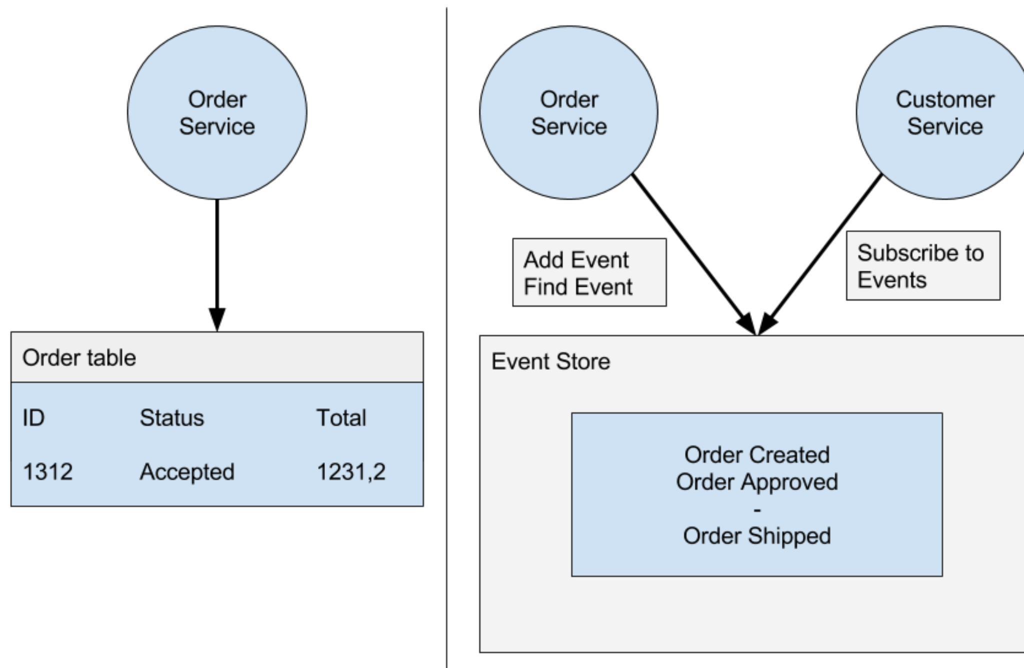


Figure 3.8: Event sourcing

thesis [2, pp. 113–114].

Command query responsibility segregation (CQRS)

The idea of CQRS is to separate the reading action from the writing action to the Event Store. The separation is achieved by grouping actors who read and write separately. Read mode is for a query and is normally used by services subscribing to the Event Store to get updates if new events are sent. Write mode is to record the events.

The benefits are resilience and scalability. The main reason is to achieve temporal decoupling. This means that you separate the writing from the reading and that they do not have to be available at the same time. There is also the benefit that the read and write services can be scaled individually, which makes the Event Store usage more dynamically scalable [23][24][2, pp. 116–122]. In Figure 3.10, you can see a separation demonstration of how the CQRS layer is placed in front of the event store.

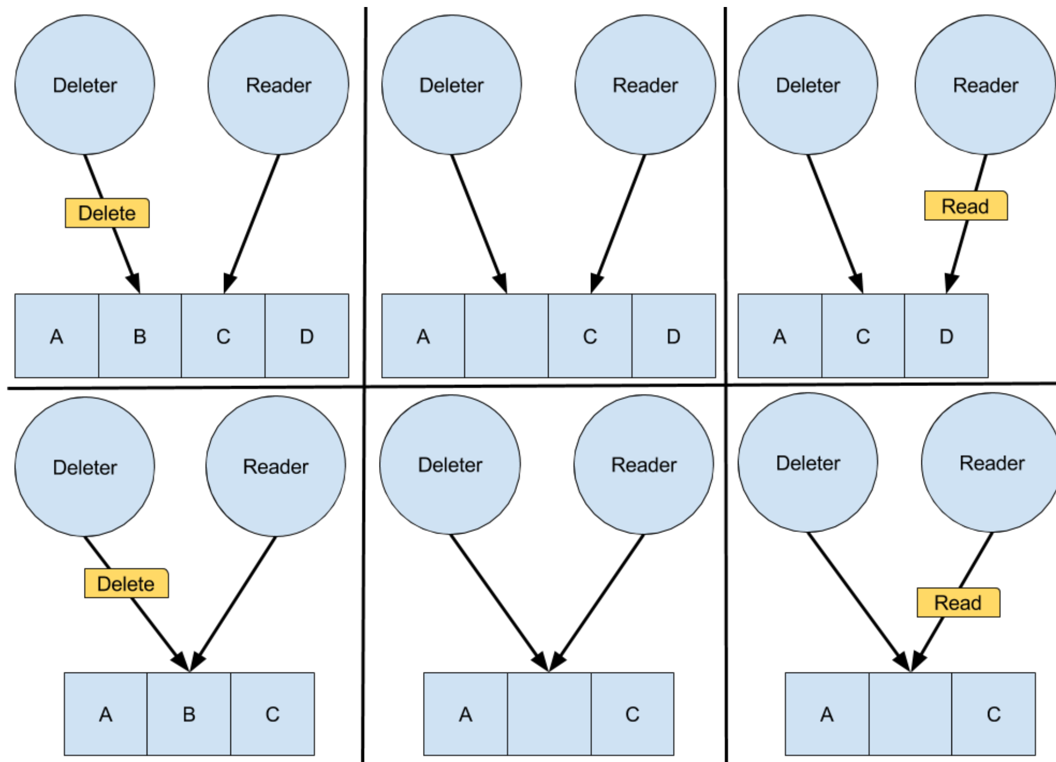


Figure 3.9: Deletion and resizing of an event log

3.3.3 Containerisation and orchestration

Containers and virtual machines

To effectively isolate applications from each other, or run several instances of the same application, it is good practice to containerise the application. What this entails is to separate the application from the operating system.

The two main categories are *containers* and *virtual machines* (VM). Containerization is an operating system feature where the kernel supports multiple isolated user-space instances, called containers. Each container can run an application. A VM contains an operating system and application. A hypervisor between the physical machine and the operating system hides the hardware from the VM and let different VMs run different operating systems. Both a container and a VM has a runtime system. The runtime system is needed to run an application. It satisfies requirements generated by the programming language.

In Figure 3.11, we can see the different layers of the VM and container. These two technologies are different in security and speed. We first consider the VM. To get from the application to the operating system, you have to go through the hypervisor and another

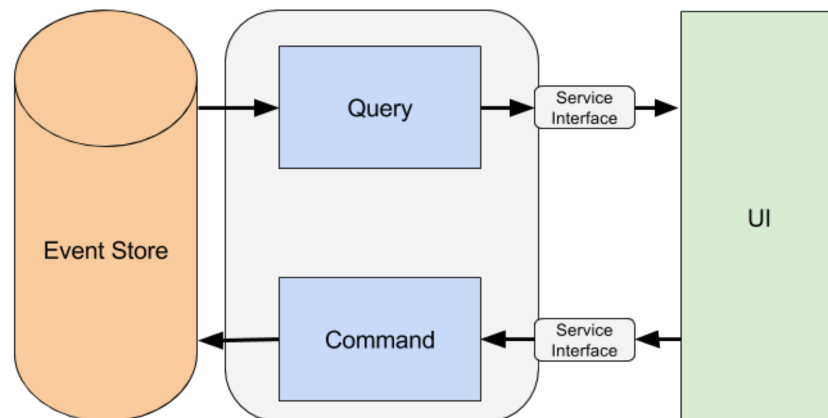


Figure 3.10: Command query responsibility segregation

operating system. These are two heavy application that create strong isolation from the host machine. But this also means that it takes a long time for the VM to start. Contrary to a VM, the container can start up in seconds instead of minutes. But in return, there are fewer levels of protection. Containers can also be inside a VM and can be configured to work well, but we will not go into how this could help [25][26][14, pp. 55–63].

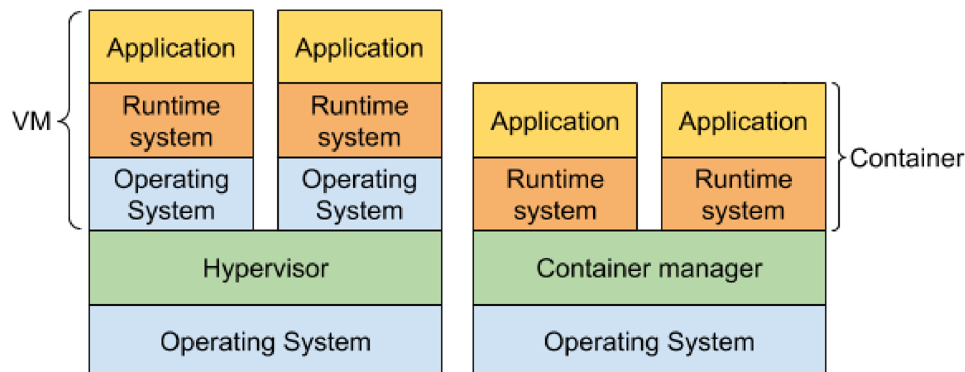


Figure 3.11: Containers and virtual machines

Orchestration

We will only discuss container orchestration. To orchestrate is to automate the application lifecycle. We can organise the different containers through a manager. This can get a container to appear at a certain time if the number of active users increases or there is a need for more processing of internal data, or something changes within.

There are differences within every orchestration platform, but Figure 3.12 shows a manager handling the workers. Here, the manager has the responsibility to start, remove,

and potentially restart the workers. The manager checks that there are enough workers, and scales if necessary. We are not going into to more details about orchestration since it is not essential for the understanding of the rest of the thesis [27][28][29][30].

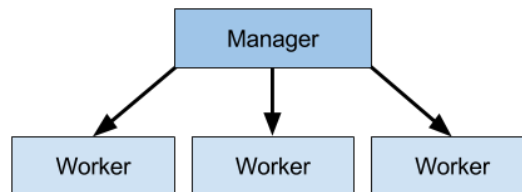


Figure 3.12: Simple orchestration setup

3.4 Summary

This chapter has introduced the general architectural methods that are used to create a microservice system. Firstly, we introduced how the services make themselves available, how to communicate with them, and how to increase the autonomy of the services by having an API gateway. Afterward, we went through how the communication within the system should function, and different attributes to give microservice systems to increase the overview and balance of the system. Lastly, we looked at how we can reduce the impact of failure on the system.

Chapter 4

Design

Before we delve into domain driven design, it is important to understand the problems we face when designing a large system. First, we need to know about the *CAP theorem*.

4.1 The CAP theorem

The CAP theorem [31] states that any networked system can have at most two of the three properties *consistency*, *availability*, and *partition tolerance* at the same time. This is one of the more important results to understand before designing a distributed system, and can be explained by considering an illustrative example.

Let us say we are going to purchase a book on a web bookstore. There is only one book left and two customers, user A and user B, both want to purchase it. The system perspective is illustrated in Figure 4.1 where you can see user A and user B, as well as their respective inventories with one book in each. In this instance, the views of the store are consistent since they have the same information, available because both can be reached and have two network partitions.

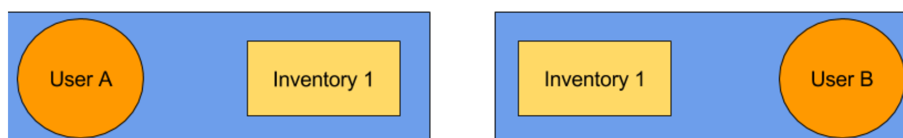


Figure 4.1: Views of store

If user A purchases the book, then the update message needs to be sent to the inventory of user B. This is illustrated in Figure 4.2. There we can see that immediately after the purchase has happened, the information presented by the two network partitions are inconsistent. If the update message between the network partitions is dropped or delayed,

the information will stay inconsistent for an indefinite amount of time.

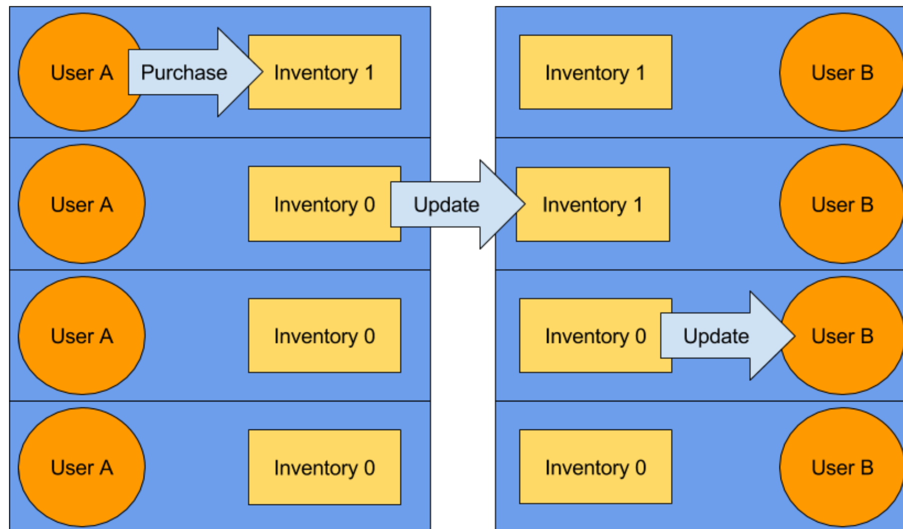


Figure 4.2: Store communication

Another way to understand the concepts of consistency and availability is to consider a system's ability to write and read data. A designer needs to understand whether it is possible to write data to a system. If the system is continuously available, then the designer must know if it is always possible to read the newest writes. The designer needs to create a system to handle the write and read limitations correctly.

The reason why partition tolerance will be prioritised in both instances (CP or AP) is because to have a system that prioritise both consistency and availability will fail. If we look at the previous example again and imagine that the two partitions lose connection to each other and one node gets a write request. The node can choose to write the data and therefore choose availability, or it can refuse to write the data and choose consistency. If one chooses both, then the system is not distributed as it would be on the same local system.

It should be noted that the choice between consistency and availability is not a binary choice, but continuous. No system is perfectly consistent in reality because data cannot travel faster than the speed of light. The design should increase the consistency and availability to the highest degree, but you will never manage to get perfection on either of them. Much is written about consistency and availability. The interested reader should consult [32][33][34][35]. Our goal is to prioritise availability and handle consistency in the best possible way.

4.2 Strategic design

The term strategic design implies that the designer first creates an overview of the system design before going into details. Strategic design is used to highlight areas of importance, how to divide the areas by importance, and the best way to integrate them [36, pp. 7–8].

4.2.1 Bounded context

A bounded context is a semantic contextual boundary. When you get large systems, words change meaning with different contexts, like the term "policy." In a large system, this distinction needs to be precise and sensible, so there will be no confusion in the end product. For every meaning of the word, a different bounded context is created. For every bounded context, there should only be one developer team, and a separate source code repository for every team [37] [19][2, pp. 46–49][36, pp. 11–17].

Within a bounded context means that the language is in context, and if there is a language that is outside of the bounded context, it is out of context. Figure 4.3 illustrates two separate ubiquitous languages and their aggregates, discussed later in the chapter.

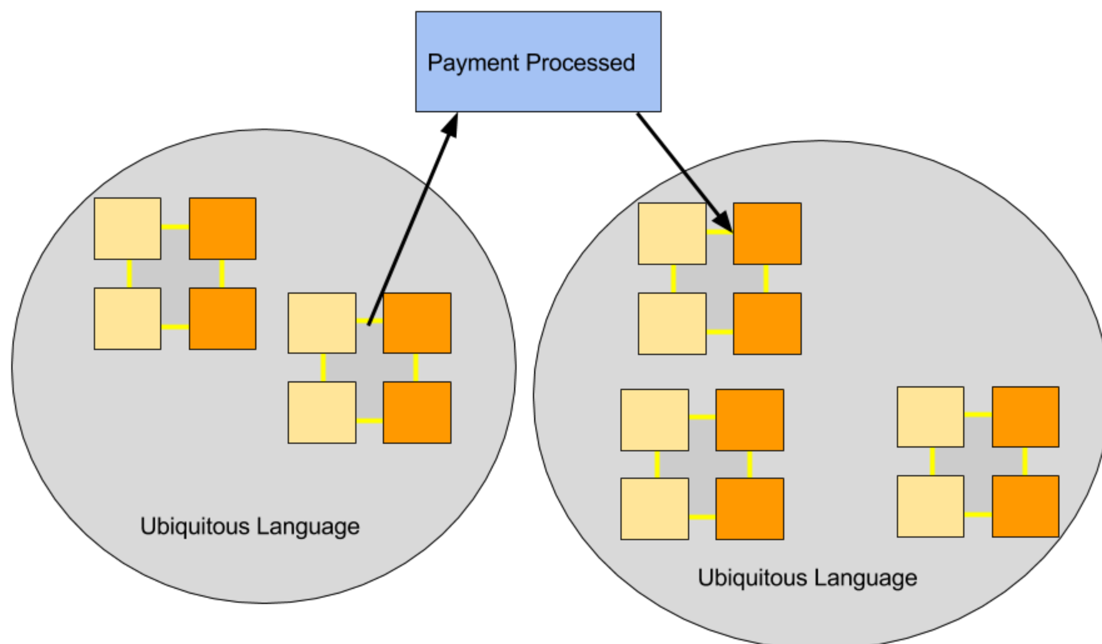


Figure 4.3: Bounded context

4.2.2 Ubiquitous language

It is the language used inside of a bounded context. The language reflects the terminology the developer team uses to discuss and create the bounded context. It is called a ubiquitous language because the spoken terminology is the same as the language used in the implementation. Therefore, it needs to be precise [36, pp. 11–17].

Designing ubiquitous language

You can use descriptions of scenarios to illustrate what each component is supposed to do. This does not mean user stories or use cases. Use cases are what a user can do. A user story is more of a description of who the user is and what the user wants and why. A scenario is a description of how the system is allowed to act. You can describe who is using the system, but it is not user-centric [36, pp. 34–38].

4.2.3 Core domain

The core domain is the key strategic initiative of the organisation. It is the centre that is always going to be used within the system, where the other bounded contexts are supplementary contexts, they are essential model elements [36, pp. 11–17][36, pp. 29–34]. In Figure 4.4, the core domain is in the centre surrounded by other bounded contexts.

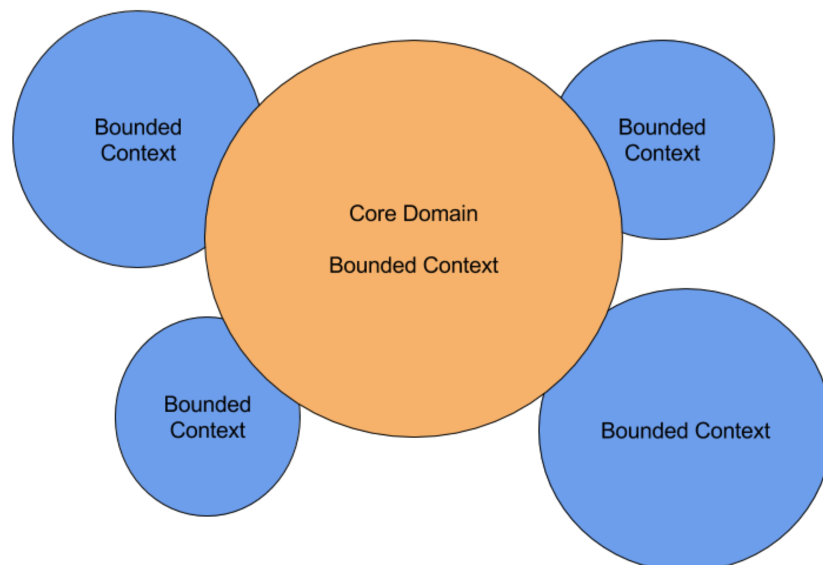


Figure 4.4: Core domain

4.2.4 Subdomain

A subdomain is a part of your overall business domain. A subdomain should be within a bounded context [36, pp. 45–47].

Types of Subdomains are:

- *Core Domain*: Well defined, with a *ubiquitous language* within a *bounded context*. The highest amount of time invested.
- *Supporting Subdomain*: Created due to needs and because no off-the-shelf solution exists. If the business was a phone operating system, it would be the podcast application.
- *Generic Subdomain*: Functionality that is important for the system but is not what the business does. A good example would be the ability to transfer money.

4.2.5 Context mapping

To context map, is to create a connection between bounded contexts. It is to effectively map one bounded context's ubiquitous language to another one. Figure 4.5 illustrates concepts associated with context mapping [36, pp. 51–54].

- A *Partnership* is when two teams with their own bounded contexts frequently meet to align code deployment and schedules [36, p. 54].
- To have a *Shared Kernel* means that two or more teams share a common model [36, p. 55].
- To have a *Customer-Supplier* mapping is to have one bounded context upstream and one downstream representing supplier and customer, respectively. The supplier holds sway because it provides what the customer wants [36, p. 55].
- *Conformist* mapping is to have upstream and downstream dynamics, but upstream has no motivation to support the needs of the downstream. Therefore, the downstream team conforms to the upstream ubiquitous language [36, p. 56].
- To use an *Anticorruption Layer* (ACL) is when the downstream team creates a translation layer between its ubiquitous language and the upstream ubiquitous language [36, pp. 56–57].
- Using an *Open Host Service* (OHS) is to define a protocol or an interface to communicate with the bounded context [36, p. 57].

- A *Published Language* (PL) is a well-documented information exchange language like JSON or XML [36, p. 58].

4.2.6 At-least-once delivery and idempotent receiver

This delivery mechanism guarantees that a message is delivered at least once. The message may be re-sent after some time because it was dropped during transmission or because the sender did not receive an acknowledgment from the receiver within a certain time window. As a consequence, the same message can be delivered multiple times. The receiver needs to be engineered to handle multiple copies of a message. An idempotent receiver ensures the same result even when it processes the same message multiple times [36, pp. 65–69].

4.3 Tactical design

This section outlines how to realise a domain driven design. The explanation will often match the earlier description of an actor system. The reader should view the introduced objects *entity*, *value*, and *aggregate root* as actors.

4.3.1 Entity

An entity is a model of a thing. It is something that can be pointed out as being different from another entity. It is mostly mutable but can be immutable. An entity can contain one or more aggregates [36, p. 76]. If domain-driven design is used to create an airplane, then a seat on the plane is a unique entity.

4.3.2 Value object

A value object models an immutable conceptual whole. It is often used to describe, quantify or measure an entity. To determine wether two value objects are equal we compare their value. A credit card would be a good representation of a value object, because two cards are viewed as the same if they represent the same bank accounts, not if they are made of the same plastic or are similar in shape [38][39][36, p. 77].

4.3.3 Aggregate

An aggregate has a consistent transactional boundary, which means that the aggregate state should be transitioned and maintained safely and correctly at all times. Two different

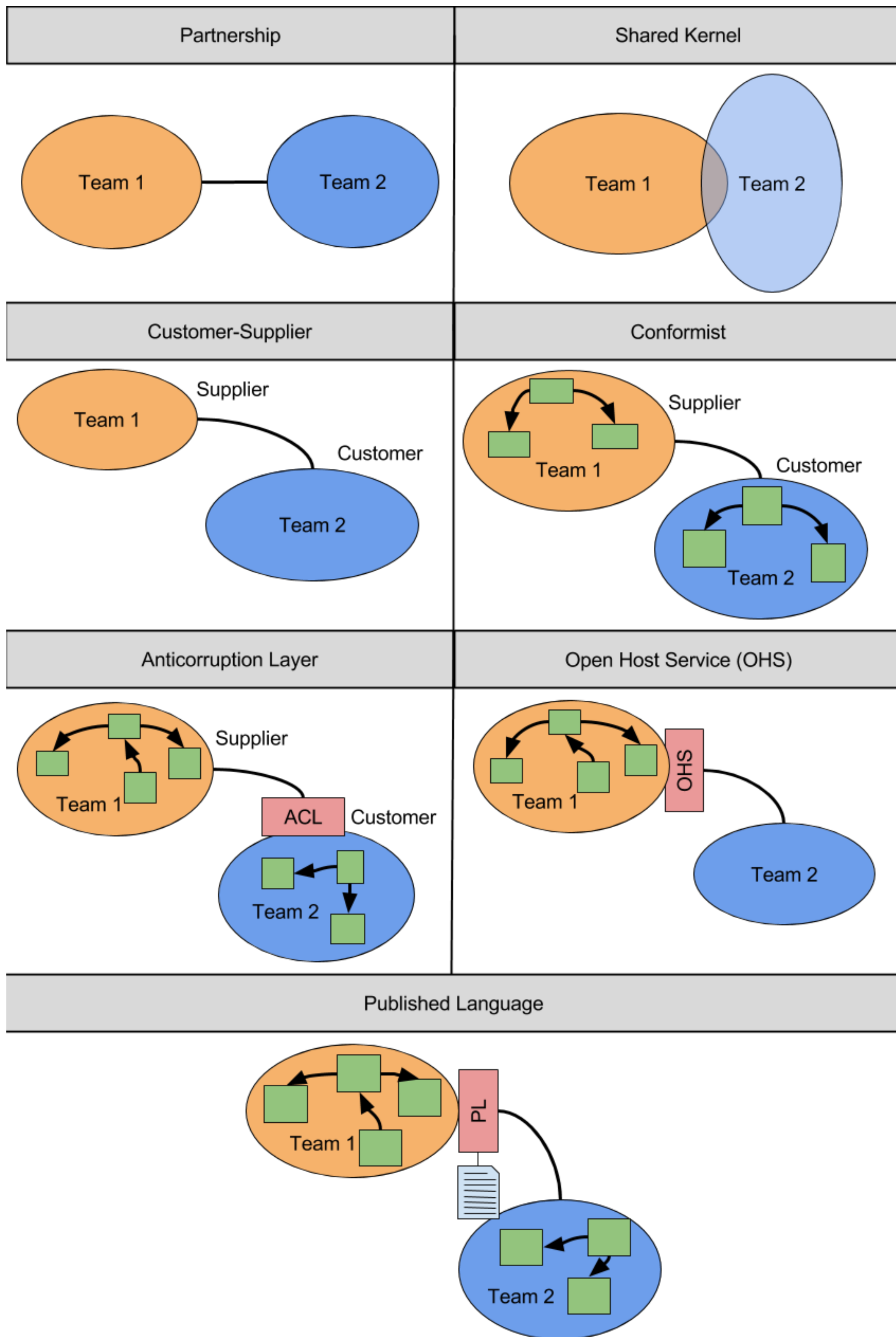


Figure 4.5: Context mapping

aggregates should not be committed in one transaction. In the actor model, an aggregate is implemented as an actor.

An aggregate is a grouping of commands, events, and reactions. Aggregates are separated into aggregate events, events that affect this logical group only, and *domain events*, meaning events that affect the wider system. We will discuss domain events later in this chapter. The reason to use aggregates is to make it easier to work with the system, as you mostly need only consider your aggregate to work on it, instead of a whole system. It separates behaviour in a sensible way [19][2, pp. 54–59]. In Figure 4.6, we can see a collection of events and commands that constitute an aggregate.

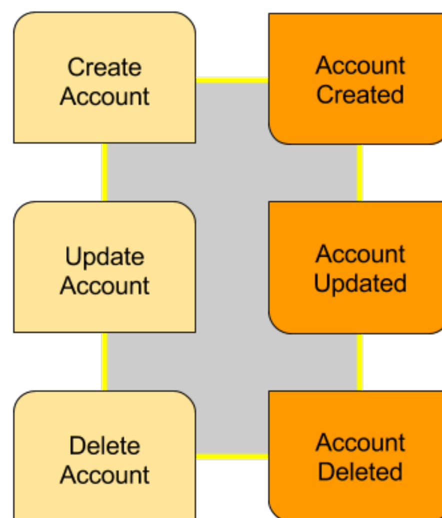


Figure 4.6: Aggregate

The aggregate is business motivated because it is the business that knows the valid states of the cluster. If the aggregate was corrupted, the business operation performed would be corrupted [36, pp. 75–79].

Aggregate root

An aggregate root is the first aggregate. In our use case, the aggregate root is the actor system. It is the first aggregate, and nothing is above it on the control hierarchy [36, p. 77]. An illustration of how a value object, entity, and aggregate root are arranged can be seen in Figure 4.7. As you can see through the modelling, it imitates the actor model very closely.

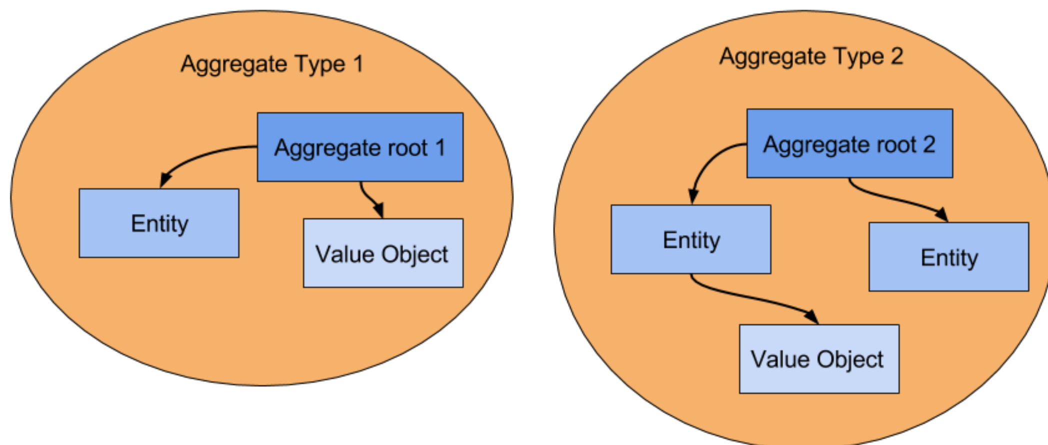


Figure 4.7: Aggregate root, entity, and value object

Implementing aggregate rules

- *Protect Business Invariants within Aggregate Boundaries*: Some invariants need to be met simultaneously. Therefore, they need to be performed in parallel. An example is if something is sold out, then it should not be possible to purchase that item from the store [36, p. 82].
- *Keep Aggregates Small*: Keeping your aggregates small give them a smaller memory footprint, transactional scope, increases the ease of using a ubiquitous language, eases testing, and uses the single responsibility principle [36, pp. 83–84].
- *Reference Other Aggregates by Identity Only*: Prevents the possibility of modifying two aggregates with the same transaction. It also makes it easier to use a journal, as one has to only store the reference to the aggregate and the change that the aggregate preforms [36, pp. 84–85].
- *Update Other Aggregates Using Eventual Consistency*: It is important to realise that not everything can happen simultaneously; making changes take time. With a large system, changes can take a long time, and an even longer time to become consistent. Therefore, it is important to design the system with eventual consistency in mind [36, pp. 85–88].

4.3.4 Consistency

There are different versions of consistency in a system. The conflict between consistency and availability was discussed at the start of the chapter within the CAP theorem. The conclusion was that availability should be prioritised and consistency should be handled in

the best possible way. The first reason was that for a commerce site of any kind downtime will be costly to the company. Even if it is a non-profit site, very often outdated information will be better than no information at all. The second reason is that partial availability is hard to do properly, and can be out of the engineers' control. Instead, there are methods to make the system available, but the information available is not fully updated but can be processed and hopefully come through without problems. If there is an important inconsistency, then there are methods to work around this as well, as refunds or apologies [40].

Client-side consistency

We will use Figure 4.8 to illustrate the different consistency versions. Here, we have processes A, B, and C that read and write values to a storage system. The storage system is treated as a black box, and the processes are independent. We will look at how processes B and C react when process A sends an updated value to the storage system, and explain what the different consistency definitions mean.

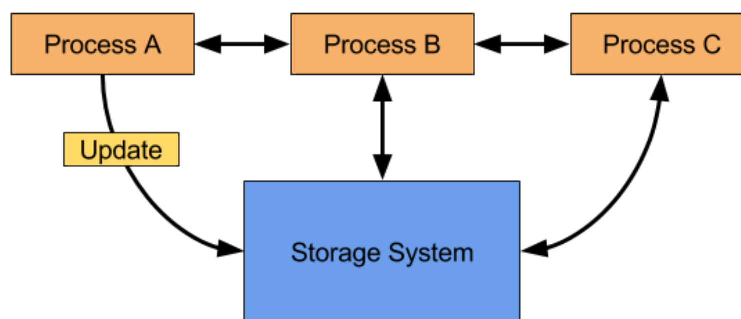


Figure 4.8: Processes communication with a storage system

- *Strong consistency* means that if process B and C read from the storage system after an updated value has been delivered, they will get the updated value.
- *Weak Consistency* means that process B and C may not get the newest value if they check the storage system because certain conditions need to be met. The time between the updated value is sent and when the updated value is ready to be read is dubbed the *inconsistency window*.
- *Eventual Consistency* is a different version of weak consistency where, if no failures occur, the maximum size of the inconsistency window can be determined. It also guarantees that if there are no new updates done to the system, then all processes have access to the updated information.

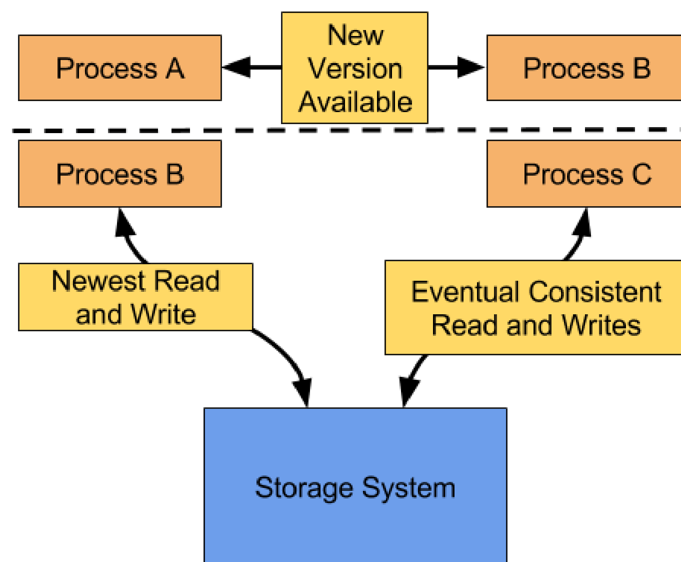


Figure 4.9: Casual Consistency

There are different versions of eventual consistency, where versions can be combined to create a better completed operation.

- *Casual Consistency*, illustrated in Figure 4.9, is when process A communicates to process B that it has delivered an updated value to the storage system, the storage will be updated and prioritised for B. This also means that any update written by B will supersede the earlier write. C is still only eventual consistent [36, pp. 99–100].

- *Read-your-writes consistency* is a special case of Causal Consistency where process A will always access the newly updated value in storage, and never an older value.

- *Session consistency* means that the read-your-writes consistency is guaranteed within your session. A session here is a period with a finite number of actions and with a certain ending action. If the session terminates while in play, a new session will be started where the guarantees do not overlap.

- *Monotonic read consistency* means that if any process has seen a value, then it will never see a previous version again.

- *Monotonic write consistency* means that the writes done by a process will be serialized.

Eventual consistency will help the storage system to relax the consistency and provide high availability.

Server-side consistency

To map the update flow through the system, we need to have control of 3 values:

- The number of nodes that store replicas of the data.
- The number of nodes that need to acknowledge the writes of data.
- The number of nodes that are contacted through a read operation.

If the sum of the number of nodes that need to acknowledge the writes and the number of nodes contacted through the read operations is larger than the total number of nodes, then it is guaranteed strong consistency. In Figure 4.10 we can see that there is a total of 3 reads and writes, but only two nodes. The example to the left shows that there is no problem if there is only one read operation, because there was two writes that updated the total number of nodes. On the right side you can see that there is no problem if there are only one write operation because the reads takes all the nodes into consideration. This is not how a relatable database looks like, but will illustrate the problems with replaceable data.

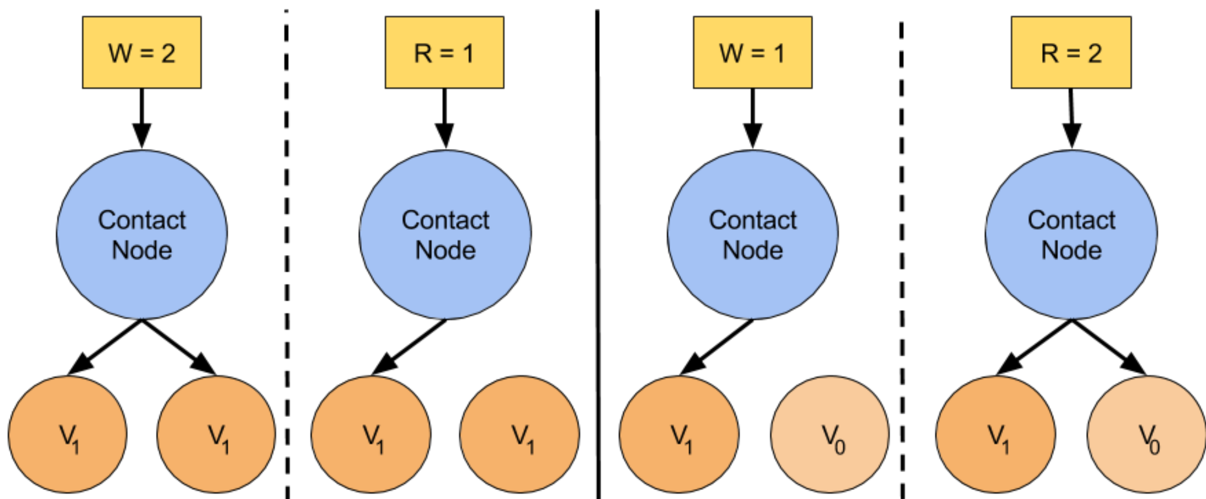


Figure 4.10: Server-side consistency example with different read and write values

If there are a equal or less amount of operations then nodes, strong consistency cannot be guaranteed. To illustrate the problem we can look at Figure 4.11 where we can see on the left a setup with three operations and three nodes, while on the right there are two operations with two nodes. In both scenarios, the read operation will receive outdated values.

In the end, if we have one read operation and as many nodes as writes, the system is optimised for fast read cases. With one write operation and as many reads as there are

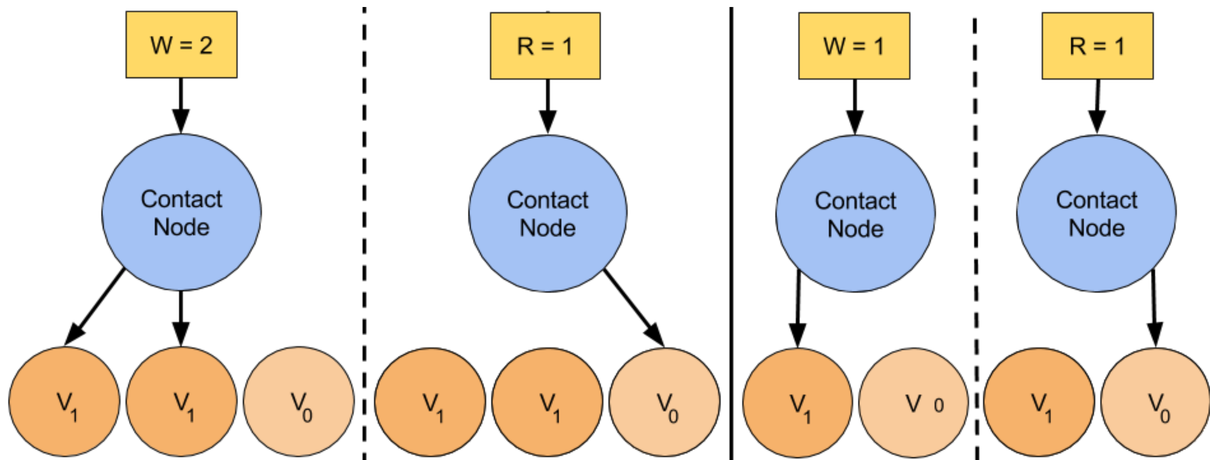


Figure 4.11: Consistency failure on the left with a total of 3 operations and 3 nodes, and on the right two nodes and two operations

nodes we optimise for fast write cases. With the later the updates normally rely on epidemic techniques to update the other replicas. Figure 4.12 illustrates a possible scenario where the number of write operations are lower than half of the nodes, creating the possibility for conflicting writes.

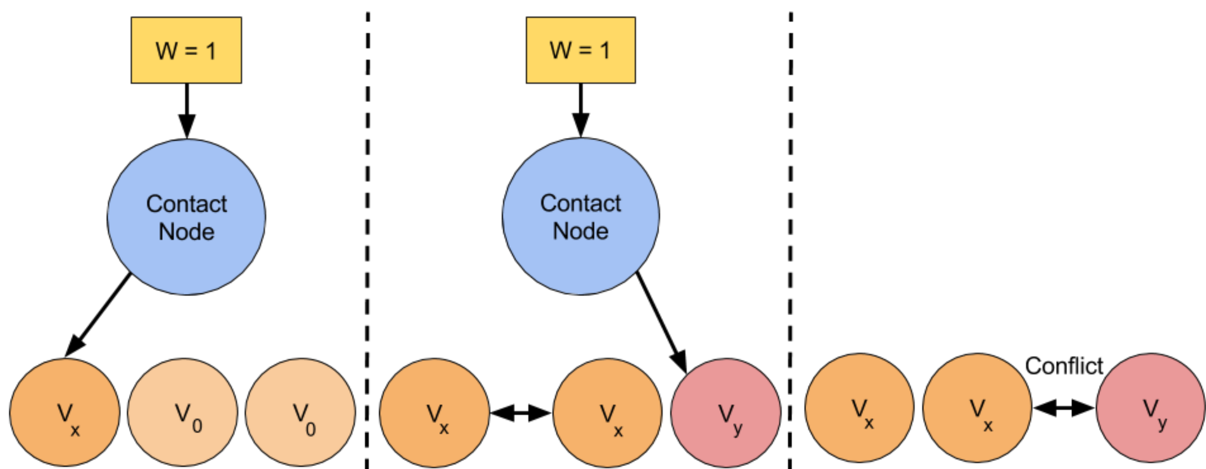


Figure 4.12: Conflicting write with write operation below the number of nodes

Weak/eventual consistency comes when the number of reads and writes are less than or equal to the number of nodes. To optimise the operations, we should have the number of writes as high as possible, and only one read operation. The low number of reading operations will mean that there is a window, the inconsistency window mention before, where one can read from an outdated node, but it will eventually be consistent. The design is desired because if one designs the system with only one read operation and wants strong consistency, then you need many more replica nodes than is necessary if one only

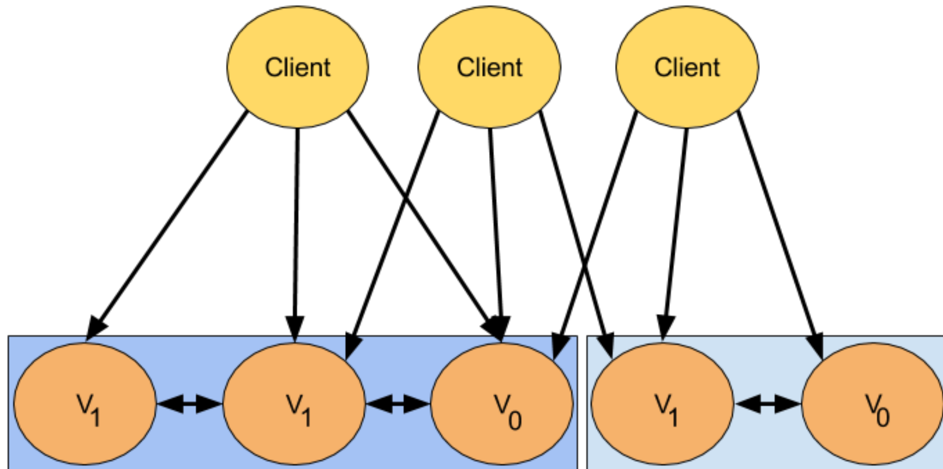


Figure 4.13: Partition consistency

design for fault tolerance. The second reason is that it is not easy to conclude what is the newest version of an object.

If failure occurs, and the system becomes partitioned, there are two main ways of handling the problem. One can be seen in Figure 4.13. Here, we see two partitions with a total of five nodes with different versions of value V, where V_1 is the newest value. Partitioning is when a group of nodes cannot reach another group. One can choose different criteria of what actions will be taken if the system becomes partitioned, but the standard way is to keep the partition with the highest amount of replicated data, while the other partition is made unavailable. In our example, if it required to have two nodes with the latest value, then the left partition with two V_1 values is the partition we keep, and the other partition we eliminate.

The other way, when lost packages are unacceptable, a new set of storage nodes are created to receive the data to be later merged with the healed system. This can be seen as the dead-letter queue we have discussed earlier.

4.3.5 Domain event

A domain event is an occurrence within a bounded context. With well-described domain events, it becomes possible to map what is happening within a bounded context. With causal consistency, we can even map what started the changes and how it is going to end. We have discussed the difference between commands (intention to do something) and events (something has happened); here it will be used. A domain event is a description of *tenantId* and *eventId*. The tenantId is what is going to happen, meaning the ID of what initiated the command. The eventId is the identification of what has been done. This means, that if

the command were to create a product, then the eventId would be productId. The domain event can be filled with other relevant information, but not contain too much information as to make it difficult to evaluate what has happened.

As mentioned before in Section 3.3.2, the domain events are sent to an event store, and the interested parties are notified. This can be to an outside bounded context, or within your bounded context.

Since a command is future tense, and a domain event is past tense, a command can be denied, while a domain event is history [36, pp. 99–107]. In Figure 4.14, we see a demonstration of domain events sent from a publishing bounded context to a messaging mechanism, while another bounded context subscribes to the domain events.

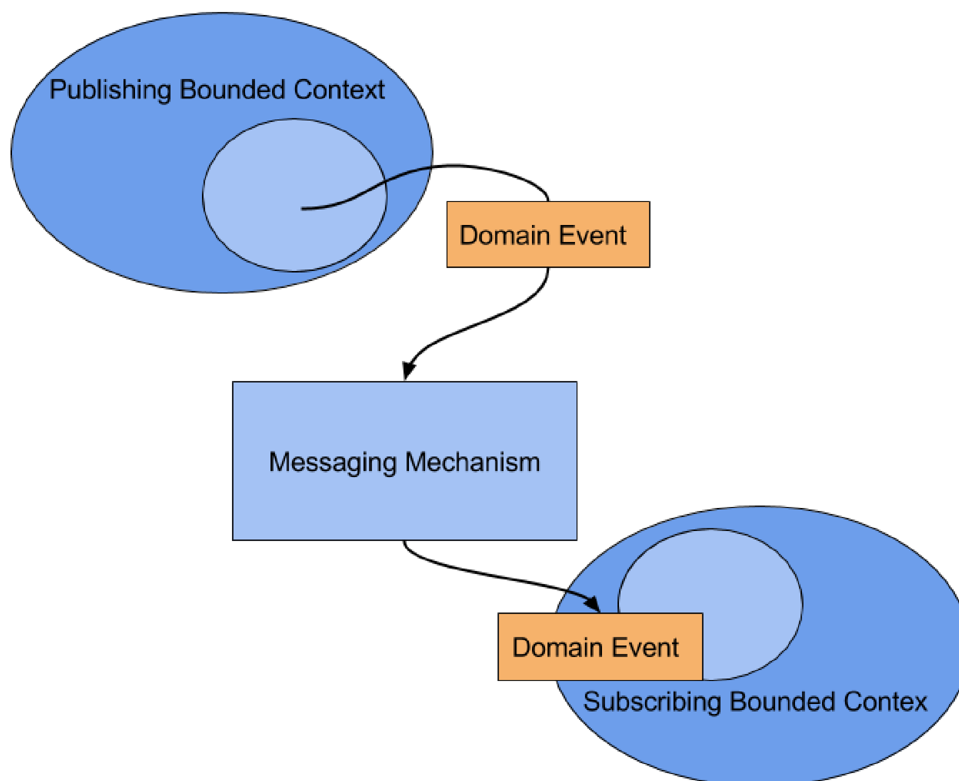


Figure 4.14: Domain events with a subscribing and publishing bounded context

According to [36, p. 107], event sourcing can be described as persisting all domain events that have occurred for an aggregate instance as a record of what changed about that aggregate instance. The nice thing about designing a system using event sourcing is that if an aggregate is removed from the system, it can be reconstituted entirely from its event stream. This is because all the domain events that have occurred in the aggregate, from the first to the last, in the correct order, are in the event stream. The events are stored in an event store [36, pp. 107–110].

4.4 Summary

In this chapter, we have discussed the CAP theorem and the limits of distributed systems. Then, we discussed the definitions and techniques needed to design a large intricate system. Later, we defineed the different structure types that can be created and what properties they should contain. We also discussed the different kinds of consistency with a special emphasis on eventual consistency.

Chapter 5

Building an Infrastructure with Akka

This chapter will focus on how Akka and Scala handle the different methods that were discussed in Chapter 3. The structures that Akka uses were discussed in Section 2.6. We do not consider everything that can be done in Akka but focus on some of the most important functionality.

5.1 Achieving asynchronous communication in Akka

Earlier in Section 3.2.1, we discussed the benefits of asynchronous communication. One of the ways we achieve asynchronous communication in Akka is by using the types *Futures* and *Promises*. These types are contained in Java [41], Scala [42], and Akka [43]. There are differences between the packages, but we will only discuss the Akka implementation.

5.1.1 Futures

A future is a placeholder for a function. When the function is completed, the future will either contain the result of the function or failure. It is a read-only placeholder that can be read more than once [9, pp. 92–101][44, pp. 247–262] (illustrated in Figure 5.1). In Listing 5.1, an event service is called and will be placed in a non-replaceable future.

The result of one future can be combined with another future. The combination of futures is called *pipelining*. In Listing 5.1, we can see Scala code being executed with a future. One future encompasses one thread, so it does not block while it is called. In the code, a service is called that returns an event that is assigned to futureEvent.

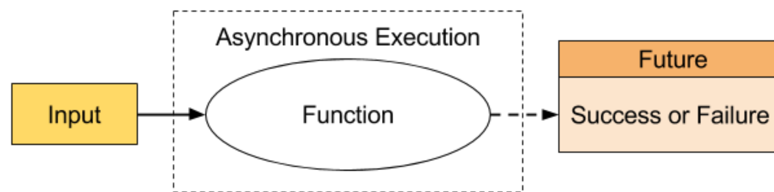


Figure 5.1: Future as a placeholder for the results of the function

What is the difference between val and var in Scala?

A val is a constant, so you can not change its content, while you can change the values declared to a var [44, pp. 4–5].

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

Listing 5.1: Future example

5.1.2 Promises

Promises are an asynchronous writing type that is a part of the future implementation. But unlike futures, they can only be read once. If a promise is completed successfully, it can be written to a future. To clarify, a function call would be in the style of Promise(Future). A Scala code example can be seen in Listing 5.2 [45][9, pp. 101–104]. Both consumer and producer act asynchronously.

```scala
// A promise is created, and the promise is used to to create its future.
val promise = Promise[T]()
val future = promise.future

// Both producer and consumer are running asynchronous

// Producer creates a value that completes the future by fulfilling the
   promise.
val producer = Future {
  val value = produceSomething()
  promise success value
}

//When the promise is fulfilled, the value is read from the future that is
   sent to doSomethingWithResult()
val consumer = Future {
  future onSuccess {
    case value => doSomethingWithResult()
  }
}
```

Listing 5.2: Promise and future example

Figure 5.2 further illustrates the example. Here, the promise completes the future.

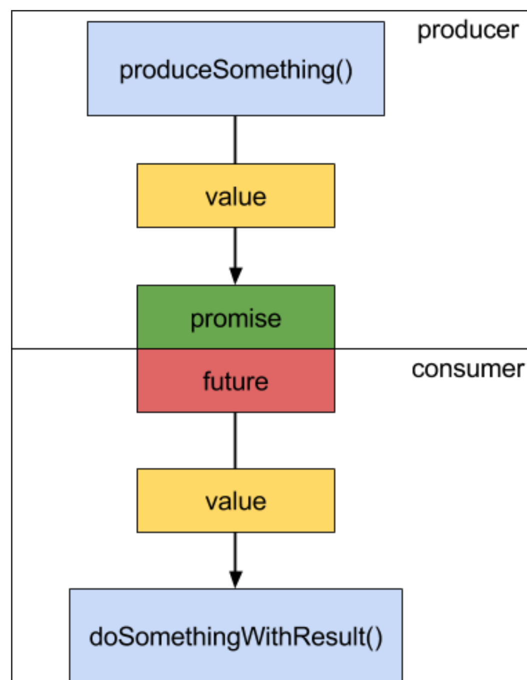


Figure 5.2: Code example from Listing 5.2 illustrated

5.1.3 Usage

There are several ways to control if everything has been executed correctly. All of the methods can be combined to make a comprehensive test method [9, pp. 104–108].

- The *Await* method is a blocking call that waits for a defined amount of time. If the future is not completed within the time-limit, it throws an exception.
- Success or Failure is a combination of checks, either as *future.isSuccess* or *future.isFailure*.
- *future.onComplete()* is a way to have the future itself rapport when it is completed.

There are different ways to combine futures as well.

- *firstCompletedOf(futures)* is a function that uses the first future that completes. The function is used when there is more than one call to different futures [9, pp. 109–110].
- You can use *future.zip* to combine two values into one tuple future with two successful futures in it [9, pp. 110–112].
- Use *For Comprehensions* to combine many futures into a future list [9, pp. 112–113].
- You can *fold* the values, which effectively combine two values or lists [9, pp. 113–115][46].

5.2 Load balancing in Akka

Here, we will be looking at two ways to load balance the workflow of the actors through routing [9, pp. 188–212]. We have discussed routing tactics before in Section 3.2.2, and we will discuss how Akka uses some of these tactics. We will use the Akka terminology *routers* and *routees* [47]. The router is the actor that arranges how the data is divided between its child actors, called routees. There are normally three reasons why messages are routed:

- Increase *performance* by dividing tasks that can be parallelized between actors.
- *Content of the received message* shows that the message should be sent to a specific routee.
- If the *state of the router* is set, then messages are sent to a certain group because a constraint was met/not met.

As illustrated in Figure 5.3, routing is a reaction to a input. The routers can be used locally and between multiple servers.

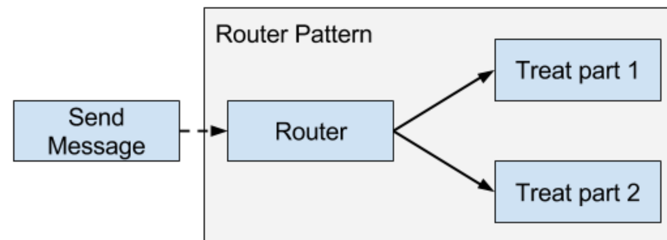


Figure 5.3: Router pattern

5.2.1 Router functionality

There are two router types, and both are illustrated in Figure 5.4. The *pool router* is responsible for managing and creating the routees. It is also responsible for removing the routees from the list when they terminate. The pool router requires less configuration than the group router, and is always the parent of the routee, making it the simplest setup. You configure the pool router by defining the number of instances of the routees and the logic, discussed later. The number of instances can be customised by turning on the *resizer* in the configuration file. An example of the resizer configuration can be seen in Listing 5.3 with accommodating comments that describe what the settings mean. As illustrated in Figure 5.5, if a routee fails, then the router will escalate the error handling to its supervisor, but the response from the supervisor will be given to the router, affecting all the routees. This behaviour can be changed by creating a non-default strategy for the router [9, pp. 198–199].

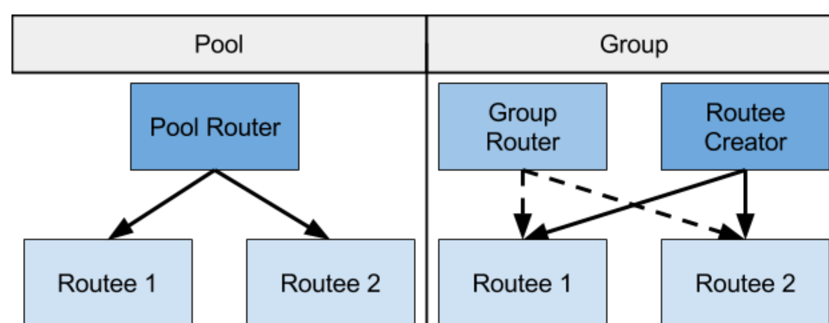


Figure 5.4: Router types

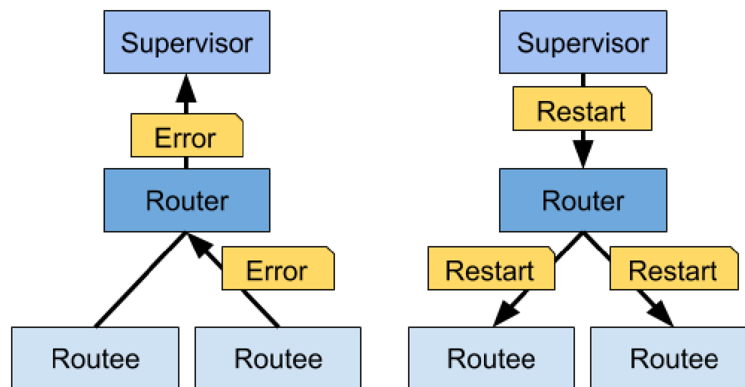


Figure 5.5: Router escalating problem to its supervisor

```
resizer {
  enabled = on

  /**The smallest and largest number of routees a router should have*/
  lower-bound = 1
  uppder-boud = 10

  /**How many messages should be in the routees mailbox for it to be
 defined under pressure. If it is 0, it is under pressure if it is
 processing a message*/
  pressure-threshold = 1

  /**How many percentages of the total amount of routees will be created
 when it ramps up. If the number of routees is always rounded up*/
  rampup-rate = 0.25

  /**The backhold threshold is the percentage of busy routees that
 triggers the back-off*/
  backoff-threshold = 0.3

  /**How many percentages of the total amount of routees that will be
 removed*/
  backoff-rate = 0.1

  /**The number of messages the router needs to receive for it to do
 another resize*/
  messages-per-resize = 10
  }
```

Listing 5.3: Resizer configuration example for a pool router

The other version is a *group router*. The group router will control the routees lifecycle and where they are initiated. The group router does not manage or watch routees. This

means that if a routee is terminated, the group router will continue to send messages to it since it might be available later on. Unlike pool routing where the routees need to be the children of the pool router, the group router is not the parent of the routees. The group router needs to find the actors through the *actor selector*. The actor selector is a method in the actor system that finds the actor children of the actor system. The management of the actors has to be implemented somewhere else in the system.

What is a Trait in Scala?

A Trait in Scala is like an interface in Java, but more liberal. Java has a separate class for an abstract class and interface, while a Trait in Scala is close to being both an abstract class and an interface. It is used as a recipe for the class [44, pp. 121–135].

Additional messages that can be sent to the router to maintain the group routees are:

- *getRouteees* makes the router reply with a routees message containing the routees that are currently used.
- *AddRoutee(routee: Routee)*, where you send a *routeeTrait*, adds the routee to the router. It is important to notice that it is a trait of a routee that is sent, not a routee.
- *RemoveRoutee(routee: Routee)* removes the specified routee from the router.

For a routee to be added to the router, it can use the *ActorSelectionRoutee(selection: ActorSelection)* method. The same routee instance type used to create the routee is needed to remove the routee. One cannot remove a routee without owning the instance type.

5.2.2 Task scheduling through router types

There are different router types within the routing functionality. These router types encompass logic that is used to divide tasks between processors. Here, it is the router which delivers tasks and expects results, and the routee is computing the task. The problem we face is that there can only be approximate knowledge about how long a process takes, calculated using available metrics.

There can also be problems with receiving the metrics if they take to long to send and receive. If it is a routee's responsibility to let the router know when it is available for work, then the routee sits idle, wasting processing time, while it waits for the response containing the task. But sending tasks ahead of time can backfire if the processing times of the tasks change. In Figure 5.6, we can see the router sending a task to routee 1 before it has finished the current task. The router expects the task to be finished before the other

tasks queued in routee 2. The tasks of routee 2 get processed before the original task in routee 1 is finished, and routee 2 becomes idle. How to fix this miscalculation is difficult, and will cost processing time. It can also become problematic if two tasks are supposed to be combined, and one finished task has to wait for the other task to be completed. These problems are not going to get a simple answer anytime soon because it is an example of the *Halting Problem* [48].

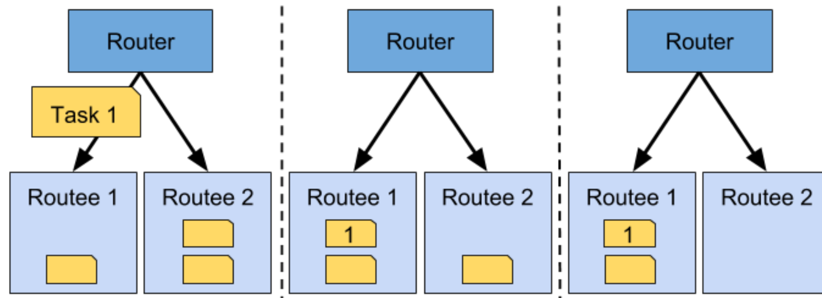


Figure 5.6: Task being processed too slowly on routee 1.

Akka have different tactics to handle the usage of router types. Some of these types can be used within both router functionalities, and some are exclusive. The two that can only be used with the pool router are:

- *BalancingPool* where the router distributes the messages to the idle routees. This is done by having one mailbox for all routees, and a special dispatcher for the routees.
- *SmallestMailboxPoolLogic* is where the router sends the message to the routee with the smallest mailbox.

The ones that are common between both functionalities are [49]:

- *RoundRobinRoutingLogic* where the router sends the messages sequentially to the routees, and repeats when all the routees have gotten one message.
- *RandomRoutingLogic* is when the router sends the messages to a randomly choosen routee.
- *BroadcastRoutingLogic* sends the received message to all routees.
- *ScatterGatherFirstCompletedRoutingLogic* where the message is sent to all the routees and the first routee to complete the task sends back the response to the router that uses that response.
- *TailChoppingRoutingLogic* which sends the message to a randomly routee, waits for a set time, and sends it to another randomly chosen routee. It waits for its first reply and sends it to the router.

- *ConsistentHashingRoutingLogic* where the hashes of the messages are sent to a specific routee.

5.3 Clustering

A singler computer is no longer enough to process the large amount of data available to modern applications. Today's distributes systems consist of multiple computers working together in *clusters*. The clusters allow applications to quickly scale the processing power and storage resources up and down according to need. While earlier clusters required special machines, current clusters contain standard, off-the-shelf machines.

A cluster in Akka is a collection of nodes joined together, where each node is a logical member of the cluster [50]. Every node can contain an actor system, but it does not have to [9, pp. 322-324]. In this section, we discuss nodes within a cluster and how they behave. In Figure 5.7, we can see a four-node cluster with four actor systems. All nodes within the same cluster have to have the same actor system in it, but can communicate with other clusters with different actor systems [51][52]. Clustered actor systems main features are load balancing as the number of nodes will increase and decrease depending on load, and fault-tolerance as there are more than one actor system to handle a task.

A cluster is set up for *at-most-once delivery*. This is a bit different than at-least-once-delivery discussed in Section 4.2.6. An at-most-once delivery message will either be delivered once or zero times, implying that messages might be lost [53]. We will discuss in depth the different roles, responsibilities, and states of the members in a cluster. Here is a short introduction to the different roles:

- *Seed role:* The node that creates and receives requests to join the cluster.

- *Master role:* The node that receives the jobs, creates the workers and supervises the workers. Within it is a *receptionist* that receives the jobs, and creates master nodes to handle the jobs.

- *Worker role:* The node that does the task given to it.

An actor system can be partitioned into separate subtree nodes, between the worker nodes and the master nodes. One can only access the top-level actors in the master node within the cluster. The workflow from when a cluster receive a task to when it completes it will be discussed later in Section 5.3.6 with more detailed illustrations of the hierarchy.

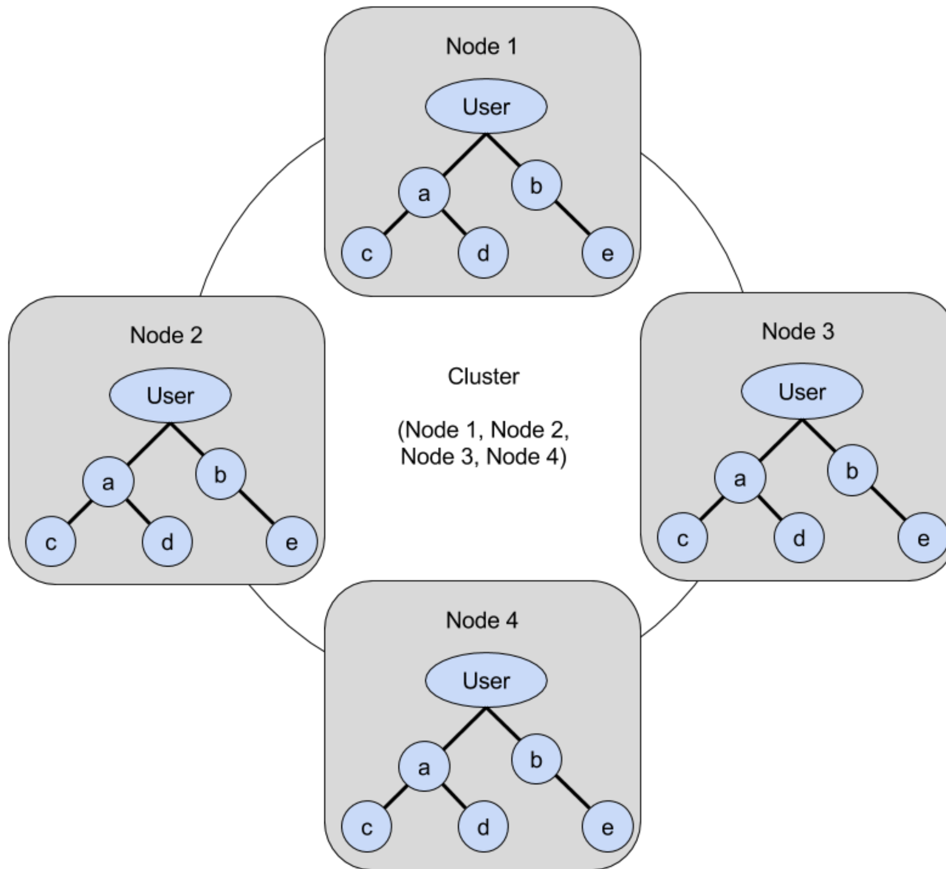


Figure 5.7: A cluster with four nodes, each containing an actor system.

5.3.1 Joining

The *seed* node must be created first to establish a cluster. A seed node creates its cluster by sending a *join* message to itself. Afterwards, other seed, worker, and master nodes can join the cluster. An illustration of the steps from initialisation to a running cluster can be seen in Figure 5.8. You will notice that when a node asks to join the cluster, the request is sent to all the seed nodes, and the first seed node to respond will be in control. There only need to be one seed node to be able to handle a request. For a new node to join other nodes in a cluster, it must have the same actor system name and use the same IP address as the other nodes, but have a different port number. To identify the different nodes, each has a *hostname* : *port* : *uid* tuple. The *leader* is a role a node can have, which decides if a member node is up or down, and manages cluster convergence and membership state transition. In our example, it is the first node [9, pp. 322–337].

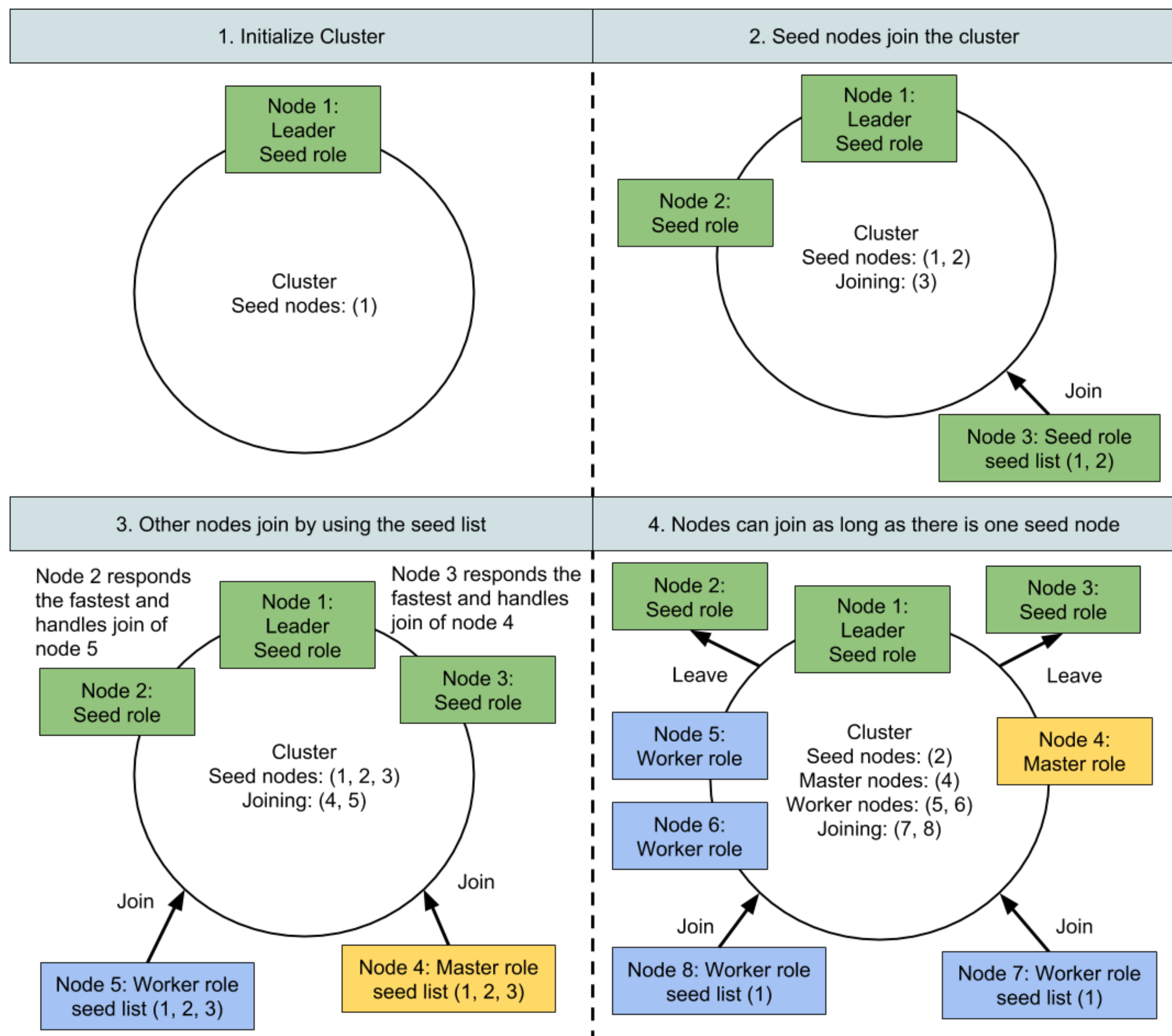


Figure 5.8: Cluster initialisation timelaps

5.3.2 Leaving

If a node is *leaving*, it will be marked by the leader node as leaving. The leader can mark itself as leaving. After a node is marked as leaving, the leader removes the node and shuts it down. If the leader is the one removed, then the next node in line will become the leader. In Figure 5.9, we can see an illustration of what happens when the leader node leaves. In our instance, seed node 2 takes over as the leader when the nodes notice that they cannot reach the leader node. After the node with its actor system has been removed, the actor system cannot re-join the cluster. For it to be able to re-join, it needs to terminate, be recreated, and given the same configurations and names as last time.

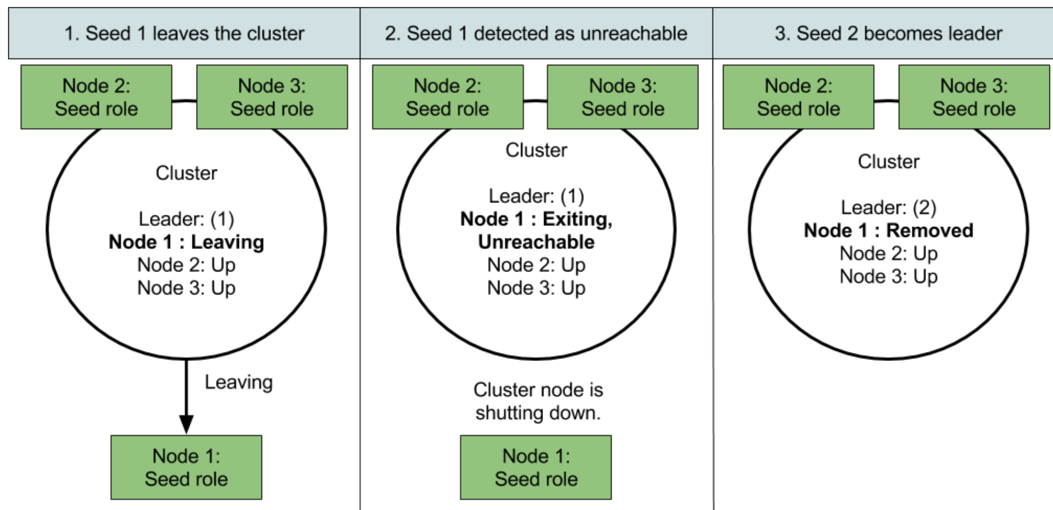


Figure 5.9: The leader leaves the cluster

5.3.3 Membership states

All nodes should converge to the same state. At the start of a membership lifecycle the node would be *joining* and set to *up*. This means that the node is up and running.

If the node is to be removed from the cluster, it is done in three steps. First, the node sets itself to the *leaving* state. Then the leader moves it to an *exiting* state, then *removed*.

A node is flagged as *unreachable* if it cannot be reached, meaning that all the nodes cannot converge on a single state and the leader cannot do an action. If this happens, the unreachable node must ether become *reachable* or marked as *down*. The node can be removed if it has been unreachable for a long time by enabling the *auto-down* configuration. This makes it possible for a new incarnation of the node to join the cluster without manual intervention.

There is also a possibility for a node to be *WeaklyUp*. If a node is unreachable and there cannot be a convergence, and a new node joins the cluster, the new node will be marked as WeaklyUp. When there is a convergence, the node is set to Up. The WeaklyUp members are only visible one the local network partition [54][9, pp. 325–337].

There are three available user actions:

- *join:* join a single node to a cluster.
- *leave:* get the node to leave the cluster gracefully.
- *down:* mark the node as down.

You can observe the different state changes and transitions in Figure 5.10.

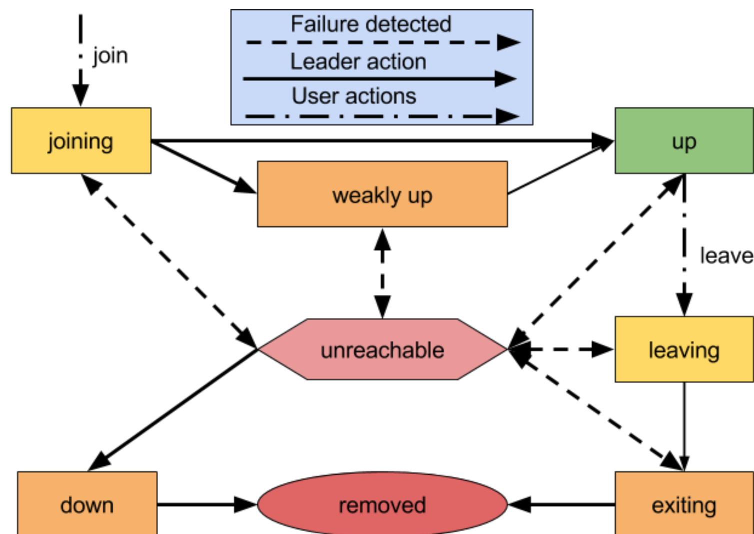


Figure 5.10: All transitions and states of a node

5.3.4 Gossip protocol

The way that the nodes share their membership is through a *gossip protocol*. The nodes communicate with random nodes once every second in an epidemic fashion, where eventually all nodes are reached. A node may be communicated with twice, which make it difficult to determine what state is the newest one.

Akka realises a *Conflict-free replicated data type*, also called *Convergent and Commutative Replicated Data Type* (CRDT) [55]. The membership state is a specialised CRDT, which implies that when concurrent changes happens on different nodes, the node states will always converge to the same state. Akka's CRDT uses a *vector clock* as a versioning variable. It generates a partial ordering of the events in the cluster. Every update to the cluster state also updates the vector clock. The gossip state representation code can be seen in Listing 5.4 [56].

```
case class Gossip(
  members: immutable.SortedSet[Member],
  overview: GossipOverview,
  version: VectorClock)
case class GossipOverview(
  seen: Set[UniqueAddress],
  reachability: Reachability)
```

Listing 5.4: Gossip state representation

A node converges to the newest state when the node communicates with other nodes which have seen the newest state. The nodes that have observed the newest state constitute the *seen set*, and when all the nodes in the cluster are included in the seen set, then there

is convergence among the nodes. If less than half of the nodes are within the seen set, the cluster gossip increases from 1 to 3 times every second. Although the gossip happens at random, it is biased towards nodes who probably have not seen the new state. If the cluster is in a converged state, it goes from biased gossip messages to small gossip messages containing the version variable. The bias probability is configurable, and is automatically reduced if the number of nodes reaches 400 nodes. It also drops messages that has been in the mailbox queue for too long.

When a node receives a gossip status or gossip state, it can use the vector clock to determine if it has a newer version and state. If it has, it will send the newer state back, as illustrated in Figure 5.11. If it has an outdated version of the state, it will send its own version to the node that gossiped, then the newest state will be sent back, illustrated in Figure 5.12. If there are conflicting gossip versions, the messages are merged and sent to the original sender [57][9, p. 333][56].

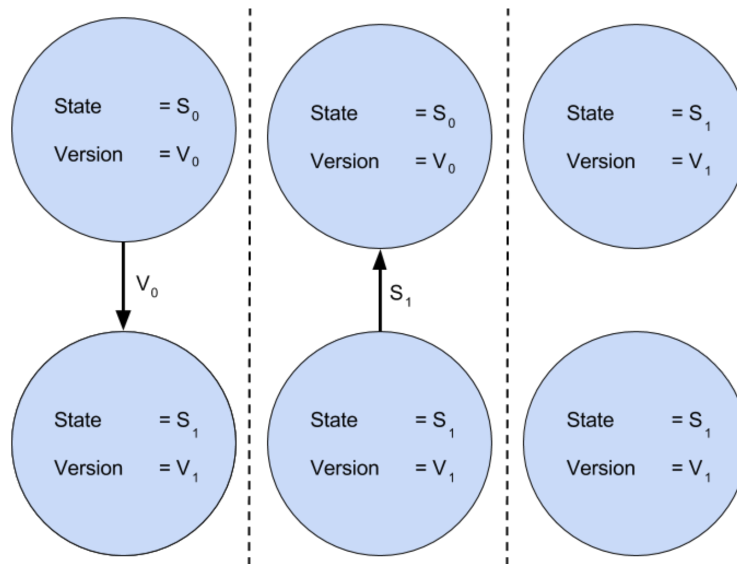


Figure 5.11: By receiving an older version of the state the node with the newer state sends the newer version to the gossiper.

5.3.5 Failure detection

Akka has implemented a version of *The ϕ Accrual Failure Detector* [58]. From the paper abstract:

"Instead of providing information of a binary nature (trust vs. suspect), accrual failure detectors output a suspicion level on a continuous scale. The principal merit of this approach is that it favours a nearly complete decoupling between application requirements and the monitoring of the environment."

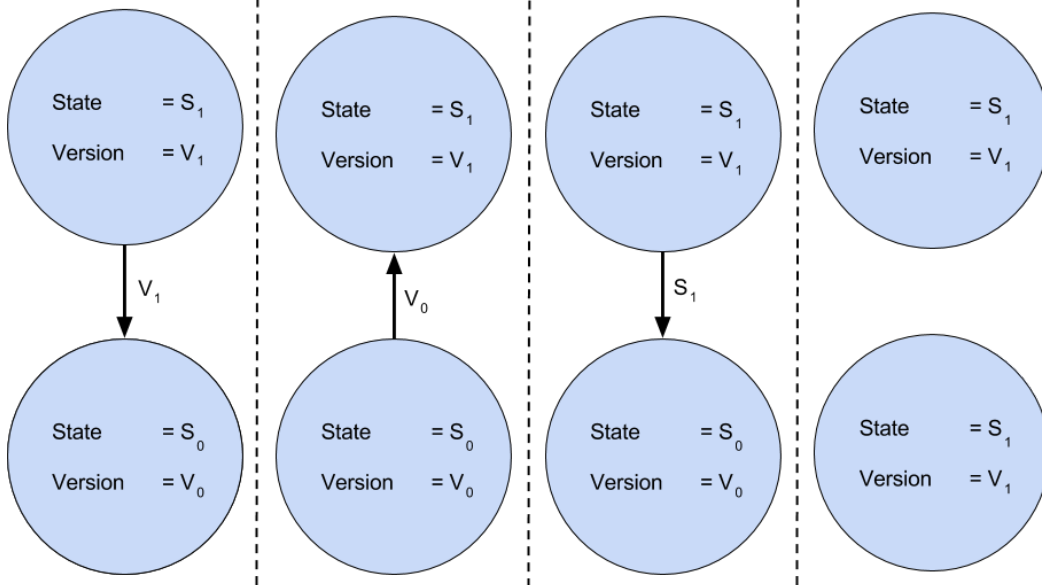


Figure 5.12: When the gossiper has a newer version of the state than the receiving node, the receiver sends back its version and obtains the new state.

In Akka, ϕ is a suspicion level that dynamically adjusts to reflect the current network conditions. To see if a node is reachable, the nodes send out *heartbeats* to signal that they are reachable. The arrival times are interpreted by the ϕ accrual. The value itself is calculated as:

$$phi = -log10(1 - F(timesSinceLastHeartbeat))$$

.

F is the cumulative distribution function of the normal distribution where the standard deviation and mean are calculated from historic heartbeat inter-arrival times. The higher the value of the function, the higher the likelihood that there is a problem needing attention. The potential problem is only investigated if the value of the function is higher than some threshold. It is possible to configure the threshold. If the value is too low, it is prone to generate a lot of false positives, and if it is too high, it will be slow to respond to real problems. You can set acceptable heartbeat pauses, where you can register no heartbeat response for a set time before ϕ increases [9, p. 336][59].

5.3.6 Working with a cluster

A simplified illustration of a cluster job from start to finish can be seen in Figure 5.13. In our instance, we use a *BroadcastPoolRouterConfig* configuration, explained in Section 5.2.2. The master uses pool and group routers as described in section 5.2.1, only here they

will be created within a cluster configuration. You can still use the same logic and the same routing methods with groups and pools. When the master node receives a *StartJob* message from the job receptionist, it starts in an idle state, and when the job is sent to the worker nodes, it is set to a *Working* state to handle the responses from the workers.

After a work message is sent, the worker node goes from idle and sends an *enlist* message to the master node to signal that it can do work, and *nextTask* to ask for the task that the master node has to give. The enlist message contains its own ActorRef so that the master knows the particular worker's reference. The worker nodes are supervised in case one or more workers crash. After a worker node enlists, it goes into the *Enlisted* state. The enlisted worker sets a *RecievedTimeout*, that is the time window where the worker expects to receive a task. If it does not receive a task within the window, it will ask again. The same is done by the master node if no worker node enlists, only then the master node stops itself. If the master dies, the workers stop themselves. When there is no more work to be done by the worker nodes, the master node sends a *WorkLoadDepleted* message to the workers.

In our example, the workers are the ones asking for the work, not the master delivering the work hoping that the workers are free. The worker asks, receives, processes, sends back, and asks for the next task. The other way of doing this has been discussed in the previous section, but you can also use *Adaptive Load Balancing Logic*. It uses the cluster metrics to determine what worker is best suited to receive the next task. We will not go into how this works, but the reader can look into the details [60].

When the master node receives task results from every worker, it merges the results and kills the workers. The merged result is sent to the job receptionist and the receptionist kills the job master.

All of the nodes in the hierarchy have a *StoppingStrategy*. An illustration is given in Figure 5.14. The master stops if its receive timer reaches zero, which stops the worker, and the worker node sends a terminate message back to the master. If the job receptionist notices that the master has died, it tries to recreate the master. If it receives the terminate message, it checks to see if the task is completed. If it is not completed, it resends to itself the original job request, and the whole cycle starts over again.

5.3.7 Distributed publish subscribe

To create a cluster with a publish-subscribe setup, one can use the *DistributedPubSubMediator* (DPSM). The DPSM is an actor that is started on all the nodes or the ones with a defined role. The DPSM manages a registry which contains actor references (ActorRef). The actors on nodes with a WeaklyUp status will get published messages if they subscribe to nodes on the same network. The changes on the nodes are eventually consistent by using the gossip protocol [61].

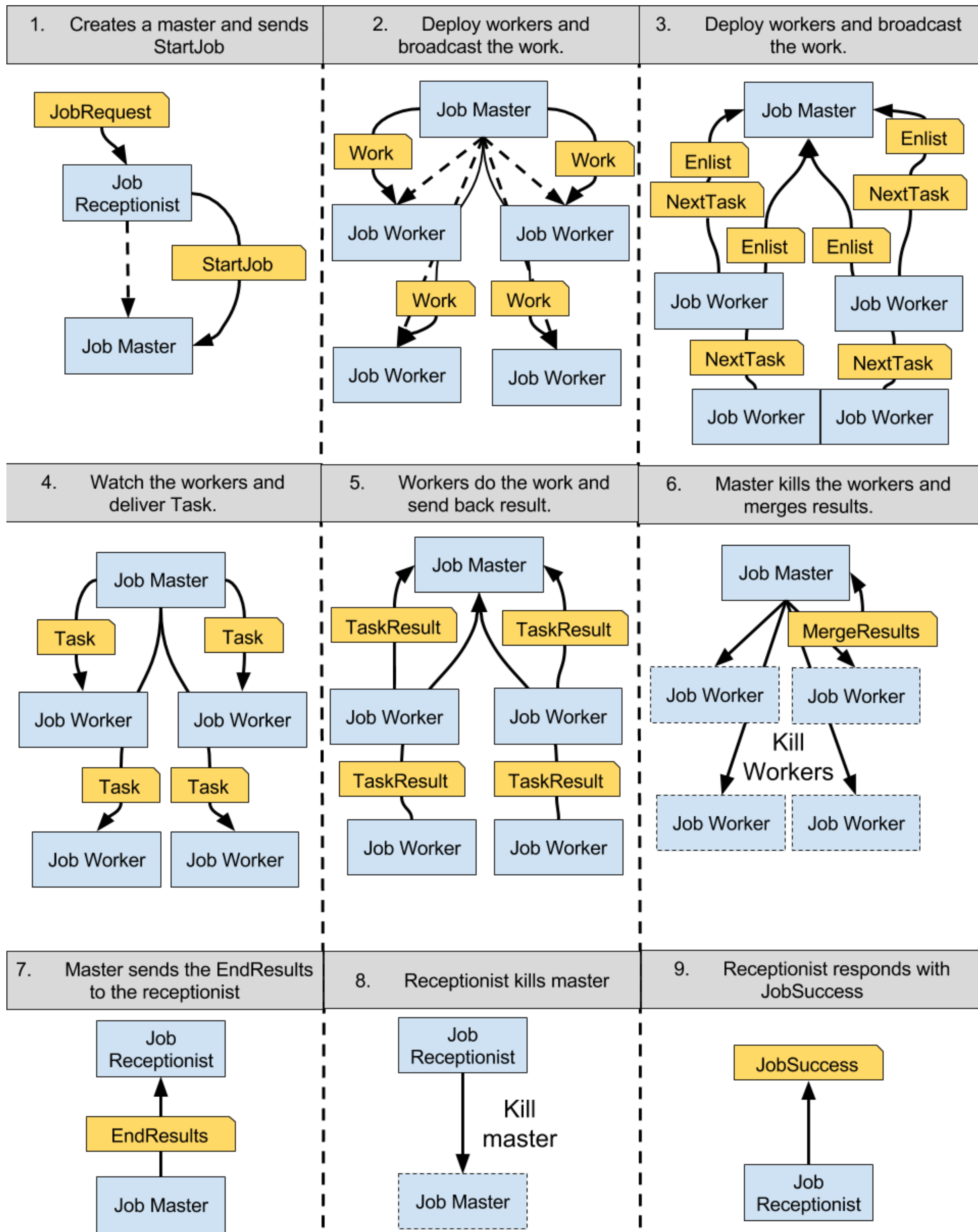


Figure 5.13: Job processing with a cluster

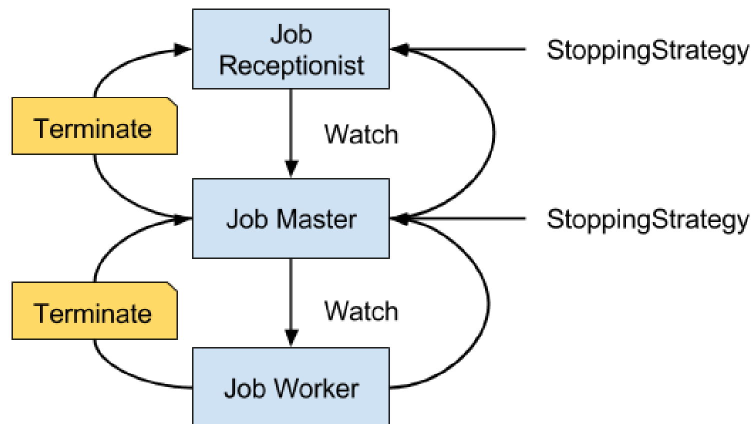


Figure 5.14: Cluster supervision hierarchy

Publish

The publish method, depicted in Figure 5.15 is the kind we have discussed earlier in Section 3.2.2 where actors subscribe to changes. An actor can subscribe to a topic by using a *Subscribe("topic name", self)* method. One can subscribe and unsubscribe, which will be followed by a *SubscribeAck* and *UnsubscribeAck*. The actor sends messages by using *Publish*. To remove actors from the registry, they have to unsubscribe or be terminated. The message is only sent to one node with matching topic over the wire and then replicated locally.

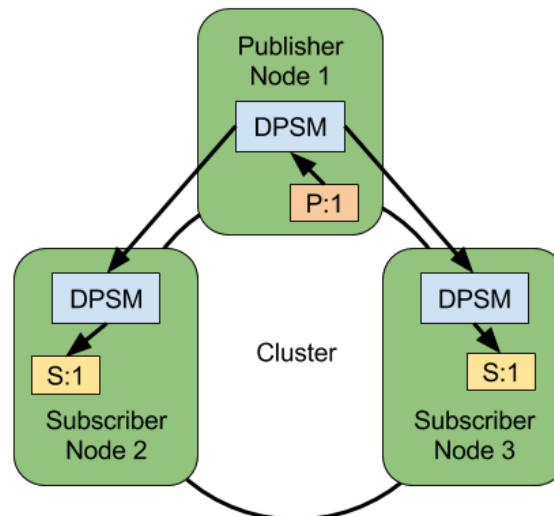


Figure 5.15: A cluster setup where one publisher sends to two subscribers

One can also use *group Id* to spread the messages differently. The group is identified with group Id, and if the message is published to a topic with *sendOneMessageToEach-*

Group (SOMTEG) set to true, it will be sent to a actor within each group. In Figure 5.16, we can see at the top where everyone has the same group name. The publisher will use the supplied routing logic, which is set to random as default, and send to only one of the subscribers. In the middle of Figure 5.16, we can see that when a publisher publishes a topic to groups with different IDs. As Group ID is a part of the topic identifier, this means that when SOMTEG is set to false, the publisher will not send to the subscribers that have identified themselves with group ID. This is illustrated at the bottom of Figure 5.16 [62].

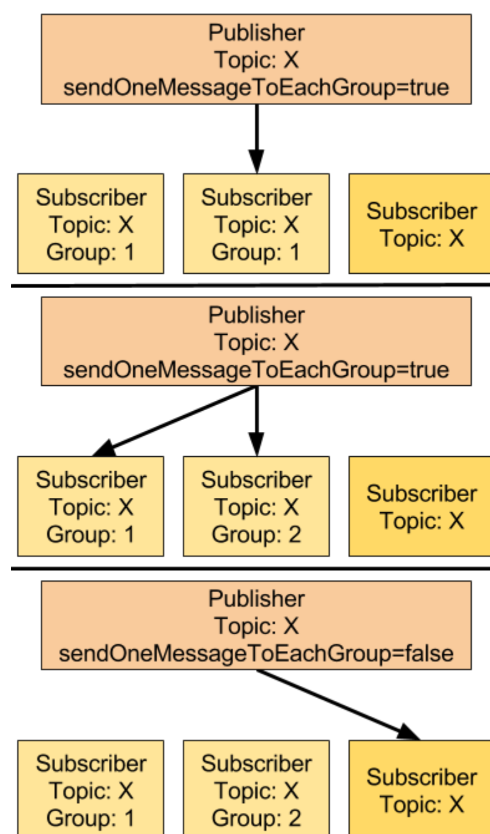


Figure 5.16: Message sent to either all or one subscriber, or only to the ones that do not have SOMTEG set to true

Send

Send is a point-to-point message that closely resembles the group scenario described earlier. It is a message that will be delivered to one receptor that has a matching path within an actor system. If there is no actor at the place that the sender is trying to send the message, then no message is handled. If there are several paths, then it will be chosen by using the supplied router logic.

An actor registers itself by putting its ActorRef in the DPSM using *Put(self)*. The DPSM needs to be the local DPSM. The actor path is the key to find the actor, as each actor system has a unique path for each actor. To send a message, one transmits a message to the DPSM with the path of the actor. If one wants to remove the actor from the DPSM, it can be removed by either using *Remove* or when the actor is terminated.

It is also possible to send to all actors that are registered in DPSM by using *SendToAll*. This means that a message is sent to every actor that has the matching path. This is normally used in redundant nodes, where there are replicas of the same actor on different nodes. There can only exist one such actor on every node.

5.4 Persistence

The Akka *persistence* library is well suited to be used with the event sourcing pattern. You can make your own event store, or use one that has already been produced.

What is a Case Class and Sealed Class in Scala?

A case class is an immutable class. If you declare a case class, many things happen automatically including:

- Every constructor parameter becomes a *val* unless specified otherwise. A val is an immutable variable, unlike var.
- The methods *hashCode*, *equals*, *toString*, and *copy* are generated unless they are provided.

A sealed class is when classes that extend the sealed class need to be within the same file [44, p. 209][44, p. 205].

5.4.1 Persistent actor

A *persistent actor*, illustrated in Figure 5.17, can recover from events or process commands by having two states with the respective possible actions in each. Both commands and events define their sealed traits, and the persistent actor has complementary methods for accepting those operations. The action of sending the event from the actor to the event store is formulated as to *persist* an event. A persistent actor requires a unique *persistent ID* which is automatically passed to the event store when the actor persists an event. This interaction is illustrated in Figure 5.18. In the figure, we can see both commands and events, discussed in Section 3.2.3 and CQRS discussed in Section 3.3.2. The important thing to notice is that the actor does not update state unless the journal has successfully

sent back the event, but this can be overwritten. The other important thing to notice is that *receiveRecover*(), the method used when recovering, is also used when you persist the events. This is to simplify the code and to confirm that the events persisted are correct [9, pp. 359–363][63].

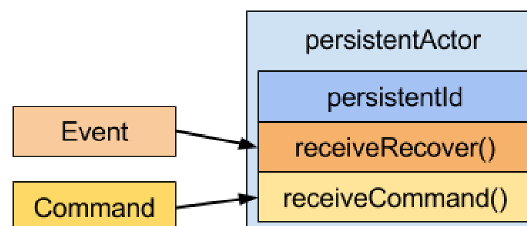


Figure 5.17: Persistent actor with recovery method and command method

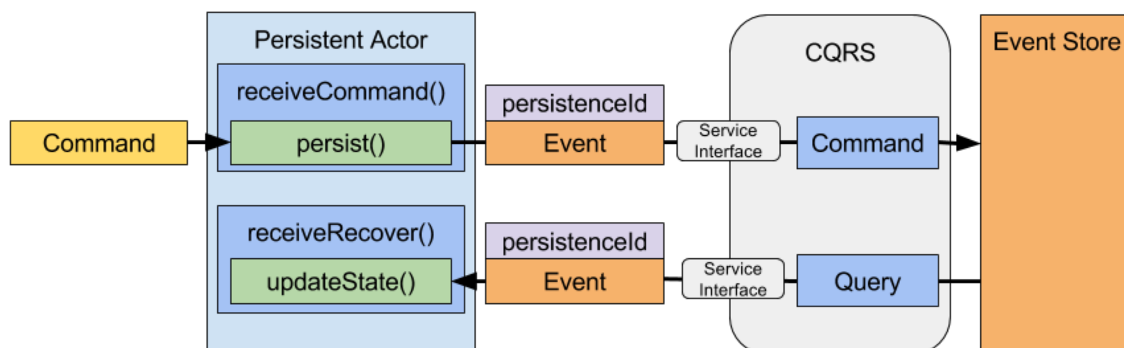


Figure 5.18: An persistent actor persisting the received command

5.4.2 Enabling at-least-once delivery

It is possible to make the persistent actors perform with an at-least-once delivery (ALOD), discussed in Section 4.2.6, by importing the *AtLeastOnceDelivery* trait, but the actor needs to be configured. If you use ALOD, then the message order will not be preserved because of the possibility of resends, and if a crash happens, the messages that were not delivered during the downtime is delivered to the new actor.

In Figure 5.19, we can see the communication protocol. Before using *deliver()* to deliver, it is good practice to persist. After the message is delivered, the actor who received the message sends back a confirmation message. This should also be persisted, before confirming the delivery with *confirmDelivery()*. The messages contain a deliveryId, making it possible to correlate the deliver and confirm events. The ID is a strictly monotonically increasing sequence of numbers that is created with the *deliveryIdToMessage()* function. This can be combined with snapshots, discussed later in this section. There is a *redeliverInterval()* method that can be configured, and *redeliveryBurstLimit()* defines the maximum

number of messages that will be sent at each redelivery burst. If the number of redeliveries exceeds a certain number, a *UnconfirmedWarning* message will be sent. The amount of retries allowed can be configured before the warning appears can be configured. Although the re-sends will continue, one can choose to cancel the re-sends by calling confirmDelivery. Lastly, the maximum amount of unconfirmed messages can also be configured [64].

5.4.3 Snapshots

A *snapshot* is a way to recover events faster by not restoring the whole history of the event store. In an application, there is often a natural point where a transaction is completed. This can be when a customer has logged off, or maybe just when the interaction within a bounded context is achieved. If this point can be identified, then it would be a natural place to take a snapshot.

If we take a movie example, as in Listing 5.5, then the movie can be in certain states. The most natural time to save a snapshot will be when the movie is finished, as the other states need to be remembered in case the movie should be resumed. If the movie is finished, and the user wants to watch it again, then the next action is to start over. If the actor is going to recover, then there would be no good reason to run through all the pauses and exits if only the last one counts. To save a snapshot you can use the *saveSnapshot*() method, and have *SaveSnapshotSuccess*() and *SaveSnapshotFailure*() return depending on the success of the snapshot save. To have *receiveRecovery*() consider the snapshot, the *SnapshotOffer*() method is used for getting the latest snapshot. The method can also be customised to not choosing the latest snapshot [9, pp. 365–369][65].

```
1 private val updateState: (Event => Unit) = {
2   case Start(movie)    => movies = movies.start(movie)
3   case Pause(movie)    => movies = movies.pause(movie)
4   case Exit(movie)     => movies = movies.exit(movie)
5   case Finish(movie)   => movies = movies.finish(movie)
6 }
```

Listing 5.5: Movie event example

5.4.4 Cluster persistence

We will discuss two of the main cluster configurations that can be combined with persistence in Akka. They are *Cluster Singleton* and *Cluster Sharding*. The cluster nodes can be configured to automatically shut down a node if it is unresponsive. One should not use this automatic downing functionality with these methods as they can lead to problems with creating more node or shard instances than intended.

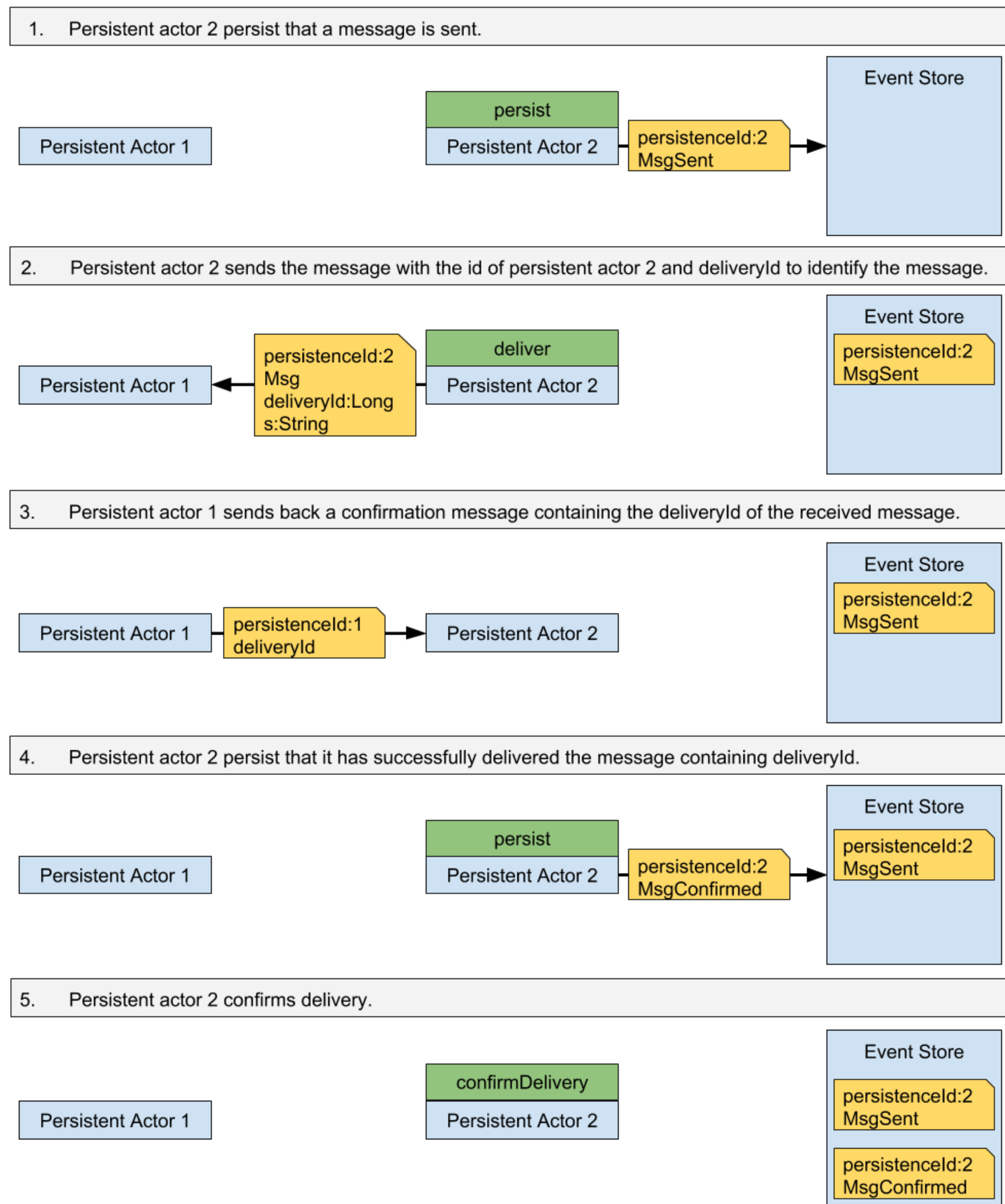


Figure 5.19: At-least-once delivery with persistent actors

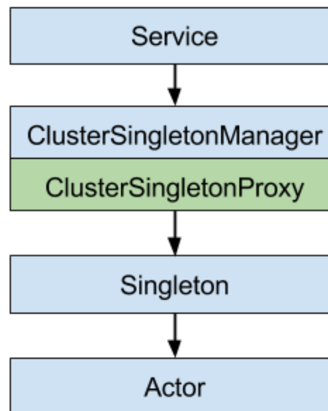


Figure 5.20: The cluster singleton hierarchy

Cluster singleton

A cluster singleton is to have a single instance of an actor on a node of a cluster. We will now discuss the different parts of the cluster singleton hierarchy illustrated in Figure 5.20.

The *ClusterSingletonManager* (CSM) actor manages the number of singletons in the system. The CSM will be created either on all the nodes or on those that have a specified role, as illustrated 5.21. So if a node crashes, is shut down, or if there is a network failure the CSM is responsible for removing the old node and creating a new singleton. The CSM creates the actual singleton actor on the oldest node in the cluster by creating a child actor. In both failure and when the actor changes places in the cluster, there will be a period without any singletons in the cluster. This is to avoid the cluster having two singletons in the cluster. A singleton is communicated with through the *ClusterSingletonProxy* which will rout all the messages to the correct node the singleton instance is placed by keeping track of the oldest node and the ActorRef of the singleton. The singleton will not run on a node with the status WeaklyUp and the messages sent may be lost. The singleton is a PersistentActor and stores events of the actors that have been created. This is important in case the actor crashes, or if the singleton should be relocated [66][9, pp. 380–383].

Cluster sharding

To *shard* a cluster is to run the actors within different nodes in the cluster, where the cluster can be divided across different machines. The Hierarchy can be seen in Figure 5.22. Specifically, a shard is groups of actors with identifiers, which in this case is *persistenceId*. These actors will be called entities. A command that is received contains a unique *ShardId* to identify the shard, and a unique *EntityId* to identify the entity.

Another illustration can be seen in Figure 5.23. Here we can see the *ShardingCoornator*

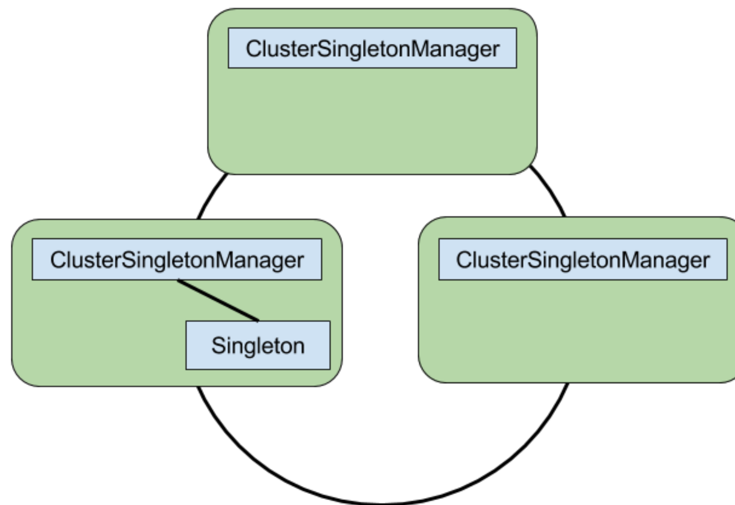


Figure 5.21: One singleton actor between three nodes, managed by the ClusterSingleton-Manager

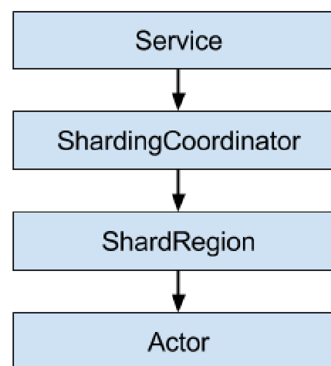


Figure 5.22: Cluster sharding hierarchy

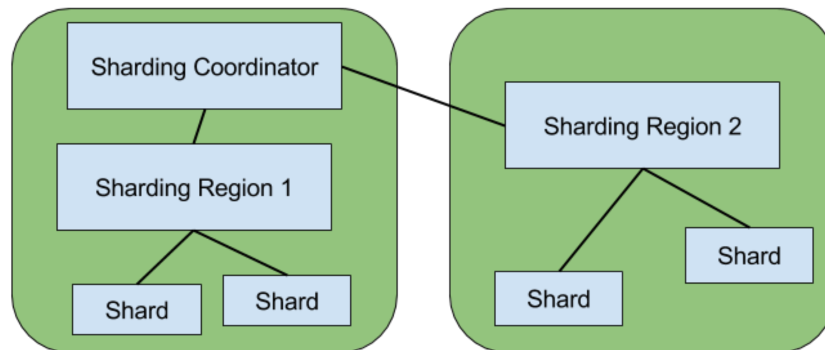


Figure 5.23: Shards in a cluster

is the actor that has control over the *ShardRegions*. Every node or node with a role has a Shard Region, and the Sharding Coordinator runs as a singleton in one node.

Sharding is a good choice if you want to be able to communicate with an entity without worrying about where the entity is located. There are two main scenarios of how the shards are created. The first is illustrated in Figure 5.24, where the request for Shard 1 is sent to the right region, but the shard is not created yet. After the shard is created, then the Region Shard can handle the incoming messages to Shard 1 with no communication necessary with the Shard Coordinator. The other scenario is when the communication is with a wrong Shard Region. This scenario is illustrated in Figure 5.25.

5.5 Summary

In this Chapter, we have discussed how to achieve asynchronous communication using futures and promises and different methods to load balance with routers. Later, we discussed how clustering functions in Akka and the different methods used by Akka to manage a cluster. Lastly, we discussed how Akka achieves a more resilient distributed system by using persistence actors, making it possible to construct a system using event sourcing and other useful methods.

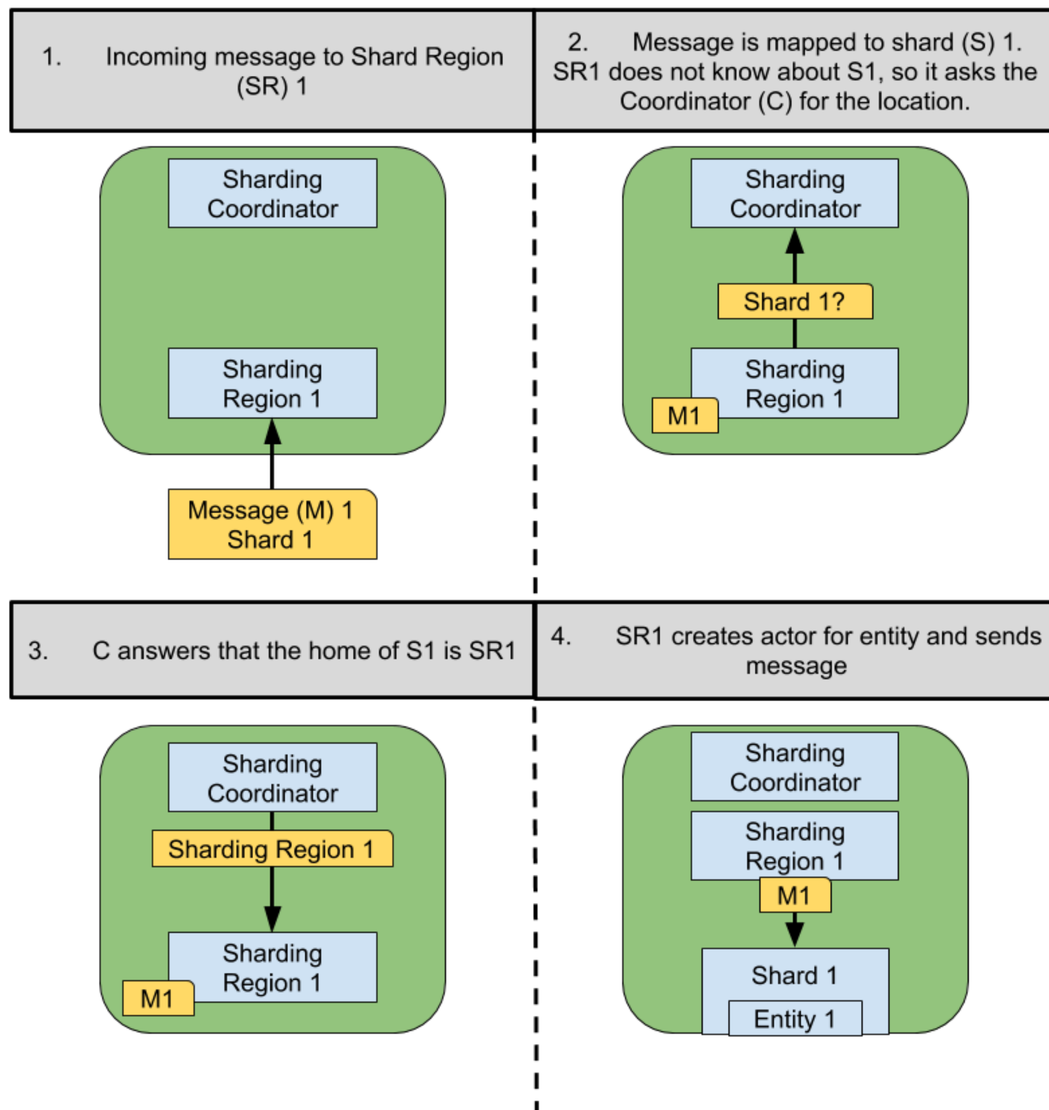


Figure 5.24: The correct shard region is communicated with, and the shard region creates a Shard for its entity

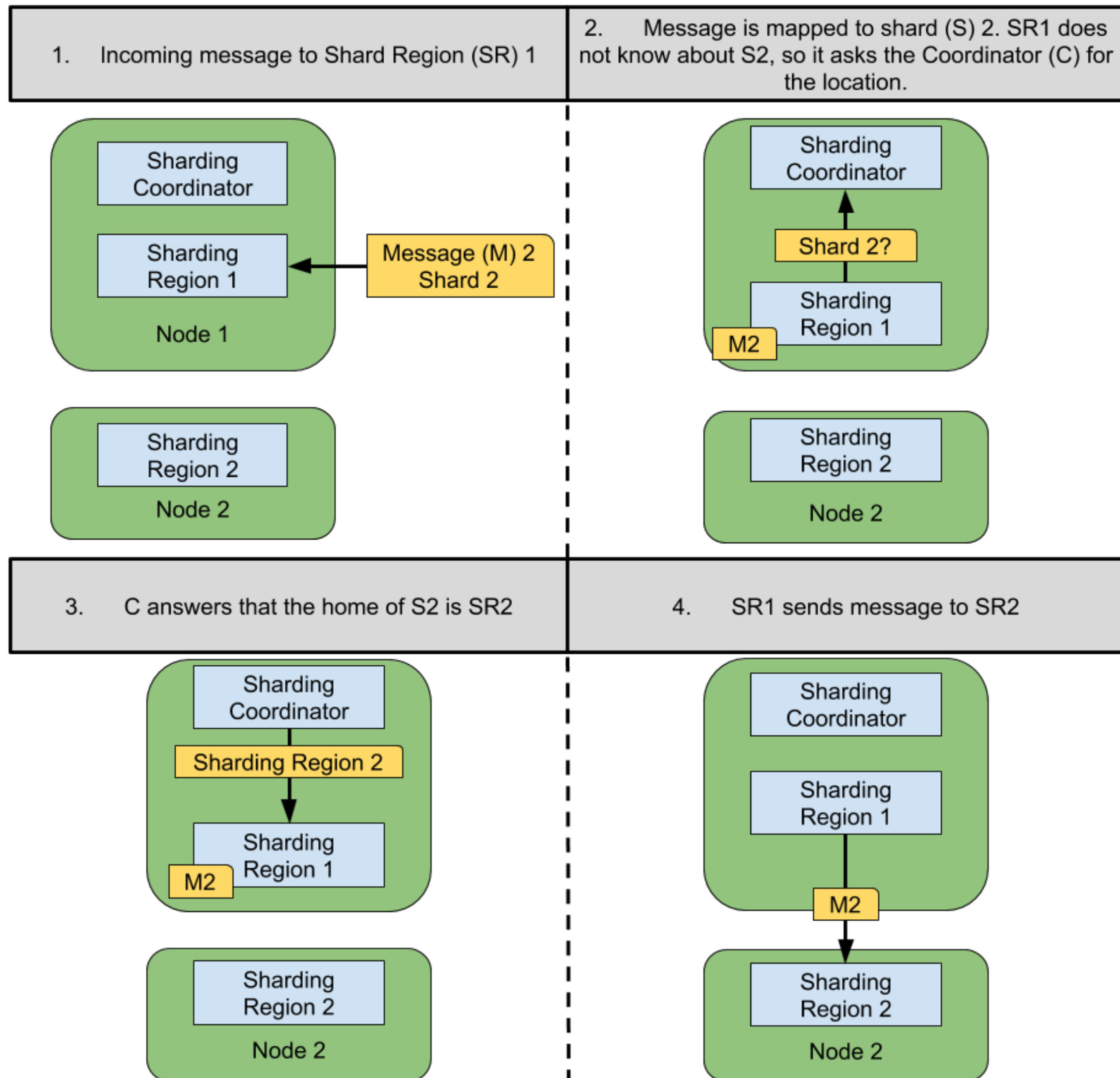


Figure 5.25: The incorrect shard region is communicated with, and the shard region sends the message to the correct shard region where the same as the previous example happens again.

Chapter 6

Analysis and Discussion

6.1 Summary and discussion

This chapter summarises the thesis and discusses the challenges of creating a distributed software system. It then suggests interesting topics for future work.

6.1.1 Summary

In Chapter 2, we discussed the terminology that is used in the thesis but might not be understood by a person unfamiliar with distributed microservice systems. Chapter 3 discussed the attributes that a microservice system should have and how the attributes function. Later in Chapter 4, we delved into many terms used within domain-driven design, what they mean, and how to design important aspects of a system. Lastly in Chapter 5, we took a closer look at Akka and how it implements the concepts discussed in the previous chapters.

6.1.2 Discussion

The first problem that will occur to a designer of a distributed system is to understand why and how to carry out the design. Although how to make distributed systems is well researched and understood by the large software companies, there seems to be a lack of explanations directed towards programmers who want to learn the methodology. This thesis is one attempt at creating a starting point for programmers, but the overwhelming information one needs to understand what to do from start to finish will still confuse many and make it hard to learn.

If you put the system in the cloud, then the system needs to be treated differently.

Some of the cluster settings in Akka will need to be specified to get all the benefit that Akka can give. Then there is the problem of deciding where the data within the system should be. If one has a local partition and a cloud partition one has to be careful not to break laws if one where to put something in the cloud. This gets complicated if one is a multi-national company, where different tax-laws need to be followed.

There is also the problem of refactoring a system after it is created. This is partially why domain-driven design is mentioned in this thesis because designing the system well is important to be able to separate the services sensibly. Maintaining a clear design from the start will help the speed of development.

It is essential to design a distributed system that tolerates inevitable failures. One of the many microservices in a system will fail sooner or later. The communication network between the services will also fail. Failure must not be allowed to propagate all over the system and cause a systemic error. Detecting errors early, mitigating the adverse effects, and strengthening the system is necessary to maintain a robust system over time [4]. If one uses Akka, one can create a supervision tree to handle local errors.

Resources can be distributed more dynamically when there is a varied amount of users. Instead of having a system at a fixed size, one can adapt to a larger system if the number of users spikes and reduce it when needed.

Compared to a monolith, it becomes easier to implement new versions of the services that have been deployed since the services are isolated and can be replaced without taking the whole system down. If the new version is not working as planned, then it becomes easy to go back to the previous version.

6.2 Future work

6.2.1 Distributed systems in the cloud

While the thesis did not discuss the cloud platform that an actor system will most likely run on, there are many interesting problems that will appear when one run a microservice system in a cloud. Instead of developing a service discovery service, one can use the Amazon Web Services (AWS) load-balancer. Amazon has launched Lambda [67] for server-less computing. Lambda takes care of the whole orchestration in the cloud, leaving the programmer with a lot less responsibility and control. If we ignore vendor lock-in, it would be interesting to look at the positive and negative effects of server-less computing.

6.2.2 Cloud security

If one knows how to create a cloud application, it would be interesting to study the different attack vectors within a cloud. As an example, in the service discoverability, how difficult would it be to introduce a malicious service, and introducing it to the service registry? Is there a common understanding of this across the different platforms? Is it possible to intercept packages and replace them? Where are the keys saved in the cloud, and can they be intercepted? Should there be certificates introduced to the system? Can you gain access to the event store and introduce malicious events to the event store that will be processed when the system recovers? This would be a big problem since the event store is immutable, so if a malicious event is in the event store, it stays there. And if event store injection where possible, would using snapshots be an acceptable mitigation tactic?

6.2.3 Distributed Data and Chaos Engineering

The two topics I did not have time to cover but would be the most sensible next step to discuss is Chaos Engineering and Distributed Data. Distributed Data is a package in Akka that uses CRDT to share data between the nodes in a cluster. The data is accessed by using a key-value store like API, and the data entries are spread to all nodes similarly to the discussion in Section 5.3.4 [68]. The limits of using this method would be interesting to explore, and if it could be used to replace the event store. Chaos Engineering is the design of a system that embraces failure and measures different metrics to understand how to maintain robustness to incidents in a changing environment. If the distributed system is resilient enough to handle failures that are bound to appear, then it is possible to test the system by introducing failures at the different system levels or introducing new features. It would be very informative to understand either how to use the services that Netflix have made, or how one can make similar services [69].

6.2.4 Different approaches

This thesis was a build up from the qualities and designs desired if one is to achieve a robust distributed system. But it is important to emphasise that Akka is a middle ground approach, with a toolkit that can be used with already well-established languages. There are languages that are using the same actor architecture as we have discussed in this thesis, including Elixir, Erlang, and Pony [70]. It might take more time to program with these languages as you need to learn more than just Akka. But you can also create a microservice system with most other languages by using tools found at the Cloud Native Computer Foundation [71]. If one wants to design the software as discussed in Chapter 4 one can read Implementing Domain-Driven Design [72].

Bibliography

[1] Finite State Machine in Akka.
http://doc.akka.io/docs/akka/current/scala/fsm.html, October 2017.

[2] Jonas Bonér. *Reactive Microsystems*. O'Reilly Media, first edition, August 2017.

[3] Jonas Bonér. *Reactive Microservice Architecture*. O'Reilly Media, first edition, March 2016.

[4] Kjell Jørgen Hole. *Anti-Fragile ICT Systems*. Springer, first edition, 2016.

[5] Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. PhD thesis, Cornell University, August 2010.

[6] Erlang. http://www.erlang.org, November 2017.

[7] Elixir. https://elixir-lang.org, November 2017.

[8] Akka. https://akka.io, October 2017.

[9] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka In Action*. Manning Publications, Shelter Island, NY, first edition, September 2016.

[10] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. Addison-Wesley, first edition, August 2015.

[11] Akka Actors.
https://doc.akka.io/docs/akka/2.5/scala/actors.html, November 2017.

[12] Chris Richardson. Pattern: Service registry,
http://microservices.io/patterns/service-registry.html, October 2017.

[13] Chris Richardson. Service discovery in a microservice architecture,
https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture, October 2017.

[14] Chris Richardson with Floyd Smith. *Microservices From Design to Deployment.* NG-iNX, first edition, 2016.

[15] Martin Fowler. Inversion of control,
https://martinfowler.com/bliki/inversionofcontrol.html, October 2017.

[16] Chris Richardson. Client-side discovery,
http://microservices.io/patterns/client-side-discovery.html, October 2017.

[17] Chris Richardson. Server-side discovery,
http://microservices.io/patterns/server-side-discovery.html, October 2017.

[18] Chris Richardson. Building microservices: Using an api gateway,
https://www.nginx.com/blog/building-microservices-using-an-api-gateway, October 2017.

[19] Kevin Webber. Modelling reactive systems with event storming and domain-driven design,
https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, October 2017.

[20] Margaret Rouse. Graceful degradation,
http://searchnetworking.techtarget.com/definition/graceful-degradation, October 2017.

[21] Chris Richardson. Pattern: Event sourcing,
http://microservices.io/patterns/data/event-sourcing.html, October 2017.

[22] Martin Fowler. Event sourcing,
https://martinfowler.com/eaadev/eventsourcing.html, October 2017.

[23] Chris Richardson. Pattern: Command query responsibility segregation (cqrs),
http://microservices.io/patterns/data/cqrs.html, October 2017.

[24] Martin Fowler. Cqrs,
https://martinfowler.com/bliki/cqrs.html, October 2017.

[25] William Wong. What's the difference between containers and virtual machines?,
http://www.electronicdesign.com/dev-tools/what-s-difference-between-containers-and-virtual-machines, November 2017.

[26] Mike Coleman. Containers and vms together,
https://blog.docker.com/2016/04/containers-and-vms-together, August 2016.

[27] Adrian Chifor. Container orchestration with kubernetes: An overview, https://medium.com/onfido-tech/container-orchestration-with-kubernetes-an-overview-da1d39ff2f91, March 2017.

[28] Docker how nodes work. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes, February 2018.

[29] Tamara Scott. Kubernetes vs. docker: Comparing containerization platforms, http://technologyadvice.com/blog/information-technology/kubernetes-vs-docker, March 2017.

[30] What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes, February 2018.

[31] Nancy Lynch Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[32] Coda Hale. You can't sacrifice partition tolerance, https://codahale.com/you-cant-sacrifice-partition-tolerance, October 2010.

[33] Martin Kleppmann. Please stop calling databases cp or ap, https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html, May 2015.

[34] Julian Browne. Brewer's cap theorem, http://www.julianbrowne.com/article/brewers-cap-theorem, January 2009.

[35] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, February 2012.

[36] Vaughn Vernon. *Domain-Driven Design Distilled*. Pearson Education, inc, first edition, June 2016.

[37] Martin Fowler. Bounded contex, https://martinfowler.com/bliki/boundedcontext.html, October 2017.

[38] Martin Fowler. Value object, https://martinfowler.com/eaacatalog/valueobject.html, January 2018.

[39] Martin Fowler. Value objects should be immutable, http://wiki.c2.com/?valueobjectsshouldbeimmutable, January 2018.

[40] Werner Vogels. Eventualconsistency - revisited, http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, December 2008.

[41] Java Interface Future<V>.
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/future.html, December 2017.

[42] Heather Miller Viktor Klang Roland Kuhn Philipp Haller, Aleksandar Prokopec and Vojin Jovanovic. Futures and promises,
https://docs.scala-lang.org/overviews/core/futures.html, December 2017.

[43] Akka Futures.
https://doc.akka.io/docs/akka/current/futures.html, December 2017.

[44] Cay S. Horstmann. *Scala For The Impatient.* Addison-Wesley Professional, second edition, December 2016.

[45] Heather Miller Viktor Klang Roland Kuhn Philipp Haller, Aleksandar Prokopec and http://docs.scala-lang.org/sips/completed/futures-promises.html Vojin Jovanovic. Scala futures and promises, December 2017.

[46] Zaid Ajaj. Exploring folds: A powerful pattern of functional programming,
https://medium.comzaid.naom/exploring-folds-a-powerful-pattern-of-functional-programming-3036974205c8, July 2017.

[47] Routing Patterns.
https://doc.akka.io/docs/akka/2.5/routing.html, February 2017.

[48] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, Volume s2(42):230–265, January 1937.

[49] Akka Tailchopping Routing.
https://doc.akka.io/docs/akka/2.5/routing.html#routing, December 2017.

[50] Akka Cluster Specification.
https://doc.akka.io/docs/akka/current/common/cluster.html, December 2017.

[51] Cluster Client.
https://doc.akka.io/docs/akka/2.5/cluster-client.html, February 2018.

[52] Akka Clustering and Remoting.
https://developer.lightbend.com/blog/2017-05-17-atotm-clustering-and-remoting/index.html, May 2017.

[53] Message Delivery Reliability.
https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html#the-general-rules, January 2018.

[54] Akka Membership Lifecycle. https://doc.akka.io/docs/akka/2.5.4/scala/common/cluster.html#membership-lifecycle, December 2017.

[55] Carlos Baquero Marek Zawirski Marc Shapiro, Nuno Pregui ca. A comprehensive study of convergent and commutative replicated data types. *Hyper Articles en Ligne*, page 47, January 2011.

[56] Jonas Bonér. Akka cluster implementation notes https://gist.github.com/jboner/7692270, December 2017.

[57] Akka Gossip Protocol. https://doc.akka.io/docs/akka/2.5.4/scala/common/cluster.html#gossip-protocol, December 2017.

[58] Rami Yared Naohiro Hayashibara, Xavier De fago and Takuya Katayama. The phi accrual failure detector. *IEEE*, 2004.

[59] Akka Cluster Failure Detector. https://doc.akka.io/docs/akka/current/cluster-usage.html#failure-detector, December 2017.

[60] Kamil Korzekwa. Akka cluster load balancing http://blog.kamkor.me/akka-cluster-load-balancing, January 2017.

[61] Distributed Publish Subscribe in Cluster. https://doc.akka.io/docs/akka/current/distributed-pub-sub.html, January 2018.

[62] Distributed Publish Subscribe in Cluster : Publish. https://doc.akka.io/docs/akka/current/distributed-pub-sub.html#publish, January 2018.

[63] Akka Persistence. https://doc.akka.io/docs/akka/2.5.5/scala/persistence.html, January 2018.

[64] Persistence : At-Least-Once Delivery. https://doc.akka.io/docs/akka/2.5.4/scala/persistence.html#at-least-once-delivery, January 2018.

[65] Akka Snapshots. https://doc.akka.io/docs/akka/2.5.5/scala/persistence.html#snapshots, January 2018.

[66] Akka Cluster Singleton. https://doc.akka.io/docs/akka/current/cluster-singleton.html?language=scala, January 2018.

[67] Amazon Lambda.
https://aws.amazon.com/lambda/.

[68] Distributed Data.
https://doc.akka.io/docs/akka/2.5.4/scala/distributed-data.html, February 2018.

[69] Aaron Blohowiak Nora Jones Casey Rosenthal, Lorin Hochstein and Ali Basiri. *Chaos Engineering*. O'Reilly Media, first edition, August 2017.

[70] Pony. https://www.ponylang.org, February 2018.

[71] Cloud Native Computing Foundation. https://www.cncf.io, February 2018.

[72] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley, first edition, February 2013.