UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Freddy Marcelo Surriabre Dick

Actor model in the IoT network edge for creating distributed applications using Akka

Master's Thesis (30 ECTS)

Supervisor(s): Satish Narayana Srirama

Tartu 2019

Actor model in the IoT network edge for creating distributed applications using Akka

Abstract:

With the upcoming wave of devices coming in the next few years to the Internet of Things (IoT), new challenges will arise with respect to the vast amount of data generated by these devices and the processing of all these data in an efficient manner. Cloud-centric architectures, that rely on the distant cloud for processing data, do not seem to fit the new requirements of this new dynamic Internet of Things, where multiple devices constantly need to interact with each other, handling real-time data processing. The edge computing model moves the computing from the cloud to the network edge, close to the source of data, reducing latency and bandwidth needs of the whole network among other benefits. Moreover, in this new model, edge devices play a central role, handling the incoming and outgoing of the data, and providing computation power to the network. Furthermore, there is the possibility to distribute the computation process among all edge devices. In this new decentralized model, a new paradigm is needed that can deal with this distributed scenario. The Actor model, which defines actors as its basic unit of computation, addresses the need of working in a distributed environment with requirements of concurrency, resiliency and scalability among others. Message passing is defined as the sole mechanism for interaction between actors in the model, allowing to perform concurrent and parallel computation without the need of locks or any thread-safe mechanisms. The Akka toolkit is an implementation of the Actor model which offers, through its platform, a series of modules and libraries than can be used to build concurrent and distributed applications. In this thesis, the Akka toolkit is used as an alternative for developing applications on the edge, applying the concept of the Actor model. Lightweight containerization through Docker is used to deploy the application on a distributed network of devices representing the edge devices. Finally, an IoT Akka system architecture is proposed along with its implementation, based on a Wireless Sensor Network IoT scenario, to demonstrate the feasibility of conceiving applications on the edge rather than using a cloud based approach.

Keywords:

IoT, Akka, Actor Model, Edge computing, Docker

CERCS: P170:Computer science, numerical analysis, systems, control

Tegutsejate mudel Asjade Interneti hajusate rakenduste loomiseks servavõrgus Akka abil

Lühikokkuvõte:

Lähiaastatel oodatava Asjade Interneti seadmete hulga kasvuga kerkivad esile uued väljakutsed arvestades seadmete toodetud suuri andmemahtusid ja andmete efektiivse töötlemise vajadust. Pilve-põhised arhitektuurid, mis toetuvad kaugel asuvaile pilveserveritele andmete töötluseks, ei täida uue, dünaamilise Asjade Interneti vajadusi, kus mitmed seadmed peavad pidevalt üksteisega suhtlema ja reaalajaandmetöötlust teostama. Servaarvutuse mudel liigutab arvutused pilvest võrgu serva, andmeallikate lähedale, vähendades nii latentsust ja läbilaskevõime vajadusi võrgu jaoks tervikuna.

Lisaks mängivad selles uues mudelis keskset rolli servaseadmed, hallates andmevoogude sisenemist ja väljumist ning varustades võrku arvutusliku võimekusega. Sealjuures on olemas võimalus jaotada arvutuslikku protsessi serva seadmete vahel laiali.

Selles detsentraliseeritud mudelis on vajadus uue paradigma järele, mis taoliste hajusstsenaariumitega toime tuleks. Tegutsejate mudel (inglise k. actor model), mille arvutuslikeks baasüksusteks on tegutsejad, vastab hajuskeskkonna vajadustele arvestades teiste seas konkurentsuse, veataluvuse ja skaleruuvuse nõuetega.

Ainsaks suhtlusmehhanismiks tegutsejate vahel selles mudelis on sõnumite edastus, võimaldades konkurrentset ja paralleelset arvutamist ilma lukustus- või lõimeturvalisusmehhanismideta.

Akka tööriistakomplekt on tegutsejate mudeli implementsioon, mille platvorm pakub mooduleid ja teeke konkurrentsete ja hajusate rakenduste ehitamiseks. Käesolevas lõputöös kasutatakse Akka riistakomplekti rakendute arendamiseks servale, kasutades tegutsejate mudeli põhimõtet. Kergeid konteinertehnoloogiad Dockeri näol kasutatakse rakenduse juurutamiseks seadmete hajusvõrku, mis esindab servaseadmeid. Viimaks esitletakse Asjade Interneti süsteemi arhitektuuri koos implementatsiooniga, põhinedes juhtmevaba sensorvõrgu stsenaariumil, et demonstreerida rakenduste loomise teostatavust servas pilvepõhise lahenduse asemel.

Võtmesõnad:

Asjade Internet, Akka, Tegutsejate mudel, Servaarvutus, Docker

CERCS: P170:Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Throughout history advances in technology have changed the way people solve problems and think about the future. Since the invention of the computer on the 1800's, many other new developments have surged based on the concept of having computing devices processing data. There is hardly any other invention in the last century that has had such a significant impact on human lives, as the invention of the Internet as global communication network.

The invention of the Internet has drastically changed the way people and machines interact with each other, enabling instant access to real-time information and services from different parts of the world. In the last few years, Internet has become almost a necessity, as multiple services are only available through Internet. Furthermore, most devices such as computers, smartphones, sensors, etc. make use of the Internet for sharing data, performing computations or providing other types of functionalities. This connectivity of devices with the Internet has led to the creation of new concepts such as the Internet of Things (IoT) which has gained popularity in recent years.

Along this line is also the concept of cloud computing, which has become an omnipresent concept when it comes to Internet services. Multiple applications rely on the computing power and other resources provided by the cloud for their correct operation. Devices such as sensors and wearables make use of cloud services to process the data they generate. This fact puts on evidence the necessity of new ways to handle the amount of data originating from these devices. Challenges do not only involve elements such as data storage or computing power, but also demand software solutions that can manage and process this large amount of information.

While many possible solutions have been proposed and implemented to deal with this scenario, most of them rely on the cloud as a service provider. Fair enough, there are multiple cloud providers such as Amazon Web Services and Microsoft Azure, that offer various solutions for different problems. However, within the next few years, a new incoming wave of devices that will generate vast amount of data, will increase the necessity of different resources and will demand different types of solutions, addressing issues related to availability, security and latency among others.

Edge computing is set to be one of the main focus of research in the upcoming years as it specifically deals with the aforementioned problems [1]. The amount of data generated from multiple devices will require a change of approach, not only in terms of network connectivity, but also in terms of software solutions.

One of these approaches is the Actor model. The Actor model was developed by Carl Hewitt in the 1973, and since then, it has been evolving from a mathematical model to a more practical solution that fits the requirements of concurrency and distribution of computation tasks in a distributed environment. A clear example of a practical implementation of the Actor model is the Akka toolkit, which encompasses a set of modules and libraries that use the Actor model as its core element to provide a comprehensive set of tools for creating applications in a distributed environment, favoring concurrency and resiliency among other features.

The Actor model, from an application model perspective, does not limit itself to a specific domain or architecture. It can be used to model almost any kind of application, on any domain, and it can be used in different types of scenarios, whether that is on the cloud, edge or other types of environments. The Actor model presents a solution that can make efficient use of the available resources within a network and distribute the processing tasks among

all the nodes that conform it, providing a robust framework which can be used to build different types of applications in a distributed environment.

1.1 Problem statement

Internet of Things (IoT) devices used on the edge are usually resource constrained devices, at least compared with cloud devices. This implies that efficient use of its resources is not an option but a central problem that needs to be addressed. Systems to be developed on the edge must be able to respond to different challenges. Among these challenges, the most important are:

Concurrency: The fact that the edge deals with resource constrained devices, makes it almost inevitable to think about the concept of concurrency in an effort to use all the available computing power of all edge devices, distributing the computation tasks.

Resiliency: It is imperative to deal with the problem of connectivity. Especially with resource constrained devices, which can run out of power or suffer other types of problems, which can affect the system functionality and/or availability. In this sense, the system needs to be reactive and self-managed, in order to deal with these type of situations.

Scalability: Given the amount of devices working at the edge layer and its characteristics, especially in terms of connectivity and power consumption, systems on the edge layer must be flexible enough to allow new devices to join the network or to leave it, without requiring major efforts in configuration or modifications to adapt to new environments.

Performance: Appropriate use of resources is mandatory. Memory and computing power with fast and reliable response are one of the main challenges on the edge layer.

1.2 Scope and Goal

The main focus of this thesis is to study the applicability of the Actor model in the network edge to build distributed applications. In order to achieve this goal, different applications are developed using the Akka toolkit, with Java as the programming language. Another goal of the thesis is to identify and analyze ways to deploy these services on the edge, for which Docker, and more specifically Docker Swarm is used. The specific research goals are:

Feasibility of the Actor model on the edge: Analyze how to apply the Actor model on the network edge.

Suitability for applications: In terms of developing applications for the edge using the Actor model with the Akka toolkit.

Deployment of the Application Stack: Analyze mechanisms to deploy the Application Stack to the different nodes on the edge.

Fault-tolerance of the system: Once deployed, how the system can heal and manage itself in different scenarios.

1.3 Related work and thesis proposal

Several works have been dealing with the computation at the network edge, developing frameworks and platforms that can be applied on different domains. Feng et al. [12] proposed a framework for edge computing on the road for vehicles, using efficiently all the available resources on the edge. Liyanage et al. [13] developed a framework for mobile devices to provide a computation service platform. Chang et al. [14] propose the idea of the *Indie Fog* infrastructure, in which user's edge devices, such as routers, are used for providing a computational service platform on the network edge. Long et al. [15] proposed an edge

computing framework for video data processing, based on the availability of mobile devices and their computation power.

Fürst et al. [16] proposed an actor based execution framework for distributed IoT applications, called *Nandu*. This framework is also based on the Actor model, however, instead of exposing to developers with a distributed application model using the Actor model, it provides an execution environment that can be used to implement sequential application logic, abstracting the underlying execution mechanism that works using actors and adapting them throughout the lifetime of the application.

Most of the related work have dealt with the computation at the edge using the typical edge model of distributed devices with no further importance to the model and the relationships behind the connectivity of the nodes in the network. The work of Fürst et al. [16], instead of using the Actor model as an application model, envisions to abstract its logic from the development process. Contrary to all these approaches, in this thesis, the focus is on the architectural model and patterns that can be used to conceive applications on the network edge, embracing the distributed nature of the environment. More specifically, using the Actor model as an application model to build different types of edge applications. The Akka toolkit is used as a main tool for providing a robust edge architecture, that can be used to conceive applications on the network edge, making use of all the available power of the different nodes that compose the network.

1.4 Thesis outline

This thesis is structured in the following manner: First, on chapter 2 the basic concepts of the Actor model and Akka are introduced in order to have a clear understanding of the concepts discussed throughout the thesis. Next, on chapter 3, two implementations developed using the Akka toolkit are presented and discussed, along with detailed descriptions of the structure of the applications and its components. On chapter 4, an edge architecture is proposed along with its implementation applied on a real IoT scenario, in order to demonstrate how applications can be conceived using the Actor model and the Akka toolkit. On chapter 5, the main application developed on chapter 4, is put on evaluation with respect to the research goals of the thesis. Finally, on chapter 5, a series of conclusions are drawn based on all the work carried out throughout the thesis, analyzing the most important aspects and considerations to be made when working with the Actor model, using the Akka toolkit, to create edge applications.

2 Background

2.1 IoT – Internet of Things

The Internet of Things represents an interconnected network of different elements such as mobile phones, sensors, vehicles, home appliances, wearables etc. The heterogeneity of these elements is represented by the word “things”, as nowadays almost anything can be connected through the internet.

The availability of the internet in a global scale has allowed manufacturers to develop products that can rely on the internet to share and process data. This has brought along the development of software applications that can leverage these devices through the use of internet, enabling communication between these devices and other components, and making it possible to process data on a large scale. These devices are present in almost any field of the human activity. An example of this are all the new smart home appliances that can be remotely controlled through internet. Another example is the car industry, where new car models are fully automated and operate sending and receiving real-time information through sensors and using the internet to share and gather data in order to make decisions.

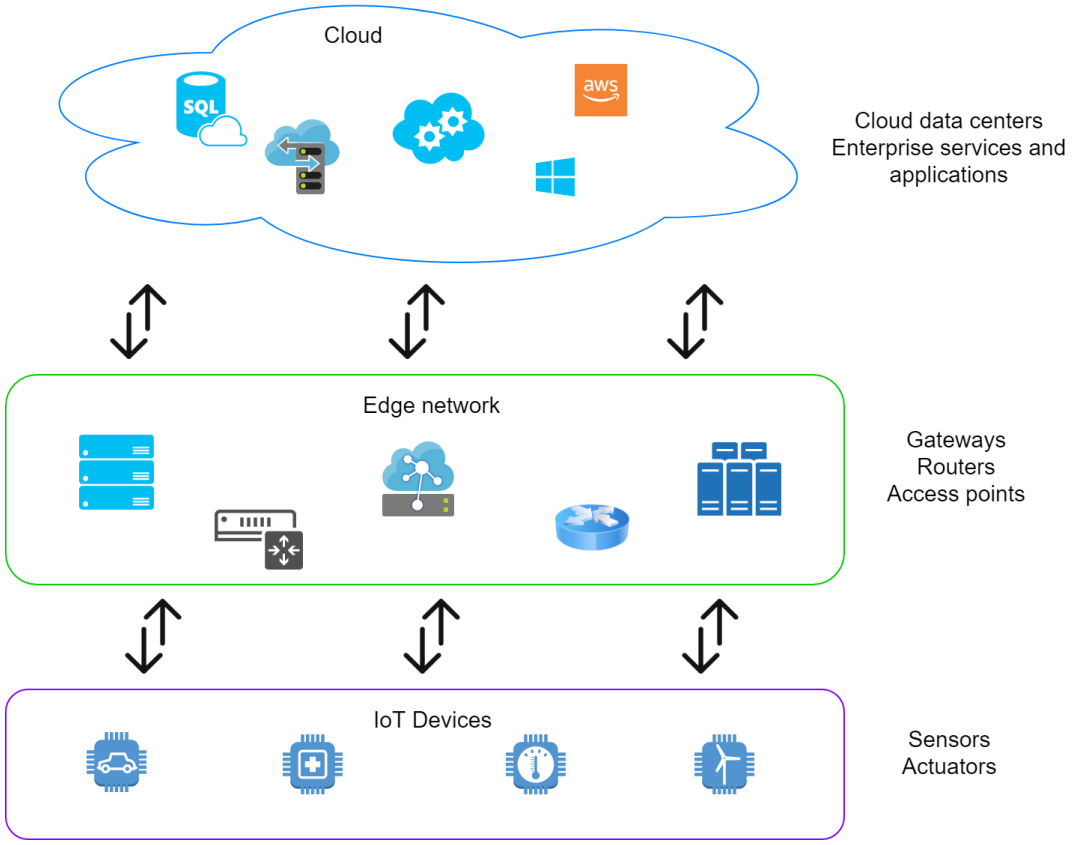

Figure 1. IoT Architecture

2.2 Edge computing

The idea of edge computing consists of giving more decision capabilities and independence to the network edge. The network edge is the closest to the devices that are the source of the data to be processed. Placing computing power and other resources on the proximity of these devices can help dealing with different problems inherent of internet communication, such as latency and bandwidth [1].

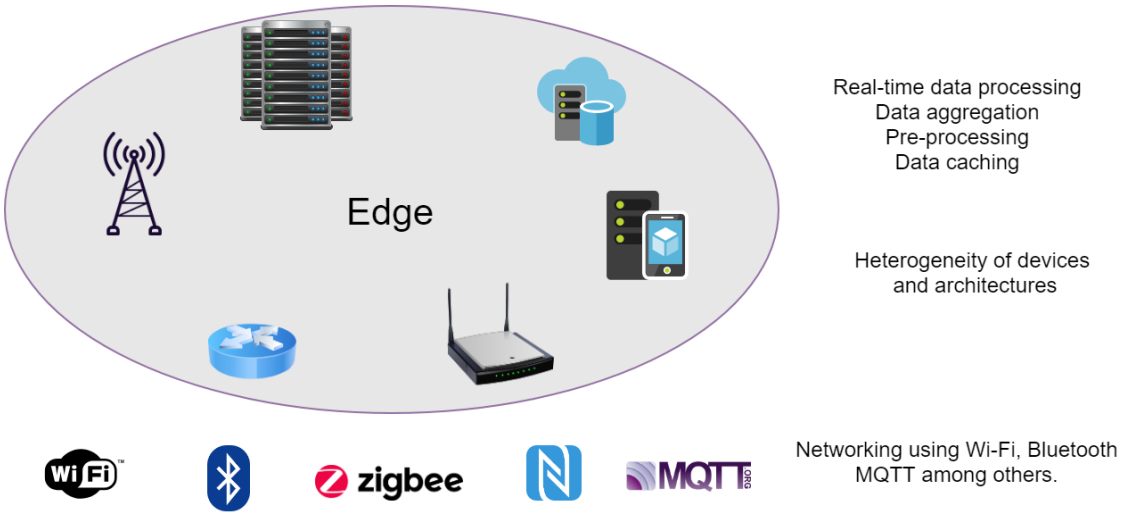

Figure 2. Edge computing characteristics

There are many classifications for the edge devices, and this could vary depending on many aspects such as the deployment architecture [2] [3]. The concept of what encompasses the edge is also blurry, with different authors extending the term to include different elements. Taking this into account, it is possible to say that the edge comprehends a wide range of elements, from small and resource-constrained devices, such as sensors and wearables, up to more heterogeneous and complex elements, such as resource-rich servers and edge data centers.

The benefits of using edge computing can be summarized in the following aspects:

- Low latency communication: As the source of data is close to the processing center, there is less time between sending and receiving packets of information.
- Reduce bandwidth of the network: As more data is handled locally, the amount of data that travels back and forth with the distant cloud is reduced, alleviating the saturation of the whole network.
- Location awareness: A device on the edge can be aware of the surrounding context where it is deployed.
- Geographical distribution: Devices on the edge can be distributed on a large area, providing uninterrupted connection, critical aspect in some application's context.
- Security: With most of the data processed locally, there is no need to send important private information to the cloud, meaning less opportunities for attackers to obtain information by intercepting communications or taking advantage of cloud data breaches.

Despites these benefits, there are numerous challenges to face on the edge. Lots of these challenges are related with the inherent characteristics of the edge, such as limited computation power or the heterogeneity of the devices.

As suggested by different authors such as Weisong [4], among these challenges there is one in particular that is of interested of this thesis, the programmability of edge applications. In this sense, applications on the edge must be in accordance with the nature of the edge, that is to say, applications must be designed so that they can be partitioned or distributed over the network. This seems like a perfect fit for the Actor model which is discussed in the next section.

2.3 Actor model

The Actor model was developed by Carl Hewitt [5] in 1973. This is a mathematical model that was inspired by physics laws rather than mathematics. The model was conceived as a universal paradigm for concurrent computation, hence, it is in nature a concurrent model of computation, which means that is suitable for creating highly concurrent and parallelizable systems in a distributed environment.

At a higher level, the model is simple. The basic unit of computation is an Actor. An Actor is an entity that can communicate with other actors through messages, and this is the only mean of communication. An actor can also create other actors establishing a hierarchy of actors within a system. An actor embodies the following three things:

1. Information processing (computation through its behavior)
2. Storage (state)
3. Communication (through message passing)

An actor has state and behavior, much like an Object on the Object Oriented Paradigm. However, in the Actor model there are some restrictions that bring some guarantees at the time of carrying out computations. The state is own completely by the actor and it is not sharable or accessible to other actors on the system. This means that there is no necessity for locks or other types of synchronization mechanisms on a multithreaded environment.

The actor can change its state in response to a message or can perform some computation depending on the message. The computation is the behavior of the actor. An actor can also send messages, which will be directed towards another actor or the actor itself, thus, allowing recursion. This receiving actor, will proceed to take a specific action, as previously mentioned. The set of actors that take part on this communication, will construct an actor system.

Messages are one of the key concepts of this model. It is the only way to communicate between actors. In concrete, an actor can do one of the following things in response to a message [5]:

- Send a finite number of messages to other actors
- Create a finite number of new actors
- Designate the behavior to be used for the next message it receives

One of the main achievements of this model is the decoupling of the actor from the process of sending messages, which can be done asynchronously. An actor can only communicate with other connected actors. Connections can be done through:

- direct physical attachment
- memory or disk addresses
- network addresses
- email addresses

Depending on the type of connection, addresses will vary, it could be MAC address in case of physical connection or a simple memory address. Messages are delivered on best efforts basis. Once an actor has sent a message, it is responsibility of the receiver to handle it, this

is the key element that allows decoupling a message from the sender actor. This type of communication is also referred as *fire and forget*.

The Actor model is an abstract concept based on some axioms that define the behavior and structure of the model. There are several properties and mechanisms working behind scenes. Implementations of the model should obey these rules and may use other concepts on top of it, to expose the behavior of the model in a practical way.

2.4 The Akka toolkit

Akka is a toolkit based on the idea of creating distributed systems using the Actor model. Akka was developed for Scala and Java, making use of the Java Virtual Machine (JVM). It is also possible to find other implementations of the Actor model for other programming languages, such as Akka.NET, for the .NET platform, using C# and F#.

It is also important to notice the amount of information and projects available using Akka in regards to the language. Most of these implementations use Scala, while very few are implemented in other programming languages. This can be explained because of the origin of the creators of Akka, who are also involved in the development of Scala, but also because of the facilities that Scala provides. For instance, code for implementing Akka in Scala is very short and succinct in comparison with Java code, which can be quite long and bloated.

Akka defines itself as a toolkit, which provides different sets of modules and libraries for exposing different types of functionalities of the Actor model, such as remoting, clustering, persistence, etc. The Akka documentation is extensive diving deep into some concepts. Most of the implementation details come from research papers and industry experience. The company that is behind the development of Akka, *Lightbend*, is a commercial company offering enterprise software solutions for distributed systems and cloud environments. This company is also behind the development of multiple frameworks and platforms such as the Play framework and the Scala programming language. Most of the examples and tools provided are oriented towards Scala with less support for Java and other languages.

In order to start building applications with Akka, it is important to have a clear understanding of the basic concepts. In this case, the core concept is the concept of an Actor and how it is implemented in Akka. On top of this basic concept other more elaborated concepts are built, such as Clustering and Sharding.

2.4.1 Actor

Akka defines an actor as a container for state, behavior and a mailbox along with its child actors and supervision strategy. All of these elements working together conform an actor in Akka. In the context of an actor system, actors need to know where to reach each other. For this, Akka uses actor references, which is an object than can be passed around as a sort of contact information, that can be used to communicate with a specific actor. This actor reference serves as the only way of communication with an actor, leaving the internal components of an actor isolated from other actors, which serves to preserve its internal state from any kind of modification from the outside, which in turns allows for parallelism of operations.

The actor's state can be defined using a Finite State Machine mechanism, which Akka provides as a library, or it can be defined by user requirements as any other object. Akka guarantees the thread safety of passing messages, thus, the actor can process each message without the necessity of using locks or any other type of thread synchronization.

Akka provides an infrastructure in which the actors can live, that is to say, Akka creates the actor system environment so that the user can deal directly with application details instead of dealing with the actor system internal details. This environment consists of a hierarchy of actors, with parent-child relationships. Figure 3 illustrates the Akka hierarchy of an actor system.

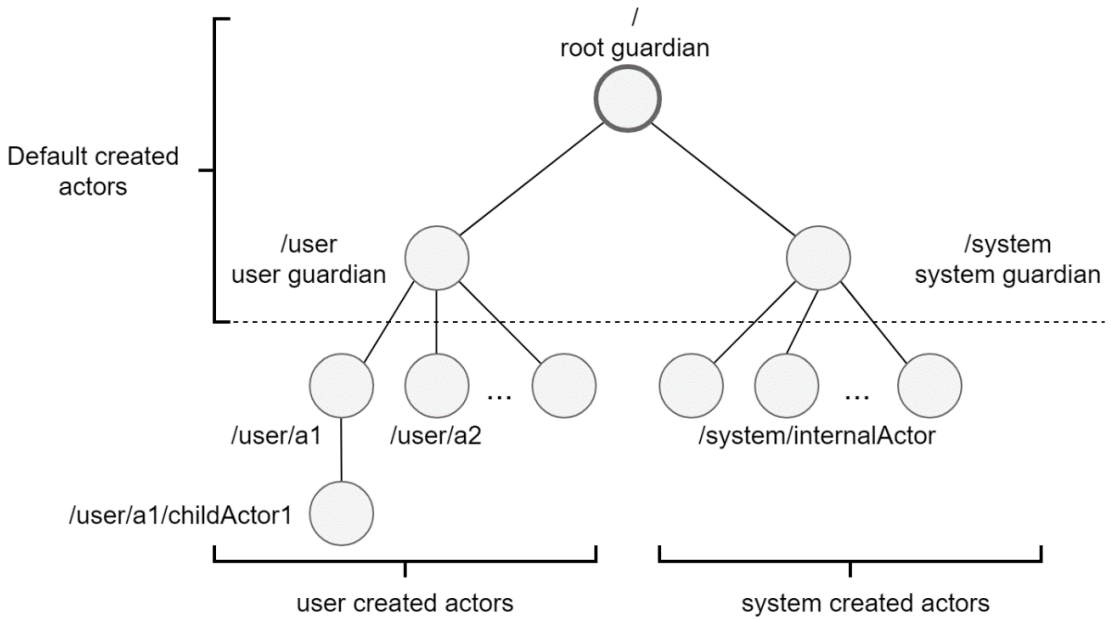

Figure 3. Actor system hierarchy

The root guardian "/" is a pseudo-actor because every actor has to have a parent actor that creates it, as shown in the hierarchy, but because this actor is the main root, it doesn't have a parent, so it cannot be a "complete" actor. This root guardian serves a parent of all other actors in an actor system.

The user guardian "/user" is the parent of all user created actors. When creating an actor, they are not created directly under the root guardian, but instead they have their own branch under the user guardian, this serves to distinguish these actors from the other Akka private actors.

The system guardian "/system" is the parent of all Akka private actors that are used to run and maintain the system functionalities. These actors are not directly accessible to user actors and it is not possible to create user actors under this branch.

The three previously mentioned actors are created by default when Akka starts an actor system. It is also important to mention how user created actors are supervised. The actor system hierarchy defines a special relationship between actors. Every actor, at least user created actors, have a parent actor, and possibly multiple child actors. In the case of failures, Akka defines different options to respond to a failure:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Stop the subordinate permanently
4. Escalate the failure, thereby failing itself

In this sense, Akka defines a supervision hierarchy which responds to the four above mentioned possible scenarios. This aspect is also related with an actor lifecycle.

2.4.2 Actor lifecycle

Actors have a lifecycle that begins when an actor is created on a specific context or by the actor system. Actors are created from "prototypes" that are defined by the user. These prototypes are classes that extend one of the predefined classes of Akka to create actors, such as *AbstractActor* or *AbstractPersistentActor*.

The created actor is also called incarnation, which has a unique identifier or UID. This identifier will be maintained by the incarnation even after it restarts. However, if an actor is stopped, and then created again, then it will be given a new unique identifier. Apart from the identifier, a unique path for an actor is assigned so it can be reachable from other actors.

Once the incarnation has been created, a method hook is called on the created actor, in this case the *preStart* hook is called. This hook can be used to initialize the actor using database connection or other types of initialization tasks.

The *postStop* hook is called when an actor has received the order to be stopped. This hook can be used to free up resources and close respective connections.

In case of restarting there are two additional hooks. The *preRestart* hook which terminates all child actors of an actor before restarting. Once all children are stopped, a new instance is created and the *postRestart* hook is called on this new instance. The *postRestart* hook, by default, calls the *preStart* hook on the new instance so that it can be correctly initialized. The whole process is illustrated in figure 4.

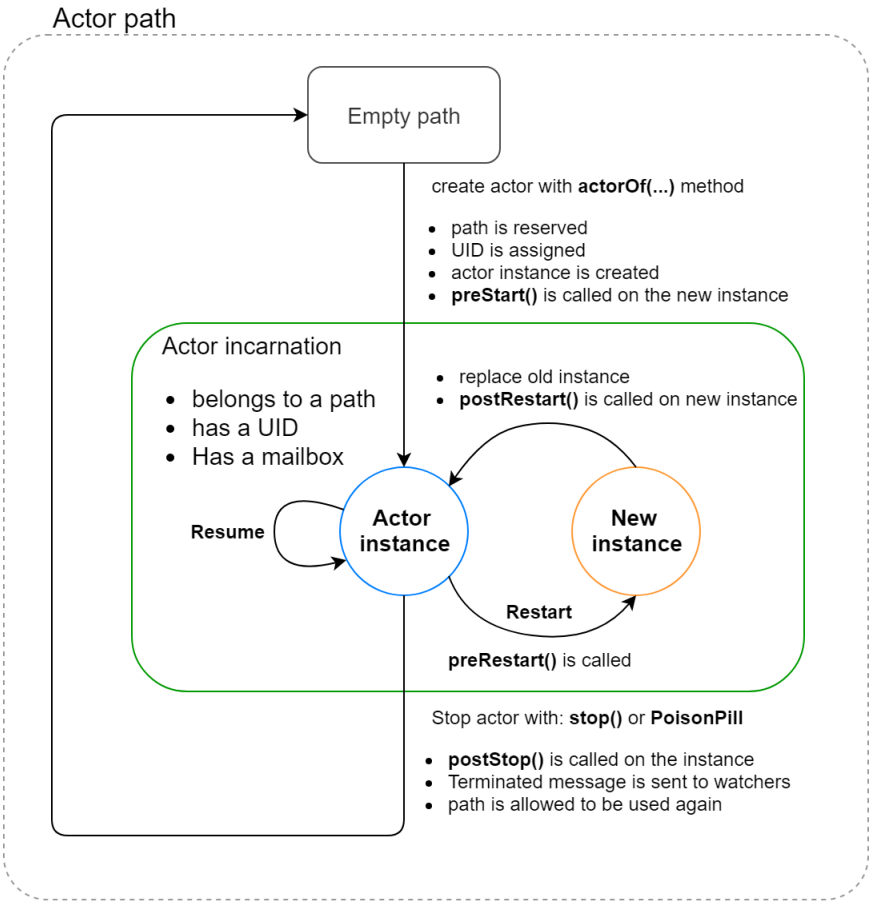

Figure 4. Actor lifecycle

2.4.3 Messages

Actors can communicate with each other through messages. To send a message from within an actor it is necessary to have an Actor reference or an *actorSelection* object, which represents the address of the recipient of the message.

There are two possible ways of using messages:

1. Using the “tell” method, which will send a message asynchronously and return immediately.
2. Using the “ask” method, which deals with “futures” and handlers or callbacks when sending messages.

Using the “tell” function is the most common way to use for simple messages, but when dealing with more complex scenarios it is better to use the “ask” method, which allows to aggregate different futures, meaning different messages, and combine the results so than it can be piped to another actor.

It is also possible to forward messages with the “forward” method which is useful when using proxy actors that act as routers or replicators.

In order to receive a message, every actor must override the method *createReceive* that returns a *Receive* object. Akka provides a useful receive builder that helps to define the behavior of the receiving actor. This *Receive* object represents the behavior of the actor when receiving messages. It matches the types of messages it can receive and defines a handler per each type as follows:

```
@Override
 public Receive createReceive() {
  return receiveBuilder()
     .match(Msg1.class, this::receiveMsg1)
     .match(Msg2.class, this::receiveMsg2)
     .match(Msg3.class, this::receiveMsg3)
     .build();

 }
```

It is necessary to always match the type of message otherwise this could generate a failure. Replying to messages is done in the same way as sending. To get the actor reference of the sender Akka provides the method *getSender*.

Actors receive messages on a mailbox. Every actor has a mailbox. Actors obtain messages from the mailbox one by one. This allows for the actor to perform in a single-thread manner with respect to message processing. The order of message arrival from one actor to another is guaranteed by Akka. That is to say that if messages: m1, m2 and m3 are send, in that particular order, from actor1 to actor2, then actor2 will receive in its mailbox m1, m2, m3, preserving the order. Then actor2 will start processing the messages in a FIFO order. Also important to mention is the fact that Akka by default offers “at most one delivery”. This means that a message is sent once, but no guarantees that it will be received. If this is needed, it can be configured to do so at expense of performance.

The above mentioned concepts are the basic ones that enable the whole Actor model on Akka. More detailed descriptions on the implementation of the Actor model and other concepts concerning it, are defined in the official documentation [6] that is quite extensive and involves different aspects to consider when implementing an actor system, such as fault tolerance aspects or changing the behavior of actors in order to respond in different ways at different stages of an application.

2.4.4 Persistence

It is common to deal with stateful actors that maintain and change its state through its lifecycle. For this reason, Akka provides the concept of persistent actors. This is another library extension that comes with some default plugins, such as memory-based journals and local snapshot-store. In order to make use of this persistent capabilities, the stateful actor should extend the *AbstractPersistentActor* abstract class or the more specialized abstract class *AbstractPersistentActorAtLeastOnceDelivery*, which offers some guarantees when delivering messages.

2.4.5 Event sourcing

Akka uses event sourcing to deal with persistence. In this scenario, a persistent actor receives commands, through messages, these commands are then validated, and once they are marked as valid, they generate events that represent the effect of the command. These events are persisted in a journal preserving its order of occurrence. A journal is the place where events are stored and become the source of events when a stateful actor is recovered. The fact that only events are stored, and not commands, guarantees that an actor recovers to a valid state, as only valid commands that later generated events are stored in the journal.

Another option to handle persistence is through snapshots. This concept can be useful in systems that have long life or handle a numerous amount of operations. For example, a ticketing system, that can provide different operations such as reservations, cancellations, modifications, etc., where each of those operations can be performed per ticket. In case of recovering this type of system, it would require a lot of time to replay all the events that occurred since the beginning of its existence. With snapshots, it is possible to persist the state of an actor at a certain point in time and to bring it all back when recovering, in a single operation instead of going through all the events one by one.

2.4.6 Routing

In Akka, routing is used as a mean for passing messages efficiently between actors. The idea is to have an actor that serves as a router. This actor is in charge of routing received messages to other actors called routees, using a specific routing logic. Akka includes a set of routing logic strategies for routing messages, such as *RoundRobinRoutingLogic* or *SmallestMailboxRoutingLogic*, each with different characteristics for different possible scenarios, making routing flexible to fit different types of applications. Figure 5 illustrates the routing mechanism in Akka.

There are two main ways to define router actors:

1. Having a normal actor, and creating a Router object inside the actor. Also creating the routees as normal children actors and then adding them as routees to the router object.
2. Creating a self-contained router actor, with the help of some configuration so that it can handle itself the routees and all routing details.

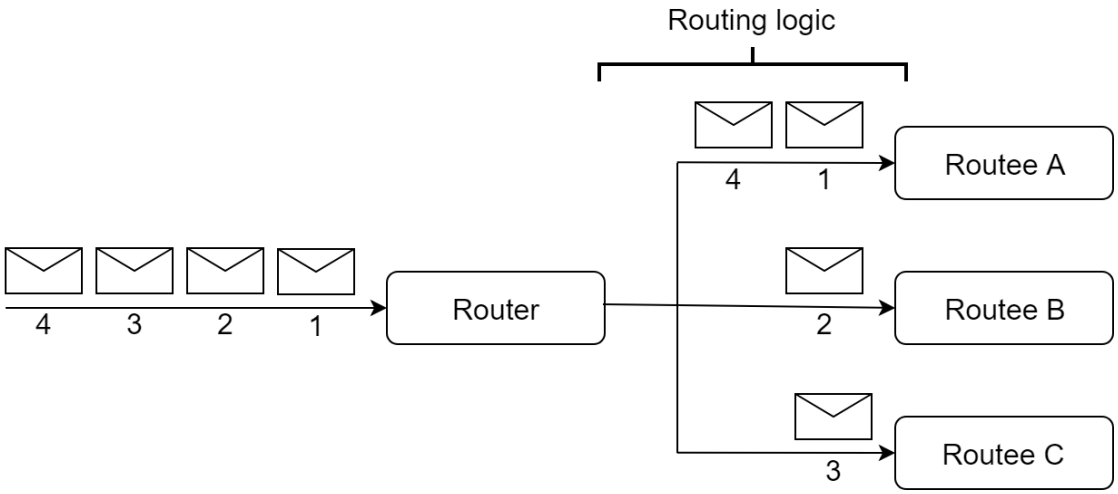

Figure 5. Routing in Akka

The first case is straight forward. Basically, it is necessary to specify in the code all the characteristics of the router and how it will handle the routees. The second case is more interesting as the actor itself acts as a router. The router capabilities, restrictions or limitations are defined in the configuration file. Akka uses two configuration files: *application.conf* and *reference.conf* (at the end both are merged into one configuration file). Different configurations can be set for routing actors, such as the routing logic, number of routees, local or remote routees, etc.

Routing actors are divided into two types: Pool and Group.

A Pool router actor have the characteristic that it creates the routees itself, and have full control and supervision of the routees. Whereas a Group router actor does not create the routees, and relies on them being created externally and being passed to it for their use.

The way to pass messages to routers is the same as with normal actors, at the end, either being a router itself or not, a router still works in an actor environment. The handling of the message, however, is somehow different. When a message is sent to a router, the router receives the message and forwards it to one of the routees, except in the case of a broadcast message, in which case all routees will receive the same message. The original sender of the message is preserved, meaning that when a routee sends a message with the "getSender" method, it will send the response back to the original sender not the router actor, even though it received the message from the router actor.

2.4.7 Clustering

The main idea behind Akka is to work in a distributed environment where all communication is handled asynchronously. From its conception, Akka was thought as a distributed tool taking into account the nature of distributed systems and how they differ from non-distributed systems [7].

In this context, one of the core concepts within Akka is the concept of Clusters. Actors live within systems and these systems can be distributed within a network. A Cluster is a group of nodes where each node represents an actor system running in a Java Virtual Machine (JVM). This is not a restriction, as it is possible to have multiple actor systems under the same JVM, but the most common practice is to use one single actor system per JVM, as shown in figure 6. This does not only serve to preserve the idea of nodes as single actor systems, in which one node is one actor system, but also helps to maintain independence between actor systems.

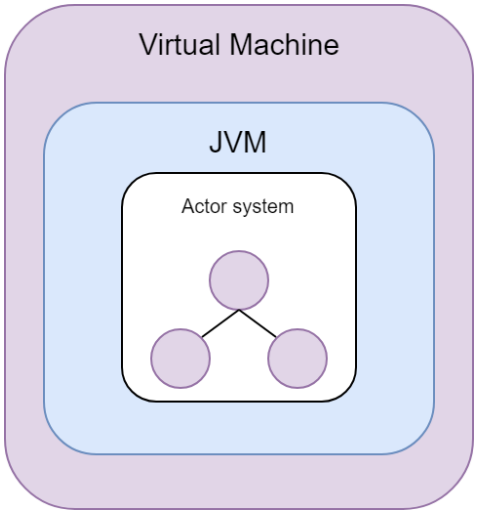

Figure 6. Actor System in a JVM

The cluster works as a peer-to-peer network, in which all nodes are peers, with no concept of master node (although the master-slave pattern can be implemented within the cluster). This helps to eliminate the single point of failure problem, or single point of bottleneck.

Akka relieves the pain of dealing with the problem of local or remote communication as it enforces distributed mechanism from its roots using location transparency. This is reflected in the way the actors communicate with each other. In this aspect, there is no difference, in terms of code, between a communication with a local actor or a remote actor. Actors are unaware if the communication is local or remote, they just send a message, using an actor reference of the receiver, and the rest is handled by the system.

Terms

The following are the main terms used when dealing with clustering:

Node

A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a *hostname:port:uid* tuple.

Seed nodes

Nodes that are used as entrypoints for new nodes to join the cluster.

Cluster

A set of nodes joined together through the membership service.

Leader

A single node in the cluster that acts as the leader. Managing cluster convergence and membership state transitions.

2.4.8 Membership

Clusters make use of a Gossip Protocol to allow new nodes to join a cluster. This type of membership is inspired by Amazon's Dynamo system [8] and Riak database [9]. This protocol makes use of communication between the members of a cluster in the form of a "gossip" between each other until all of them converge into a state where all of the member nodes are aware of the "gossip".

2.4.9 Membership lifecycle

The cluster membership lifecycle can be represented with the states of the nodes, as shown in figure 7, where “fd” stands for failure detection. There are 6 possible states. At the start, when a node wants to join the cluster it is in the “joining” state. At this stage, the leader has to make sure that the gossip of the new node entering the cluster has converged. Once this happens, the state of the node changes to “up”.

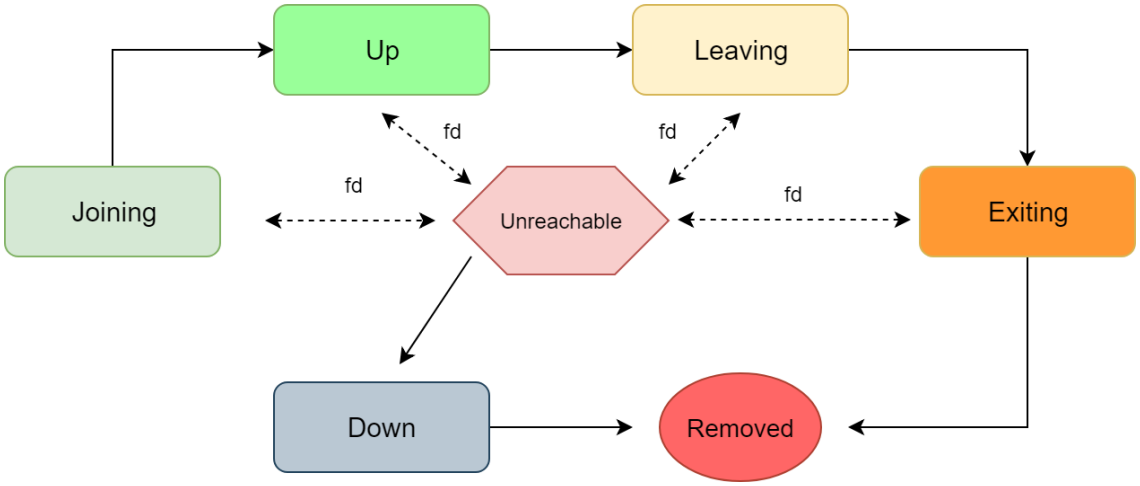

Figure 7. States of a node in an Akka Cluster

When a node leaves it is in the “leaving” state, and then once the gossip has converged with the leaving of the node, the leader node puts it to the “exiting” state and finally to the “removed” state. Regarding the “down” state, it is possible to reach this state from all the other states, except for the “removed” state, as the removed node is no longer taking into account as part of the cluster. In order to set a node down, the node first has to be in a quasi-state called “unreachable” which acts as a flag indicating that it is not possible to establish communication with the node.

2.4.10 Seed nodes

Seed nodes are defined as the entry points for a node to join the cluster. These nodes take a special role within the cluster, as they are the ones in charge of building up the cluster. The definition of which nodes are seed nodes is done in the configuration file, in the form of a list of remote addresses. The creation of the cluster is as follows: At the beginning of the cluster, when there are no members, and in fact, there is no cluster yet, the first seed node on the list is the one that has the responsibility of creating the cluster. Obviously it has to be up in order to start the cluster. Later other seed nodes and normal nodes can join the cluster by contacting any seed node that is reachable. Only the first seed node is capable of starting the cluster. This is done in order to avoid other seed nodes creating new clusters in the case of a network partition. Once the first seed node on the list is up, and other seed nodes have joined the cluster, any node trying to join the cluster can do it by contacting any seed node, not only the first on the list, as the cluster is already started.

2.4.11 Cluster singleton

This a pattern that is used on clusters when there is the necessity for a Singleton Actor across the cluster. Akka offers the possibility to create a singleton actor, but it also warns of some shortcomings of doing so, such as: Single point of failure, bottleneck, relying into its existence at all times and multiple singletons created in case a network partition occurs.

Nevertheless, the documentation also provides some guidance on how to circumvent these problems. Furthermore, there could be some useful cases in which singletons are needed, such as having one master node for centralizing some functionalities. Akka itself uses this concept when dealing with Cluster Sharding,

2.4.12 Remoting

Remoting is the communication module that works underneath a cluster. This module is what makes it possible to have a peer-to-peer communication between actor systems. Akka defines remoting based on the idea of symmetric communications, in the sense that both ends of the communication can accept and initiate connections.

This module is now an essential part of Akka, and is no longer intended to use as a standalone module. Most of the concepts and configurations are already present in the cluster module and it is recommended to use the cluster configuration instead of just remoting [10].

Akka offers two ways of remote interaction: Lookup and Creation.

Lookup deals with finding remote actors. In order to do so, Akka uses the concept of *ActorSelection*, which works like an actor reference. To obtain the *ActorSelection* it is necessary to specify the location of the actor in the following format:

akka.<protocol>://<actor system name>@<hostname>:<port>/<actor path>

And can be used in the following manner to obtain the ActorSelection:

ActorSelection selection = context.actorSelection("akka.tcp://app@10.0.0.1:2552/user/serviceA/worker");

With the selection, it is possible to send messages to the remote actor in the same way as with an Actor reference. It is also possible to obtain directly an actor reference from the actor selection. It requires an exchange of messages with the identity of the remote actor.

The second type of remote interaction, Creation, refers to the case possibility of an actor system to remotely deploy actors on other (remote) system. The location of the nodes to deploy can be configured in code or within the configuration file. The remote creation of actors allows for a distributed approach when dealing with actor systems and routing, enabling load balancing and other benefits that are also used when sharding a cluster.

2.4.13 Sharding

The next big concept in Akka is Sharding. Sharding goes hand in hand with Clustering and it seems like a natural progression of the concept of Clustering. Sharding consist in distributing actors of a specific type across different nodes in a cluster in order to properly distribute the use of resources of a cluster. The type of the shard acts as a label that represents the types of entities that are handled by a shard. In order to use sharding in a cluster, all nodes should create a shard region for the corresponding type.

The Sharding concept involves the following main terms:

- **Entity:** An actor with an Id in a Shard

- **Shard:** Group of entities that are managed together. A shard also has an Id.
- **Shard Region:** A region within a node that holds Shards
- **Shard Coordinator:** An actor in a node in charge of managing the Shard Region. This is a singleton actor.

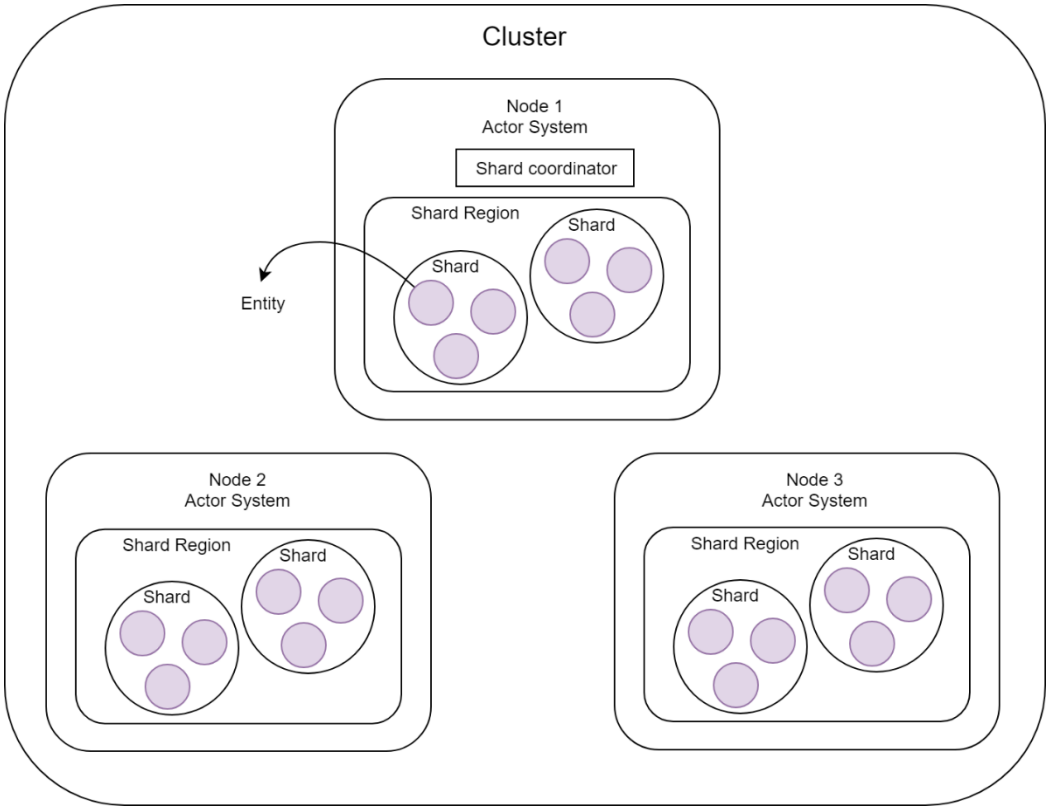

Figure 8. Akka Sharding within a Cluster of 3 nodes

Cluster Sharding comes as a module, so it is necessary to import the required library in order to use it. The user does not have to deal with all the internal details of the sharding process. Akka provides a clean solution to maintain the user focus on the business logic while hiding the internal complexities of the sharding mechanism, which can be configured by setting different parameters.

In concrete, the user should only care about creating a shard region with its corresponding configuration settings. At creation time, the user must define a *MessageExtractor*. In order to do so, it is necessary to create an object of this class overriding the following methods:

- entityId
- entityMessage
- shardId

The implementation of these methods can vary depending on the application requirements. The idea is that when a shard region receives a message, from this message is should be possible to: extract the id of the receiving entity with *entityId* method, so that it knows to which actor the message is destined, extract the message or payload with *entityMessage* method and identify to which shard the message should be directed to with the *shardId* method.

That is all the responsibility of the user, it should only care about creating the proper configuration for the shards. All the setup for creating entities, routing and balancing the shards is handled by Akka.

Behind the scenes, Akka uses the help of the *ShardCoordinator* to manage the shards, this coordinator is a singleton actor per type of shard. When a shard is created, the *ShardCoordinator* will decide which shard region should manage the shard, and it will notify it. The shard region in response, creates the Shard actor. This actor will create the individual entities and will become its supervisor.

The whole process works as follows: when an actor sends a message (directed to an entity of a shard), it sends it to the shard region actor instead of the entity. The shard region uses the *MessageExtractor* to extract the details of message, such as the *shardId* and *entityId*. Then it communicates with the shard coordinator asking for the location of the shard with the extracted *shardId*. The shard coordinator knows all the locations of the shards so it will reply to the shard region with the correct location of the shard. Next, the shard region receives the message and redirects, if necessary, the message to another shard region. If the shard is residing under the same node of the shard region, then it just passes the message to the corresponding shard actor. Finally, the shard actor will pass the message to the corresponding entity using the entityId.

It is also important to notice that the messages to be handled by the sharding mechanism must be in accord with the *MessageExtractor*. In case the *MessageExtractor* cannot extract the corresponding ids or the payload message from the message, the the communication will fail.

The other important concept in sharding is rebalancing. This consist in the migration of a shard, with all its entities living under it, from one node to another. The decision of when to migrate can be configured by using strategies and setting some threshold values. While doing the migration all messages to the shard are retained until the hand off is concluded. Once the sharding has being recreated the messages will be redirected to the new location of the shard.

Persistence plays an important role in sharding. At least in the case where persistence is required. This is because, when a shard is migrated from one node to the other, all the entities and its states are destroyed. Later, the entities will be recreated on the new shard location, but its state will be lost. In this case, it is necessary to use persistence to store the state so that the new entities can replay the corresponding events to recover its previous state.

2.4.14 Configuration files

Akka makes extensive use of the configuration files(s). There are two main configuration files: *application.conf* and *extension.conf*. Both use the HOCON (Human-Optimized Config Object Notation). This is a configuration format developed by “Typesafe”, the same company behind Akka. There is no particular difference between both configuration files, and both can be present. The only distinction mentioned in the Akka documentation is that the *extension.conf* file should be used in case of creating Akka libraries, meant to be used by other Akka applications. While the *application.conf* file should be use in cases where the main goal is to build Akka applications.

2.5 Lightweight virtualization

It comes naturally to think that in a heterogeneous environment such as the edge, it is necessary to use some kind of tool that would assist in the process of deploying services to the

edge, considering all the constraints and characteristics of edge resources and the complexities of the network. In this context, Lightweight virtualization technologies arise as a good alternative to deal with this problem.

Lightweight virtualization, applied through Containers, allows for a decoupling of hardware and software, allowing for software to be deployed on different types of hardware architectures. This scenario seems to fit into the description of the requirements for creating edge services.

There are multiple benefits of using virtualization technologies, especially in the current context where many specialized tools have been developed throughout the past years reaching a point where they are suited to be used in production environments. A very clear example of this is Docker, that offers a rich set of functionalities to conveniently deploy applications in distributed environments, for instance, the use of Docker Swarm.

There are plenty of research papers that have studied the use of Containerization on IoT context, deploying services at edge nodes and gateways. An excellent reference in this context it the paper of Roberto Morabito [11]. In this document, the author evaluated in terms of performance the use of lightweight virtualization, using Docker, in the context of IoT applications, using different Single-Board Computers (SBC), including the Raspberry Pi 2 model B and the Raspberry Pi 3 model B. The author takes one step further, using as base other related works and adding other metrics such as power consumption and energy efficiency.

In particular, among several conclusions, the following conclusions are of interest:

- Employing container-virtualization does not incur in a significant impact in terms of performances when compared with native solutions, this includes the scenario when several containers are running at the same time.
- Raspberry Pi's boards are highly efficient dealing with low volumes of network traffic. This aspect can be useful for deploying applications at the gateway level and other messaging protocols such as MQTT (Message Queuing Telemetry Transport).

It is also important to mention, that the author also makes reference to some specific points that were not fully considered during this research, such as the interaction between multiple gateways and the security using containers.

Nevertheless, the study provides a sufficient background to assert that it is not only practical but also appropriate to use lightweight virtualization through containers in the context of developing IoT applications in the edge.

3 Implementation

This chapter deals with the implementation of the Actor model on the network edge using Akka. The developed applications aim to serve as base frameworks for future applications on the edge, taking into consideration elements of availability, resiliency, and scalability among others, which are desired on the network edge.

3.1 General considerations

Programming language

The official Akka toolkit provides support for Scala and Java, although in theory, any JVM language could be used. The implementations developed for this thesis use Java as the implementation programming language. However, it is important to mention that Akka itself is implemented using Scala, and it uses several concepts that are more in accordance with the functional approach that Scala provides, such as using functions as first-class citizens.

Build Tool

Gradle was chosen as the build tool for the applications. Although Maven is another option, Gradle is a tool that is being used more often in several new platforms and applications, such as Android. Gradle uses Groovy as a DSL (domain specific language) to define the build script, which is easier to use, as Groovy itself is a programming language.

Akka version

Akka version 2.5 is used for the project. It is important to mention that the Akka toolkit is in constant development. Additional features are still being developed and others are on testing stages. For this reason, only the stable features are used for the projects, leaving other features out of scope in order to conceive stable applications.

Devices

A Linux machine, with enough resources, such as CPU computation power, is used as a representative of a more powerful device on the edge.

2 Raspberry Pi model 3B+ are used as a representative of constrained resource devices.

Implementations

There are 3 different types of applications developed. The first one consist of a simple cluster application. The second one adds the concept of Routing and Docker Swarm. Finally, the third application involves a more complex IoT scenario using Docker Swarm. These projects are available through public repositories[1][2][3] where each project is independent of the others, which allows to use them independently and in accordance to the requirements of the applications to be built on top of them. The last implementation, which models an IoT scenario, uses the concepts and techniques used in previous implementations as guidelines.

3.2 Docker Swarm setup

The idea of a distributed system is to have a group of interconnected devices that can communicate and share information between each other. On the network edge, many different

[1] http://github.com/marcelo-s/akka-cluster-basic/
[2] http://github.com/marcelo-s/akka-cluster-swarm/
[3] http://github.com/marcelo-s/akka-iot-wsn/

aspects must be considered, such as heterogeneity devices and network related issues. Managing all these aspects can become a complex task, for which some kind of mechanism must be used to face this challenge.

As previously stated, lightweight virtualization helps to address these concerns. The idea is to have the edge devices connected using Docker. In order to accomplish this, it is necessary to have Docker running on every device. For the developed applications, two Raspberry Pi devices were used along with a Linux machine. Each Raspberry Pi runs the Raspbian Operating System (OS), which is the default OS when installing the OS through the Raspberry Pi software tool NOOBS (New Out Of the Box Software). Raspbian is a light OS based on the Debian Linux distribution. Installation of Docker on Linux devices is quite simple following the instructions provided in the official Docker website. The advantage of using a Linux operating system is that Docker works natively when installed on Linux devices and takes advantage of the Linux architecture to create containers.

Docker offers a feature called "Docker Swarm", in which several machines running Docker, also called nodes, can be connected through Docker forming a cluster, as shown on figure 9. Nodes in a swarm can be one of two types: manager nodes or worker nodes. Manager nodes, as the name implies, are in charge of carrying out the orchestration of the swarm, such as scaling and managing services among others tasks. Worker nodes are mainly containers that run tasks or services. Manager nodes can also act as worker nodes along with its administrative tasks, this is the default behavior, while worker nodes can also be promoted as manager nodes.

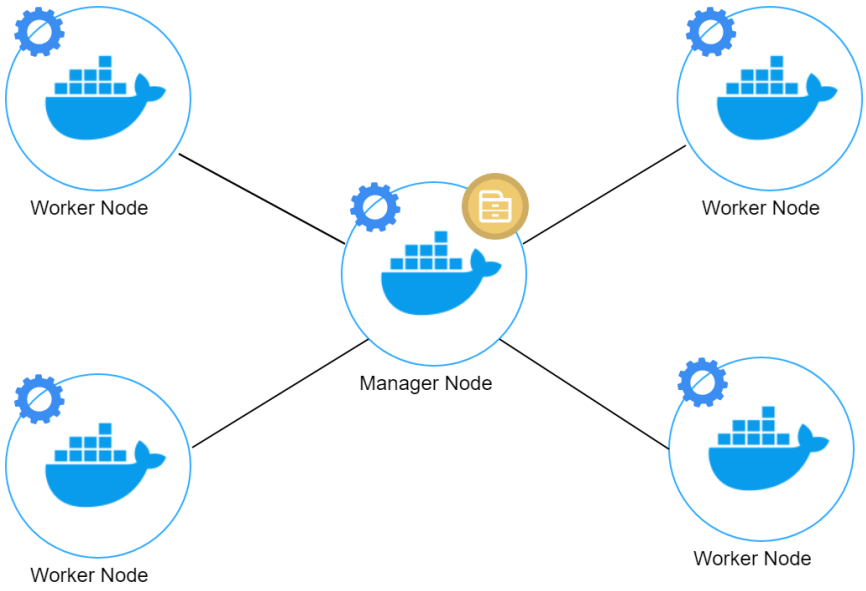

Figure 9. Docker Swarm nodes

There are several things to consider regarding a cluster swarm, such as the correct number of managers and load balancing. According to the characteristics of the network edge, these settings can be customized in order to best fit the requirements. For the swarm application, the Linux machine acts as a swarm manager and the Raspberry Pi devices act as workers, as illustrated on figure 10. Correspondingly, this scenario can be easily scaled-out using more devices as long as they can run Docker in swarm mode.

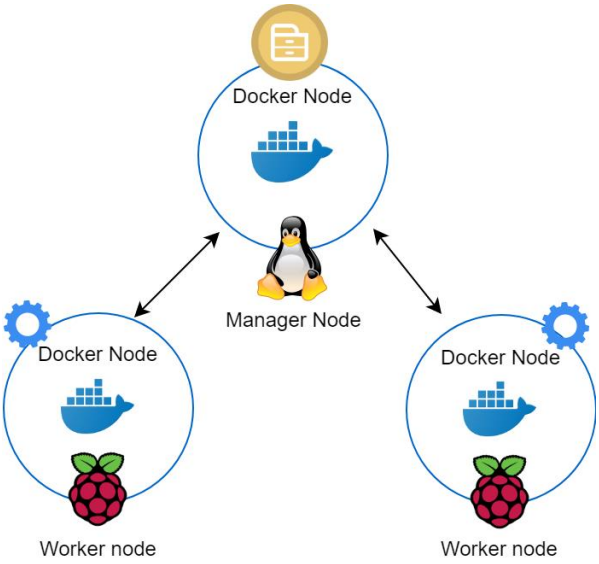

Figure 10. Docker Swarm setup

The benefits of using Docker are put on evidence when trying to add more devices to a system. Whether it is a more powerful device or a resource constraint device, any device, as long as it is capable of running Docker in swarm mode, can join a swarm and become reachable within the network established for the swarm. This facilitates the deployment of applications on multi-node scenarios, where multiple nodes can join or leave the swarm.

3.2.1 Overlay network

Docker swarm creates an overlay network by default sitting on top of the host network. An overlay network helps to bind together the nodes of the swarm creating an internal network for the containers participating on the swarm. All the nodes in the swarm are connected to this network and they communicate using this network, even though externally they both may be in different networks, namely the respective networks of the hosts. Figure 11 illustrates how the overlay network sets a direct communication among nodes inside a swarm, having each one a specific IP address on the overlay network.

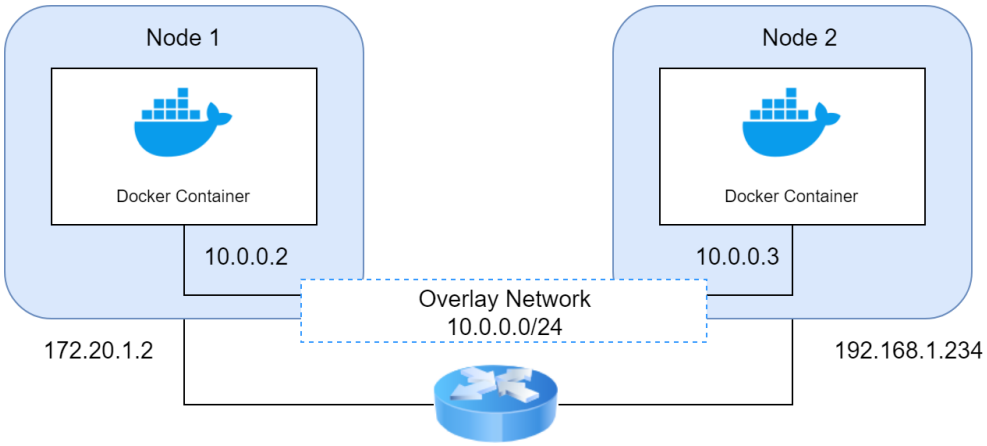

Figure 11. Docker Swarm nodes

To further isolate and keep control of the network, it is better not to use the default overlay network called “ingress”. Instead, it is better to create an overlay network exclusively for the swarm. This can be done externally, meaning creating the network independently and then assigning it to the swarm, or automatically along the definition of the services using a *docker-compose* file.

3.2.2 Docker compose file

The *docker-compose* file is a YAML[4] file that defines all the services, networks and volumes that are going to be used and deployed to the swarm. This file is similar to a *dockerfile* in its syntax, but instead of using it to build an image, the *docker-compose* file is used to bootstrap the swarm.

The following is an extract of the *docker-compose* file used for the second application, which makes use of Docker swarm:

```
version: '3'
services:
 seed1:
   image: marcelodock/akkaswarmarm32v7
   ports:
     - "2550:2550"
   environment:
     CLUSTER_IP: seed1
     CLUSTER_PORT: 2550
     SEED1_TCP_ADDR: seed1
     SEED2_TCP_ADDR: seed2
     ROLE: backend
   networks:
     - akka-cluster
   deploy:
     replicas: 1
     placement:
       constraints: [node.role == manager]
   command: gradle run

networks:
 Akka-cluster:
```

The *docker-compose* file defines the configuration settings for each of the services to deploy on the swarm. In this case, there is a service called *seed1*. The configuration settings for this service are:

image: This service will use a custom image created for the swarm, which will be discussed later. This image contains the application to be run on this node. The image must be available on docker hub so that remote nodes can pull the image.

ports: The ports are defined in congruence with the application ports that are defined in the *application.conf* file on the Akka application.

[4] https://yaml.org/

environment: These are the environment variables that are set for the application. These variables and their values are accessible to the container running the application, therefore, they can be used by Akka for its configuration. All the variables defined are in accordance to what is needed in the Akka configuration for the specific node. In this case, the seeds IP addresses and ports are defined along with the role of the node.

networks: As mentioned before, a specific network is defined for the swarm called: *akka-cluster*. All the services that should join this network should define the name of the network in order to join. The creation of the network is done at the end of the file with the top-level option *networks,*

deploy: Defines options for the deployment of the service. In this case, only one replica is defined with the constraint that it should be deployed on a manager node.

3.2.3 Docker images

Different types of images were created for the different projects; this was required as for the specific characteristics of each of the projects. Giving that Raspberry Pi devices have a different architecture, namely Advanced RISC Machine (ARM), all software associated with the service to be deployed on a Raspberry Pi device must be compatible with ARM hardware. For example, to use Gradle, an ARM compatible image has to be used for the Raspberri Pi devices, while the Linux machine uses the "normal" Gradle version.

3.3 Akka modules

Akka offers a large set of tools to build distributed systems. The selection of which capabilities to use varies depending on the requirements of the applications to build. The network edge faces different challenges in different aspects such as latency, availability and connectivity. In order to provide a suitable application environment, taking into account these characteristics, the following modules were considered for the projects:

- **Routing**: To increase the throughput of the system.
- **Persistence:** To persist data used by the system.
- **Remoting:** To enable the communication of actors on different nodes.
- **Cluster:** To build up a cluster, composed of the different edge nodes.
- **Cluster Sharding:** To load-balance actor across the swarm.

Despite being a simple concept on the surface, the Akka implementation of the Actor Model involves different kinds of concepts to provide a solid and robust distributed framework, such as CQRS (Command Query Responsibility Segregation) and Reactive Programming.

In this sense, Akka provides multiple modules and libraries that can be used alongside each other. The toolkit is large and its use depends on the specific requirements of an application. For the projects developed for this thesis, the previously mentioned modules were selected as they address directly the aforementioned problems regarding the network edge.

3.3.1 Akka application architecture on a Docker Swarm

Following the Swarm architecture provided by Docker, the network edge devices can be mapped to Akka cluster nodes (figure 12) in the following manner:

Manager nodes as Seed nodes or Persistence nodes: Manager nodes are the ones in charge of managing the swarm. These nodes are not simple workers, as they already have special responsibilities within the swarm. Manager nodes can be defined to be special nodes in the swarm. This could mean that these nodes may have better capabilities such as resource-rich

servers and that they could be located and deployed in such form that they are easily maintainable and accessible, and less susceptible to failures or outages. Given this context, these nodes can be a good fit for seed nodes within the Akka cluster. Another good use for these nodes could be for persistence of data of the cluster, for instance, to create a distributed Cassandra cluster.

Moreover, depending on the architecture of the system, micro data centers could be established where all, or most of the servers, would act as manager nodes given the special characteristics of these devices.

Worker nodes as normal Akka nodes: Worker nodes are given tasks or jobs to perform. These nodes can be very heterogeneous, ranging from very resource constrained devices to more resource-rich nodes. These nodes could be set up as normal Akka nodes that perform different types of computations. Depending on the specific requirements of these computations, it is possible to set these nodes to handle only specific types of computations. This can be accomplished by using routers within the Akka cluster.

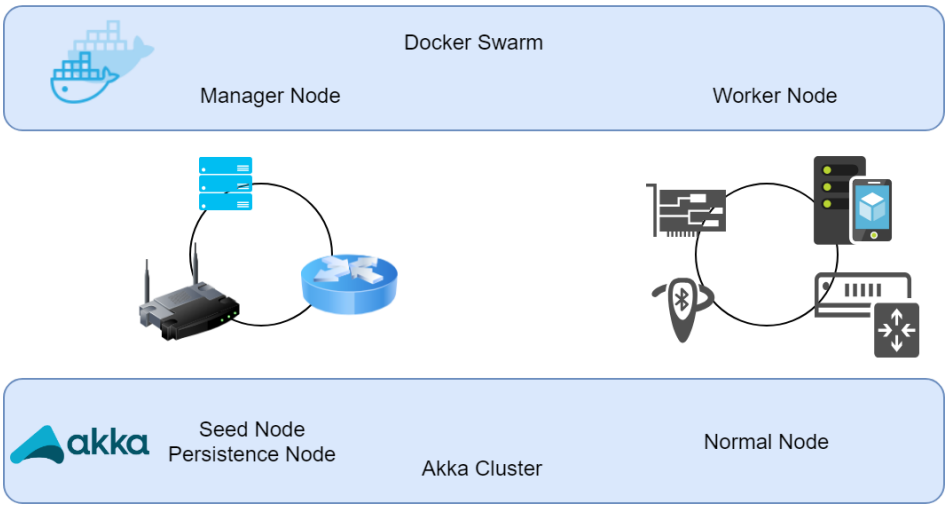

Figure 12. Mapping of Docker Swarm nodes to Akka Cluster nodes

Because of the nature of Akka, based on a peer-to-peer communication, the nodes on the cluster do not need to be different. In fact, a cluster of only Raspberry Pi devices can be established, where any node can act as a seed node, persistent node or a normal node. However, the different the limitations and restrictions of some of the devices on the edge must be considered as some of them are better suited for different roles in an Akka cluster, increasing the availability and scalability of the system among other benefits.

3.4 A simple Akka cluster

The network edge is composed of different interconnected devices. These devices can be grouped in cluster(s), where each node represents a device on the network edge. The decision of how to group nodes could be done in different ways. One of the most obvious ways to group would be to do it by proximity of the nodes. This would be a better fit in cases where low latency is required as less hops would be required for passing data through the network. Another case could be to group nodes by services they provide, in order to have clusters of services. The decision of how to cluster nodes should consider the requirements of the application and taking into account the physical and logical distribution of the devices on the network edge.

The Cluster module of Akka make use of the Remoting module. This is no surprise, as the idea of having a cluster is to group nodes that are on remote machines. The following is a description of the configuration to have a basic cluster running with Akka.

The configuration file *application.conf* is where the application configuration is defined. The configuration file uses the HOCON (Human-Optimized Config Object Notation) format. The format works similar to JSON, as it is a superset of it. All the Akka configuration is defined under the top level key “Akka”. For enabling the cluster mode, the following configuration is required:

```
actor {
  provider = "cluster"
 }
```

For the configuration of the communication protocol and IP, the following configuration options needs to be set:

```
remote {
  log-remote-lifecycle-events = off
  netty.tcp {
   hostname = "127.0.0.1"
   port = 0
  }
 }
```

The important part is the definition of the TCP protocol, where the hostname and port is defined. If running the cluster locally, the hostname can be set to localhost. If running remotely the IP address of the machine should be used instead. The port is also defined. If set to 0 a random port is assigned.

Another important part of the configuration is the definition of nodes for the cluster:

```
clustering {
  cluster.name = ClusterSystem
  seed1-ip = "127.0.0.1"
  seed1-port = 2550
  seed2-ip = "127.0.0.1"
  seed2-port = 2560
 }
```

These variables can be referenced on other places of the configuration. For the application, the cluster name, seed addresses and ports are defined. As mentioned before, this configuration changes when used with remote nodes which will be explained later in more advanced scenarios.

To illustrate how the communication can be done in a cluster two types of actors were designed:

Frontend: These type of actors act as interface between the cluster and the outside, gathering request for jobs and assigning these jobs to the Backend. The cluster application provides the service through these actors.

Backend: These actors are the worker actors in the cluster. They perform some kind of computation, based on the jobs received by the frontend. These actors are the ones that actually do the processing.

This simple application structure, shown in figure 13, has 2 Frontend actor systems, and 3 Backend actor systems. The frontend simulates receiving jobs with a scheduler that creates a new job every second. As there are 2 frontend actor systems, there are 2 jobs created every second.

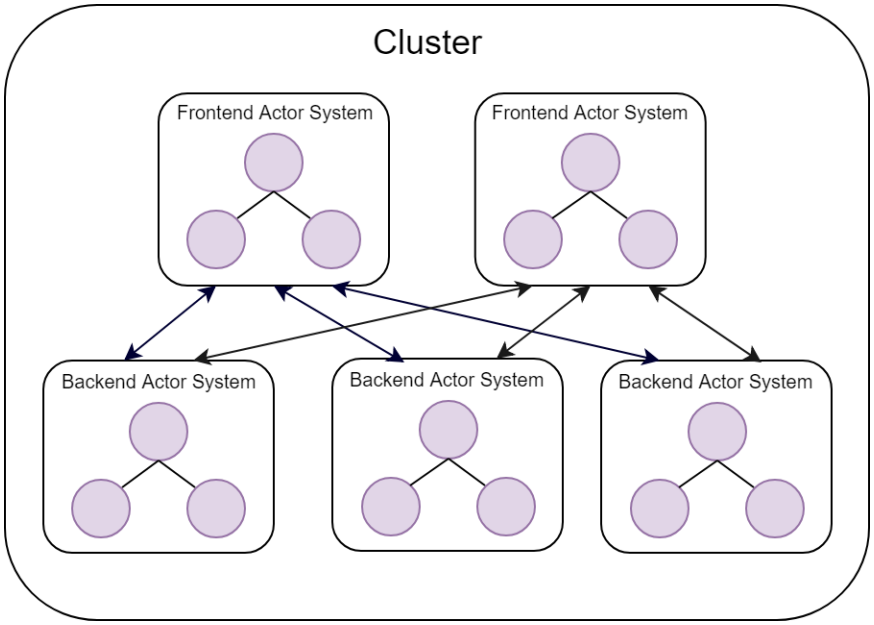

Figure 13. Cluster using Frontend and Backend actors

The sequence of actions in the cluster proceed in the following manner:

1. Frontend receives a job to be done.
2. Frontend checks if there are backend nodes available to assign the job.
3. If no backend node is available, Frontend responds with a message that the service is not available.
4. If there is a node available, Frontend delegates the job to an available backend node.
5. Backend node receives the job, process it, and returns the result
6. Frontend receives the result of the job

There are three types of messages on this application:

- **JobMessage**, which carries the job to be done.
- **ResultMessage**, message that carries the result of the computation
- **FailedMessage**, message that carries information about the failing computation

All the messages are grouped together in an interface called *AppMessages*. This is done in order to have one specific place to look for the messages that are used in the application, instead of having them dispersed on different classes. This kind of pattern is called “Messaging Protocol”, and can be useful when dealing with complex scenarios with multiples types of messages.

The main idea of the basic cluster is to have different actor systems and connect them to form a cluster. The frontend nodes serve as a base to which other applications can be built on top of it. For instance, in the case of a Raspberry Pi, a sensor can be installed on it, and

send the readings of this sensor to the Backend systems for processing and later, when results of the processing are received, act based on the results.

3.4.1 HTTP Management

Akka offers another module to manage the cluster through an HTTP API. This module is useful to see the state of a cluster using a web browser. Among other things, the API allows to:

- List all nodes on the cluster
- Join a node to the cluster
- Put down a node from the cluster
- See the state of a specific node

This module can be useful to develop web applications to query the state of a cluster using the API. This would allow to have a bit of control of the cluster without using terminals or having deep knowledge of the inner workings of the cluster. The HTTP management module is used in this simple cluster application. It is started on one of the backend nodes that act as a seed node of the cluster.

3.5 Cluster-aware routers with Docker

The previous setup was used to illustrate the basic configuration of an application using Akka. The next step is to use Docker to run the application. A new concept is introduced in this next version of the application: Cluster-aware routers.

With cluster-aware routers it is possible to deploy routees in other nodes of the cluster, as shown in figure 14. This increases the availability and scalability of the application. There are different ways of deploying the routees. For this application, a pool router actor is used, so that it can handle itself the creation of its routees.

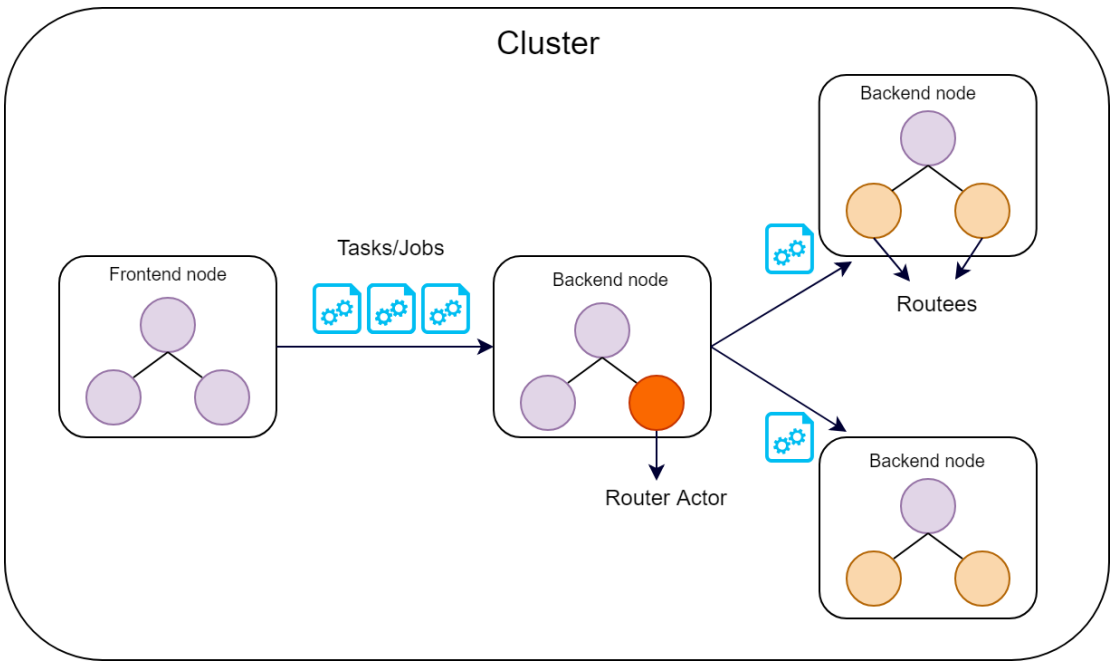

Figure 14. Cluster-aware routing with remote deployed routees

To explain how this application works a brief description of the components and their inner workings are described.

Frontend node receives jobs and send them to the router. The router is in a different node as shown in figure 14. The Backend node containing the router actor receives the task, but does not perform the computation itself. With the help of Akka, the router has deployed "routees" to other nodes. The router actor sends the task to one of the routees. There are several options and parameters to configure the process of routing the messages to the routees, defining specific routing logics. For the application, the "RoundRobinRoutingLogic" is used. Akka takes care of the deployment of the routees.

The routees are the ones of doing all the processing. Routees are created as simple actors on the remote nodes under the "remote" path of the actor system, indicating that these nodes have a remote supervisor, which is the router actor residing on a remote node. Routees reply directly to the Frontend instead of the router, but will set themselves as senders. However, it is also possible to hide the fact that routees are used by setting the router as the sender of reply messages. In this case the Frontend only sees the Router actor as the one doing all the job.

The cluster awareness come from the fact that it is possible to configure a router so that it automatically deploys routees to new nodes joining the cluster. This concept also takes into account the scenario where a node containing routees may exit the cluster or when a node in the cluster may become unreachable, in which case the routees of that node are no longer reachable and are removed from the router automatically, so that the router will no longer route to those routees.

Using this model, it is possible to perform several computations on different nodes. It can be used to implement a master-workers pattern kind of application, where a complex problem can be divided in small sub-problems and these can be processed independently and in parallel.

This application runs locally using Docker. The *docker-compose* file is defined with three services:

- seed
- backend1
- frontend

The seed node has also a backend role, so in total there are: 1 frontend and 2 backend nodes. The local deployment is done with only one command: *docker-compose up*. This simplicity of defining different services through one file, demonstrates the benefits of using Docker for managing the deployment of an application.

3.6 Akka cluster in a Docker Swarm

With the services defined and all the configuration of the application in order, it is easy to take the next step and deploy the application on a Docker swarm.

Docker needs to be running in *swarm mode* in all the nodes that are going to be part of the swarm, and eventually part of the Akka cluster. Initially, a node is chosen as the first manager of the swarm. After the swarm is created, other nodes can join as worker nodes, and if desired, some worker nodes can be promoted to managers.

With a swarm environment created, it is possible to deploy the Akka application using the swarm environment created (figure 15), in which all the services of the application can be distributed to the different nodes of the application.

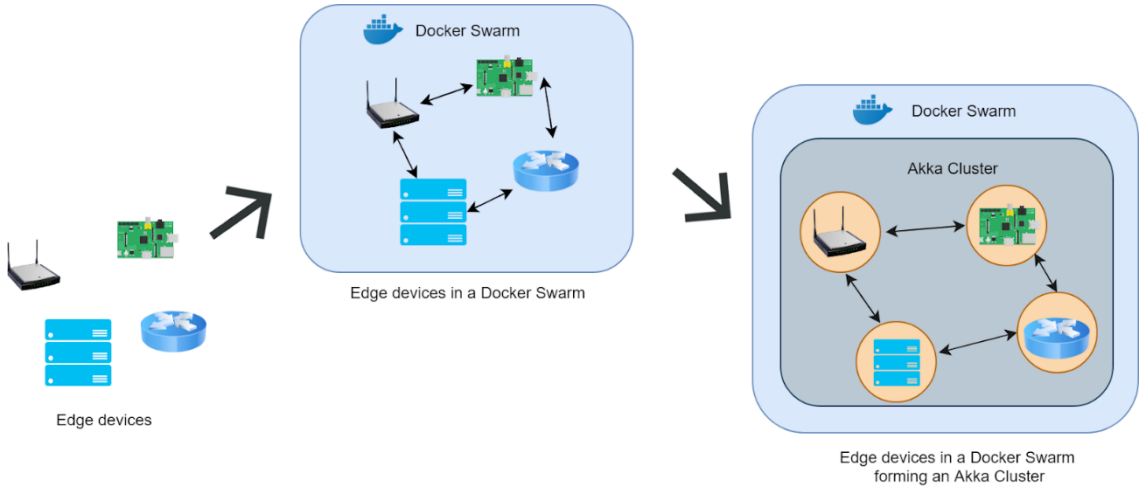

Figure 15. Edge devices forming an Akka Cluster in a Docker Swarm

All the components that are part of the application, such as the frontend and backend systems, must be defined in the *docker-compose* file as services (each system is a service). All the parameters that each system needs in order to initialize the system must be defined in the *docker-compose* file as environmental variables. Later, these variables can be accessed on the specific configuration files of the application, such as the *application.conf* file, or any custom configuration file. There are no further changes on the code. This is achieved thanks to the location transparency of Akka, which always assumes that the application lives in a distributed environment.

One of the major benefits of using Docker to create the swarm environment is that the network connection details are handled automatically by Docker. For instance, on the *docker-compose* file, it is no longer necessary to know the specific IP addresses of the devices and statically bind them to the services, although this is possible as well. This allows for edge devices to form a cluster with no necessity of setting up complex networks configurations. With this robust environment it is possible to do more complex things in Akka such as Cluster Sharding.

3.7 Cluster Sharding and persistence with Akka

Further down the path of distributed systems is the concept of Sharding. This concept is usually associated with databases, in order to separate data in shards and distributing these shards of data across different nodes. Akka takes this idea and applies it to the Actor model distributing live actors across the nodes of a cluster.

This scenario changes the way applications are conceived. First of all, all the nodes on the cluster must create a *ShardRegion* actor. This actor is the one in charge of all the sharding activities on the actor system where it is created. All the messages directed to an actor on a shard, called entity actors when defined on shards, must go through the *ShardRegion* actor. Second of all, there is no need to create the actors “manually” in the shard. At the moment of creating the *ShardRegion*, a props object is passed so that it can create entity actors of that class. The creation of actors by the *ShardRegion* is on demand. When a message is send to an entity on the shard, the *ShardRegion* will determine if it already exists, if not, a new entity actor will be created.

Another important aspect that goes along with the concept of Sharding is the persistence of the entity actor’s state. This is because of the ability of rebalancing and migrations of the shards. These processes include the termination of actors in one shard, of one node, and

migrating them to another shard on another node. This implies that if an entity had some kind of state, this state will be lost, as the entity will be recreated as a new entity in the new shard location.

Akka has a specific persistence module that allows to keep the state of an actor. This module enables to preserve the state of an actor using journals. Akka make use of the CQRS (Command Query Responsibility Segregation), which makes a distinction of what is saved. Akka journals do not store commands, as they can be invalid. Only valid commands are persisted on a journal in the form of events, that are defined as actions that have occurred. These events allowed the actor to go from one state into another. Once the journal has persisted all the events for an actor, they can be replayed in case the actor needs to start from zero. Furthermore, the concept of persistence is not only attached to Sharding, and can be used in any type of scenario where it is required to keep state of an actor.

The persistence module is used on the IoT application in next chapter in order to persist the state of the Master Cluster. The sharding module is also used in the same application, but in a different cluster. In this case, the sharding mechanism is used in order to distribute actors across the Worker Cluster. The details of this implementation and other important considerations are presented in detail in the next chapter.

3.8 Summary

Two different applications were presented on this chapter. Different Akka modules were used in these applications in order to demonstrate how it is possible to construct basic edge applications using Akka. On the next chapter, a more complex application is developed, using an IoT scenario where an edge architecture is proposed to handle the processing of data as opposed of a cloud-centric approach.

4 IoT System scenario

In order to apply the different concepts discussed on chapter 3, an Internet of Things scenario that relies on cloud computing is used as a base architecture, so that later it can be modified so as to work on the network edge with the help of Akka's toolkit and using the Actor model as an application model.

4.1 Cloud and edge architectures

The base cloud architecture was taken from a self-managed architecture for Wireless Sensor Networks (WSN) proposed by G. M. Dias et al. [17]. This architecture considers the hardware limitations of wireless sensors as well as the communication of the WSN with other components in the architecture. Moreover, the proposed architecture defines specific components to perform real-time data analysis, using data collected from the sensors, so that later the processed data can be used by the WSN to self-manage, and also provide access to this information to interested parties over the internet.

The architecture of this scenario consists of the following main components:

- **Cluster of wireless sensor nodes:** cluster that contains a group of sensors that communicate with the outside via a Gateway.
- **Gateways:** mediators between the WSN and the cloud. They forward the data collected from the sensors to the outside and also receive instructions and new data targeted to the sensors.
- **DAS-dashboard:** This is the main component of the architecture, in charge of collecting, storing and publishing data transmitted by the WSN. It serves as a provider of raw data for interested parties, such as the WSN owners, or to other services such as the processing component.
- **Data Analytics Server (DAS):** Component in charge of the real-time data processing. It acquires the data from the WSN through the DAS-dashboard and can communicate with other services on the internet to gather more data for its processing.

The main limitation of this model is that it relies on cloud services for real-time data processing. The DAS and the DAS-dashboard are considered as cloud services that may be located in distant locations which most likely increase the round-trip time, which can be crucial for real-time applications. Moreover, it does not consider the computation power on the proximity of the data, namely edge devices, that can process the data without using the distant cloud.

Assuming that Gateways on the edge are powerful enough to carry out the processing of the data, and that there are enough of them, the whole process, or at least the most critical processing, can be performed on the network edge. This transformation can be done efficiently using Akka to build an application that work across the multiple gateways on the edge, as illustrated on figure 16.

The proposed Akka architecture consist of the following components:

- **IoT Cluster:** basically the same as the base architecture. This cluster has a manager or Cluster Head, in charge of receiving all the data from the sensors and passing it to the master cluster.

- **Master Cluster**: a cluster that acts as a main controller of the whole application. This cluster replaces the DAS-dashboard. Among its main functions are to receive the data from the IoT cluster and send it to the Worker cluster for processing, and to send the results of processing back to the IoT manager. It also maintains a record of all the events in a distributed journal as well as any other relevant data that can be accessed or queried by interested parties.
- **Worker Cluster**: a cluster in charge of the real-time data processing. This component replaces the DAS. This cluster distributes all the workload of processing the sensor data among all the nodes in the cluster.
- **A distributed persistence journal**: A cluster that acts as a data storage, which is implemented using a multi-node Cassandra on a data center. This cluster can be implemented using the gateways on the edge or other external data centers.

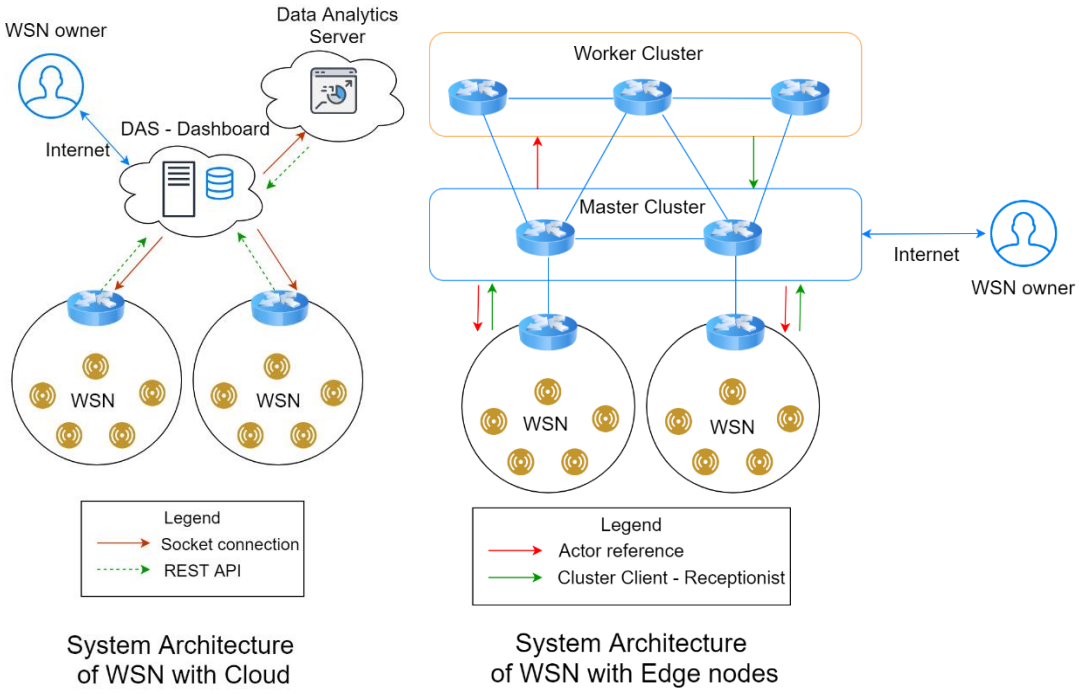

Figure 16. Cloud and edge IoT architectures

The Master and Worker clusters are composed of gateways on the network edge, while the IoT cluster consist mainly of sensors. The clustering of edge devices can be done based on location proximity, capacity of the corresponding gateways, or any other suitable criteria that can leverage the architecture. Communication between the clusters is done via Akka's Cluster Client mechanism, which allows for actors of different systems and external clusters to communicate with each other. In the proposed architecture, the IoT cluster and Worker cluster are defined as *cluster clients* whereas the Master cluster is defined as a *cluster receptionist*. A more detailed view of the proposed Akka architecture is shown on figure 17.

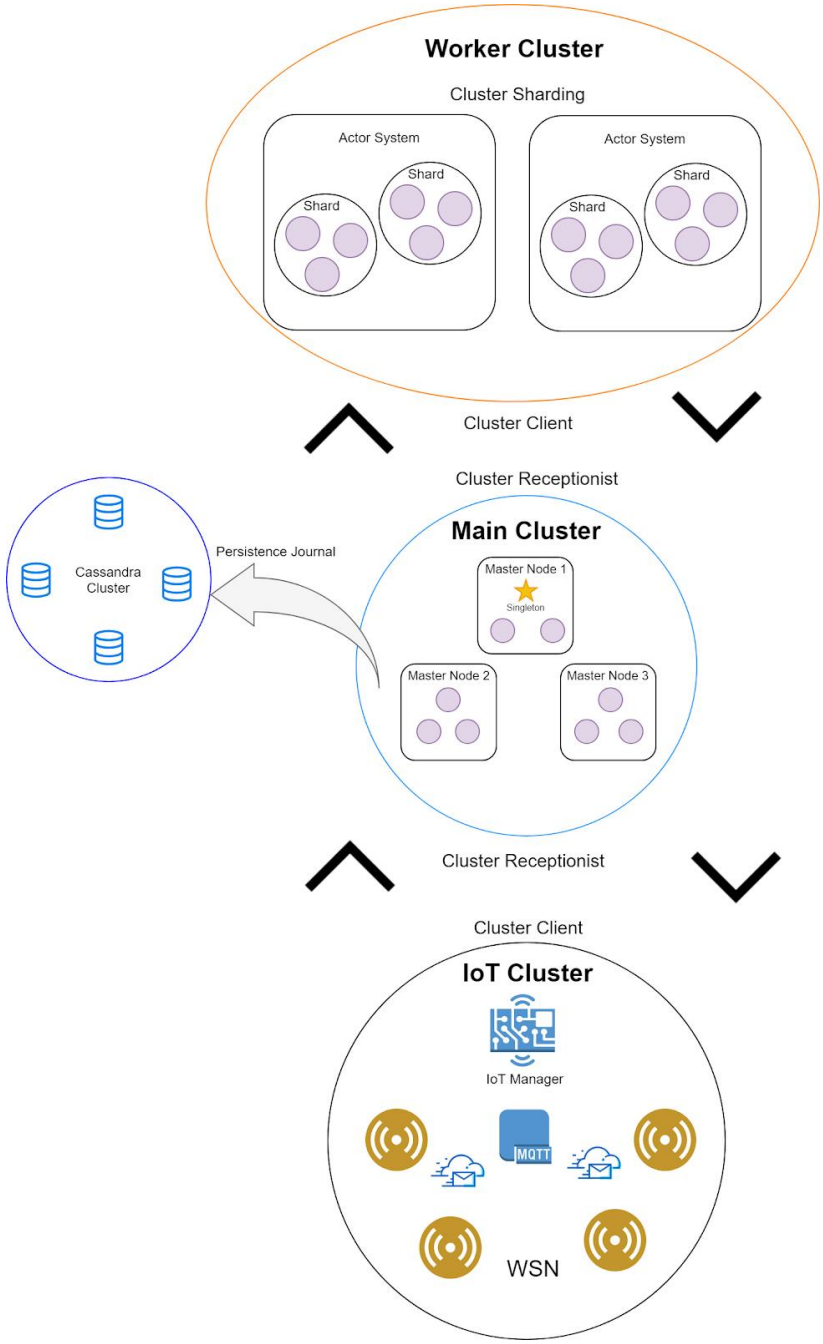

Figure 17. Akka IoT system architecture

4.2 IoT Cluster

Different research on the field of IoT, focus on the problem of how to deploy and distribute sensors on the network [18] in order to deal with different challenges such as low energy consumption and efficient communication between the devices. Most of the research on the topic propose schemas and algorithms based on a clustering topology to face these challenges. Taking this into account, it results of no surprise that Akka uses the concept of clustering at its core to provide a reliable architecture to build distributed applications.

Having said that, the proposed Akka solution for the IoT cluster is modeled using this concept as well, grouping all the sensors of the system under a cluster, or possibly multiple

clusters if desired. The sensors are modelled as any type of sensor that is capable of generating some data, i.e. temperature, humidity, pressure, and also capable of modifying its state/settings according to data that are received through the manager of the cluster. The IoT manager in the cluster works as a Cluster Head [18] [19], which serves as a gateway between the cluster and the external world, gathering data to and from the sensors on the WSN. The sensors are represented as actors within the IoT cluster.

The sensor data consist of unique information of the sensor, such as the sensor Id, and the readings from the sensor which is represented as an array of values. This sensor data is published using the MQTT (Message Queuing Telemetry Transport) protocol, which is a lightweight messaging protocol that works well in resource constrained devices. This protocol uses a publish-subscribe model that allows for communication between multiple devices. In the cluster, the sensors publish the generated data to a specific topic, to which the IoT manager is subscribed. Figure 18 illustrates this process. Once a message is received by the manager, it is sent to the Master cluster for further processing. In order for the publish/subscribe model to work, a MQTT broker has to be set in order to handle all the reception and delivery of messages. The open source broker server *Mosquitto* is one of the most common brokers in use for the MQTT protocol, and it is used as the message broker for the application.

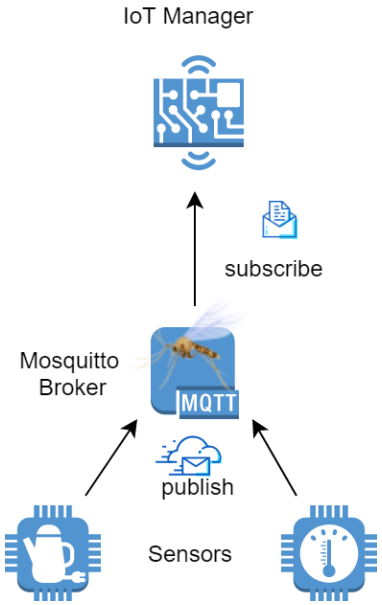

Figure 18. Publish-subscribe with MQTT broker on the IoT Cluster

4.3 Master Cluster

The Master cluster receives all the sensor's data coming from the IoT manager. The logic of how this data is going to be processed is handled by the Workers on the Worker cluster. The Master cluster has a persistent state, with the sensor's data received from the IoT manager. This state is persisted on a distributed journal so that it can keep track of all the sensor data that it handles.

The journal to store the state of the Master uses an Apache Cassandra database. The Cassandra database can be modelled as a multi-node datacenter in almost the same fashion as an Akka cluster. The resemblance of the configuration and concepts behind the creation of a Cassandra cluster, makes it ideal to use it in the context of a distributed application, as it follows the same philosophy of working in a distributed environment.

The persisted state can include different data. For instance, specific data about the sensors and their readings can be persisted, so that later this information can be send to the cloud for further analysis. In this sense, the Master cluster acts as a gateway of the whole application, in case it needs to communicate with other applications, systems or other interested parties. The simplified model of how the sensor data is handled by the Master cluster is shown figure 19.

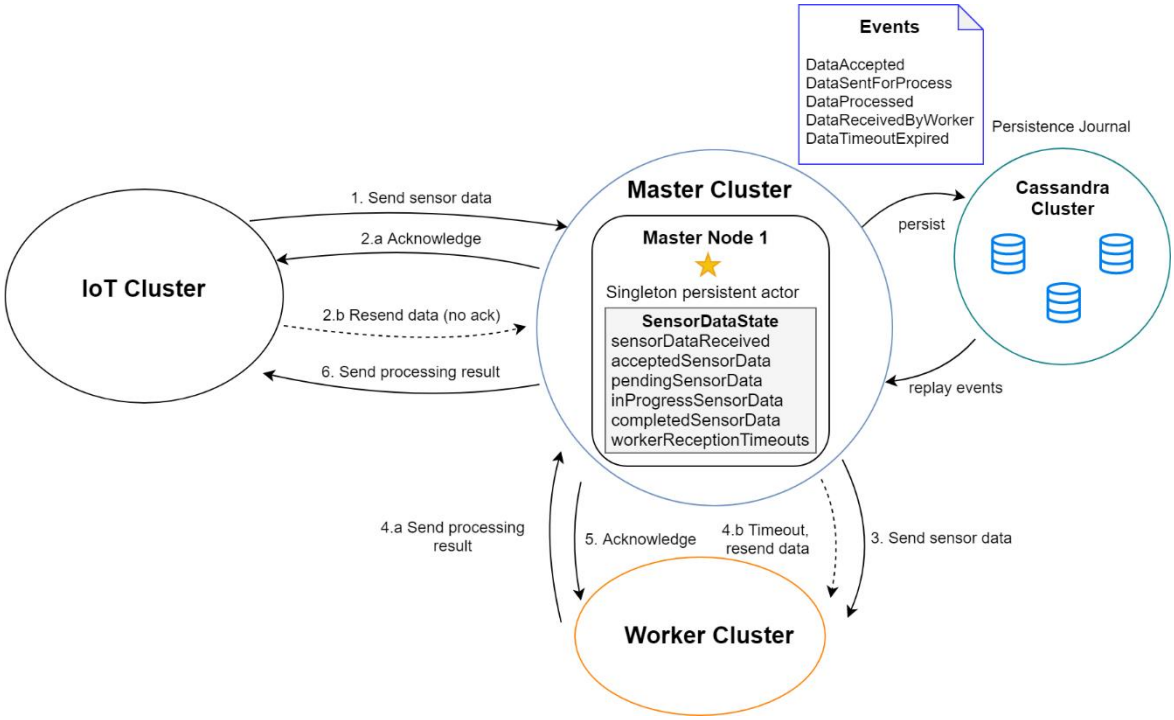

Figure 19. Sensor data handling process by the Master cluster

The whole process starts when the Master cluster receives a sensor data form the IoT cluster. The reception of the data is confirmed with an acknowledgement and the data acceptance is persisted on the state of the cluster. The handling of the state is described with more detail later on this section.

Next, the data is sent to an available/reachable Worker node on the Worker cluster. When a sensor data is sent to the Worker cluster, the Master cluster sets a timeout for the reception of the data by the Worker cluster. This is done in order to guarantee the processing of the data. In case no acknowledge is received, the data is sent again. Once the processing of the data is completed by the Worker cluster, the result of the processing is sent back to the Master cluster, and then the Master cluster forwards the result back to the IoT cluster.

Throughout this handling of the sensor data, the singleton actor of the Master cluster keeps its state in a state object called *SensorDataState*, which is persisted in the distributed journal, namely the Cassandra cluster. There are five types of events that are persisted:

- **DataAccepted**: When a sensor's data has reached the Master cluster.
- **DataSentForProcess**: When the data has been sent to the Worker cluster.
- **DataReceivedByWorker:** When the data has been received by the Worker cluster.
- **DataProcessed:** When a work result is received from the Worker cluster.
- **DataTimeoutExpired:** When no acknowledgement of the reception of the sensor's data by the Worker cluster is received in a determined amount of time.

These events are persisted when the singleton actor receives a message corresponding to the mentioned actions, either from the IoT cluster or from the Worker cluster. A special case is

the timeout event. For this event to happen, there is no response from the Worker cluster, that is to say, no message needs to be received for the event to happen. To handle timeouts of sent data, a specific map is maintained, with the data identifier and the timeout for each data message. A special task is scheduled to go through this map over a certain period of time to make sure that the data has not overdue its timeout. If it has, it will persist the event *DataTimeoutExpired* and will proceed to send the data again to a new available worker.

In order for the *SensorDataState* to keep track of all the data, it makes use of different collections: *sensorDataReceived, pendingSensorData, inProgressSensorData, acceptedSensorData, completedSensorData* and *workerReceptionTimeouts*. These collections are self-descriptive. The data state is updated whenever an event has happened. These events are persisted so that they can be replayed in case of recover of the master singleton in the Master cluster.

A simple case of how the update process works is explained: when a particular sensor data has been accepted it is registered in the *sensorDataReceived, acceptedSensorData* and *pendingSensorData* collections. Then later, when the work has been started, it is removed from *pendingSensorData* and added to *inProgressSensorData* collection. Finally, when the processing of the data is completed, it is removed from the *inProgressSensorData* and added to the *completedSensorData* collection. The *sensorDataReceived* collection stores all the sensor data received by the Master cluster. This collection serves as historical data and can be used as source of information for further analysis. The other collections are used as control mechanisms for data handling.

The master cluster uses a Cluster Singleton, which is a mechanism provided by Akka that enables to have only one specific instance of an actor among all the nodes of the cluster. This is done in order to have only one source of truth in respect to the handling of the messages. In this sense, multiple nodes can be added to the master cluster in order to increase the availability of the singleton actor. For instance, in the case where the node where the singleton actor lives is taken down, the singleton actor will migrate to another available node in the cluster. With the persistence facility enabled with the Cassandra cluster, the state of the singleton actor can be recovered by replaying all the stored events or using snapshots, which will bring back the new singleton actor's state to its previous state before the failing of the original one.

4.4 Worker Cluster

The Worker cluster is the processing cluster. It receives sensor data from the Master cluster, process it, and then returns the result back to the Master cluster. The Worker cluster has one specific duty which is to process sensor data received from the Master cluster. This separation of concerns, in terms of clusters, can be useful on the edge, where a group of powerful edge devices can be grouped to form a processing cluster, capable of processing information as fast as possible.

Worker nodes in the cluster are registered to the Master cluster so that the Master can send the sensor's data to the workers. The Worker cluster is configured as a Cluster Client in the same way as the IoT cluster, so that it can send messages to the external Master cluster.

The Worker cluster is modelled using the concept of Cluster Sharding. The idea is to distribute the work processing evenly among all the nodes in the cluster. In order to accomplish this, all the worker nodes have a worker region, which is the shard region to which all the work messages must go through before arriving to the corresponding work entity actor that lives within a specific shard (actors within a shard are also called entities). The series of steps of how sensor data is handled are shown in figure 20. Some communication details

such as acknowledgment of workers and shard communications are omitted for the purpose of maintaining the simplicity of the diagram.

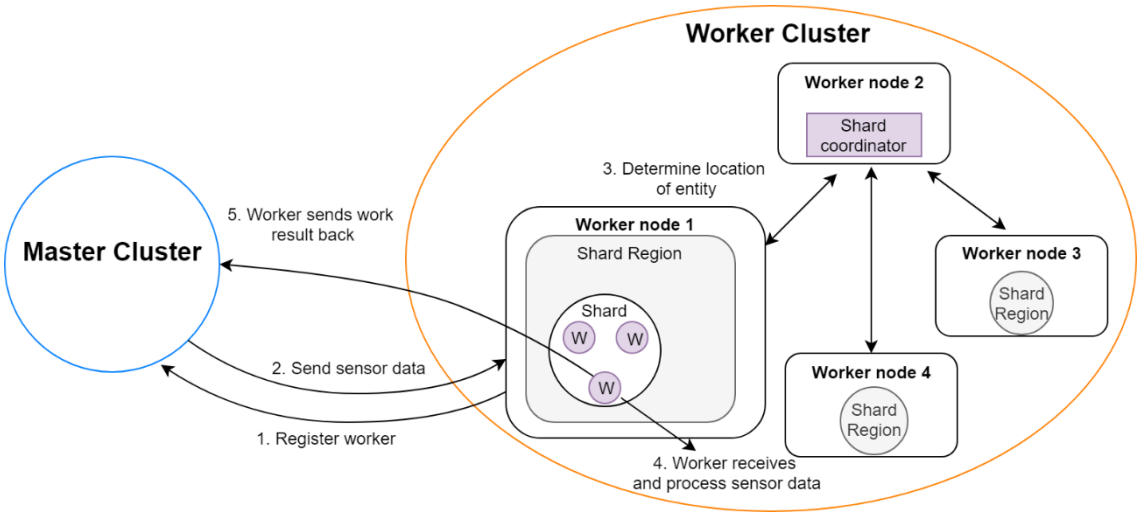

Figure 20. Handling process of sensor data between Master cluster and Worker cluster

On the first step, when a worker node is started, it sends a message to the Master cluster with an actor reference of the Worker system that lives in the node. The Master cluster keeps a list of Worker systems so that it can later send works to them. On the second step, once a Worker system is registered, the Master cluster, through its cluster singleton instance, sends sensor data to a registered Worker system. The third step deals with the distribution of the work between the Worker nodes, this aspect is handled by the Akka sharding mechanism, with the help of a message extractor, which is described with more detail later. On the fourth step, a Worker entity receives the sensor data and perform the required computation with a *WorkProcessor* that uses the Routing mechanism to distribute the processing among all nodes of the cluster. The final step is carried out once the work is done processing by the Worker, in which case it sends the result back to the Master cluster. It is also important to mention that mechanisms to handle failures are also implemented using a supervision strategy on the child *WorkProcessor* routees.

A worker message extractor is defined in order to extract the messages as well as the entities and shard ids. The *entityId* corresponds to the *sensorId* of the data that is going to be processed. This means that an entity within the shard will correspond or represent a sensor from the IoT cluster. The *shardId* is determined based on the hash code of the *entityId* with the modulo operation. This is done in order to properly distribute work among the shards. Using the hash modulo operation is a safe bet in general cases and it is recommended by Akka. As per the message, it is just passed as it is. Messages going through the shard regions carry the sensor data, so no further change is required. On a side note, Akka documentation remarks how difficult and challenging can be to define a good sharding algorithm, hence, using the best practices is the best way to go for general cases.

As mentioned before, the Worker actor uses the Akka's Routing mechanism to distribute the processing of the data, which consist of different forecasting tasks. In order to do so, a Router actor is created in each Worker node. This Router actor, called *WorkProcessor*, is a self-contained router that contains the logic to do the forecasting task. The *WorkProcessor* is created as a child of the *ShardRegion* actor. Having the *ShardRegion* actor as a parent allows to handle exceptions produced during the processing of a work through a supervision strategy.

The processing task is to compute a forecast model using the ARIMA (Autoregressive Integrate Moving Average) method, which is a commonly used technique for generating forecasts on the WSN [20] [21] [22]. The idea is to have good forecast models to reduce the frequency of readings and transmission by the sensors in order to reduce the energy consumption of the devices and prolong its lifetime. In the application, the *WorkProcessor* uses the java library "java-timeseries"[5], which can compute ARIMA models using sensor's data as input. Several models are computed in order to choose the one that fits the best for each sensor data. The computation of these models can be done in parallel as they only depend on the initial sensor data. Later, when all models are computed, they can be compared using different indicators such as the AIC (Akaike Information Critera).

Given than this process can be computed in parallel, the application uses Akka's routing mechanism to split the processing work into tasks, where each task corresponds to the computation of one specific ARIMA model.

Once the processing of the data is completed, the result is sent back to the Master cluster using the cluster client mechanism. Results reception is also handled. The Master cluster must send an acknowledgment that the result is received. In the case that the acknowledgement is not received in a given period of time, a new message with the result is send to the master until it acknowledges the reception of the result.

4.5 Application workflow

With all the components properly described it is possible to explain how the whole process works within the application. A simple case, for the processing of one sensor data, is described in detail in this section. The graphical workflow is shown on figure 21.

1. A sensor actor in the IoT Cluster generates data readings to be processed. These readings consist of an array of values. This array of values is put in a "envelope" called *SensorData*, which includes other relevant information with respect to the data readings, such as the *sensorId.*
2. The sensor publishes the *SensorData* to the corresponding MQTT topic on the IoT Cluster.
3. The IoT manager, which is subscribed to this topic, receives the *SensorData* and proceeds to send the *SensorData* to the Master Cluster.
4. The Master Cluster, through its singleton actor, upon reception of the *SensorData* adds the *SensorData* to a queue of sensor data waiting to be send to the Worker Cluster.
5. The Singleton actor takes the *SensorData* from the queue (assuming it was the only element in the queue, otherwise it would wait accordingly), and sends it to an available Worker node in the Cluster Worker.
6. The Worker node, through its *ShardRegion* actor, receives the *SensorData* and extracts the *entityId*, *shardId* and *message* from the *SensorData* using the *MessageExtractor.*
7. The *ShardRegion* actor, with the help of the information of the *entityId* and *shardId*, proceeds to forward the *SensorData* to the corresponding Shard. The Shard contains all entity actors that correspond to each sensor. In this case, the corresponding sensor entity actor, called *Worker*, receives the *SensorData.* The location of the entity actor can be in the same node that received the message or in another node in the cluster. The owner of this information is the *ShardCoordinator* of the cluster which resolves these concerns.

[5] https://github.com/signaflo/java-timeseries

8. The *Worker* entity receives the *SensorData* and starts the handling of the forecasting tasks. First, it creates an Aggregator actor to aggregate the results of the forecasting tasks. Then, it starts sending all the forecasting tasks, one by one, to the *WorkerRouter* actor. The *WorkerRouter* routes each forecasting task to one of the deployed routees of the router, distributing all the computing tasks of the *SensorData* among all cluster nodes.
9. The *WorkerRouter* route receives the *SensorData* and proceeds to do the corresponding computation task. Once it is done with the processing, it sends the results to the Aggregator actor.
10. The Aggregator actor, receives one by one the results of the forecasting tasks. Once the last one is received, it sends the final result back to the *Worker* entity actor.
11. The *Worker* entity, upon receiving the final result, sends the result back to the Master Cluster.
12. The Master Cluster, through the singleton actor of the cluster, receives the result and publish it to a specific topic using a *mediator* actor that handles the publish-subscribe mechanism in the cluster.
13. A *ResultProcessor* actor, that is subscribed to publish-subscribe topic, receives the result and sends it back to the IoT manager.
14. The IoT manager receives the results and send it back to the appropriate *Sensor* that created the data.
15. Finally, the *Sensor* receives the results of the processing.

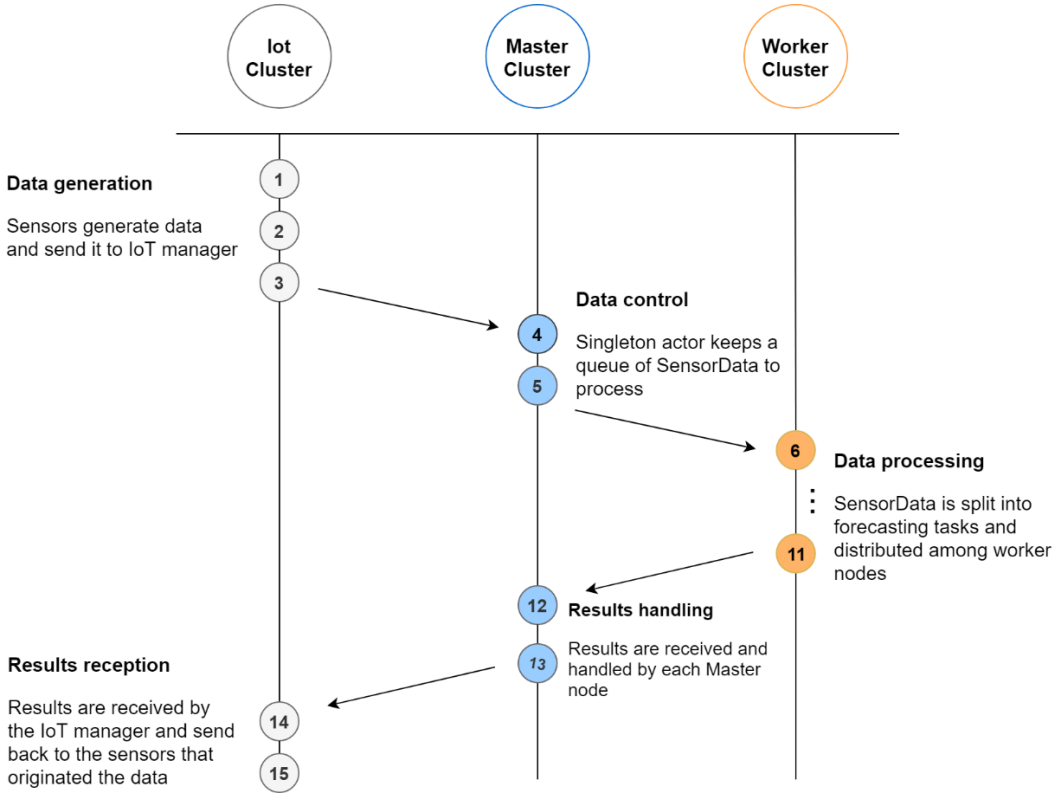

Figure 21. Workflow of the Akka application

The workflow of the application is relatively complex, as evidenced in the series of steps mentioned above, as many aspects are handled by the application using different Akka modules to achieve high throughput in delivering and processing messages, and to optimize the available processing power of all the nodes.

4.6 Deployment of the application with Docker

The deployment process is relatively simple when using Docker in a Docker Swarm environment. With the Akka application developed and properly configured, it is only necessary to define a *Dockerfile* that will take the application code and copy it to a location in the container, and finally execute the corresponding command on the container. The following is the *Dockerfile* used for the Raspberry Pi:

```
FROM arm32v7/gradle
COPY --chown=gradle . /akka/cluster/app

WORKDIR /akka/cluster/app
```

The base image will change depending on the underlying architecture of the device. In this case an ARM compatible Gradle base image is necessary, given the characteristics of the Raspberry Pi. The COPY command changes the user of the file to copy to gradle and copy the application into a specific location in the container. Finally, the working directory is changed to that location in order for the application to be executed directly from this location. The *Dockerfile* allows to easily create images according to the hardware architecture of the device, changing only a few values, so that later it could be used to deploy multiple services with it.

Once the Dockerfile is properly configures, it is necessary to build the images and publish them in Docker Hub, so that the images can be available to remote nodes.

Finally, in order to deploy the whole Application Stack to the Docker Swarm, it is only necessary to use the following command:

$ docker stack deploy –compose-file docker-compose.yml akkaclusterswarm

This will deploy the full stack, defined in the *docker-compose* file, to the Docker Swarm.

The series of steps to deploy the Application Stack can be summarized as follows:

1. Install Docker on all devices.
2. Initialize a Docker Swarm, creating the first the manager node, and then adding one by one each of the worker nodes.
3. Build the corresponding Docker images. These images vary depending on the hardware architecture of the devices.
4. Publish the Docker images to Docker Hub.
5. Configure the *docker-compose* file according to the required configuration of the application. For example, for the Worker cluster, each worker must be defined as a service, using different environmental variables to set the addresses, ports, and other configuration details used by Akka to form the cluster.
6. Deploy the application using the *docker stack deploy* command

Figure 22 summarizes the whole deployment process using Docker, starting from an "empty" Docker Swarm and ending with the Application Stack deployed on the Swarm.

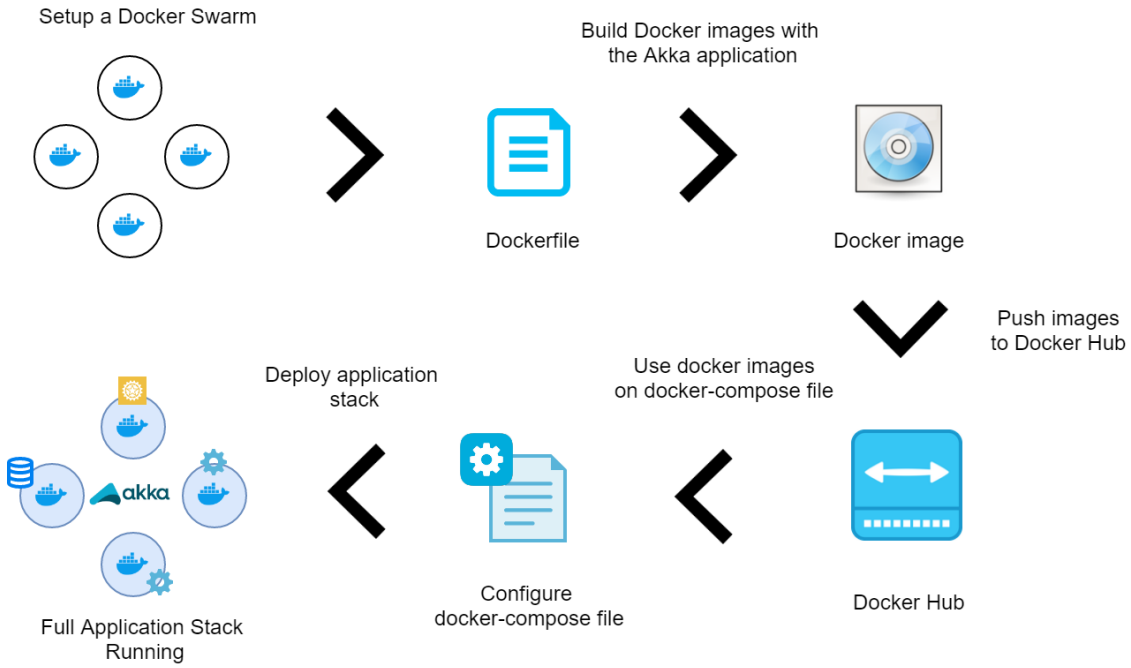

Figure 22. Deployment process

After the Application Stack is deployed, all the services will be allocated to the corresponding Swarm nodes as defined in the *docker-compose* file. All the nodes will start their corresponding Akka systems. This process could take some time, especially in resource constraint devices. This is because Docker first needs to download the image from the public repository, then, once the image is downloaded, start the container with the image, and finally start the Akka application on that container.

As each node starts their corresponding Akka system, they will start forming the corresponding Akka clusters and start providing the corresponding functionality. This process can become cumbersome as some nodes can start faster than others generating bottlenecks or other cluster related issues. Thankfully, Akka provides a configuration mechanism to start an actor system only after the cluster has reached a certain size. This is done defining the minimum number of members of the cluster, which delays the initialization of the actor system on each of the nodes of the cluster, until a specific number of members have joined the cluster.

Using Docker to create and manage the Swarm environment, allows to use several distinct actions on the Swarm, such as restarting services, adding nodes, scaling specific services, etc. What is more, with Docker handling the deployment of the Application Stack, there is no need to deploy and start each system individually, which can be troublesome on the edge giving the amount of devices and their characteristics.

4.7 Summary

On this chapter, an IoT Edge architecture was proposed, as opposed of a cloud-centric approach [17], using the previous applications as guidelines to model a more complex and robust distributed application that can be deployed on the network edge, maximizing the use of the available computation power of the different edge nodes. Docker was used as the main tool for creating and managing the connectivity between the nodes, as well as for deploying the Application Stack to the network edge.

5 Evaluation

Based on the IoT scenario proposed in the previous chapter, the application is evaluated according to the research goals defined for this thesis.

5.1 Feasibility of the Actor model on the edge

First is the aspect of using the Actor model on the edge. Applications on the edge require working in a distributed environment, with all its implications, i.e. network failures, latency, etc., where multiple devices need to be connected, constantly sending and receiving data between each other.

These types of requirements can be quite challenging, especially considering the actual models used to conceive applications, which generally use object oriented programming or other types of paradigms. The Actor model, on the other hand, works naturally in this type of environment, where multiple actors, distributed in nature, can interact with each other.

Actors are by nature independent units of computation. Actors do not share their state and work in a single-threaded "illusion" which facilitates the idea of working concurrently with multiple actors processing information at the same time with no necessity of synchronization or locks. Working with these conditions, or perhaps restrictions, help to visualize how distributed applications can work and be maintained.

For instance, In the proposed scenario, the Worker cluster is conceived as a multi-node cluster where each node can hold multiple actors, distributed using cluster sharding. Each worker actor on each shard can process work messages independently and concurrently. In this context, it is easier to think about scaling the cluster, adding new worker nodes, without modifying the underlying architecture or even the code behind these actors. Thinking about these requirements from the beginning not only help to avoid future problems but also allows to define more robust architectures from early stages of a project.

Furthermore, the idea of working in a peer to peer network using the Actor model, goes hand in hand with the architecture on the edge, where multiple devices need to interact with each other.

All things considered, the Actor model fits well in the schema of the network edge, especially comparing with other programming models that are not conceived in this distributed model of computation.

5.2 Suitability for applications

Akka provides an implementation of the Actor model with multiple modules, addressing different types of requirements for different types of applications. Thanks to its rich platform it is possible to develop almost any type of application where having a distributed system plays a central role. Akka has a small footprint, which allows to create an application environment with multiple actors and high performance when it comes to passing messages between actors. Aspect that is required on the edge, working with resource constrained devices. There are different benefits that can be accomplished using Akka, and these will be discussed in the following sections.

5.2.1 Distributed computing

One of the most useful characteristics of Akka, is the possibility of using it as a tool for distributing computation, which can be very useful in domains such as edge computing,

where trying to use all the available computing power in all the nodes is a must. Akka enables to distribute processing basically on any step of the process chain. Whether it is splitting domain entities, with Akka Sharding or diving work with Akka Routing.

The proposed Akka application, handles the distribution of tasks in different parts using different mechanisms. First of all, is the representation of the sensors for processing their data. It is important to notice that the system can grow larger over time, considering that the number of sensors can be large and so is the data they generate. This situation could eventually become a bottleneck if only one or a restricted set of nodes would be in charge of processing all the information for all the sensors. Even more daunting, could be the task of managing and distributing the processing among these nodes.

In the proposed architecture, Cluster Sharding is used to distribute the work among the Worker cluster, as shown in figure 23. This Akka mechanism, allows to easily manage large amount of actors. In this case, an entity actor on a shard represents a sensor. Using the sharding mechanism, Akka handles the distribution process, maintaining a load-balance in the cluster without any further intervention. Moreover, the fact that every actor represents a sensor, along with its data readings, can be used to store state for each sensor, which can be useful for caching or used as a source of information for further analysis

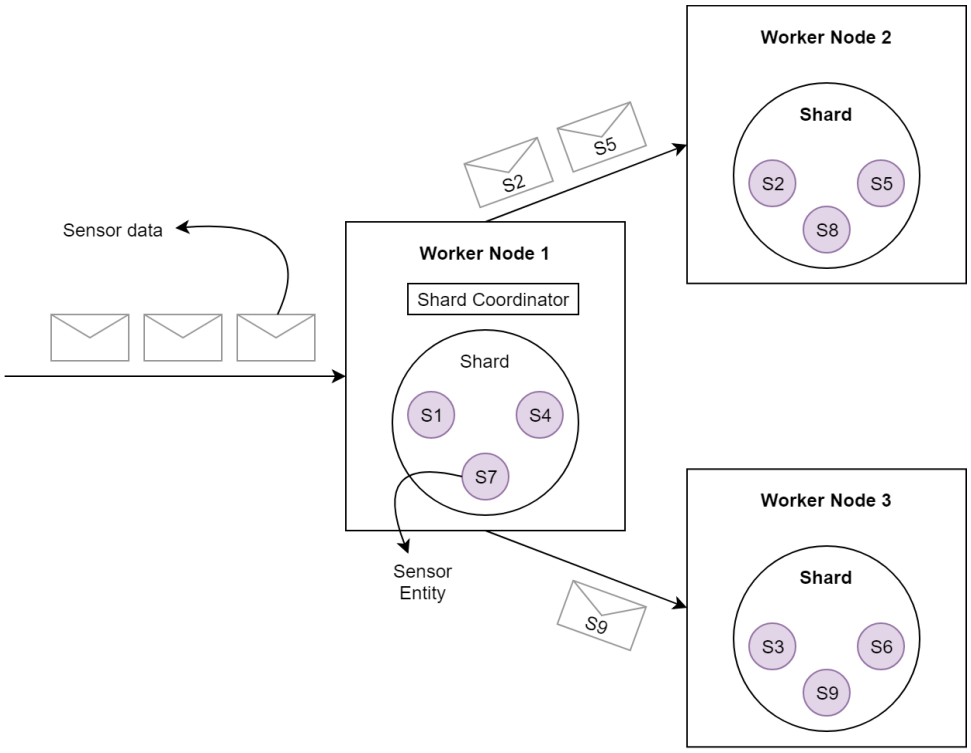

Figure 23. Distribution of sensor data in the Worker Cluster

Furthermore, the proposed architecture makes use of all the computing power of the cluster by splitting the work itself among the nodes on the cluster. This is done using the cluster aware routing mechanism of Akka, as shown in figure 24.

The idea is to distributed all the different computations that need to be carried out for the sensor data. In this case, for any given sensor data, which contains the reading of a sensor represented as a set of numeric values, different forecast models must be constructed and then compared against each other in order to determine which one fits better the data. The computation process of a forecast model, using the ARIMA technique, can require a significant amount of computation power [17], and if performed by only one node, namely an

actor in a shard region, can lead to delays and poor use of the computing power of the whole cluster, as other nodes may be idle while one is doing all the heavy work.

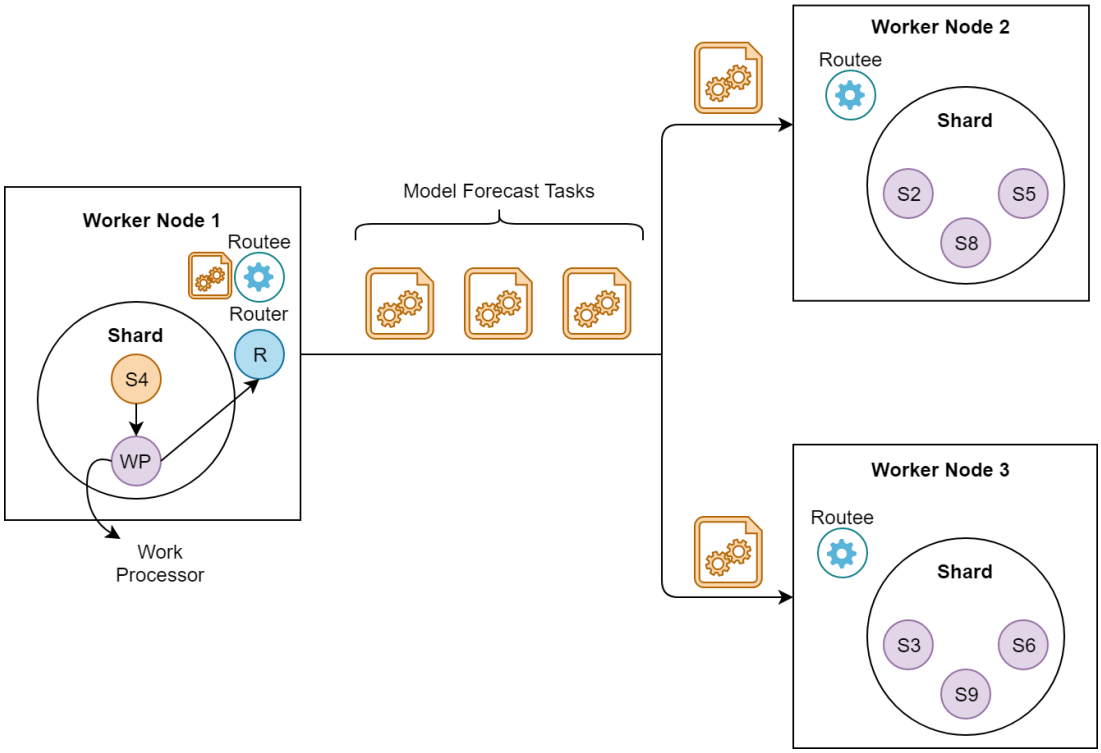

Figure 24. Distribution of computation tasks of a sensor entity in the Worker cluster

In this case in particular, each forecast model can be computed independently of others, so this process can be distributed among all worker nodes. The whole process begins when an entity actor, representing a sensor, receives data to process, the work in this case is to create different ARIMA forecast models. Then it forwards one by one all the forecasting tasks to the Work Processor, which is a Router actor. This Router actor delegates each forecasting task to all the deployed routees in the cluster using a specific routing logic. Each task message contains a reference to an aggregator actor.

Once processing is done by each routee, the result is sent to the aggregator actor. Finally, when all computations are received by the aggregator, it will return the final result to the entity actor that initiated the whole process. The use of the aforementioned mechanisms guarantees that at any point in time, if there is a work, then all the nodes in the Worker cluster are processing a part of that work, provided there is sufficient work for all nodes.

Not only the processing of the work is distributed but also the handling of the results can be distributed as well. In this case, the Distributed publish-subscribe mechanism of Akka was used to accomplish this task. The singleton actor on the Master cluster is the publisher and the *result processors* actors are the subscribers. Each node on the Master cluster has one result processor actor that is created when it is initialized. Upon the reception of the processing results, the singleton actor publishes the results to a specific results topic. Using Akka's configuration, each result message is received by only one subscriber, who will finally send the results back to the IoT cluster. Figure 25 illustrates how this distribution of results works.

The distribution of the results handling can help to increase the response time, as multiple results can be send back in parallel, making use of all the available nodes in the Master

cluster.

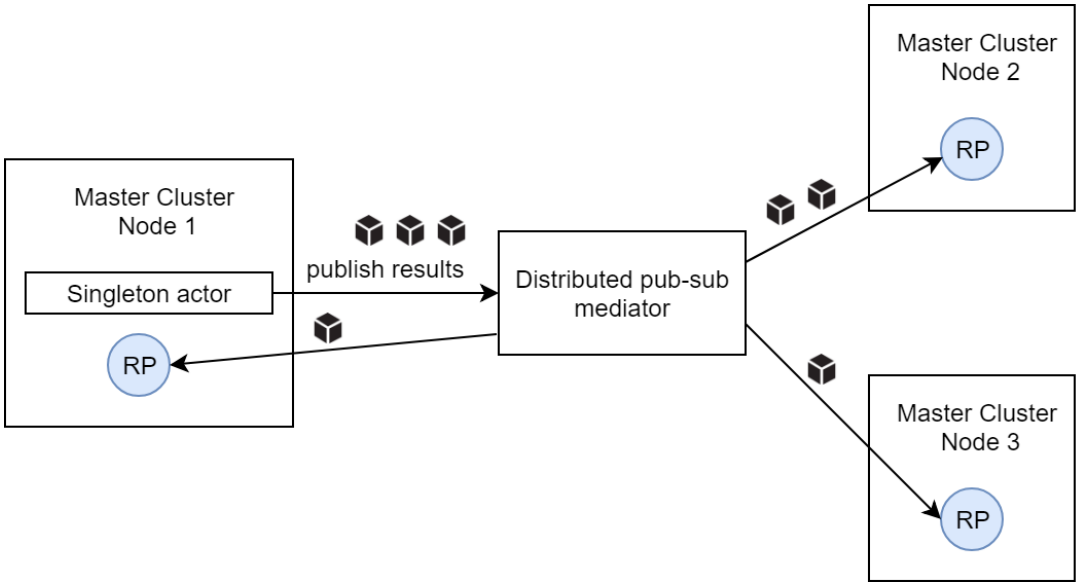

Figure 25. Distribution of results using Distributed pub-sub mechanism

Considering all the components of the architecture and all different things going on at the same time, the whole process can become complex and difficult to handle. Nevertheless, Akka offers the adequate mechanisms to ease the pain of handling all this logic manually, allowing to focus on the optimization of the process, such as the distribution of the computation among the nodes of a cluster.

5.2.2 Experimental results

The application has been tested using synthetic data. The synthetic data corresponds to measurements from the sensors. The specific characteristics of the experiment were the following:

- Synthetic data: Using normal distribution with mean 25 and standard deviation 3
- 3 Edge nodes: 1 Linux Machine, 2 Raspberry Pi devices
- Number of simulated sensors: 10
- Number of routees per node: 10
- Number of data input for models: 80000
- Number of ARIMA models to compute per sensor data: 4
- Frequency of data generation: 10 to 30 seconds per sensor data
- Number of Worker nodes: 1 for the non-distributed case and 2 for the distributed case. Both running on the Raspberry Pi devices.

The non-distributed case uses only one worker node. Whereas the distributed case scenario uses 2 worker nodes, using Cluster Sharding and Routing for distributing the computation among these nodes.

Indicators

***Processing time*:** From the moment the data was sent for processing, from the Master cluster, until the moment the processing result was received, by the Master cluster.

***System load average*:** The system load average taken every 20 seconds using the Akka metrics extension.

Heap memory usage: The application heap usage on the JVM taken every 20 seconds using the Akka metrics extension.

Processing Time

In order to measure the time, the *java.time.Instant* class was used, that is available since Java 8. This class uses a specific time-scale which is more precise than using other methods such as the common *currentTimeMillis*.

The results of the tests are shown in figure 26, where 25 requests of sensor data were processed. The time was measured in milliseconds. The results show how the time of using one worker tends to grow linearly with more data. Whereas having 2 workers produces a variance of time within a certain range. In order to understand these results, it is important to consider how the processing of forecasting models works.

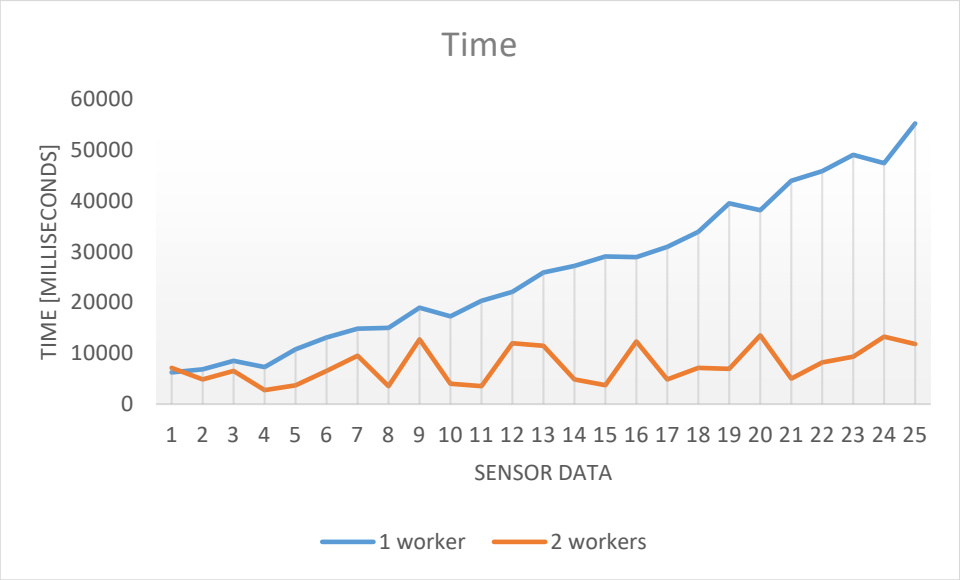

Figure 26. Time taken to process a sensor data

Each sensor data needs to compute 4 ARIMA models. With 10 devices there are 40 models to compute at a certain point in time. As time passes, the sensors keep sending sensor data to process, in the range of 10 to 30 seconds. In the cloud non-distributed case, as there is only one worker, that worker keeps accumulating the forecasting tasks in its mailbox queue, waiting to be processed, hence, taking more time to process.

With 2 workers, the processing can be distributed, not only thanks to the sharding mechanism, but also using the routing of the modeling tasks. The result shows that the time does not grow as with the 1 worker scenario, but it ranges between certain values. These results also depend on the number of routees. With more routees available, there are more actors capable of processing, however, this can overload the whole capacity of the system at some point creating other problems.

System load average

In regards to the system load, the Akka metrics extension was used. The measurements were taken every 20 seconds. The results are shown on figure 27.

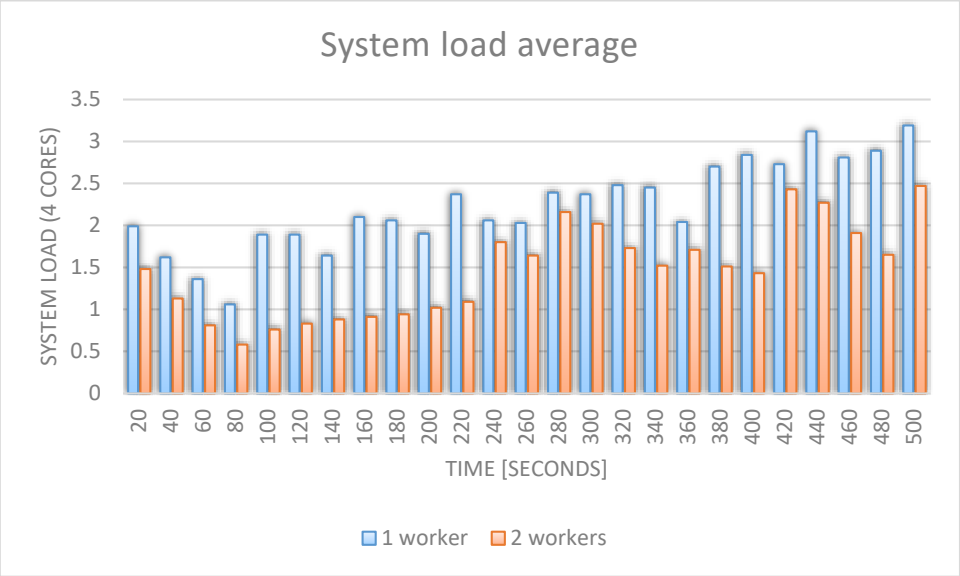

Figure 27. System load average of SensorData processing

It is evident that adding another worker reduces the system load. Nevertheless, the difference is not very significant, especially as time passes. This can be explained with the amount of task to compute. As more time passes, more computing tasks arrive and these are distributed between the two workers, making them use more CPU resources.

Heap memory usage

The same analysis can be applied to the heap memory usage. Figure 28 shows the results. Initially, when there are still no incoming messages, the usage is low. The difference in respect with the system load is that, in the case of the system load average, the system requires to initialize the cluster and perform initial internal tasks of the whole Akka system, which requires CPU resources. In the case of the heap memory, at the beginning only the basic actors are created. As times passes, sensor data arrives to the Worker cluster, and more actors are created as entity actors in the Shards.

One important aspect to notice, is how the 2 workers scenario uses more memory. This can be explained with the amount of routees that need to be created when a new node joins a cluster. For example, node1 is the seed node and creates the cluster. Later, node2 joins the cluster. Node1 notices this fact, and it deploys routees do node2. Node2 does the same, as soon as it enters the cluster and realizes that there is another node in the cluster, it deploys routees on node1. This mechanism is done automatically with the cluster-aware routers configuration.

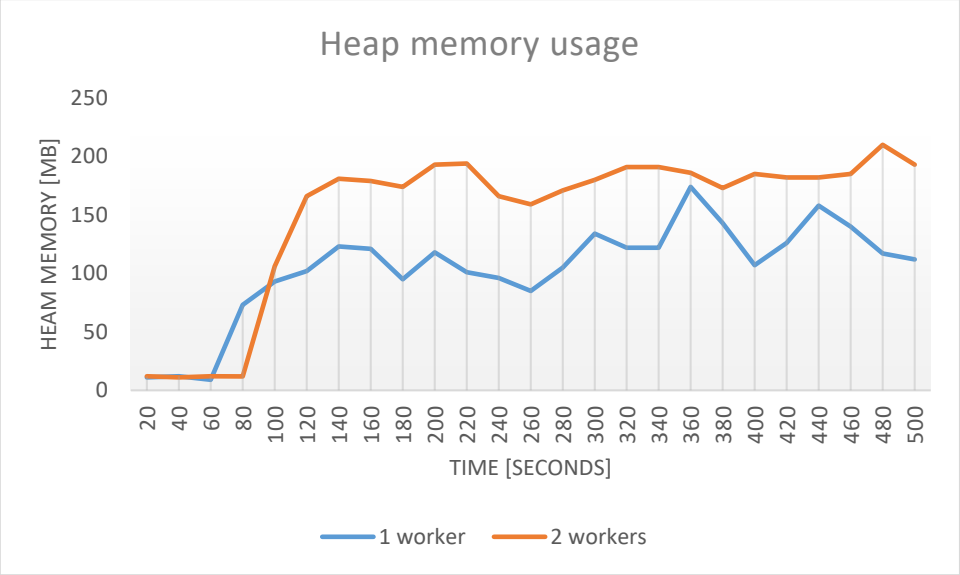

Figure 28. Heap memory usage with respect to time

Trying to find a perfect balance requires a trial and error approach. Changing some parameters, such as the number of routees, or even the configuration of pool routees vs group routees can cause significant changes in all metrics. As Akka documentation mentions and also different literature on the topic, there is no simple solution for all cases. The appropriate set of configuration settings and application structure, will depend much on the requirements of the application.

For the purpose of giving insights for analysis, the scenario proposed was enough to illustrate the benefits of using a distributed scenario. From this experiment, it follows that adding more nodes and distributing the computation process, as with the Worker cluster, processing times can be reduced, which may very well be the most important metric when dealing with real-time data processing.

5.2.3 Programmability for applications

Regarding programmability, Akka's toolkit forces to think in terms of actors and messages for creating applications. All the modules and libraries that Akka provides, enforce the concept of the Actor model of working with actors and messages. This approach seems appropriate in order to develop more robust and consistent applications.

Application code tends to be more succinct, and the concept of using actors help to encapsulate or define more clearly domain concepts. For instance, in the Work Cluster, an entity actor on the shard represents a Sensor. This helps to model the whole application using the same abstraction level used on the domain design. Furthermore, every actor clearly defines its message protocol of what types of messages it sends and receives, placing the corresponding response logic, or computation tasks, behind each type of receiving message in a specific manner.

Another important fact, is that Akka establishes the same code structure for any type of actors. This means that despite two actors belonging to different and possibly completely unrelated concepts, the code structure for both actors, can be read and understood in the same manner. This facilitates the maintenance of the application, looking for possible issues within the code, and also makes it easier to update the behavior of an actor, having specific places where to define the new logic and handling of messages for an actor.

5.2.4 Location transparency

Another benefit of Akka is location transparency. This means, in programming terms, that the code does not change if the system is working locally or remotely. As mentioned previously, the whole concept of using actors, is to have these actors working in a distributed environment. This aspect benefits the programmer in the sense that it is easier to build an application that works locally, as done in the first implementation, and then with the help of some configuration settings, as shown in the IoT implementation, an application can properly work in a remote and distributed environment. This aspect can be of special interest to easily develop edge applications, where no major change is needed to deploy a fully functional local single-node application, to a multi-node scenario in the edge, provided all the application requirements, in terms of resources or others, are met by the edge devices.

5.2.5 Difficulty and challenges using Akka

Despite all these benefits, using the Akka toolkit can result challenging. First of all, it can take a long period of time before being able to create relatively complex applications. For instance, the proposed Akka architecture for the IoT scenario, involves several different concepts, such as clustering, routing and sharding. Trying to develop an application with these characteristics all at once can be really challenging. The best approach is to start with basic applications, handling one module or concept at a time, such approach was used in this thesis. This will allow for better understanding of how Akka uses these concepts and how it implements them, so that later, it would be easier to use the corresponding modules that Akka offers.

Second of all, documentation is quite extensive, and it requires lots of time to read it through. Some topics are more complex than others, and most of them are related in some way or another, which implies that multiple concepts must be considered when trying to learn a single one. While most of the concepts are provided with small examples, most of them work without modification only in Scala. The examples provided for Java need some kind of modification, which can be cumbersome when trying to learn and use complex concepts such as Cluster Sharding.

Additionally, special care must be taken regarding configuration settings and other programming details to avoid breaking the actor encapsulation or the single threaded logic. Luckily, Akka documentation provides special recommendations and best practices to avoid these pitfalls.

From a programming perspective, working with Akka applications requires a certain knowledge of the toolkit and its modules to understand how an application works. In this sense, it is hard to explain to other programmers the semantics of the code and how the different modules work together do build an application. Different to what happens with other frameworks or tools, an Akka application cannot be explained only with code. Essentially, any programmer trying to understand an Akka application, must first have a basic understanding of how the Actor model works and how Akka implements it.

5.3 Deployment of the Application Stack

The Application Stack refers to the set of software subsystems or components required to have a fully functional application. In the case of the proposed Akka solution, the Application Stack is composed of the following components:

- IoT cluster
- Master cluster

- Cassandra cluster
- Worker cluster

Each of these clusters can be composed of one or more nodes where each node represents an actor system, or a service provider, such is the case of the Cassandra cluster, where the set of nodes in the cluster provide the persistence service, through the distributed database.

All the clusters, except for the Cassandra cluster, have the same code for running the Akka application. Depending on their roles defined in the *docker-compose* file, they will be deployed and assigned to a specific device on the Docker swarm. This is very convenient as it possible to easily define which types of devices can hold which part of the application.

The Cassandra cluster can be deployed independently as can work as a standalone database, or can be deployed at the same time, in the same *docker-compose* file of the Akka application. As previously mentioned, given that the Cassandra cluster deals with data management it would be a good idea to deploy the service on more stable and powerful edge devices.

The platform that Docker offers for application deployment allows to easily distribute the application components among different devices. Docker also provides a set of tools and services, that can help to have more control of the application as a whole. Trying to handle these types of tasks manually in the edge could be very challenging, considering all the characteristics of the devices and the complexities of a network communications.

Relying on Docker for the underlying connectivity between the devices creates a solid platform on which Akka applications can be executed. Not only this, but also the configuration management and tools, and small memory overhead of using Docker, makes it a very useful tool to use in the edge.

Certainly, it is also possible to think about using other orchestration mechanism besides Docker Swarm, such as Kubernetes or Apache Mesos. However, Docker Swarm works very well with Docker containers and being part of the same platform helps to keep things simple.

5.4 Fault tolerance of the system

Responding to failures on the system is one of the challenges that edge systems need to address. This means that a system needs to possess the ability to self-heal, self-adapt and, in more general terms, self-manage. Akka acknowledges this necessity and developed the concept of Supervision Strategy along with other features in order to address these requirements.

5.4.1 Isolation of failures

Using actors to construct an actor hierarchy helps to deal with failures. The idea is to isolate different areas or segments of the system. If there is a problem with a particular branch on the actor hierarchy, the problem will be handled by the parent actor of that branch while other branches can continue to work normally. This way of dealing with failures also puts emphasis on design decisions within the application.

For example, on the Worker cluster, workers are modelled as entity actors living in shards. Each worker entity has a child actor which is in charge doing the aggregation of all the model forecasting tasks. If there is a problem with this actor, then its parent actor, the corresponding Worker actor, will resolve the issue using a specific supervision strategy. Other worker actors are not affected by this failing actor and can continue its normal processing as illustrated in figure 29. In a major failing scenario, where the node itself becomes unreachable, then the whole shard is migrated to another node in the cluster, providing high

availability in respect with working actors in the system. This is, by no means, the only way of achieving high availability, as Akka offers other types of features and modules, such as cluster- aware routers, used for distributing work in the Worker cluster, that not only deal with issues regarding the availability of the system but also can help in terms of scalability.

It is also important to notice how the failure handling process is done without the intervention of any other entities other than the system itself. This is a characteristic that is much appreciated on remote systems where accessibility and maintainability are hard to accomplish.

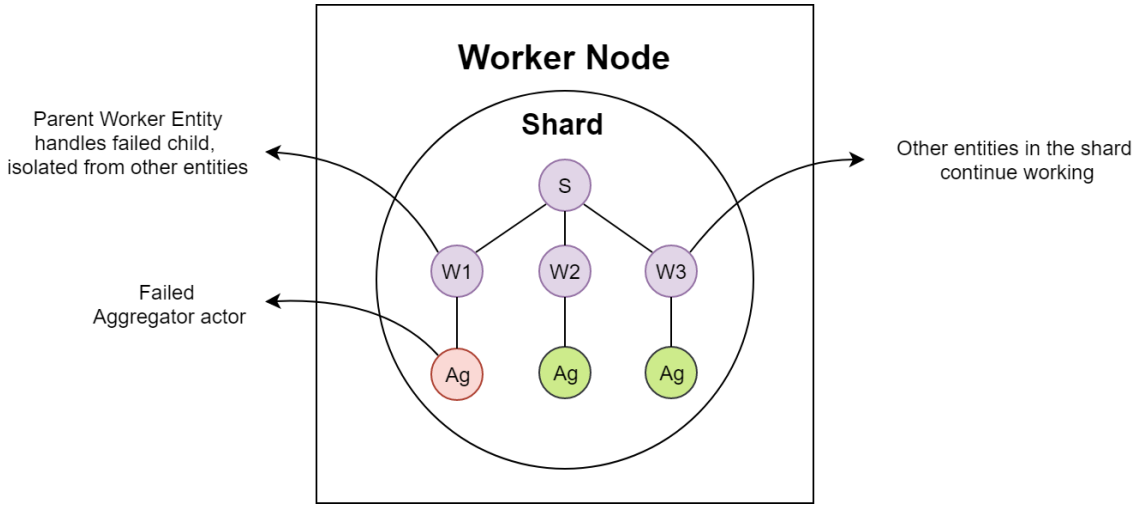

Figure 29. Isolation of failures on a shard

Applications on the edge, and more specifically the actors/entities of an application, need to be designed in a way that the actor hierarchy reflects the domain entities. Most likely, using bounded contexts. This could potentially help to better visualize and address particular problems in specific areas of the application. For instance, in the context of the network edge, an edge device can define specific actors for different services, and in the case one of those actors have a problem, only one service would be down while the rest of the services would still be available.

5.4.2 Use of replicas to update and maintain the system

Using replicas can help to keep the system available in different common scenarios; for example, when a service needs to be updated or during migration of data. To illustrate this idea, it is possible to think on a service deployed using Akka clusters on 5 different nodes. After a while, there could be a requirement for the service to change in according to some specific domain requirements.

While a normal full update would require to stop the service, making it unavailable for users or other components using it, a partial and controlled update can be carried out without requiring the whole service to stop, and ultimately converge into a full system update. This process can be done by taking down one actor system replica at a time, or depending on the distribution of the service, it could be done taking multiple nodes at a time. Figure 30 shows this scenario. These nodes can be updated with the new application logic and all the setup required for the new service. Meanwhile, the service is still active during this partial update thanks to the remaining replicas of the service.

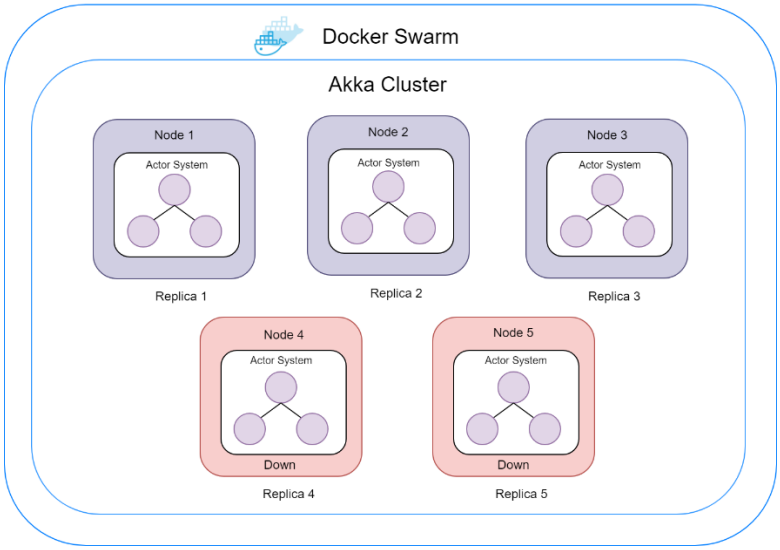

Figure 30. Service with 2 replicas down in an Akka Cluster

From within Akka, it is possible to stop the actor system running the application on the node and deploy the updated actor system with the new service. In this case, the Akka system is down, but the node on the Docker swarm is still connected and reachable, no need to fully restart or take down the node itself.

If a greater change is needed on the node, such as hardware update or maintenance is required, then apart from taking the Akka node down, it is also possible to remove the node from the swarm. The situation of the service is still the same, the replicas are still up and running making the service available during this period.

Another possibility to apply an update would be to start another actor system on a new container, but on the same swarm node, with the updated service, until the old one is no longer needed and then taken down. Figure 31 illustrates this idea. This would mean that a node would be running two actor systems at the time. This is possible using Akka. In addition to this, Akka allows to run more than one actor system on the same JVM meaning that it would be possible to put the updated service on the same container (two actor systems on the same JVM). Nevertheless, it is better to keep the isolation that Docker containers offer and simply deploy one actor system on one container. There could be also other issues with this approach, such as impact on performance or other undersired results.

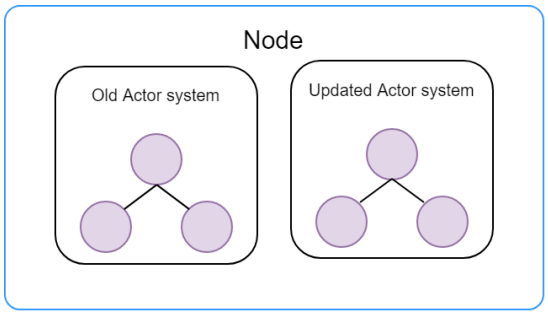

Figure 31. Two different actor systems deployed on the same node

Applications are constantly changing and evolving and so are the technologies on which these application are running. Using the combination of Akka and Docker allows for a system to keep up with this fast change, without repercussions on the availability of a service by its consumers.

5.5 Summary

On this chapter, the proposed IoT scenario was used to evaluate how the Actor model can be implemented in the network edge using Akka, responding to the research questions proposed for this thesis. A detailed analysis of the different aspects considered for the application were discussed and possible alternative scenarios were also taken into account for the different evaluations.

6 Conclusions

The main goal of this thesis was to investigate how the Actor model can be applied in the network edge using Akka for building IoT applications. After going through the main concepts of the Actor model and how Akka implements this concept, with its different modules and libraries, it was demonstrated that the Actor model complies with the different requirements of the network edge, in regards to working in highly distributed environment. Three different applications were developed. The last one proposes an edge solution to an IoT Wireless Sensor Network scenario, that previously used a cloud-centric approach. Different considerations were taken into account to evaluate this solution in order to demonstrate the benefits of using the Actor model, through Akka, on the network edge. Throughout this process, several ideas were proposed and different scenarios were considered with respect to the possible types of applications/architectures that can be built using the Actor model concept. The result of this analysis is summarized in the following paragraphs.

First of all, as described in this thesis, the Actor model involves a different approach to model applications, in which distribution or partition of an application plays a central role. Different literature [23] [24] try to address this specific challenge. Most of this literature propose to start with the perspective of using Domain Driven Design to decompose an application into more refined parts. The concept of bounded contexts, aggregates and entities, are useful to start thinking in terms of actors and define their hierarchy. Having said that, putting these ideas into practice is not a trivial task. Decomposing a domain into actors, may very well be, one of the most difficult challenges trying to work with the Actor model.

Thankfully, Akka forces to think about these concerns from the beginning of the development process avoiding more complex problems in more advance stages of the process. This favors making important decisions early on, so that, if implemented correctly, can bring along different benefits such as high availability, fault tolerance, and scalability among others. On the other hand, if these concerns are not carefully analyzed, additional complexity can be added to an application that can make working with actors a complicated venture.

It is also important to consider the context of the domain of the applications. When dealing with applications in the edge, applications must include the modelling of different devices in a distributed environment. In this sense, the domain of edge applications is already distributed and partitioned, which makes it easier to think in terms of actors. Nevertheless, there are other important aspects to consider, such as the granularity of the application with respect to the actors.

Another important dilemma originates at the moment of deciding which Akka features to use. Selecting the appropriate modules and applying them on the right components of an application, can improve the system's overall performance, increasing availability and scalability among other desired qualities. This aspect was demonstrated in this thesis, where a cloud IoT scenario was modeled with an edge architecture, maximizing the use of the available computation power of all the devices in the edge.

Applying the concepts available through Akka, and essentially from the Actor model, requires good understanding of Akka's Actor model implementation, plus the knowledge of the domain to model and creativity when it comes to ensemble all these elements together.

In regards to the use of the Akka toolkit, first and foremost, a good basic knowledge of how the Actor model works is required. This can be quite challenging given it involves a different way of thinking about applications. But even more challenging can be applying the Actor model using Akka. Implementations of the Actor model, add a layer on top the model in order to make it more practical. Akka uses a vast amount of concepts to accomplish this.

Several of these concepts were mentioned and used in this thesis, such as remoting, clustering, persistence and sharding. The implementations of these concepts involve different types of configuration details that have to be well defined in order to use them properly. While some are simple, others can be quite hard to understand. The documentation of Akka is abundant, which can be overwhelming for new developers, trying to understand all the concepts at once.

Equally hard is the idea of creating libraries on top of Akka. As previously mentioned, this task would not only require to have a proper knowledge of the Actor model but also of the inner details of how Akka implements the model. Developing small libraries do not seem like a good approach either, as they would not add anything significant to the toolkit. In fact, Akka already handles many aspects and it could be even hard to find a new one that Akka does not already provide some kind of solution. Most of the libraries or frameworks built on top of Akka are robust frameworks that handle specific types of applications, such as the *Play framework* for web applications or the *Lagom framework* for microservices. Other small projects focus on creating drivers or connectors, such as the ones for integrating different types of databases to Akka.

Selecting a programming language to work with Akka can play an important role in the short and long term. While there is an API for Java and different resources using Java, most of the official examples, projects and bibliography, use Scala as the main programming language. This becomes a sort of disadvantage trying to learn and apply different concepts using Java, as there are no clear working examples to use as prototypes. Moreover, Akka was written in Scala, using its concepts and features. Most of these are adapted for the Java API. As a result, parts of the code can end up in a mix of Scala-Java code. While both languages are compiled to Java bytecode, and any other JVM language can be used on top of it for that matter, it is clear that Scala is the predefined language for implementation, and in case of dealing with serious long term projects, it would make sense to learn and use Scala.

All in all, Akka provides a very complete set of tools for building distributed applications. The developers of these tools tried to consider almost every possible scenario, and are continuously improving and adding more modules in order to provide a more comprehensive toolkit.

Regarding the deployment process, Docker helps with the process of managing the different nodes of the system. It provides a robust platform on which Akka applications can easily be deployed relying on all the tools and guarantees that Docker provides, as shown in the applications developed for this thesis. Docker Swarm is the orchestration mechanism that Docker uses to manage multiple containers, and it is useful when thinking in terms of multi-node clusters, and how to manage them in order to deploy Akka applications.

On another note, the distributed nature of Akka, makes it a good candidate for integration with other distributed applications and tools. For example, in the proposed edge architecture, an Apache Cassandra cluster was used as a distributed journal. Cassandra uses the same concepts of nodes, seeds and clusters, which makes it an excellent candidate to integrate with Akka. The idea of working in a distributed environment does not constraint to databases. Different frameworks and technologies are being developed in terms of highly distributed systems, which facilitates the process of implementing applications on the edge.

In summary, throughout this thesis, the Actor model was studied and used as an application model for developing applications on the edge. An IoT Akka system architecture was proposed along with its implementation, based on a Wireless Sensor Network IoT scenario, in

order to show how applications can be conceived in the edge rather than relying on the cloud. As a result, Akka presents itself as a comprehensive set of tools that can be used to implement distributed applications using the Actor model as an application model, which can be applied on different types of domains, and it results particularly useful in the domain of IoT systems, and more specifically on the network edge, as it directly addresses some of the issues and challenges of edge computing, such as distribution of computation and self-managing of the edge nodes.

7 References

[1] C. Chang, S. N. Srirama, R. Buyya: Internet of Things (IoT) and New Computing Paradigms, Book title: Fog and Edge Computing: Principles and Paradigms, Editors: R. Buyya and S.N. Srirama, Wiley, 2019, pp. 3-23.

[2] Yu, Wei & Liang, Fan & He, Xiaofei & Hatcher, William & Lu, Chao & Lin, Jie & Yang, Xinyu. (2017). A Survey on the Edge Computing for the Internet of Things. IEEE Access. PP. 1-1. DOI: 10.1109/ACCESS.2017.2778504.

[3] Premsankar G., Francesco M.D., Taleb T. Edge Computing for the Internet of Things: A Case Study. IEEE Internet Things J. 2018;5:1275–1284. DOI: 10.1109/JIOT.2018.2805263.

[4] Shi, Weisong & Dustdar, Schahram. (2016). The Promise of Edge Computing. Computer. 49. 78-81. 10.1109/MC.2016.145.

[5] Hewitt, C. Actor model of computation: scalable robust information systems. arXiv: 1008.1459 v28, 2010.

[6] Typesafe. (2019, February) Build powerful concurrent & distributed applications more easily. [Online].

http://www.akka.io/

[7] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. 1994. A Note on Distributed Computing. Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Wer-ner Vogels. 2007. Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41, 6 (October 2007), 205-220. DOI: https://doi.org/10.1145/1323293.1294281

[9] Rusty Klophaus. 2010. Riak Core: building distributed applications without shared state. In ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10). ACM, New York, NY, USA, Article 14, 1 pages. DOI: https://doi.org/10.1145/1900160.1900176

[10] Hugh McKee. (2019, February) Akka Clustering and remoting [Online].

https://www.lightbend.com/blog/akka-clustering-remoting

[11] R. Morabito, "Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation," in IEEE Access, vol. 5, pp. 8835-8850, 2017. DOI: 10.1109/ACCESS.2017.2704444

[12] J. Feng et al., “AVE: Autonomous vehicular edge computing framework

with ACO-based scheduling,” IEEE Trans. Veh. Technol., vol. 66, no. 12, pp. 10 660–10 675, Dec. 2017.

[13] M. Liyanage, C. Chang, S. N. Srirama: mePaaS: Mobile-Embedded Platform as a Service for Distributing Fog Computing to Edge Nodes, The 17th International Confer-ence on Parallel and Distributed Computing, Applications and Technologies (PDCAT-16), Guangzhou, China, December 16-18, 2016, pp. 73-80. IEEE.

[14] C. Chang, S. N. Srirama, R. Buyya: Indie Fog: An Efficient Fog-Computing Infrastructure for the Internet of Things, IEEE Computer, ISSN: 0018-9162, 50(9):92-98, 2017. IEEE. DOI: 10.1109/MC.2017.3571049

[15] C. Long, Y. Cao, T. Jiang and Q. Zhang, "Edge Computing Framework for Cooperative Video Processing in Multimedia IoT Systems," in IEEE Transactions on Multimedia, vol. 20, no. 5, pp. 1126-1139, May 2018. DOI: 10.1109/TMM.2017.2764330

[16] J. Fürst, M.F.Argerich, K. Chen, and E.Kovacs. (2018). Towards Adaptive Actors for Scalable IoT Applications at the Edge. OJIOT, 4, 70-86.

[17] Dias GM, Adame T, Bellalta B, Oechsner S. A self-managed architecture for sensor networks based on real time data analysis. In: AA.VV. FTC 2016 - Proceedings of Future Technologies Conference. 1 ed. IEEE; 2017. p. 1297-1299.

[18] O. Boyinbode, H. Le, A. Mbogho, M. Takizawa and R. Poliah, "A Survey on Clustering Algorithms for Wireless Sensor Networks," 2010 13th International Conference on Network-Based Information Systems, Takayama, 2010, pp. 358-364. DOI: 10.1109/NBiS.2010.59

[19] Seema Bandyopadhyay and E. J. Coyle, "An energy efficient hierarchical clustering algorithm for wireless sensor networks," IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428), San Francisco, CA, 2003, pp. 1713-1723 vol.3. DOI: 10.1109/INFCOM.2003.1209194

[20] G. M. Dias, B. Bellalta and S. Oechsner, "Using data prediction techniques to reduce data transmissions in the IoT," 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, 2016, pp. 331-335. DOI: 10.1109/WF-IoT.2016.7845518

[21] Chong Liu, Kui Wu and Min Tsao, "Energy efficient information collection with the ARIMA model in wireless sensor networks," GLOBECOM '05. IEEE Global Telecom-munications Conference, 2005., St. Louis, MO, 2005, pp. 5 pp.-2474. DOI: 10.1109/GLOCOM.2005.1578206

[22] Bhandari, S., Bergmann, N.W., Jurdak, R., & Kusy, B. (2017). Time Series Data Analysis of Wireless Sensor Network Measurements of Temperature. Sensors.

[23] Michael Nash and Wade Waldron. 2016. Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications (1st ed.). O'Reilly Media, Inc..

[24] Vaughn Vernon. 2015. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka (1st ed.). Addison-Wesley Professional.

Appendix

I. License

Non-exclusive license to reproduce thesis and make thesis public

I, Freddy Marcelo Surriabre Dick,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Actor model in the IoT network edge for creating distributed applications using Akka,

supervised by Satish Narayana Srirama.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Freddy Marcelo Surriabre Dick

16/05/2019