Edge computing for disaster relief operations

Nicolai Vatne

Thesis submitted for the degree of
Master in Programming and System Architecture
30 credits

Department of Informatics
The Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Edge computing for disaster relief operations

Nicolai Vatne

© 2022 Nicolai Vatne

Edge computing for disaster relief operations

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

This thesis is the final product of my master's degree in Informatics at the University of Oslo.

The work was supervised by Carsten Griwodz. Frank T. Johnsen was helpful as a subject matter expert.

I would like to thank Frank and Carsten for all their help and support.

Thank you to my family for all the motivation.

Finally, thank you to Nora and Mika for always being there.

Abstract

Humanitarian Assistance and Disaster Relief (HADR) operations are needed to alleviate the effects of natural disasters on the local population. In HADR operations, multiple branches of government and non-government organizations all need to collaborate toward the same goal. Therefore, there is a need for a collective C2, which ensures everyone is coordinated and has the necessary situational awareness to complete the mission. Historically, this has been done through voice communication, but developments in modern technology can provide valuable new resources for first responders. In this thesis, we explore the capabilities of an ad-hoc distributed networking solution built using single-board computers. We explore its capabilities and limitations as we try to enable new forms of communication. In addition, in this thesis, we investigate Kubernetes as the resource framework for edge computing HADR operations.

List of Figures

Listings

Contents

Chapter 1

Introduction

1.1 Topic and Research Questions

In most parts of the developed world, continuous connectivity is a given - but the tools enabling the connectivity still reside within ground cables or even cable pylons in residential areas. In the last 20 years, the way people communicate has shifted to handheld devices, which most commonly rely on a group of communicating towers that distribute signals wirelessly. Cellular networks are estimated to account for over 50 percent of website access in 2025, and has been steadily growing ever since its inception [68].

The modern world needs to become ever so accustomed to natural disasters, which have increased by a factor of five over the last 50 years [42]. Natural disasters have a devastating effect on the area and infrastructure, which has the indirect effect of making the human response even harder. In Humanitarian Assistance and Disaster Relief operation, a quick and effective response can prove vital to ensure that human lives are saved - and preserve the safety of those responding. Communication and coordination are arguably one of the most important aspects in HADR operation, which is what this thesis will research.

HADR operations will range in scale depending on many factors such as country, existing infrastructure, technological capabilities and so on. HADR operations are however normally a joint operation between multiple departments such as the Military, fire and police, but also including humanitarian organizations such as the Red Cross. Operations like this are commonly called CIMIC, which means Civil-Military co-operation. It is a tactic deployed by countries all over the globe, as situations have to be adapted to differently, which requires a wide range of expertise.

In CIMIC operations, communication is vital to the outcome of the operation. Secure and separated communication also play a very important role, as different agencies should not be able to access the others classified communication. Enabling this type of data-driven communication, specifically in the first few

hours after a disaster has been a subject of research for military organizations such as NATO.

With the massive increase in IoT devices, NATO established a research task force that has been exploring the applicability and usability of IoT in the military domain, NATO STO IST-147, called "Military applications of IoT" [44]. During its three years of operation, the group worked on experimenting and demonstrating how different components of IoT could be utilized from a military point of view. The group has concluded that IoT has and is expected to have a significant impact on future military operations and collaborative efforts such as HADR operations, counter-terrorism, and even the logistical aspect of running a military.

In this thesis, we will explore the capabilities and processing power of the Raspberry Pi to be utilized as an ad-hoc networking solution for HADR operations. Raspberry Pi is a low-end off the shelf product used all over the world, which is something we will discuss in subsection 3.5.1 of this thesis. We will implement a cluster a nodes consisting of Raspberry Pi's using a lightweight version of Kubernetes, called K3s. Kubernetes, which is described in more details in subsection 3.2 is an open-source container system that allows everyone to utilize the power of distribution [7]. We will explore the opportunities and limitations related to this approach and subsequently evaluate the solution as a whole.

With the description given above, we derived a few questions which we aim to answer

- **In what capacity is it possible to utilize affordable single-board computers with Kubernetes?**
- **Will a system of this sort be sufficient to operate on a variety of network configurations, applicable to those relevant during a HADR operation?**

1.2 Related work

Internet of things which is more commonly referred to by its acronym, IoT has been a growing field of interest for researchers ever since its inception. With the staggering amount of devices, researchers are trying to uncover new opportunities, and testing its capabilities. We covered IoT and its various components more comprehensively in chapter 2. As an offspring of the significant developments done within the field of cellular technology, new areas of interest such as Edge and Fog computing have sprung out, as discussed in subsection 2.2. The advancements in hardware has also contributed to IoT proving itself as a valuable field to enable these technologies.

The main conceptual idea of IoT refers back to the idea of having ubiquitous computing and a constant pervasive presence of things connected to a network. These things fall into categories such as micro-computers, processors, RFID

devices, tags, sensors and actuators, all serving its own unique purpose in both the private and public sector. These things have to be able to collaborate with each other to reach common goals within computing, sensing or actuation [46].

Cloud-based computing, has in many cases, replaced the traditional on-site servers that businesses historically relied on to maintain their services. With this change, computing at the edge of networks has become essential for the performance of IoT applications so that it can perform real-time computations of filtered data at greater speeds. Offloading computations closer to the edges of the network does not only benefit businesses and organization, but also the end-user, as the necessary computations are done physically closer to the end-users device.

Cisco [51] estimates that the number of ubiquitous devices in the world reached 50 billion as of 2020 [51]. This is a staggeringly high number, which poses questions to the massive capabilities the industry now possesses. Researchers have been investigating different options, which would allow this massive industry to collaborate with each other. This has been done due to the fact that the actors in the industry historically have not collaborated with each other. Establishing a way for all of these devices to communicate with each other, could be beneficial on many levels, as it would allow for businesses or even governments to get a real-time data about several aspects in society. [69].

Looking at the capabilities of the IoT ecosystem, there are several factors to consider to prove its feasibility for use in HADR operations. These factors fall into categories such as IoT's usability in cases where existing infrastructure is non-existent or wiped out. Researchers at NATO have conducted significant work that looked into the IoT infrastructure's capabilities [30].

Several groups within NATO have been extensively researching and demonstrating the capabilities of IoT in Military context. Within this field of study, technologies such as Edge and Fog computing prove to be vital in its success [46]. NATO IST-147 worked on demonstrating the capabilities of IoT devices for military HADR operations and concluded its work in late 2019 [30]. Their work was later succeeded by IST-176, which is still on-going. IST-176 primary focus of research lies in the interoperability of existing IoT systems with existing military command and control systems [45].

Achieving federation within such a divided industry poses certain difficulties. The difficulties lies in a very divided approach to production and software from the companies behind the devices. Many of the devices are contained within its own metaphorical silo. This silo serves to preserve the companies best interest in terms of not sharing its design, functionality and otherwise valuable intellectual property [70]. IoT devices do, however share its fair share of commonalities, such as being ubiquitous, limited to no computing capabilities and low energy. To achieve this set of specifications, messaging paradigms such as MQTT has proven vital in its success, and MQTT has become more and more of an industry standard as time has passed [55]. Because of this, MQTT has become a heavily researched topic both within the business aspect, and for military capabilities,

as it offers general ease of use with other technologies such as JSON [5].

Achieving interoperability between existing IoT infrastructure has been the topic of research for organizations, both military and civilian - as it would be highly beneficial platform to utilize in cases such as natural disasters. Achieving interoperability would also require that it could be utilized in combination with military command and control systems, to externally monitor devices, and having a centralized location where the devices would report back. As previously stated, NATO IST-176 is currently researching different approaches to interoperability. Their latest publication "Concepts and Directions for Future IoT and C2 Interoperability" provides insight into the different guidelines they suggest to achieve this [45].

Most of these works are oriented around the use and applicability of devices in a less than perfect circumstances in the terms of network capabilities and capacity. Exploring the use of Commercial of the shelf products in a distributed manner is an interesting possible development in the field, as state of the art distributed computing distributions such as Kubernetes will offer a picture of how devices can be utilized in unison with each other.

1.3 Methodology

The work in this thesis follows Peter J. Denning's Design paradigm [10]. According to Denning, computer science is an interdisciplinary field of study consisting of three different archetypes [10]. These are theory, experimentation and design.

Theory is oriented around making conceptual frameworks and understanding the theory of the different relationships between areas within the given field of study.

Experimentation is about researching models of computer architectures within an application domain and doing related testing to explore the capabilities of those architectures in terms of exploring new opportunities.

The design archetype is described as an iterative process consisting of 4 main steps, formulated in the collaborative article "Computing as a discipline" [9].

- State Requirements
- State Specification
- Design and implement
- Testing

We utilized these four stages during the time of this thesis. Each stage proved

helpful in terms of making the necessary adjustments along the way for our previously untested theory.

1.3.1 State requirements

During this stage, we researched and documented the various requirements for our system, and looked at the different support technologies we would require and how they would interoperate with what we envisioned. This is due to the infancy of the software available for ARM64 architectures which can be read more about in section 3.5 and 6.2, several issue were discovered and required further research to solve.

1.3.2 State specifications

Based on the issues and solutions we uncovered in the first stage, we laid out the plans for the system's architecture, while doing comprehensive research into all the different supporting technologies required for the system. We covered the various architectural elements throughout chapter 3

1.3.3 Design and implementation

With the research conducted in sections 1 and 2, we started implementing the various services and applications required to make the system operate. This included both the set-up of the hardware and implementing the underlying software architecture, services and coordinators. We extensively covered the various implementation stages in chapter 4

1.3.4 Testing

Several durability and performance testing were done to the complete technological solution to evaluate its applicability as an extension on the edge of the network, specifically in terms of its applicability in a disaster relief operation. Testing was done to ensure both its durability in terms of failing nodes, and its performance with various networks, ranging in bandwidth and loss. We aimed to test various components of our system. Both related to K3s and memory usage, along with a variety of network configurations to get the broadest image we could. This is covered in chapter 5.

1.4 Scope

This study aims to explore the boundaries of an IoT based networking solution aimed explicitly at humanitarian relief situations. Due to the unpredictability which normally occurs with existing infrastructure, the ability to have locally configurable and accessible devices offers a unique opportunity for teams on the ground, as they would easily be able to deploy or replace devices without the need for complex configuration. The study will be conducted using Raspberry Pi's units grouped together in a Kubernetes cluster. Due to the size of related frameworks and packages, K3s will be utilized - which is a lightweight distribution of Kubernetes that can be utilized in the same fashion and to the same extent as regular Kubernetes.

This thesis aims to host related services that could prove beneficial in HADR situations, along with doing rigorous testing to ensure the flexibility, durability and reliability of such as system.

This study will not cover the various IoT technologies that could further enhance the systems or hardware alternatives that are comparable to the Raspberry Pi. This thesis will also not explore or measure the outcome with different hardware and software, other than what is being utilized in this paper. With respect to the networking protocol, we will work with the assumption of a federated network utilizing MQTT [55] as a messaging paradigm. MQTT will be discussed, in detail in section 2.3, as it is an essential part of the ongoing research into enabling edge computing for HADR operations.

The study was conducted over a five month time frame from January to May 2022. In the coming chapters, we summarize the possibilities that were explored, and how the system was designed and operated to cover the cases we identified and solved, like latency, distribution and architecture. Next, we will talk about the implementation of the Kubernetes cluster and the related technologies and services that we implemented on the network. Finally, we will go into detail about the testing of the system, where we will utilize different technologies to test the durability, stability and reliability of the system.

1.5 Outline

Chapter 2 provides the user with the necessary background information on the different topics, terminologies and architectures related to this thesis. The chapter starts off with an overview of the IoT ecosystem, defining the high and low level concepts of how communication is conducted, before moving on to the work done by NATO's Research Agency for the applicability of IoT infrastructure for HADR operations.

Chapter 3 discusses the practical aspects of the thesis, with an insight into the underlying architectures and hardware that was utilized in the thesis. Our

primary focus here is to introduce the user to the vast amount of elements that comprises the Kubernetes Architecture, and how each component plays a vital role in how applications operate in a Kubernetes cluster.

Chapter 4 in this chapter, we extensively cover the work that was put into running the K3s cluster with the various services and applications we required. We detail the various implementations, and how we approached them, along with providing insight into how storage is handled under various circumstances in a K3s cluster.

Chapter 5 in this chapter we discuss the various tests we performed on the system and how its performance can be interpreted in context to the use-case we are working with. This chapter also includes the results of the testing, which was done with a variety of network configurations, so that we can more accurately draw a picture of how the system would perform in worse than best circumstances.

Chapter 6 the last chapter summarizes the key points and findings throughout the thesis and elaborate on our thoughts for future research.

Chapter 2

Background

2.1 Defining IoT

The Internet of Things, more commonly referred to by its acronym IoT is a terms used to describe the new architecture of single-purpose devices connected to the Internet. There are many different definitions used to describe this vast ecosystem of devices, whereas Oracle defines it as the following,

> "The Internet of Things (IoT) describes the network of physical objects—"things"—that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the internet [41]."

IoT devices in private homes are normally aimed at making everyday life more connected, and automate tasks that previously required human focus and interaction. IoT in the public sector is normally used as monitoring devices, reporting statistics in agriculture, production or similar sectors, along with a range of similar purposes.

The IoT industry has experienced an exponential growth over the last few years, as new use-cases show up. IoT is unique in many ways, with maybe the most notable being the ability for individuals and businesses to get a real-time look into how systems work, along with providing valuable insight into the performance aspect of the business.

IoT has already become an essential part of many people's lives and exists alongside everyone, all the time. This pervasive presence makes it ideal for multi-usage purposes such as the one being proposed in this thesis. In this chapter, we will look deeper into the key aspects that define IoT in 2022, along with looking at some of the proposed architectures and research that has been done in context to making IoT applicable for HADR operations.

2.2 Edge computing

Edge computing is another important paradigm for this thesis. Edge computing has many names within modern technological solutions, and is considered more than just a paradigm, but rather a philosophy [8]. Since Edge Computing is used so widely, there has been made some misconceptions like Edge computing being regarded as IoT or even Fog computing, but in reality - Edge computing is more synonymous with running less computations and processing in the cloud, instead off-loading it on local devices, which in many cases are very capable of doing it by themselves. These local device include laptops, desktops and IoT device. On the basis of this, we regard the use of devices in this described fashion as Edge computing throughout the entire thesis.

Gartner defines Edge computing in the following way,

> Edge computing is part of a distributed computing topology where information processing is located close to the edge, where things and people produce or consume that information [24].

The goal of Edge computing lies in the quote above, being to move the computing closer to where the data is being collected or consumed, which in terms minimizes the amount of long-distance communication that has to happen before an end-user gets the desired result or response.

Within Edge computing, there is also a popular term called the network edge. The term is used to describe the device that connects the local network to the wider internet, which can be viewed as the node at the edge of the network [8].

When off-loading computations, lower transmission costs become a given, since the need for larger capacity in a data center becomes unnecessary. Expanding locally is normally easier and cheaper than in the cloud, considering you can utilize devices under the same implementation [4].

Edge computing does come with some disadvantages as well. For one, the security in Edge devices does not necessarily meet the same rigorous demands that many data centers have to deal with in terms of authentication, and general capabilities, which makes it a more vulnerable target than large cloud provider's data centers [4]. Following along at that, for large companies managing a wide variety of applications, the infrastructure costs at the launch of the platform can be massive, which is something that has to be considered. In the long term however, the cost of running your application at a data center might exceed it.

In the modern world of smart homes, and large IoT infrastructures, Edge computing has become a popular paradigm for many companies looking to expand in the early stages. Some experts theorize that it might represent a paradigm shift in terms of how we process data [4].

2.3 Defining MQTT

MQTT is a lightweight messaging protocol utilizing the publish and subscribe messaging pattern [55]. MQTT specifically suits the needs of low-power and resource-constrained IoT devices and applications, since it has a minimal code footprint and network bandwidth utilization [55]. MQTT brokers can be deployed anywhere it would be required, being in a cloud solution, enterprise environment or a container architecture. Various implementations of MQTT are available, which has enabled it to be run on a wide variety of architectures and solutions.

The strength of the MQTT protocol lies in the fact that it allows persistent sessions, and also a dynamic variable Quality of Service setting, which in turn makes it suited for networking situations at the edge of the network [44].

In its core, MQTT is based on a client-server architecture, which means that there is a centralized server that is responsible for the delivery and communication paradigm between the different endpoints on the network [44], this can be seen in other networks as the broker. However, this was deprecated in the third iteration of the MQTT protocol, where it is now called the server. This was an attempt to standardize and normalize it for widespread use in the IoT sector, along with complementing new functionality that was added. [44]. This new version also brought new terminology compared to the previous version, publishers and subscribers were differentiated, whereas now both of them are referred to as clients.

There are also new elements to complement a typical use-case in modern IoT devices, being lost messages and congestion at the server. Prior to version 5, MQTT had a problem with published message deliveries that ended up not being received by the nodes, which could happen in cases where nodes failed or published messages simply ended up being lost [55]. The solution that was proposed and implemented in version 5 was the concept of shared subscription. Shared subscription allowed a set of nodes to participate in a group to share a common subscription, instead of needing to individually send to every single participating node, as this in many cases led to duplicated messages which could strain the network [40]. The unique part of the subscription group is that all participating nodes on the subscribed topic has an equally distributed load, allowing for reduced traffic and computations to be performed on networks during run-time.

Other prominent changes that came with the last version were a focus on better supporting offline or persistent sessions, with the release of clean start mode, which allows for the servers to request connections from devices so that they can clean up their session state. This helps reduce the server loads that occur in massive networks where clients may appear and disappear often. Along with this MQTT was released with functionality that nicely complements the idea of utilizing it in HADR operations. This includes flow control, that has the preliminary goal of reducing the amount of messages and negotiations involving both the client and the server.

For our thesis, we used the latest released MQTT Broker image from Eclipse, allowing use to utilize the new aforementioned developments in the protocol.

2.3.1 Publish/Subscribe

The Publish/Subscribe paradigm, from now on called pub/sub, is a messaging pattern that involves publishers, subscribers and brokers. pub/sub is available through a variety of paradigms. The purpose of our research is the Publish/Subscribe paradigm implemented in MQTT.

In its simplest terms, publishers do not create messages directed towards specific receivers, instead it uses a predefined set of rules to categorize messages into classes without knowledge of which subscribers are listening. This is contrary to a traditional client-server architecture where a client directly communicates with an endpoint [56].

A pub/sub networks provides great flexibility and scalability due to the aforementioned nature of how publishers and subscribers are inherently unaware of each other. This aspect happens on multiple different levels,

- The publisher and subscribers do not need to know anything about each other.
- The publisher and subscribers can run in separate time-frames, and do not need to run simultaneously.
- Operations and calls that happen locally on a node are not interrupted during publishing and receiving.

In terms of scalability, Pub/sub has some unique abilities contrary to traditional client-server approaches, which stems back to the fact that it allows the broker to work highly parallelized and in an event-driven way, which makes it ideal for small low powered devices that would not handle congestion, and a constant high load recurrently.

MQTT is just one of many protocols that utilize the publish / subscribe pattern. Each has their own variations in terms of how the protocol operates in its code. MQTT follows the pattern of decoupling the publisher and subscriber, but only to the extent that, if they want to participate in the network, publishers and subscribers only need to know the host-name and port of the broker, or server, as it is referred to from version 5.

Figure 2.1 is the publish subscribe paradigm in its simplest form, consisting of multiple MQTT Clients. As you can note by the illustration, the measurement device in the top right corner has no use for subscribing to anything from the server, so it only performs publishing towards the central entity. In contrast, you could have a device like a phone that would benefit from doing both publishing and subscribing, to either control or verify measurements from other clients.

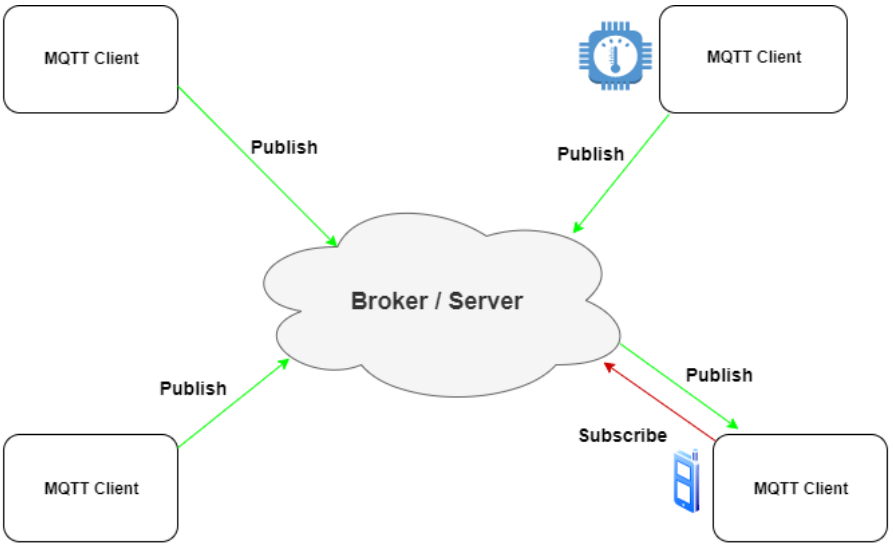

Figure 2.1: Publish/Subscribe Pattern

2.3.2 QoS

Quality of Service, or QoS for short, is a description and measurement used to define the quality of a service, specifically in the area of networking. QoS can be primarily seen as a quality seen by the users of the network.

Quality of Service has become a much more significant aspect of delivering good software in the modern age of computing, as more and more service make the total transition to cloud, and users on opposite sides of the world require the same quality of service, even with the distribution challenges that can pose.

QoS plays a key role in delivery of the MQTT protocol as well, where it is divided into 3 levels[57]. These 3 levels are,

At most once The lowest QoS level is zero. This service level guarantees a best-effort delivery. There is no guarantee of delivery. The receiver does not need to acknowledge receipts of the message and the message is not stored and re-transmitted by the sender. This QoS level is often referred to as fire and forget, and has no relatable technological aspects as the underlying TCP protocol, which is also a fire and forget protocol without many types of guarantees[57].

At least once The next QoS level in MQTT is level 1, where delivery is guaranteed at least one to the receiving client. The sender saves the message locally until it gets a acknowledge packet from the receiving client of the message. With

this in mind, it is important to note a message can be sent of delivery multiple times. The sending client uses a unique identifier from each packet and matches it to the PUBLISH packet. If a sending client does not receive a packet in time, it will send another message contained the PUBLISH packet[57]. Lastly, When a receiving client gets a message at this QoS level, it can process the message right away, without having to wait for additional acknowledgements.

Exactly once The last QoS level is the highest possible level of service in MQTT. This level guarantees that each message is received only once by the intended nodes, to prevent unnecessary load on the system and at the nodes. With this added precaution, it is however the slowest quality out of the ones listed, but also the safest if messages contain sensitive instructions related to running and operating the end-node device.

The extra functionality compared to the other layers happen during the initial PUBLISH message that goes from the sending node to the receiving one, and both nodes use this one to coordinate with themselves. If the first PUBLISH messages does not get acknowledged by the end-node, the sending node will keep sending until it gets a message containing the PUBREC packet. If a sending node has to send multiple PUBLISH messages, it will also include a DUP flag to indicate that it has been sent multiple times.

When the PUBREC packets have been received by the sending node, it can remove the original PUBLISH packet, as it does not need it anymore. Ones the sender has received sufficient PUBREC packets, it sends a packet containing PUBREL, which indicates to the end-nodes that it can discard the previously exchanged messages.

And lastly, once the PUBREL message has become the new default for marking the messages, we can be sure that message wont be processed twice, as it become publicly identifiable on the network for re-usability. All of these principles are valid for both MQTT client and brokers to make sure that both parties have their messages delivered and sent [57].

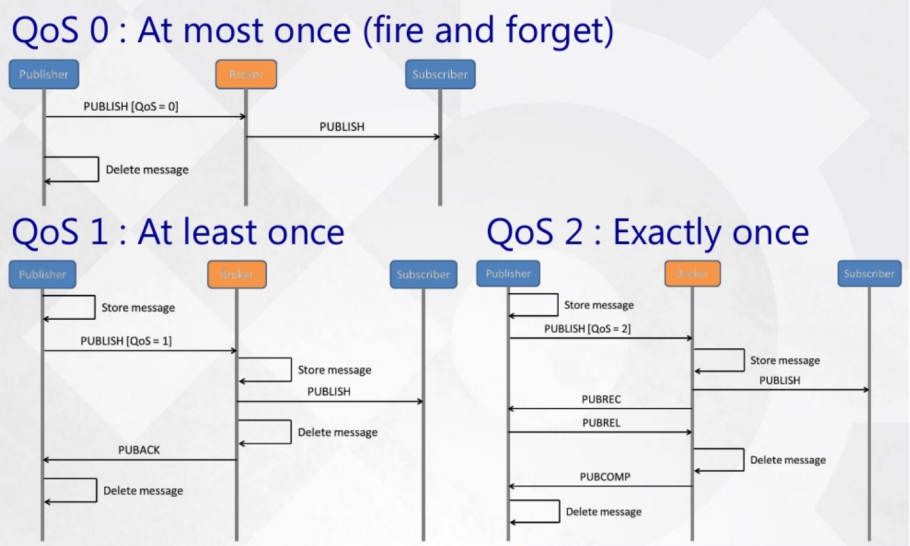

Figure 2.2: Quality of Service in MQTT [43]

Figure 2.2 shows the various levels of QoS within the MQTT messaging paradigm. The progression from 0 to 2 becomes apparent through the increasing amount of segments being transmitted, from the "fire and forget" policy up until QoS level 2, which aims to guarantee the deliver exactly once, requiring the subscriber to acknowledge, and the broker to delete the message once it has reached the theoretical end of the round-trip [43].

The difference separating these levels of Quality of service might not seem obvious at first, considering that we are working on a such a flexible network in the first place. There are however some things that stand out about the different ones.

QoS 0, is for instance most usable in situations where you have a completely stable connection between all the nodes, and similarly to TCP, it is most usable in situations where your application will still deliver even with a few lost messages [57]. QoS 1 is applicable for situations where it is critical for your application to get every message and the application wont be too affected by handling duplicates. A typical scenario, when it comes to determining whether QoS 1 or 2 is most suitable for the application, is to determine whether it is critical for your application to get all messages exactly once, as the name of the QoS level suggests. The final important thing to mention is the difference in overhead between QoS 1 and 2, which makes QoS more widely used in modern IoT solutions, where end-users is utilizing for example an app, where response time and low overhead is a preferred state [57].

Determining QoS level, and the related work that goes into making the system handle duplicates, and time-critical messages is something we will touch back on in the implementation part of the thesis, where we more closely dive into the various aspects surrounding the implementation of MQTT we used on our system.

2.4 Military Federation of IoT

2.4.1 NATO IST-147 and 150

This specific research can have implications in many sectors, and have been researched since IoT systems started exponentially growing. NATO or the North Atlantic Treaty Organization has shown a particular interest in this topic, to the extent that it has established task forces to look at the opportunities for federated inter-operability between military systems such as C2 and IoT networks in and around cities[30].

NATO IST 147, more commonly called "Military applications of IoT" was the task force working on this problem.

NATO IST-147, which was a part of the NATO Science and Technology Organization worked on investigating a proving the applicability of COTS IoT systems and Standards, which could be applied to Military context, and in September of 2019, they concluded their work successfully[30]. The group was succeeded by NATO IST-176, which is still on-going. The group is primarily researching methods to enable both interoperability and integration for existing IoT systems[45].

In the group latest publication “Concepts and Directions for Future IoT and C2 Interoperability” [45] the group presents the key design requirements needed in developing data models that would provide the necessary interoperability and extensibility for Military systems and Commercial of the shelf IoT devices. One of the fundamental elements used within NATO to maintain a cooperability between the participating nations at STANAGs, which simply is the abbreviation for Standardization Agreements, these have been used as means to maintain compliancy and standardization for different military applications, such as Technology. These Agreements are made by experts within their respective fields to best suit the current state and the future state of the area of interest[45]. In their latest paper, they do not propose new standards for the already comprehensive and intricate IoT ecosystem, instead, their work has been oriented around identifying and making clear guidelines for concepts of reusability of existing standards that can prove helpful for future developments within the smart systems sector.

STANAGs are identified as one of the causes that might have contribute to the widespread gap that has risen in IoT over the later years, due to the fact that STANAGs are made to complement existing use cases and technological domains[45]. STANAGs are however not as resilient to change, and it often takes a very long time to merge or re-purpose previously applied principles to, for instance, the every developing IoT sector. The idea of incorporating commercial digital ecosystems into military systems has therefore become challenging.

The group IST-147 has previously proposed the use of a simple messaging paradigm based on Topic formats, that is employed using the MQTT

paradigm. They subsequently demonstrated the applicability of this topic based communication which could support IoT and C2 data alike at a demonstration in 2018[45]. In this presentation, they mapped out the topic paradigm from MQTT over a much more standardized JSON hierarchy, which proved to be successful when employed over a range of coalition systems, consisting of a wide variety of different sensor. Even with the advancements by the previous groups working on the challenges with federating IoT platforms, all of them equally acknowledge that one of the fundamental problems lies in the lack of open-source components, such as data models, which is specifically acknowledge by the authors[45].

The proposed design considerations proposed by the group is adding descriptions for device publishing data, automatic recognition and remote configuration, network awareness and traffic distribution/management, security and privacy and storage of processing of data.

Descriptions for Device Publishing Data concerns itself with how metadata is is gathered, what format it is along with information about the owner. This information will be vital for a NATO affiliate to determine the to what degree the information can be utilized in a military situation, as precautions have to be made with respect to security and validity in crisis situations [45].

The idea behind Automatic Recognition and Remote configuration lies in the field of ubiquitous computing, and how IoT devices fall within the category. The group suggests that there should be capabilities in the Shared Data models to allow for remote configuration and event based responses based on the information that would be received in the Military Systems [45].

Enabling network awareness and traffic Distribution is another aspect that should be included in the Shared Data Model, which allow for the network state to be assessed along with controlling network distribution and management [45].

The storage and processing of Data relates to how the Shared Data Model should support different types of storage and processing assessments such as how the data should be stored and used for fusion with other data sources [45].

Lastly, the paper mentions how a shared data model should be helpful for providers to ensure privacy and security of corresponding IoT data streams with respect to anonymizing data that should not be accessible from a military point of view [45]. This may include data such as personal information, which serves no particular benefit to military systems, and is protected by legislations in many different regions of the world.

In summarization, the initial paper by IST-176 [45] suggests the use of a open Data-Model and a continuation in the work of STANAGs to be key contributors towards enabling more interoperability within the IoT sector towards military C2 systems. Through the use of open specifications in terms of how data should be stored, processed and translated - along with allowing remote monitoring of networks will be play a vital role in the usefulness and usability of systems from a military context.

2.4.2 Challenges with Federation

During a demonstration in 2018, NATO IST-147 performed experiments proving the applicability of IoT can be integrated in a urban HADR operation, but the demonstration left some open questions [44].

In the 2021 article "Federation Based on MQTT for Urban Humanitarian Assistance and Disaster Recover Operations" [44] they list the open ended questions yet to be researched in terms of validating the use of IoT platforms for HADR operations, which is listed below.

- How can multiple command and control (C2) system federate with each other?
- How can an IoT protocol be leveraged for federation between the coalition partners?
- Which C2 systems and which interface from the C2 systems can be used for federation?
- How can the civilian data be accessed and at what granularity?

This section will however look at the questions from an IoT point of view, and not in terms of the the corporation with military C2 systems, as most of the literature regarding C2 systems lies outside the scope of this thesis.

IoT consists of a wide variety of platforms which is engineered and operated as individual solution, with a lack of interoperability and cooperation, and a lack of infrastructure in terms of collaboration will negatively influence future use cases which can require larger scale IoT deployments, such as it's applicability for HADR operations[70].

As previously discussed, the IoT ecosystem is growing at a very fast pace, and the need for cross-domain IoT applications have been ever growing. Utilizing today's vertically isolated platforms to cover a wider variety of domains has become an area in which businesses need ever so much expertise[70]. One of the proposed solutions, which is not federation based is for companies to engage in strategic partnerships, to better be able to tackle larger distributions.

The interoperability in context of the IoT systems has been defined in the ETSI whitepaper as "the ability of organizations to effectively communicate and transfer (meaningful) data (information) even though they may be using a variety of different information system over widely different infrastructures"[27].

IoT platform federation was described in the paper Collaboration Mechanisms for IoT Platform Federations Fostering Organizational Interoperability [70] as associations between two or more platforms, which are willing to share access to their IoT resources in order to facilitate their fair interaction and collaboration.

Since the nature of most large IoT platforms in actuation and sensing, platforms would benefit from having a collaborative effort, since devices are spread out over large geographical areas. Enabling such a collaboratory effort would however require the platform to adopt a decentralized privilege approach, not allowing for one part of the network or one individual platform to use all of the capacity to limit other parts.

A federated IoT platform would enable applications to use resources that would also be managed and operated by other federated platforms as they were their own, which offers a great advantage in prospect to large infrastructures for sensing and actuation, such as Smart Homes and industrial plants. By federating platforms, it would remove the tedious task of interacting with multiple different interfaces and platforms across a wide variety of applications.

When federating IoT platforms several aspects have to be taken into account, such as resource utilization and fair usage properties. Research has been done as to rectifying this, through the use of Service Level Agreements, first proposed in "Collaboration Mechanisms for IoT Platform Federations Fostering Organizational Interoperability" [70].

Service Level Agreements is a solution that enables multi-level trust and reputation, along with providing a a fair usage sharing process among the federated platforms to allow each individual platform to gain the most out of the federation. Service level agreements can also be seen as a service where performance requirements of the different applications are noted in a standardized way that suits the nature of the end-to-end architecture that is the IoT domain. This agreements would play a vital role when accounting for things like Quality of Service metrics, which has be accounted for at every level such as the network gateways, edge nodes and in the cloud [1].

Interoperability within IoT platforms has so far not had much attention in research [70], but with the staggering amount of devices increasing daily, the need for federated IoT platforms grows. With the primary reason being to better complement the future developments in IoT.

Chapter 3

System Design

For the purpose of this thesis, several different software components were made, such as data emulation software and testing suites, with the most significant work being the Kubernetes Cluster made up of Raspberry Pi's. This section will cover the various discussions and aspects that went into creating the applications for this thesis.

3.1 Scenario

3.1.1 Disaster relief operation

Natural Disasters is an umbrella term to describe an unusual intensity of a natural agent. It can range from avalanches, droughts, tornadoes, volcanic eruptions to landslides.

Norway faced one of its latest disasters on 30th of December 2020, when a landslide struck a significant area in Gjerdrum, a small town north of Oslo [39].

The devastating landslide happened early in the morning hours, dragging 13 buildings and homes with it [23].

The first few hours after a natural disaster is often critical. If humans are trapped in the rubble, every minute can be vital for their survival. At Gjerdrum, the first few hours after the landslide were chaotic, as infrastructure and communication was destroyed. This is where an ad-hoc networking implementation can be valueable, such as the one we are proposing. It is an intermediary solution until the surrounding infrastructure is re-deployed. With an approach aimed to be utilized as a theoretical bubble on the Edge, our theory is to roll out the detached ad-hoc solution, which can later be utilized in the coalition with the surrounding infrastructure, once it returns.

Search and rescue at Gjerdrum was an ongoing effort until January 5th. During the span of the week, several organizations participated in the operation to find the residents of the homes that were struck by the landslide.

The search and rescue operation was comprised of multiple state departments, private organizations, voluntary organizations and the Norwegian military. This made the communication efforts complex with the resources at hand.

Ten people lost their lives in the landslide, while another ten were saved from the rubble. The infrastructure in the area was completely destroyed, forcing many people to relocate from their homes. The aftermath of the landslide still affects the area at the time of writing this thesis.

3.1.2 Reflections on the need for digitization and edge computing

One of the primary means of communication among the search and rescue teams is voice communication, specifically for the Gjerdrum incident where it would be "Nødnett".

"Nødnett" is a product of The Norwegian Public Safety Network, which is a public network system based on Terrestrial trunked Radio, or more commonly referred to as TETRA. TETRA is a formal standard for a voice-based radio system, where actors in the network are grouped and encrypted [32]. The TETRA network used in Norway was made to be a replacement for the existing radio-channel based communication that persisted in many departments around the country [32]. There have been some proposed technologies that might be viable as an alternative to the TETRA network, 5G being one of them.

5G technology has been under discussion in terms of acting as a replacement for traditional standards such as TETRA, which with its increased bandwidth and modernized standards, would open up for future emergency networks to include more than just voice based communication [13]. Traditional 5G networks are based on existing infrastructures in urban cities and areas. However, the alternative is 5G core, which can be operational on the edge, acting as a beacon, connecting the rest of the devices out of reach back to the infrastructure. This can be related back to the concept we covered in subsection 2.2, being the network edge. This is per now a proprietary technology, such as the 5G Vinni project, which is currently in experimental stages[67]. The 5G Vinni project is working on speeding up the innovation in the sector, by providing industries and businesses with a end to end solution that lowers the general entry level to 5G. They have solutions which can contribute to innovation on the edge of the 5G networks, along with the idea of providing separate slices within 5G networks. This is to allow communication to flow freely within those dedicated slices[67]. This research also has culminated to 5G Vinni, along with other independent researchers experimenting with mobile facility sites, such as vehicles, which would allow for a quick response to a HADR situation[67].

To limit the need for voice communication, which is currently a tightly inter-department system and extend the intercommunication between departments, we propose a system in which there is an increased use of data, contrary to voice. Transitioning information to data channels would open up for more participating actors, which would allow for higher coordination across departments. This can be assumed to be a way of streamlining communication, allowing for missions to be completed faster, and with a higher degree of cooperation.

We believe that utilizing the IoT infrastructure and its capabilities can profoundly affect this type of scenario. Devices come in all shapes and sizes with varying degrees of robustness. We can, however, assume for the purpose of our proposal, that the raspberry pi units can be retrofitted with an IP68 or military standard battery pack to increase the robustness of the devices in the field, along with an appropriate casing solution.

We aimed to have our solution be based on completely open-standards and open-source software to allow for maximum interoperability. Doing this in such a manner would allow a system of this sort to communicate with any end-systems wishing to utilize it, given the proper procedures.

3.1.3 Proposed high-level technical architecture

We devised a scenario, shown in figure 3.1 where we would have both internal and external devices transmitting over MQTT. Our initial idea was comprised of a centralized system oriented around the Kubernetes architecture. Still, not all devices would be included in such a system, so to better emulate a realistic scenario for testing, we included some stand-alone devices reporting to the broker.

We wanted to have services inside and outside the cluster reporting various activities and metrics. The idea behind having both internal and external service reporting to the Broker that lies on the cluster, was primarily to see if they would respond differently in cases of lackluster connectivity. It would also provide some realism to the scenario.

As discussed in subsection 3.1.2, data can become a new driving force in this specific use case, which is now primarily based on voice communication. The system we are proposing would enable voice, video and sensory data through the use of the MQTT protocol, along with WebRTC. Utilizing MQTT, with its favorable aspects in terms of disseminating large amounts of data, would allow for an autonomous collection of data throughout the cluster, giving responders a clear view in terms of what the situation is like through the use of data. In addition, enabling voice and video would allows responders to get an overview of the situation from afar and maintain a high level of communication between the participating responders.

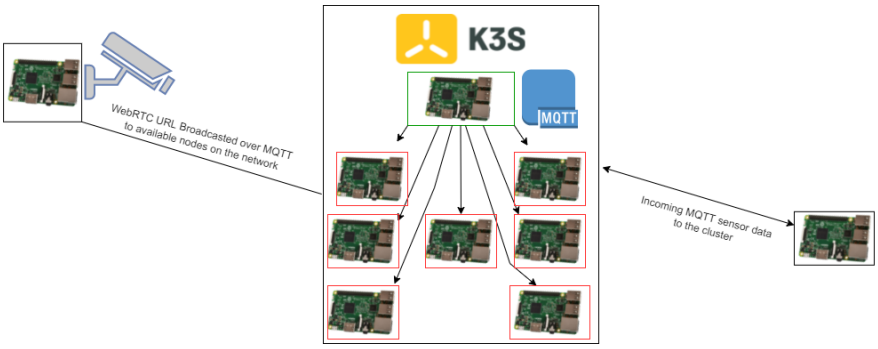

Figure 3.1: Scenario

Figure 3.1 depicts the base architecture for our system. This simple high-level schematic shows two external devices reporting back to the interconnected cluster where the coordinating MQTT broker resides. Within the depiction, Green represent the Master node of the cluster, and red represent the worker nodes.

3.2 Kubernetes

Kubernetes is an open source software used for managing containerised applications across multiple nodes or hosts. It has basic mechanisms for all aspects of application maintenance [60].

Kubernetes results from 15 years of research and development at Google, based on the workloads from Google's own internal cluster manager called Borg which consists of thousands of applications, running over many clusters [66].

In 2014, Google open-sourced the Kubernetes project, and the Cloud Native Computing Foundation currently maintains it.

Kubernetes has proven to be useful in many deployments, specifically in systems that should be able to rectify faults by automatically deploying a new container to prevent unnecessary down-time [60].

Looking back at the different deployments in modern computing, companies originally ran applications on physical servers. In these servers, there was normally no clear way to define boundaries. Obviously, this caused issues if multiple applications ran on the same server. If one server would take up most of the servers resource, the other applications would suffer the consequence. The way to solve this issue would be to separate the applications across different servers, but this would, in many cases, cause hardware to be under-utilised and very expensive in the long run.

A solution to this was the introduction of Virtualisation, which allowed you to

run multiple Virtual Machines or VMs on a servers CPU. VMs allowed for better utilisation of servers through better scalability through being able to present a set of physical resources as a cluster of disposable virtual machines [60].

Containers, such as those used in Kubernetes, are fairly similar to Virtual Machines, but have more abstract properties to share the host operating system among the applications in the containers. Furthermore, due to the nature of containers and VMs, they are generally portable across clouds and operating systems. With all this in mind, Kubernetes and the container deployment is a great choice for this thesis since it offers a lightweight architecture for edge devices [60].

3.3 K3s

K3s was developed by a company called Rancher Labs as an alternative distribution aimed at enabling Kubernetes for small, low powered IoT devices. The K3s distribution is a single binary of less than 40MB, which completely incorporates all the elements of the Kubernetes API [52].

To incorporate the otherwise large executable at 40MB, the extra drivers that did not need to be part of the core executable were removed, but is easily added if the developer needs it. K3s have very low resource requirements, so it is possible to run it on machines as long as it has at least 512MB of RAM [52]. Due to the small size of the binary, it is very fast and easy to install compared to a regular Kubernetes cluster, making it perfect for Raspberry Pi's and other small sized IoT devices.

Considering the size of K3s, it would be reasonable to assume that there would be a considerable difference in terms what is offered compared to Kubernetes. In reality - both have the same capabilities in terms of load-balancing, isolation and deployment procedures [52].

Considering that the distributions are similar in many respects, there are some notable differences, one of them being the database. The default database is used in Kubernetes is a key-value database called etcd. In K3s however, the default database is SQLite. SQLite offers great performance for smaller clusters, with a notable drop-off when reaching a certain threshold [52].

3.4 Kubernetes Architecture

Kubernetes was a platform made by Google, and was released in 2014 under a open-source license. Kubernetes is, in simple terms, used as a tool to help developers and companies to manage and deploy newer versions of software without requiring any downtime. In this section, we will explore the different

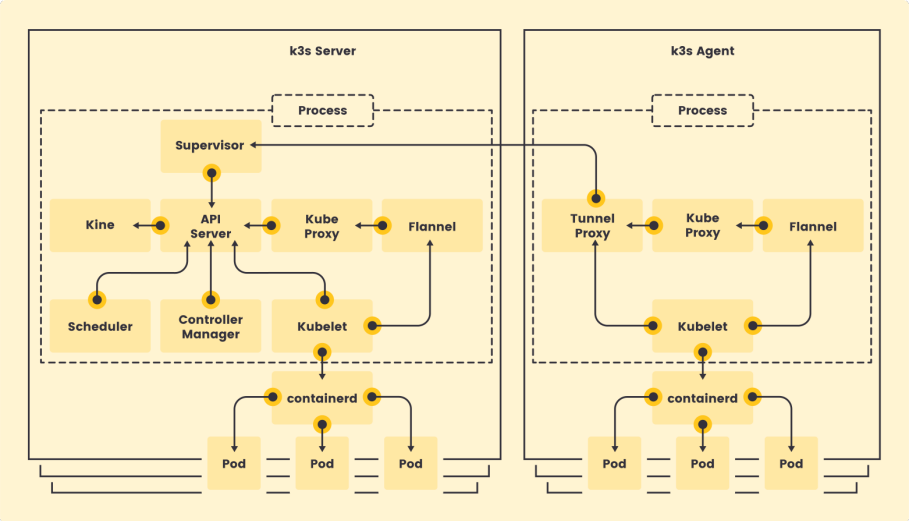

Figure 3.2: K3s Architecture [52]

architectures of K3s and Kubernetes, to better uncover their capabilities for our thesis.

We used K3s [52] as an alternative to the traditional Kubernetes deployment. We will refer to K3s and Kubernetes interchangeably unless there is a significant difference.

One of the most fundamental aspects of Kubernetes is the idea behind a Cluster. A cluster is just an alternative word to describe a system of nodes you propagate using Kubernetes. A cluster is made up of two main types of elements, which are the Master and Worker nodes. The Worker nodes serve a different purpose than the master, as the master contains the control plane.

Figure 3.2 shows the basic high-level architecture for K3s. It mainly consists of two components, the server and the agent, which is synonymous with the master and worker node terminology in the standard Kubernetes Implementation [11]. One of the fundamental things that make K3s more lightweight and easy to run than its counterpart is the fact that in both the server and the agent, the components are grouped together and run in the same process. Whereas in traditional Kubernetes, every component run in a stand-alone process. Even in cases where the cluster only consists of a single node, K3s have the option to run the server and the agent within a single process, making it an ideal distribution for typical low-power IoT devices. As previously stated, K3s and Kubernetes share a fair amount of similarities in terms of functionality, with the differences illustrated in Figure 3.2 as SQLite, Flannel, and Tunnel Proxy components [11].

Regarding the components separating the two deployments, we can start by looking at Flannel. Flannel is K3s own abbreviation for an encapsulation protocol called VXLAN [11]. VXLAN is a protocol that addresses the scalability problems associated with large cloud computing systems [37]. VXLAN is a com-

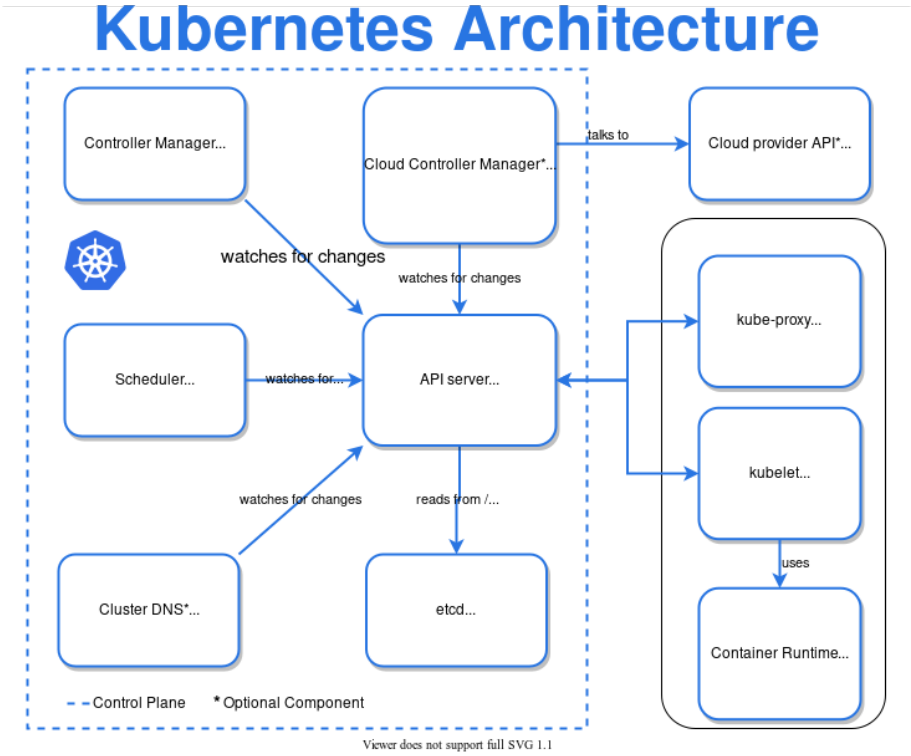

Figure 3.3: Kubernetes Architecture [6]

mon protocol used to create overlay networks that sit on top of existing physical networks to enable virtualization [37]. Within the Kubernetes distribution, it is commonly referred to as a Container Network Interface or CNI [11]. Another precautionary measure in the K3s deployment was to remove the dependency of the etcd database, which is a distributed key-store database in the regular Kubernetes deployment. Lastly, there is the Tunnel Proxy component. The tunnel proxy is a component that enables secure communication between the kube-proxy on the agent and the API server on the k3s server. This allow secure bi-directional communication over a single port, compared to Kubernetes, where the kube-proxy uses a number of ports to communicate with the API-server [11] which would not be suitable for smaller scale IoT clusters.

The differences above are the only aspects where the K3s deployment differs from regular Kubernetes. Figure 3.3 shows an illustration of how the same architecture looks with traditional Kubernetes. For the purpose of this thesis, the Cloud Controller Manager, and the Cloud provider API is excluded as it does not benefit us in our approach. When comparing figures 3.3 and 3.2, we see similarities in the remaining components making up the control plane being the Controller Manager, Scheduler, API-server and Cluster DNS.

The API-server is the component that, as the name suggests, deals with API requests and is the working frontend server of the cluster. The API service also serves to track the state of the different cluster components and deals with the

interaction element between them all [15]. The scheduler is the component in charge of deciding where to run the newly created pods and applications, along with distributing the unscheduled workloads across the worker nodes available in the cluster [17]. Finally, there is the Controller Manager, which is a very essential component in the Kubernetes architecture. The Controller Manager or CM is responsible for running all the built-in internal controllers in the cluster, which handles aspects such as the node and the replication controller. This controller plays a vital part when a node should eventually go down and fail, it will then communicate to the rest of the system that a replication is required to set up a new node [16].

The last essential part of the Kubernetes architecture lies in the Worker node. The worker node is responsible for deploying and running the containers containing the applications. In addition, worker nodes run all the different services that are required for different container to communicate with each other. Finally, the worker nodes communicate with the control plane, which asks the scheduler to delegate resource capacity for containers so that they can be run and installed on particular nodes.

Workers consist of a few different components such as the kubelet, kube-proxy, and the runtime environment for the containers. A kubelet is a component that directly enables the Kubernetes ecosystem, and is responsible for managing the local run-time environment of the container, and doing scheduled work related to aspects such as monitoring and surveillance of resources. The runtime environment starts and stops containers and is responsible for their communication. A typical example of a container runtime that is widely used today is Docker. The final component in the K3s Agent architecture is the networking proxy that serves to routing responses and requests between the pods and the internet.

3.4.1 Pods/Containers

Within the Kubernetes Distribution, some essential linguistics are used to explain the different components, for how it runs, and how to deploy and regulate resources on the cluster.

Before explaining the Kubernetes unique terminology, it is important to know about containers. Containers can be described as a form of enclosure for an application. The enclosure is in a more technical aspect, a form of virtualization [54]. The enclosure can contain components of a larger microservice-based architecture, or a larger executable consisting of an entire application. All the building block within the enclosure are for the code to run as intended, such as binaries, libraries, and config files [54]. Containers are operating system neutral, meaning it can be run across all the platforms that it is being built for, such as X86, AMD64 or ARM. The process of building container images is normally done through the use of software such as Docker, but there are many different alternatives.

Containers are often praised for their many benefits within software devel-

Figure 3.4: Pods Overview [14]

opment, such as less overhead, efficient development, consistency ,and as mentioned earlier, the ability to run across multiple platforms without any added labor. These benefits play back to the fact that containers, compared to alternatives such as virtual machines, does not contain a operating system image. This makes it more lightweight and suited for lower capacity devices, such as the ones existing on the network edge. This, along with the fact that container processes are normally automated, such as building, patching or scaling enhances its capabilities with IoT.

Containers especially gained traction when more and more companies and organizations made a move over to a cloud-based architecture, where most of the ubiquitous computing requires the use of images [54]. With this transition, architectures based on separating components into micro-services became the new normal. A container-based architecture also adds value by allowing applications to be more resilient to failures and allowing applications to be scaled more efficiently than before [54], which we can see with the use of distributions such as Kubernetes.

In Kubernetes, the notion of containers is expanded into the concept of a pod. A pod is a group of one or more containers. Within a pod, there are also components such as shared storage, which is commonly called Persistent Volumes within Kubernetes[14]. Persistent Volumes will be covered in subsection 3.4.6, as it is a reasonably comprehensive component.

The containers within a pod share a common IP address and port space in the cluster. The address and port are, however just their identifiable address from within the cluster, not from the outside. With this approach, containers run in parallel with each other and co-schedule their routines fairly. This process is automatically done by the master and is determined based on the available resource that is present on the node [14].

Illustration 3.2 represents the base overview of a pod is structured within Kubernetes, and shows the steps in which a pod is deployed, in simplistic terminology, the Kubernetes team refers to a pod as an atomic deployment, which can be

noted from the illustration [14]. Pods are also bound to a node within the cluster that they are deployed in, where the pod lives until the node fails, or the pod is deleted. A node can contain multiple pods, as distribution allows for device of lower capacity to host several systems when utilizing Kubernetes. A node, is normally a worker with the cluster and is commanded and maintained by the Master node, or more specifically, the control plane of the master node.

3.4.2 Load Balancing

Load balancing is a familiar concept within modern networking, and more specifically within cloud computing. It describes a paradigm to efficiently distribute network traffic across a series of networked back-end systems [38].

Load balancing has been around since the 1990s, but back then it was a hardware component installed across server parks. The need for load balancers stems back to organization's wanting to improve the performance of applications running on servers [2].

A modern software-based load balancer acts as the symbolic gate in front of the server. It provides routes for incoming requests across all capable and operative servers, which in turn should be helpful to maximize the performance and utilization across the servers so that one should not be overwhelmed in times of high traffic. Along with this, it also serves the purpose to mitigate against failure, which can happen if one server should go down, then the load balancer can compensate for this event by routing the traffic to another server [38].

A Load balancer can be divided into three main categories based on functionality [38].

- The Load balancer contributes to distributing traffic across localized servers
- Provides organizations and applications with high availability and reliability by sharing the traffic.
- Gives the flexibility to add or decrease capacity during high or low activity times.

Building on the premise of high availability and reliability, it becomes clear why it is a central building block in the Kubernetes distribution. Kubernetes and its different distributions, such as the one we utilized in this thesis, K3s all have different approaches to Kubernetes. Within the distributions, load balancing is an aspect that is highly flexible and configurable to suit each cluster's need [38]. Load balancing serves as a flexible solution within clusters to expose deployments, pods, and services to the outside world, and provide the functionality mentioned above.

Kubernetes is a widely used component within modern cloud architecture, where it is being utilized on virtual machines with a designated capacity. The other option is what is called a bare-metal approach [35]. A bare-metal approach is a terminology used to describe clusters that live and run on physical hardware, such as our cluster, running across a set of Raspberry Pi devices.The Kubernetes distribution, both the official and sub-official, like K3s, does not offer a implementation of a network based load balancer for bare-metal clusters. Instead it just offers implementations that are compatible with cloud solution such as Azure, AWS and others [35]. There are however many different open-source alternatives available, MetalLB being one that stands out, since it attempts to provide a network load balancer that is compatible with traditional networking equipment found on regular computer hardware. This opens up new opportunities for experimentation with bare-metal clusters [35].

A more in-depth explanation and comparison of Metallb will be discussed in section 3.4.2 of this thesis. We also covered the steps we used to implement it on our cluster.

3.4.3 Deployments

Deployments are another vital element in Kubernetes. A deploy is used to describe a resource object that provides updates to existing applications, which resides in pods across the system. A deployment consists of a wide range of parameters, the life-cycle events, what images to pull, and the initial number of replicas or pods that should be initialized [19].

A deployment can be looked at as a way to command the cluster of how you, as the developer, wants the computation to be performed. By applying a deployment, it is a way to ensure that the system strives to maintain the applications desired state [19].

Listing 3.1: Deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: foo-deployment
   labels:
        app: foo
spec:
   replicas: 3
   selector:
      matchLabels:
           app: foo
   template:
     metadata:
          labels:
```

```
            app: foo
spec:
  containers:
      - name: foo
        image: foo:1.14.2
        ports:
        - containerPort: 80
```

The code sample 3.1 shows a simplified example of how a deployment script is structured in Kubernetes. Here we have the makeshift application foo-deployment which is defined to replicate itself three times in the system. The application is retrieved from the example image foo:1.14.2 and exposed internally on the cluster through containerPort 80. Most of the scripted code you deploy to Kubernetes share a reasonably similar buildup. The notable exception is the parameter "kind", which separates a deploy from a service, ingress, persistent volume, or any other scripted calls to that is aimed at the system.

3.4.4 ConfigMaps

A configuration file in Kubernetes is called ConfigMaps, and is considered an API object used to store data in key-value pair. Configmaps can be considered as regular environment variables and application arguments[58].

Configmaps work to decouple the configuration containers so that applications are more easily portable both across the cluster and across systems.

Configmaps are restricted in size, not exceeding 1Mib of data to prevent an exceeding amount of data from continuously passing through the system.

Configmaps prove especially useful in scenarios where applications have different declarations and configurations for development and production, allowing for separate sets of the same application to share similar code, but not similar configurations [58].

We utilized Configmaps in several cases. The most important being the TLS certificates required to run a secure MQTT-broker.

Listing 3.2: ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mosquitto-config
  namespace: mosquitto
data:
    foo.conf : /-
```

A typical Configmap deployment manifest can be seen in listing 3.2 and is pretty

similar in buildup compared to the other kinds of objects in Kubernetes, with the notable exception being the data declaration towards the end of the file.

3.4.5 Services

A service in Kubernetes is an abstraction used to represent a set of deployed pods within a Cluster. A service's main job is to connect a given set of pods to a higher service name, along with the port number and IP address. When a service groups together a collection of pods, it opens up for discovery and routing internally between pods in a cluster [64]. There are several different kinds of services, like LoadBalancer, which we covered in section 3.4.2, along with Cluster-IP and Node-Port. Node-Port opens up a service through the use of a static port on all the participating nodes, while Cluster-IP, which will be listed below, opens up a service to the rest of the cluster [64].

A typical example of how a service operates can be thought of as a composition of a back-end and front-end system, both co-existing within a Kubernetes cluster. Since the front and back end of the application might be run across different pods and nodes, then the service helps group together the co-existing pods across multiple nodes within the cluster [64]. Therefore, a service can be seen as a binding element between applications that require each other to function.

As all other Kubernetes objects, a service is defined in the YAML format.

Listing 3.3: Service

```
apiVersion : apps/v1

kind: Service

metadata:

name: service-frontend

spec:

  - port:4000

protocol: TCP

targetPort: 5454

selector:

run: service-frontend

type: ClusterIP
```

Listing 3.3 shows a typical setup for the makeshift object "service-frontend." When applying this object to the cluster, any pods within the cluster would be able to access it on port 5454 under the name "service-frontend"[64]. This example shows a typical ClusterIP build, which would also open for pods to utilize the main specification port, set to 4000.

Many objects within Kubernetes are bound to each other, such as Deployments, Services, and LoadBalancers. Still, as seen here, even abstracted services serve their unique purposes when deploying applications to a cluster [64].

3.4.6 Kubernetes Volumes

Kubernetes, like most other systems, are based on storage. Within Kubernetes it is crucial due to the nature of the fault-tolerance.

The umbrella terminology used to explain the different ways to persist data in Kubernetes is simply called Kubernetes Volume. Under this terminology, Persistent Volume, Persistent Volume Claim and Storage class, are the 3 main approaches to persisting data in Kubernetes[36].

The concept of persistent storage plays a vital role in determining the viability of Kubernetes for HADR operations. Persistent Volumes will be the answer in terms of reconciling data and information from being lost on nodes that might fail.

The basic needs for storage within applications are often provided through a database, such as MySQL. If you put that context into the bigger Kubernetes picture, things change. This is because applications reside in a container image that is enclosed within a pod that resides on one or many nodes throughout a system. This fact pertains to storage as well, which could be deployed as a separate pod residing together with application specific containers, which rely on each other for functionality[36].

Persistent Volume, however describes the next layer above this, which deals with storage related to when or if nodes should fail then the data defined and stored within these pods, have to be retrieved and set up on a new node, but still keeping the information that was stored prior to the nodes failure, which is exactly what persistent volume deals with in Kubernetes.

Three characteristics often characterize Kubernetes Volumes. First of all, as mentioned earlier, Storage must not be dependent on the pod life cycle, it must be available across all nodes, and storage needs to survive even if the entire cluster crashes [36].

Persistent Volume is the first of the three main pillars of Volume storage. Persistent volume is a cluster resource equal to CPU or RAM capacity. Like all other Kubernetes objects, it is defined and applied as a YAML file.

Listing 3.4: Persistent Volume

```
apiVersion : foo/v1

kind: PersistentVolume

metadata:

name:

spec:

capacity:

storage:

volumeMode:

accessModes:

ReadWriteOnce

PersistentVolumeReclaimPolicy: Recycle

storageClassName: slow

mountOptions:
```

Listing 3.4 shows a typical template YAML file that would be used to define the Persistent Volume resource. You can note from the listing, that the kind is specified as a PersistentVolume, and the spec is used as a sheet to define the capacity, volumeMode, and accessMode for the storage.

Examples of Persistent Volumes are typically cloud-based storage such as Amazon AWS or NFS servers. The alternative that is popularized on bare-metal clusters such as ours is simply using the local storage on the participating nodes.

A Persistent Volume claim, which is the second way of defining storage, is done similarly to a Persistent Volume. A Persistent Volume claim is a request for a new container, pod or application. Persistent volume claims are defined as a request to already existing Persistent Volumes on the cluster. Based on the desired specifications defined in Persistent Volume Claim, a suitable Persistent Volume will be chosen. This is done so that applications have easily scale up their persistence as they grow. A persistent volume claim can be done dynamically from other operations and compared to adding persistent volumes; it can be done after an application image has been deployed to the cluster [36].

A Persistent Volume Claim has to be deployed in context to a Pod deployment

or any similar deployment. In this YAML used to deploy the Pod, there has to be a reference to the Persistent Volume Claim, for them to be matched up with each other. Otherwise the Claim will reside in the Cluster until a pod has referred to it [36].

Storage Class is the final pillar of Persistent Volumes in Kubernetes and is typically the case for large-scale applications and enterprise-level systems. In addition, a Storage class can provision Persistent Volumes dynamically when a Persistent Volume Claim is made on the Cluster, henceforth automating the entire process mentioned above [36]. A Storage Class is, however a component that largely resides outside of the scope of this thesis, and on lightweight clusters such as the ones covered in this thesis.

Persistent Volumes offer a fair bit of challenges due to the amount of computations required. This makes it a challenge to incorporate on lightweight devices such as the Raspberry Pi. Projects are, however, ongoing and being released to meet the requirements of lightweight edge-computing.

3.4.7 Secrets

Applications and software in general relies heavily on security, both in terms of communication and to protect internal systems.

To keep keys, certificates and other types of classified information hidden from prying eyes, software usually retrieve them from remote locations or keep them stored encrypted within the source code.

Kubernetes uses the principles of secrets to store confidential information such as tokens, keys, certificates or password. Utilizing secrets to keep and preserve the confidential aspects used in application code allows the developer to not include the data directly in the application code [59].

Secrets are created and deployed independently of the Pods that use them, which allows for a less chance for secrets and its data to be reveled during creating, editing and maintaining Pods[59].

The Secrets object in Kubernetes also has different added features, allowing for configuration for several use-cases, such as preventing the data contained within secret to be written to nonvolatile storage, and special reading writes to prevent unauthorized access [59].

The overall, base implementation of the Secrets object in Kubernetes is not a security measure per se, as it is stored un-encrypted in the API servers data store, namely etcd on Kubernets, and SQLite on K3s [59]. Storing it in this manner would allow for anyone with access to the essential data store, to query and view the underlying data.

3.5 Hardware

3.5.1 Raspberry Pi

The Raspberry Pi is a small computing device made by a foundation under the same name. The foundation is based in the United Kingdom and is a charity that works to provide children, schools and youth organizations with the tools to help them learn coding and digital creation [18].

The Raspberry Pi is a low cost product available in almost every country in the world. It has the appearance of a circuit board, but is in reality a full-fledged computer at about the size of a credit-card, and has the ability to do most things a regular desktop computer can, only being restricted by the capacity of hardware on it [18].

The Raspberry Pi has been through several iterations since they came about around 10 years ago, and since then, they have aimed to be prized at around 15-40 dollars, depending on which version [18].

In this thesis, we utilize multiple Raspberry Pi 3 Model B as the worker nodes in our Kubernetes cluster, and a Raspberry Pi 4 Model B as a master node and control plane . The Raspberry Pis used for this project have the following specifications [33],

Raspberry Pi 3 Model B:

- ARM Cortex-A53 1.2Ghz processing unit
- 1GB SRAM
- Integrated Wi-Fi supporting 2.4Ghz, and Ethernet support
- Bluetooth 4.1
- Integrated SD-Card Reader

Raspberry Pi 4 Model B:

- ARM Cortex-A72 1.5Ghz processing unit
- 4GB LPDDR4 SDRAM
- Integrated Wi-Fi supporting 2.4Ghz and 5Ghz, and Ethernet support
- Bluetooth 5.0
- Integrated SD-Card Reader

For our devices, we used a San-disk Ultra 32GB SD-Card with a Read/Write speed of 98MB/s.

Raspberry Pi has also become a very popular device in the general market as well, due to it's unique ability to interact with other devices and have been used in everything from robots, weather stations, cameras.

Due to the massive popularity that the Raspberry Foundation received for their devices, many different alternatives to came on the market which we will not cover in this thesis. The alternatives also have very suitable qualities, yet the Raspberry Pi remains a favorite on the market due to its ease of use and low cost of entry.

3.5.2 Raspian Buster

The Raspberry Pis used in this thesis runs the same operating system, Raspian 10, more commonly called Buster.

The Raspian operating system is a free OS specifically suited for Raspberry Pi hardware. It contains all the necessary functionality to utilize the most out of the Raspberry Pi hardware [50].

Contrary to belief, the Raspian OS is not affiliated with the creators of the hardware but was created and is maintained by a small team of developers.

The operating system has primarily been distributed in a 32-Bit version, but as of February 2022, Raspberry Pi OS became available as a 64Bit version [50].

Raspian OS is based on the Debian, being one of many popular distributions for Linux. Since its release back in 2021, it has gone through 4 iterations, with the latest one being released in 2022, based on Debian 11 "Bullseye" [50].

Chapter 4

Implementation

The utilities deployed to our cluster stems from pre-compiled images compiled and available from Docker's open-source repository. There are still several projects in early stages of development due to the continuous innovation happening with the Kubernetes architecture. We made adjustments to existing images to complement our use case when we deemed it necessary. If changes were made to existing images, we covered it in its respective section. For example, in section 4.6 we built ARM64 images, and published them to Docker Hub.

We made adjustments to existing images to complement our use-case when we deemed it necessary.

4.1 Preparation

4.1.1 Hardware

Once the Raspberry Pi was ready, we started flashing the 32Gb SD Cards to become identical copies of themselves. The devices use the 64Bit version of the Raspian Buster operating system. Raspian Buster is the predecessor of the current version called Bullseye. However, there are some reports about incompatibility when running K3s with Raspian Bullseye, which made it a safer choice to go with the lastest version of Buster, which is based on the Debian 10 Linux Kernel

In retrospect, we have noticed that we could have rather used Raspian OS Lite. Raspian Lite is a smaller OS in terms of space, only occupying 400Mb. K3s does however always suggest using a 64Bit OS as most images are prebuilt for that, contrary to 32Bit, which historically has been the standard.

We utilized Raspberry Pi Imager to easily flash all the devices with the same distribution, speeding up the process significantly compared to command-line based approaches. During the imaging process, we automatically enabled SSH

and added our network configuration to configure them from the terminal on our local machine directly.

To preserve most of the RAM capacity on the Raspberry Pi's we deactivated the processes related to having a graphical UI on the Raspberry Pi's and enabled them to boot into terminal mode instead. We also went into the rasp-config and decreased the default RAM allocation to the GPU from 64mb to 16, to further allocate memory for our system. When validating, we noticed around 8 percent less RAM usage, compared to when the GUI was activated. During the configuration, we also added unique host-names to each individual Raspberry, as it is a formal requirement for running K3s.

4.1.2 K3s

We used the official documentation available at Rancher's website[48] to set up the K3s Cluster. There are several different ways to install K3s, as there is a thriving community of open-source developers backed by the creators of Kubernetes.

The official documentation has some prerequisites prior to doing the installation. For linux distributions published after 2015, one has to enable legacy Iptables, as K3s rely on the original getsockopt/setsocketopt-based interface in the kernel to allow for horizontal distribution between nodes.

For devices running ARM64, we also had to enable the control group feature, more commonly known by is abbreviation as cgroup, which has to be added in the kernel.

Listing 4.1: cmdline.txt

```
cgroup_enable=1 cgroup_enable
    =memory cgroup_enable=cpuset cgroup_memory=1
```

Listing 4.1 provides the necessary features that have to be appended in /boot/cmdline.txt. K3s also occupies port 6443, which has to reserved across all participating nodes in the cluster. Port 6443 is the port which makes nodes accessible to each other in the cluster.

4.1.3 Installation

Installing the traditional Kubernetes distribution is usually a tedious job, consisting of multiple steps of setup and installation. K3s differ quite significantly from this by having downloading and installation grouped in one command.

Listing 4.2: k3s-install-master

```
export INSTALL_K3S_EXEC
    = --no-deploy servicelb --no-deploy traefik
```

```
curl -sfL [https://get.k3s.io](https://get.k3s.io/) | sh -

grep /var/lib/rancher/k3s/server/node-token
```

After using SSH to access the device we wanted to use as our Master node, we ran the curl command in listing 4.2 downloading and initializing all the necessary components for the master node and control plane. During our discovery, we noticed that the integrated traffic distributor and load-balancer, namely servicelb and traefik had a significant memory overhead on bare-metal clusters, which opted us to find an alternative load balancer more suited for our use-case. We, therefore, chose not to deploy servicelb and traefik as optional arguments during the installation on the master node. To set up worker nodes corresponding to this newly set up master node, we also had to get the unique token generated during installation, as it will be used as a command line argument to install workers. The token can be retrieved from the master node by using the grep command provided in listing 4.2.

Listing 4.3: k3s-install-worker

```
export K3S_URL=https://myserver:6443
export K3S_TOKEN=mynodetoken

curl -sfL [https://get.k3s.io](https://get.k3s.io/) | sh -
```

To install a worker corresponding to the master, we ran the curl command provided in listing 4.3, where the K3S-URL corresponds to the IP Address of the master node, and the K3S-TOKEN argument was the value retrieved from the final command provided in listing 4.2.

4.1.4 Kubernetes command line

Rancher's setup only goes as far as setting up everything locally on the nodes. To be able to access and perform operations on the cluster from our local machine on the same network, we had to set up the Kubernetes command line client, called kubectl.

Once Kubectl was installed on our machine, we copied the content of the configuration file over to our local machine in the appropriate path for the client to find it, which can be seen in listing 4.4. This enabled us to perform operations on the cluster with the use of the kubectl client on our local terminal, making the cluster operational.

Listing 4.4: Move config to local machine

```
scp pi@master
    -ip-address:~/etc/rancher/k3s/config.yaml ~/.kube/config
```

4.2 Implementing a Load Balancer

We worked with devices that generally had low capabilities in terms of storage, memory and processing power. Therefore, some modifications had to be made to the original K3s distribution, with one of them being the Ingress Controller called Traefik. An Ingress controller is a specialized version of a LoadBalancer that serves a number of purposes in Kubernetes. For a more detailed description of LoadBalancer and Ingress Controllers, please refer to subsection 3.4.2.

During initial testing, we discovered that the traditional implementation of Traefik had quite a significant overhead in terms of RAM usage, peeking at upwards of 180 Mb continuously when running very few pods. It accounted for roughly 20 percent of the entire memory usage on the control plane when running, we wanted to explore alternatives.

MetalLB presented itself as a nice complementary LoadBalancer, which can be used in harmony with Traefik. However, considering our initial problem, we chose to run MetalLB as a stand-alone LoadBalancer, as we could not benefit from the cloud connectivity Traefik offered. For our thesis, we did not identify any particular case in which a LoadBalancer would be instrumental. A load balancer would however be beneficial for development and testing, as it would autonomously delegate resources and services between nodes in case any of them would disconnect.

MetalLB is an open-source LoadBalancer made especially for bare-metal clusters, using traditional routing protocols that exist locally on the networking cards on the hardware [35]. This separates MetalLB from the network LoadBalancers that is implemented in the standard distribution of Kubernetes and K3s, as those LoadBalancers are made and shipped with the intent of running it against Infrastructure as a Service platforms such as Microsoft Azure and Amazon AWS [35].

So for developers that wish to not be reliant on a IaaS platform, LoadBalancers will remain in a Pending state, and never reach its full potential. MetalLB however, does the work locally on the networking equipment, and aims to provide applications and services on bare-metal implementations to be exposed as good as possible [35].

The implementation of MetalLB, along with many existing sandbox projects are fairly easy to implement, given the correct preconditions, as mentioned above.

We initialized MetalLB by pulling the latest manifests from their official Github repository

Listing 4.5: MetalLB Install Manifest

```
kubectl apply -f [https
    ://raw.githubusercontent.com/metallb/metallb/v0.10.2/
    manifests/namespace.yaml](https://raw.githubusercontent
    .com/metallb/metallb/v0.10.2/manifests/namespace.yaml)
```

```
kubectl apply -f [https
    ://raw.githubusercontent.com/metallb/metallb/v0.10.2/
    manifests/metallb.yaml](https://raw.githubusercontent
    .com/metallb/metallb/v0.10.2/manifests/metallb.yaml)
```

The first manifest simply contains a name-space declaration. Name-spaces offer a unique feature of grouping related tasks together, and also isolating them within a single cluster. In this way, they are able to run in a parallel execution with other tasks, and their failure primarily affects other resources also contained within the given name-space [53].

Listing 4.6: MetalLB Namespace configuration

```
apiVersion: v1

kind: Namespace

metadata:

name: metallb-system

labels:

app: metallb
```

Listing 4.6 shows the simple implementation file for creating a dedicated namespace. This follows the same notion throughout all namespace implementations, except being a unique name under metadata.

The next manifest file contains the unique code implementation and images of the MetalLB load-balancer.

MetalLB also required a unique configuration file containing the implementation and declaration of a pool of IP addresses. It could utilize it to expose applications outside of the local cluster nodes.

Listing 4.7: MetalLB Configuration file

```
apiVersion: v1

kind: ConfigMap

metadata:

namespace: metallb-system

name: config

data:

config: *|*
```

```
address-pools:

- name: default

  protocol: layer2

            addresses:

            - "192.168.0.240-192.168.0.250"
```

The listing 4.7 shows a typical configuration file, where data contains a unique name for the address pool, to make it easier to reference from other declarations, along with a protocol and address declaration that suits our network and router. When this was deployed, MetalLB utilized the address space declared in listing 4.7, across nodes, pods, and namespaces.

4.3 Kubernetes Dashboard

Kubernetes dashboard is a web-based user interface that provides the same functionality as the command-line client, along with several other helpful features such as visualization, troubleshooting options, and general management of clusters and the pods within them.

The dashboard is a beneficial tool when deploying various applications, since it provides links between different pods, services, and deployments to help visualize their interactions.

We deployed the Kubernetes dashboard to our cluster, specifically for the purpose of error handling and logging, since some of the containerized applications we were running had some experimental functionality that need to be monitored.

Listing 4.8: Kubernetes Dashboard Install Manifest

```
kubectl apply -f https://raw.githubusercontent.com
    /kubernetes/dashboard/v2.0.5/aio/deploy/recommended.yaml
```

Listing 4.8 references the official executable made by the Kubernetes team, which is compatible across the different distributions of Kubernetes, such as K3s.

Once the installation is complete, we had to configure a root account to access all fashboard metrics. This account is commonly called a a Service account in Kubernetes, since it most often is a widely distributed set of applications, with dedicated personnel doing maintenance.

Listing 4.9: Kubernetes Dashboard Configuration

```
apiVersion: v1
kind: Namespace
metadata:
      name: kubernetes-dashboard

---

apiVersion: v1
kind: ServiceAccount
metadata:
      name: admin-user
      namespace: kubernetes-dashboard

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
    name: admin-user
roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: cluster-admin
subjects:

    kind: ServiceAccount
    name: admin-user
    namespace: kubernetes-dashboard
```

Listing 4.9 shows the setup that we used to make an administrative account, which would then contain the token that we used to access and gain administrative privileges with.

4.4 Mosquitto MQTT

The implementation covered in this section utilizes the official docker image published by the creators of the Broker, Eclipse [20]. Existing documentation from the creators does not cover implementation details for Kubernetes.

4.4.1 Open Implementation

The MQTT broker from Mosquitto Eclipse is a well known broker within the IoT community. It offers fair scalability for smaller systems, with a considerable drop off when reaching around 10 000 devices. This is due to the coordination mechanisms in the broker, making it generally unsuitable for enterprise level implementations. For the purpose of this thesis it was suitable, as we were working with less than 10 devices in total.

Initially, we created a simple deployment file which contained a reference to the Eclipse docker image.

Listing 4.10: Mosquitto Deployment File

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: mosquitto-deployment-open
    namespace: mosquitto
spec:
  selector:
    matchLabels:
        app: mosquitto
replicas: 1
template:
metadata:
   labels:
     app: mosquitto
spec:
  containers:
    - name: mosquitto
      image: eclipse-mosquitto:1.6.15
      imagePullPolicy: Always
      ports:
      - containerPort: 1883
```

Listing 4.10 shows the deployment manifest required for deploying a mosquitto-broker, which will listen on port 1883, which is what the Eclipse Foundation recommends for as an unauthenticated port.

Listing 4.11: Mosquitto Service Deployment File

```
apiVersion: v1
kind: Service
metadata:
    name: mosquitto-service-open
    namespace: mosquitto
spec:
    type: LoadBalancer
    ports:
        - name: mqtt-1883
          port: 1883
          targetPort: 1883
selector:
    app: mosquitto
```

To expose the mosquitto deployment, we had to make a service manifest to make it accessible. Since we already implemented MetalLB we implemented the service as a LoadBalancer, which can be under spec in listing 4.11. This allows MetalLB to read the configuration of Mosquitto with its pre-defined

ports. With this approach, we achieved a outwards facing broker, and a logical service between the nodes on the cluster.

4.4.2 SSL / TLS Implementation

Due to the nature of our use case, an important aspect was security - both across devices and also for the systems being used. During development, we decided to add SSL certificates to enable the MQTT broker to use TLS security.

Existing documentation listed a TLS implementation as experimental, due to the nature of how IP-addresses and services are exposed to outside the cluster.

We initially started of by creating,

- A Certificate of the CA or Certificate Authority that has signed the brokers certificate, which will be used on both the client and broker side.
- CA certificate for broker.
- Server private key that will be used for decryption

This approach allows us to only use trusted server certificates for our applications. In larger scale applications, adding individual username and password authentication and generating unique client certificates and keys would be more suitable. This is because sharing a common server certificates among a large deployment of devices would jeopardize security.

To create the necessary certificates, we used the software by OpenSSL.

Listing 4.12: OpenSSL Commands

```
1 - $ openssl genrsa -des3 -out ca.key 2048

2 - $ openssl
    req -new -x509 -days 1826 -key ca.key -out ca.crt

3 - $ openssl genrsa -out server.key 2048

4 - $ openssl req -new -out server.csr -key server.key

5 - $ openssl x509 -req -in server.csr -CA ca.crt
    -CAkey ca.key -CAcreateserial -out server.crt -days 360
```

Listing 4.12 shows the necessary commands needed to make the certificates. The first command produces a ca.key file, which is a key pair that we used to verify the certificate authority. The second uses the ca.key file to request a certificate for the CA. Command 3 generates a server key, like we did with the first command. The fourth command creates a certificates request for using the server.key file to

complement the ca.crt file. Lastly, the fifth command uses the ca.key file that we created with the first command to verify and sign the server certificate, which in terms creates the server certificate that we used on the broker on our system.

Listing 4.13: Creating Kubenetes Secrets

```
kubectl create
    secret tls mqtt-tls --cert=server.crt --key=server.key

kubectl create secret generic mqtt-ca --from-file=./ca.crt
```

When the necessary certificates was made, we had to experiment with how correctly implement the certificate on to the cluster to be read by the broker. Several approaches exist, such as defining them individually through the use of a ConfigMap manifest. We eventually decided upon implementing the certificates as secrets internally on the cluster using the commands provided in listing 4.13. We covered secrets in subsection 3.4.7.

With the recent added complexity, several modifications had to be made to the deployment and service we covered in subsection 4.4.1.

The updated deployment manifest, which can be viewed in Appendix A shows the differences, primarily in terms of mounting the secrets that we deployed to the cluster, along with a unique Mosquitto configuration file. In addition, to initiate locally persistent storage for the node, we also allocated a persistent volume claim to be used by Mosquitto. This is particularly helpful in cases where the node disappears and re-appears, so it could continue execution without having to set itself back up.

Listing 4.14: Mosquitto.conf

```
apiVersion: v1

kind: ConfigMap

metadata:

name: mosquitto-config

namespace: mosquitto

data:

   mosquitto.conf: /-

log_dest stdout

protocol mqtt

tls_version tlsv1.2
```

```
listener 8883

allow_anonymous true

cafile /mosquitto/tls-ca/ca.crt

certfile /mosquitto/tls-server/tls.crt

keyfile /mosquitto/tls-server/tls.key
```

In the Mosquitto configuration we defined port 8883 for secure communication, as per the official documentation [21]. Due to a fall-back error in the official mosquitto image, we specifically specified that the broker should use TLS version 1.2 to prevent the broker from misinterpreting certificates in past formats. We did not enable user authentication with username and password for this thesis.

The Mosquitto configuration in Listing 4.14 is similar to traditional broker systems on any other arbitrary operating system, but it is loaded as a ConfigMap in Kubernetes and referenced in the Deployment, which can be seen in appendix A under section Volumes and volumeMounts as Mosquitto-config and mounted in the virtual container under /mosquitto/config/mosquitto.conf.

Appendix B shows the changes made to the service file, which is no more than another declaration for allowing for incoming traffic to the cluster over port 8883, which again is delegated towards the mosquitto deployment.

Appendix C is the necessary code for the persistent volume claim on our system, which contains the reference to the local-path storageClassName that we previously configured on the cluster.

4.5 Storage

4.5.1 Local Persistent Storage

In subsection 3.4.6 we introduced the concept of Kubernetes Volumes, along with the challenges of implementing a distributed persistent storage solution on a bare-metal cluster.

Most Pods and containerized applications do, however require storage in some fashion, for either momentary data or configurations required to run the containers. As for our implementation, we chose to utilize Persistent Storage available locally on the device.

Having locally enabled storage would enable the cluster to configure a local-path on the node to store data belonging to the container running on that given node. On typical Raspberry Pi implementations, it is configured by making a

hostPath backend persistent volume claim, which is a reference to a part of local storage which can be utilized for storing data locally on nodes.

Listing 4.15: Persistent-Volume Local Path

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: local-path-pvc
    namespace: default
spec:
  accessModes:
      - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 15Gi
```

Listing 4.15 shows the base implementation of a local persistent volume claim from Ranchers documentation [49]. To enable Rancher's Local Path Provisioner [49], we also had to implement a pod, which would allow for resources on the cluster to access the local path storage, which can be seen in listing 4.16

Listing 4.16: Persistent-Volume Local Path Pod

```
apiVersion: v1

kind: Pod

metadata:

name: local-path

namespace: default

spec:

containers:

-name: local-path-test

 image: nginx:stable-alpine

 imagePullPolicy: IfNotPresent

 volumeMounts:

name: local-path-pvc
```

```
mountPath: /data

ports:

-containerPort: 80

volumes:

- name: local-path-pvc

  persistentVolumeClaim:

        claimName: local-path-pvc
```

Once these manifests were applied, they were configured in a bounded state. This allowed for external containers to access the storage through creating a claim, referencing the name found under volumemounts, which we configured to **local-path-pvc**.

4.5.2 Longhorn

Longhorn is a software developed by the same company as the one that published and maintains the K3s distribution[47].

Longhorn is a lightweight distributed block storage system created especially for K3s, but it is also compatible across other Kubernetes distributions.

As of March 2022, Longhorn is currently marked as experimental if implemented on a cluster consisting of devices with ARM64 architecture, such as the ones we utilize for our testing [47].

Longhorn is based on a paradigm of block storage. The fundamental principal is that each storage volume acts as an individual storage unit. Based on the block storage paradigm, data is saved to the storage drive as fixed-sized chunks, referred to as blocks [12]. Furthermore, every block in the storage unit is identified with a unique address, which is the only identifiable aspect of the each block [12].

Longhorn is an innovative approach to the idea of distributed storage, which previously was unheard of concept within lightweight solutions such as bare-metal clusters [47]. Longhorn simplifies the perceived storage model by implementing a large block storage controller into a small number of storage controllers allowing for a powerful storage paradigm to be utilized on low-capability devices such as Raspberry Pi's. This in terms culminates to significantly increased capacity on Edge devices compared to before.

One of the examples brought up by the developing team at Longhorn, regarding

its capabilities for ARM64 devices, is the trouble related to the capacity of the storage units on smaller single-board devices. The problem lies in the significant amount of read and write operations required by the Longhorn Manager, which is the component responsible for coordinating the blocks across the cluster. The significant amount of read and write operations puts a heavy strain on devices, that in many cases relies on smaller capability SD-card which is considered unsuited for these operations [47].

As an alternative approach, Longhorn suggests putting the computations required by the software to dedicated storage nodes in the system. In regards to our use-case, we propose implementing a stand-alone storage unit. This node would serve no other purpose than acting as a unit of storage for the containerized applications in the cluster. This could solve the problem of storage, allowing for a separate entity to restore the data that might get lost, as nodes could drop-off or fail.

4.6 Scripts and code

In subsection 3.1 we covered a fairly simple architecture overview for the different services and programs we wanted to use as a basis for this thesis. In this section we will cover the various services and applications, along with how we implemented them.

The code samples referenced in thi section will be available in the appendix section at the thesis's end.

4.6.1 K3s MQTT Clients

For testing, we made three different containerized versions of the code in appendix D.

The code follows the structure provided by Eclipse Paho, which is the same developer responsible for the Mosquitto broker we implemented in the K3s cluster.

The essential parts of the code lie in the connect and publish functions, where we establish the connection to the broker and validate our certificate.

In the publish method, we generate a variable "msg-count" which we utilized to see in the server log for dropped messages during testing. The messages are pseudo generated with a timestamp and a range of values for a pre-determined set of devices, such as a fan controller, temperature controller, and a Becquerel controller.

When the values are read, we packaged tehm using json.dumps from the JSON framework available in Python. We then publish the payload with the pre-

defined topic set at the top of the code sample.

Lastly, we call the run method that connects the client to the broker. Since we wanted the clients to continuously report, we made a call to the function "loop-start" establishing a new thread and handling re-connections automatically before publishing the payload generated in the publish method.

This code is replicated three times to simulate three different clients reporting to the network.

4.7 Versions

Software	**Version**
Raspi-OS Buster ARM64	Published: 07.05.2021
K3s distribution	v1.19.13+k3s1
Kubernetes Dashboard	v2.0.5
MetalLB Load Balancer	v0.10.2
Eclipse Mosquitto MQTT Broker	v1.6.15
Grafana	v7.5.2
Nginx Ingress	v4.0.1
Python	v3.9
Docker	v20.10.8 Build 3967b7d
Rancher Longhorn	v1.1.0

Table 4.1: Table of Versions

Chapter 5

Testing

The concluding step in our thesis was testing. Our goal was to get a impression of how a distributed computing approach would operate under various circumstances, reminiscing of the network conditions it would face in a real-world scenario.

This section will cover various metrics we gathered from the cluster. The most significant part of our testing was done with various network configurations, but we do however include other interesting metrics such as memory usage, as it could be a valuable resource to determine what type of hardware to use.

5.1 Memory Usage

An important part of verifying the validity of our proposed solution was to document the memory usage across the cluster. Mainly since we wanted this to be running on typical IoT devices with limited hardware capabilities.

During development, we continuously worked on implementing the lowest resource-intensive alternatives what was included in the K3s distribution. This also included the extra frameworks we used to make the cluster operate autonomously. Choices like MetalLB, Eclipse MQTT Broker and the removal of unnecessary connections to infrastructures such as Amazon AWS and Microsoft Azure contributed to the low memory footprint we achieved across the devices.

Name	CPU(cores)	CPU%	Memory(bytes)	Memory%
HADR-Worker-1	212m	5%	570Mi	62%
HADR-Worker-2	170m	4%	570Mi	62%
HADR-Worker-3	187m	4%	472Mi	51%
HADR-Worker-4	165m	4%	558Mi	62%
HADR-Worker-5	204m	5%	568Mi	62%
HADR-Worker-6	197m	4%	569Mi	61%
HADR-Worker-7	199m	4%	562Mi	62%
HADR-Master	373m	9%	1384Mi	36%

Table 5.1: Average metrics

Table 5.1 illustrates an average reading we collected on 10 separate occasions during run-time. This was a helpful, as it gave us a general idea in terms of how much memory was actually needed to operate a cluster of this size.

What we can denote from this is that the latest developments of Kubernetes, in which K3s is tightly related to, has brought with it some more resource intensive aspects. From the table, we can see that 1GB will be sufficient for operating a MQTT Broker and a fair bit of clients, while also leaving some space for running other services.

The Master node, which is listed at the bottom of table 5.1 illustrates that using K3s for its intended purposes requires a fair bit of capacity, specifically in terms of memory. For earlier reference, this was the node that was substituted during development, in favor of a Raspberry Pi 4 with 4Gb of local memory, compared to the Raspberry Pi 3 with 1GB, which we initially started using.

5.2 Gilbert-Elliot Model

Gilbert-Elliott model is a simple burst error model formalized by Edgar Gilbert and E. Elliott back in the early 1960s at Bell-Labs [29].

The model is based on a mathematical model called the channel model [34], which is described in a variety of ways. The model introduced by Gilbert and Elliot is widely used for describing burst error patterns for transmission protocols, and is commonly used to simulate digital errors in communication links, which is what we used it for.

A channel model is , simply put, a mathematical representation of how a communication channel with wireless signals is affected. Gilbert and Elliott's model is based on a Markov Process or Chain, where you have two states, often represented by G and B, for good and bad. When we exists in the G state, there is a probability determining the chance of transmitting a bit correctly, which is represented by k. At the same time, if the channel is in a B state, the probability is derived by h.

The Gilbert-Elliott model has some different representations based on the use case. For our usage, we utilized the function within TCPDump 5.3.2, where the model is represented in equation 5.1.

$$<p>[<r>[<1-h>[<1-k>]]] \tag{5.1}$$

In equation 5.1, there are a few different parameters to note,

- **p** is the probability of going from a good to a bad state
- **r** is the probability of going from a bad to a good state.
- **1-h** is the probability of a transmit when in a bad state.
- **1-k** is the probability of a loss when in a good state.

We applied this model to the network to simulate an unpredictable set of events on the communication links. We will touch closer on the actual usages in subsection 5.5.1

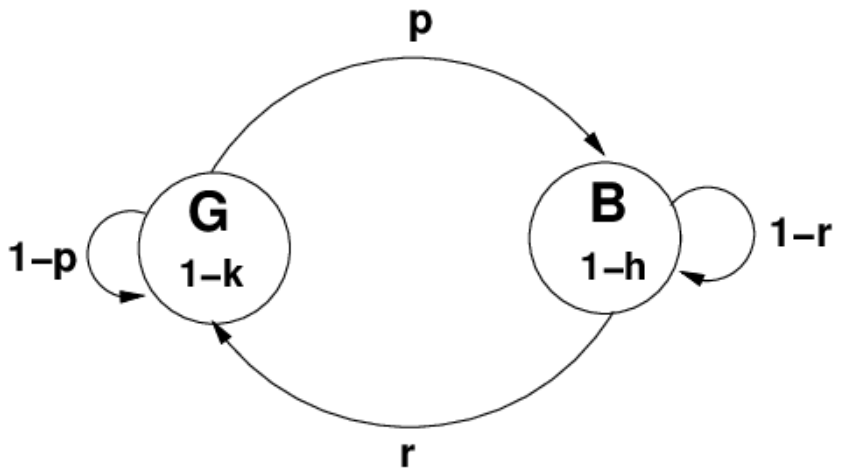

Figure 5.1: Gilbert Elliott Model [28]

Figure 5.1 shows the flow of the model, representing the two different states that the model can traverse between, with the **p** and **r** parameters representing the probability of transitioning between the good and bad state. **1-p** has a technical difference compared to the Linux netem implementation, where it is referred to as **1-k**, representing the probability of a loss in a good state. Lastly, **1-r** also has a slight difference from netem, where it is represented with **1-h**, being the probability of a transmit in the bad state.

5.3 Utilities

We had to utilize multiple different libraries and frameworks to conduct the testing. However, most of the tools we used are available in any Linux distribution, and can be used out of the box is most of the operating systems based on a Linux kernel.

5.3.1 Linux Netem

NetEm is a tool in the Linux operating system which allows you to modify the traffic control facilities on the link layer protocol [22]. This includes adding packet loss, delays and duplication of packets, and a range of other modifications, such as the Gilbert-Elliott Loss model, which we utilized for our testing.

5.3.2 TCPDump

TCPDump is another utility we used during testing. It is a tool that captures network traffic and prints it out, or writes it to a file [26]. TCPDump outputs every line to represent one packet of data. For the purpose of analyzing and creating metrics, we used the built in tool to write the output to a .PCAP format [26]. We wrote it to the PCAP format so that we could analyze the results in Wireshark 5.3.3.

5.3.3 Wireshark

Wireshark is a free and open tool created by The Wireshark team. Wireshark has several use-cases, such as network troubleshooting, analysis and general software development, where networking plays an essential role [63].

Wireshark allowed us to see the networking packets at an abstracted level, which is one of the unique features that has made it the industry standard packet analyzer [63].

We used Wireshark to identify various performance aspects of the different networks we tested. Considering that we ran many tests using automated scripts, verifying the results before producing the graphs proved beneficial.

5.3.4 Wondershaper

Wondershaper is an open-source command-line utility to limit bandwidth on machines with Linux-based kernels. It was made by Bert Hubert, Jacco Guel and Simon Séhier [3]. The script was initially released by Hubert in 2022 and

has been continuously iterated over since then, with contributions from the two authors mentioned above, along with Hubert [3].

Wondershaper utilizes iproute's tc command, which is a core utility in the Linux operating system, it does, however simplify the process of limiting bandwidth, as Linux netem can be quite overwhelming, requiring a lot of work to layer limitations on interfaces like eth0 or wlan0.

Wondershaper simplifies the operations down to a few simple parameters, as shown in

Listing 5.1: Wondershaper [3]

```
wondershaper [-hcs] [-a <adapter>] [-d <rate>] [-u <rate>]
```

We utilized wondershaper to limit the bandwidth of all the participating nodes when testing the cluster. More information and metrics regarding the different networks will be covered in the first part of section 5.5, where we discussed our results.

5.4 Parsing the PCAP files

To produce comprehensible results that could be compared properly, we wrote a application using Python, with libraries that could take PCAP files as inputs, and produce suitable graphs. We covered the relevant steps to re-produce the application in the subsections to come.

5.4.1 Pyshark

Pyshark is a wrapper based on the Wireshark interface, provided by the company responsible for the all popular networking tool, Wireshark.

Pyshark utilizes TShark, which is a network protocol analyzer [62]. Tshark works in many cases in the same manner as the TCPDump framework which we covered in subsection 5.3.2, TShark allows for analysis of pcap formatted captures.

5.4.2 Implementation

We wanted to produce metrics and visualizations that were more comprehensive than what was available in the Wireshark application.

To solve this problem, we wrote a python application utilizing Pyshark 5.4.1 where we could access and filter out the information we needed through the use of their Python package. Since we had a significant amount of files to parse,

we implemented a queuing mechanism, which would handle every file as they finished running on the cluster.

We applied three different filters to the files to help uncover the possible causes of problems or issues, tcp.analysis.lost-segment, tcp.analysis.retransmission and tcp.analysis.flags. These three filters would allow us to visualize the number of packets lost while also providing enough data to form an opinion about the of the network during run-time.

Once we parsed ten sets of each run, we determined that we wanted to visualize packet loss, since it would be an appropriate metric to represent the TCP state.

We utilized the matplotlib library [61] to visualize the output from the Pyshark parsing.

The source code is available in the Github repository [65]

5.5 Results

5.5.1 Scope of testing

We limited the scope of our testing to the network configurations listed in subsection 5.5.2. These configurations were the basis of our networking tests, along with the Gilbert-Elliott Burst loss configuration.

For the Gilbert-Elliott model, which we covered in section 5.2 we used the base parameters when running these tests. This entailed that we only provided the **p** parameter for the model. The Gilbert Elliott model implemented in Linux Netem, defaults to the other parameters in the model accordingly, based on earlier representations [29].

We tested the network configurations with four different loss values, namely 0, 1, 5, and 10% which can be noted as the **x** values in the graphs in appendix 5.6.

We recorded ten different series of each execution, spanning 10 000 packets each time to get a fair range for the plots. In addition, each execution was configured to only record TCP-packets, to prevent unnecessary duplication packages that might occur during handshakes and broadcast messaging. This was also done as the services on our cluster primarily use TCP, or MQTT over TCP.

5.5.2 Network configurations

For this thesis, we wanted to utilize network configurations that would be suitable for the use case were working with. We therefore decided to run a set of tests on very limited networks with data rates far below what is typical for mod-

ern networks, along with a configuration which is derived from modern 4G/5G networks, that can be found in most countries around the world. We did this to explore the differences in terms of performance of our solution for networks that are widely different in terms of bandwidth and close to what would be the applicable situation in which our solution would be used in the real world.

Network	Data Rate	Latency
Mid-Band 5G	100 mbit/s	20ms
Tactical Network	2 mbit/s	20ms
NATO Narrowband Waveform	16 kbit/s	20ms
Combat Network Radio	9.6 kbit/s	20ms

Table 5.2: Network Configurations

In terms of latency, we chose to run all the different networks with a 20ms latency due to the nature of our use case. A HADR operation is typically confined to a certain area, or the current operating area is at least restricted in such fashion that testing anything above 20ms latency would make the theoretical physical area too large for an operation of this sort.

5.6 Results

As discussed in subsection 5.5.2, we worked with four different network configurations as the basis for our testing, running them with 0, 1, 5, and 10% loss, using the Gilbert-Elliott loss model.

In the coming subsections 5.6.1 and 5.6.2 we illustrated our results using boxplots. Boxplots is an excellent way to display descriptive data such as networking metrics. Boxplots consist of two whiskers, the one above illustrates the upper 75% of values, while the one below illustrates the lower 25% of values. The orange line passing through the boxes illustrates the median value. On the outermost edges of every whisker, there is a black horizontal line crossing it. This is used to represent the absolute maximum and minimum value of each box.

5.6.1 K3S Master Graphs

As mentioned in subsection 5.5.2, the network configurations vary significantly in terms of bandwidth, ranging from 9.4kb/s to 100mb/s. This fact becomes quite clear when seeing the graphs. Mid-Band 5G shows a stable loss rate of about 52 packets lost during the entire execution of 10 000 packets, which occurred late in the execution when a burst of packets got lost right towards the end.

The Master node of the K3s cluster is the primary entity that deals with coordin-

ating and delegating the other worker nodes. K3s primarily use TCP, with some broadcast messaging when bursts of coordination is required. When opening the PCAP files in Wireshark, we noted that most of the connection on this node was used to send larger payloads and segments, contrary to other tested nodes that primary dealt with MQTT over TCP.

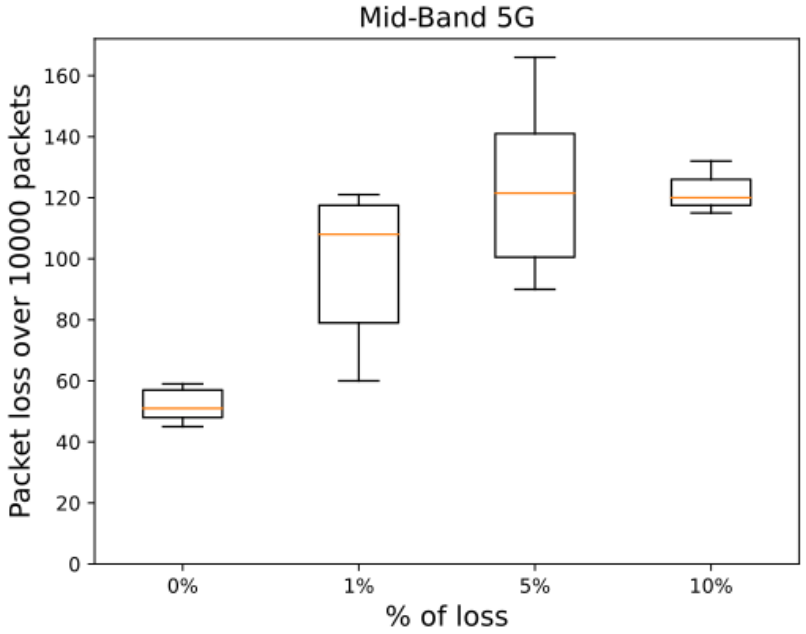

Figure 5.2: Mid-Band 5G Results

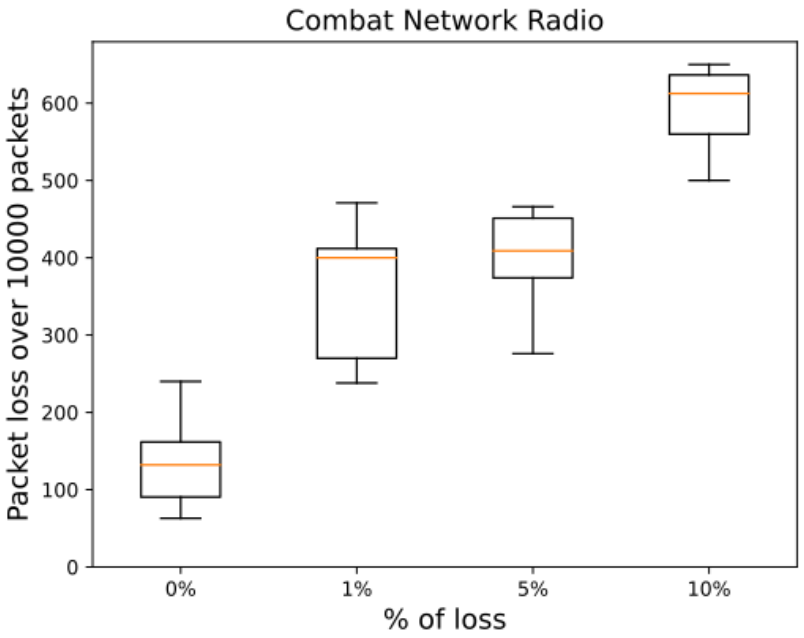

Figure 5.3: CNR Results

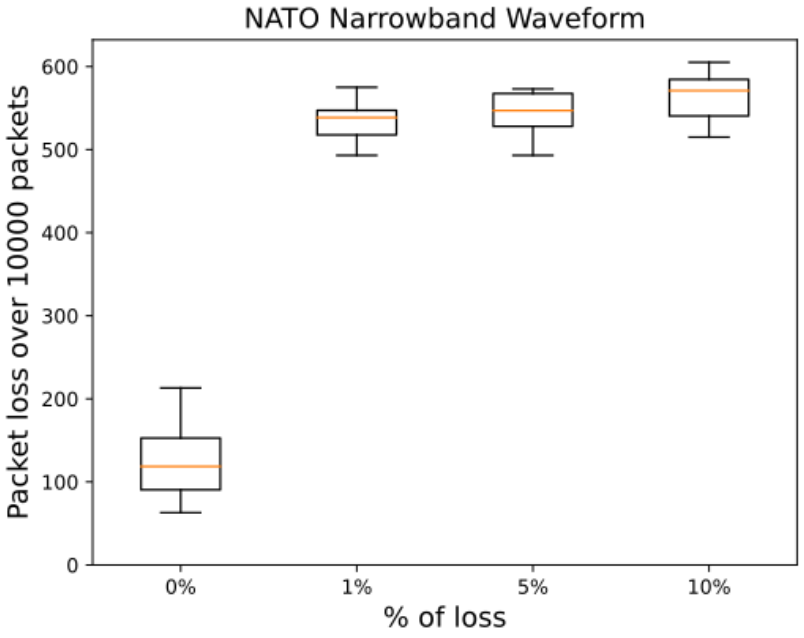

Figure 5.4: NBWF Results

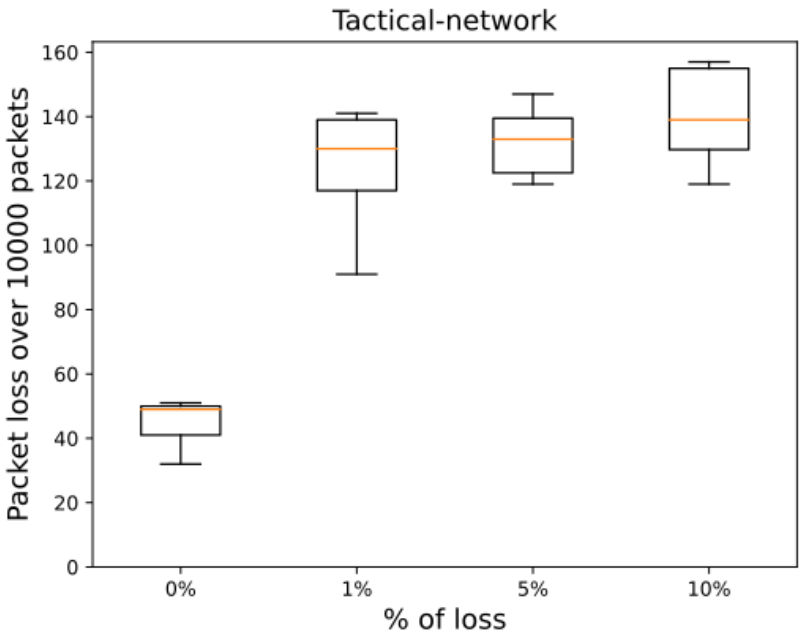

Figure 5.5: Tactical Network Results

The graphs provided in this subsections 5.3, 5.2, 5.4, 5.5 contains estimations of the packets lost during testing according to the output of our test suite. It is particularly interesting to note the slight differences when increasing the Gilbert-Elliott loss model parameter. We believe a variety of factors can contribute to the fact that loss only increased slightly.

In appendix E, we listed out a rough average of the number of packets lost during testing, which also follows the pattern mentioned in the paragraph above of not having a statistical linear progression in terms of the amount of packets lots.

We covered our interpretation and assumptions related to the outcome of the results in subsection, 5.7.

5.6.2 MQTT Broker Graphs

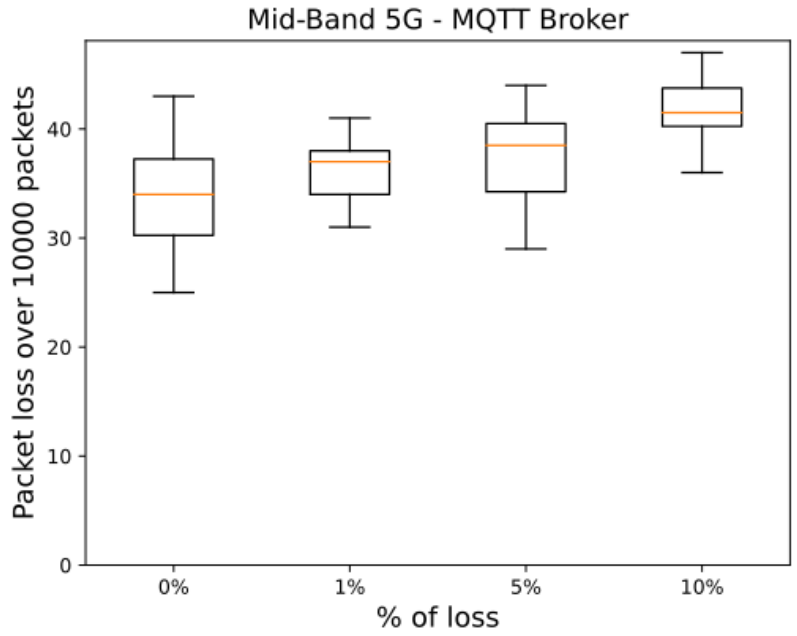

Figure 5.6: Mid-Band 5G Results

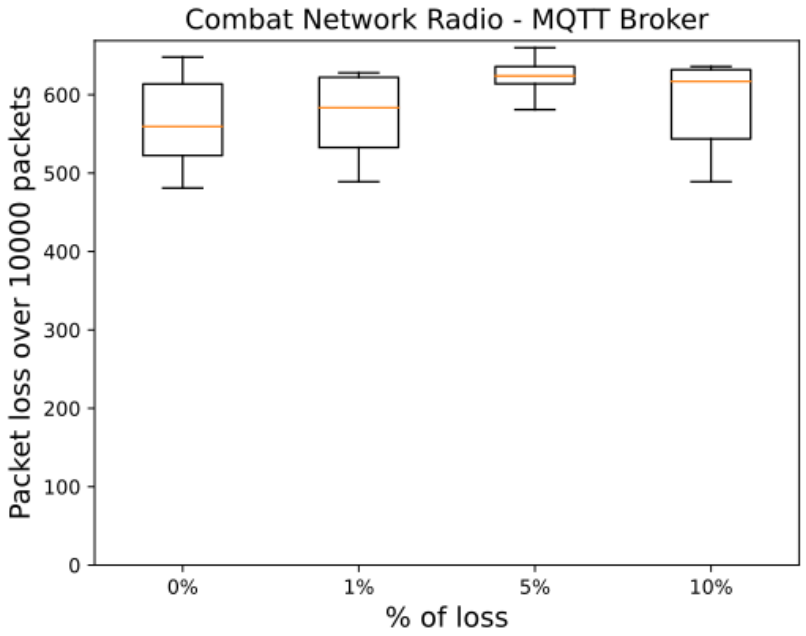

Figure 5.7: CNR Results

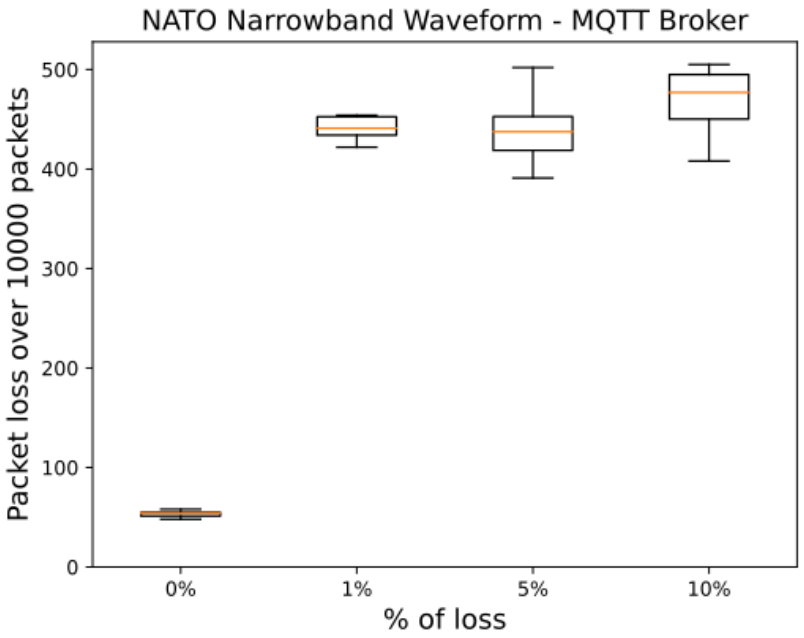

Figure 5.8: NBWF Results

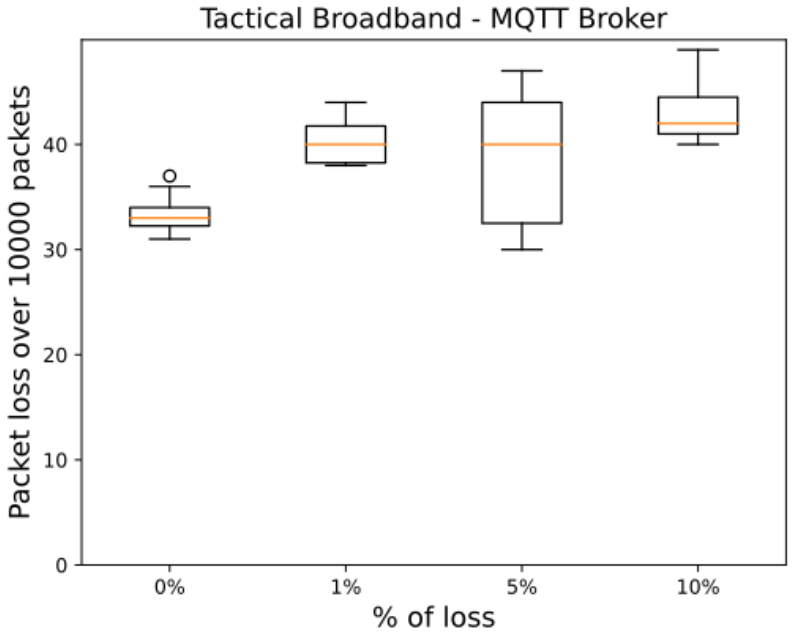

Figure 5.9: Tactical Network Results

Figures 5.7, 5.6, 5.8 and 5.9 show graphs from running the testing suite on the MQTT broker that resides in the cluster. Initially, it shows a reasonably similar pattern in terms of many packets that got lost during execution, with some slight

variations for the networks with a higher bandwidth capacity, the Mid-Band 5G and Tactical Network configuration. The MQTT broker primarily transmits MQTT packets, which are smaller segments than traditional segments over TCP.

The tables provided in appendix F shows a rough estimate of the average packet loss on the MQTT Broker during the execution. As noted from the graphs, the primary difference between these results and the aforementioned results from the master, is the lower general average on the higher capacity networks, with a significantly worse result for the CNR network.

We also covered the results, along with our interpretation of why the results differed slightly across the two nodes in subsection 5.7.

5.7 Discussion

When reviewing the results in section 5.6 it becomes apparent that the higher the bandwidth, the fewer packets are dropped, which is the obvious answer when presented with network configurations on each of its spectra in terms of bandwidth capabilities.

An interesting aspect to note is that when decreasing the bandwidth, other aspects of the TCP protocol become the issue. In the case of a packet-switching network, congestion control presents itself as the next area of focus. Congestion is a state that can occur in a network when the traffic is so overwhelming that it decreases the overall response of a given network [25]. If a network is lackluster in terms of bandwidth, congestion issues arise, and if congestion is present on the network, it leads to the loss of packets in transit [31].

Congestion control varies in terms of performance and implementation across different protocols. For example, TCP utilizes both a congestion policy and a window to avoid congestion [25], the window is commonly referred to as **cwnd**. In TCP, the congestion policy is built up of three phases being the Slow Start Phase, Congestion Avoidance Phase and the Congestion Detection Phase.

During the slow start phase, packet's sender starts with a very low rate until it increases exponentially to detect a threshold, and when it reaches it, it reduces the rate to avoid congestion [31]. During this phase, if congestion is detected, it has fallback mechanisms to start over again with slow start or move on to congestion avoidance [25].

Congestion avoidance is a more conservative policy than slow start and is based on the principle of increasing the congestion window at the end of every ACK. If all the segments are correctly acknowledged, this policy increases the congestion window by 1. This increase will continue until congestion is detected, whereas it will have fallback mechanisms similar to slow start [31].

Congestion detection is the final policy, which we believe plays a role in the

outcome of our networking testing. Congestion detection is a policy where, if congestion is detected, the window size has to be lowered. In TCP, the only way a participating sender can find out if congestion has occurred is to re-transmit its segment. To get to the retransmission stage in TCP, one of two things must have happened, either 3 similar ACKs are received for the same segment, or the time-out mechanism expires before even receiving the ACKs [31]. In all of these cases, the window size is cut in half, which is significantly larger than it can rise, given the policies we just covered.

Congestion detection can prove to be a problem in networks with weak bandwidth such as ours. During manual review of the PCAP files, we noticed, on multiple occasions that the network reached time-outs, also indicating that the packets never came through. Based on our knowledge of TCP, this is an indication that the cwnd must have been halved so many times that most of the testing we conducted on the weaker network configurations simply bottle-necked the network. If this is the case, it can provide an explanation as to why the nodes presented a fairly similar pattern in terms of loss, which differed from our initial assumption, which would be that they would have a progressive increase, as we increased the percentage of loss with the Gilbert-Elliott model.

If our assumption is correct, preventative measures can be tested, utilizing alternative congestion control algorithms such as fast recovery and fast re-transmit, which potentially could speed up the time it would take to recover lost segments on the network. Alternatively, there is also the case in which the bandwidth is simply too constrained for the cluster to operate normally.

Given the results, we can, however make an assumption in terms of how constrained the bandwidth can be, given the circumstances. Considering that we made no further modifications to the hardware or software to complement any particular use-case, we can say that both the Mid-Band 5G and Tactical Network on the nodes we tested had positive results in terms of its utilization and packets lost. We believe that further configurations to prevent congestion on the network can help reduce the packet loss even further. The NBWF and CNR network configurations showed varied results, specifically when testing with the 0% configuration. Our findings indicate that most of the traffic simply seizes when packet loss is configured higher than 0%. This enhances our opinion from earlier that due to a continuous loop of timeouts, the congestion control mechanisms in TCP is simply working on such a restricted network capacity, that the mechanism reaches a point where it simply discards packets. This is one possible explanation to the increased packets lost during some of our testing on the more restricted network configurations.

K3s - Master Node	0%	1%	5%	10%
Mid-Band 5G				
Tactical Broadband Network				
Nato Narrowband Waveform				
Combat Network Radio				

Figure 5.10: K3s Master network results

MQTT Broker	0%	1%	5%	10%
Mid-Band 5G				
Tactical Broadband Network				
Nato Narrowband Waveform				
Combat Network Radio				

Figure 5.11: MQTT Broker network results

In general, figures 5.10 and 5.11 represent our overall opinions in terms of the results we collected. Green represents a result that met our expectations in terms of number of packets lost, while red is used to indicate an inadequate result that did not meet our expectation. As previously discussed on in this section, many of the tests we ran showed similarities in terms of congestion, but we feel that the overall impression of the networking tests is well presented using the figures.

Our theory and assumptions is based on a sample of the PCAP files we produced during testing. We manually analyzed significant pieces of the PCAP files using Wireshark, and our assumption regarding congestion was based on observations of larger bulks of marked retransmission packets, where we utilized Wiresharks TCP Stream tool, which indicated that the congestion had occurred.

We sampled that fact from several PCAP files, specifically those produced when running the weaker bandwidth configurations like CNR and NBWF.

Chapter 6

Conclusion & Future Work

6.1 Conclusion

Our thesis was oriented around exploring the capabilities and opportunities of edge computing in a less-than-perfect environment to test its capabilities as an ad-hoc solution for HADR operations. This thesis aimed to shed light at recent innovations in modern computing, such as the one we utilized in this thesis, called K3s. Additionally, we wanted to answer the questions proposed towards the end of section 1.1.

Our first question was as follows,

In what capacity is it possible to utilize affordable single-board computers with Kubernetes?

Proving the usability of a modern edge-based approach for this scenario breaks down into multiple parts. Initially, we researched and discovered a well-established open-source distributed solution that would suffice when grouping together a set of affordable single-board computers, with our choice ultimately being K3s, the lightweight solution to Kubernetes.

Secondly, based on the research conducted by the NATO research groups, we used MQTT as a messaging paradigm, due to its favorable factors in the terms of size, complexity and flexibility. MQTT's conversion to other common data types, along with its varied levels of QoS offered favorable aspects that suited our use case well.

On-wards, our focus turned to the services we wanted to host on the cluster. Considering the choice of hardware on typical IoT devices, this became a question of simply what was available at the current time and what could be modified to be run on ARM64 architectures. We utilized our somewhat modified and experimental deployment of the MQTT broker from Mosquitto, deployed on our fully portable set of networked devices.

With our extensive research into the matter, we conclude the research successfully considering our findings in terms of both what is available and what the future might hold for bare-metal clusters 3.4.2. When utilizing the K3s architecture in the future, the only real limitations will be in terms of what is compatible or available for devices running the ARM architecture.

With regards to our question, we consider the research we did a success and have concluded that single-board computers such as the Raspberry Pi certainly has the capabilities to operate in a cluster, and future research into the manner can further its usability as a viable alternative in modern computing. We believe that the only real limitations will be what is compatible or available for devices running the ARM architecture.

Our second question was as follows,

Will a system of this sort be sufficient to operate on a variety of network configurations, applicable to those relevant during a HADR operation?

Testing the solution with various network configurations closely representing what would be available in a real-world scenario was a key part our work. We provided the metrics and graphs in subsection 5.6 with the subsequent discussion in subsection 5.7 to shed light on our discoveries when reviewing the files we collected. Our findings indicate that a system of this sort has the capabilities to be run in a variety of network configurations and be able to operate successfully. In section 6.3, we propose suitable areas to research to further enhance our system, both in terms of capabilities and performance.

Our findings culminate to what we considered the goal of this thesis. We explored the capabilities of an distributed ad-hoc networking solution, capable of being deployed on the network edge. We consider the thesis successful due to what achieved, and also for what all the different enabling technologies we were able to explore.

The repository and thesis can be a valuable resource for any further research, as it contains fully reproducible steps for further research.

6.2 Experiences/Contributions

Our work was oriented around a vast amount of technological components within all software and hardware aspects. Many of the technologies we utilized were still in the early stages of development or marked as experimental.

The experimental software we used was in many respects, used to complement either the Raspberry Pi or the K3s distribution.

In several cases, we were met with different bugs, as the software did not complement other equipment or software we were using.

Initially, using K3s as an alternative to the regular Kubernetes distribution brought its fair share of challenges in terms of compatibility since most third party images rely on components that are removed in K3s as a precautionary measure for it to run smoothly on ARM64-based devices.

One of the most challenging aspects of development was related to the ARM64 architecture, which is present on Raspberry devices. Most of the popular containerized applications on Docker are built for AMD64 and other desktop architectures.

There is, however a large open-source community supporting the development of these images for ARM64 architecture, which was very helpful in in terms of finding and using compatible images for the MQTT broker and load-balancer. In some rare cases, we had to manually pull a version and make the necessary adjustments to build it for ARM, which often required changes to the intermediary storage solution over to what has been popularized on IoT devices, being SQLite.

With the nature of our use case, which was operability in disaster-struck areas, we wanted to account for nodes falling out of reach, or otherwise stop working. This obviously entails a fair bit of intercommunication between the devices, for them to be able to restore the services or information contained in the given node. Kubernetes, but more specifically K3s has accounted for this issue by introducing Block-Storage, which we covered in subsection 4.5.2. However, with such a complicated system of blocks scattered throughout the system, the most prominent problem became the read and write capacity on the SD-cards.

Our device were fitted with high quality SD-Cards, which proved to weak for Rancher's block storage solution. Eventually ending up decreasing the overall capacity of the nodes significantly to the point where it could not operate at the intended capacity.

This presented an issue for us, as the autonomous nature of K3s would otherwise allow for a system, as long as it was configured properly, to essentially fix itself if a node died, or a malfunction occurred.

As we discovered, however, block based storage was simply too demanding for regular SD-Cards. and forced a complete wipe and re-install several times during testing, ultimately forcing us to abandon the implementation towards the end of our work. We came to the conclusion that our system of Raspberry Pi devices simply could not handle the amount of computations required to run block-based storage solutions.

In chapter 6.3 we aimed to further elaborate on our findings, and propose our ideas on how some of the issues can be resolved or how it naturally resolves itself as projects go beyond early stages of development, as many are as of writing this thesis.

6.2.1 Hardware challenges

One of the most significant challenges we faced during development was related to hardware. Our initial scope in terms of hardware was 7 Raspberry Pi 3 Model B units, where each one was suppose to have a dedicated role, as either Master or Worker.

The K3s distribution is marketed as having a very low footprint [52], requiring as a little as 512Mb of storage.

During the building, and implementation process we discovered that this was not the case. K3s have had several iterations over the last 2 years, keeping up with the current release of the traditional Kubernetes, and somewhere along the line, it seems that the general memory requirement has significantly increased.

The platform we planned out consisted of several small containers, with a limited amount of frameworks to best incorporate even the lowest capacity devices out there. When initializing and applying these different components we reached maximum memory capacity on the master node, which was a Raspberry Pi 3. The cluster started halting, delivering time-out errors as a result of this deficit.

Luckily, The Raspberry Foundation released the Raspberry Pi 4 back in 2019, which was a significant change compared to the previous iterations, most especially in terms of memory – as they now deliver 2, 4 and 8 Gb configurations. To maintain and work within the scope of the thesis, we acquired one of these devices in the 4 Gb configuation to use as our master node, which significantly changed the development experience, and also provided us with the necessary memory to complete our goal.

6.3 Future Work

When working on the thesis, we identified several different interesting areas, which we wanted to complete. Due to the time-frame of the thesis, we were not able to accomplish them. This section aims to give an overview of the most significant areas that can improve the already established cluster implementation.

6.3.1 Longhorn & Persistent Distributed Storage

Throughout the thesis, we have made several references to Longhorn, which we introduced in subsection 4.5.2. In this subsection, we also covered some of the problems we faced with implementing Longhorn as a service on our cluster. In short, this was most likely related to a restricted read/write capacity on the SD-Cards we used in the Raspberry Pis.

Considering the vital role that persistent distributed storage could potentially have for our use-case, we have come up with a theory to be explored further.

As most of the applications and services we ran were fairly lightweight both in terms of storage and performance, having one or multiple dedicated storage nodes can present itself as a viable solution. With reference to subsection 3.1.2, we talked about recent innovations in the field of 5G and how it can be applicable for our scenario. With this development in 5G technology, it is not a stretch to assume that the standalone 5G core could also act as an intermediate storage node to enable the use of technologies such as Longhorn.

This idea requires some research, as Longhorn has to perform computations on the API-layer of the master node, in order to operate sufficiently. Due to the limited time-frame of this thesis, we were not able to test this on our implementation. Further research into the matter is required to implement distributed block storage on the cluster.

6.3.2 WebRTC Broadcasting node

Voice and video was identified as a very important part of what we aimed to include in our current system, but due to a lack of time, we were not able to complement this. We prepared a Raspberry Pi with a local on-board camera, broadcasting its surroundings using WebRTC. Our initial idea was that we would host this as a service in the cluster. We realized quite early that this would be a challenge, as WebRTC requires a significant amount of ports, that would have to be designated and delegated in Kubernetes. Using WebRTC, these ports would also change frequently, posing a problem when hosting the application in a constrained architecture, where it is delegated a port manually or through the use of a load-balancer 3.4.2.

An alternative approach, however, would be to host WebRTC directly on a participating node, allowing it to operate in a non-constrained environment. This, along with making it communicate its URL back to the cluster using MQTT as a means of messaging, would allow for the idea of voice and video to be realized, as the implementation could be built in the same manner. but also communicate back to the MQTT broker and the other clients of on the cluster.

6.3.3 Alternative congestion control

In section 5.7 we extensively covered our assumptions as to explaining why some our testing concluded with a significant amount of packets lost. This was more noticeable on the more constrained network configurations such as CNR and NBWF 5.5.2. In section 5.7 we elaborate about our findings when manually reviewing the PCAP files, which indicated to us that there was severe congestion on the network, likely due to the severely limited bandwidth.

Testing with alternative congestion mechanisms can prove helpful in reducing the amounts of packets lost, overall, improving the performance of our solution.

6.3.4 Real-world simulated tests

To prove its usability in an HADR situation, a real-world simulated test would be highly beneficial. A real-world simulated test would, for instance, be composed of two teams performing the same task, one utilizing the current approach, while the other team utilizes a data-driven approach through the use of a version of our system. Performing a simulated test such as this would be beneficial to figure out its capabilities in a less than perfect environment, not only network wise, but also in an environment with obstacles and challenges. Making it as user-friendly and autonomous as possible will be a key factor in determining its success as a real-world application.

6.3.5 Messaging Application

The MQTT messaging paradigm is an essential component of our proposed system. In subsection 2.3.2, we elaborate about the different levels of Quality of Service in MQTT. For the purpose of our initial prototype, we utilized QoS level 0, as we did not reach the point of developing an application that could operate on top of the MQTT broker.

Our idea was to create a messaging application, which could utilize MQTT to broadcast messages between participating nodes. With this in mind, we wanted to implement a mechanism, allowing for some vital messages to be sent using either QoS level 1 or 2, ensuring that it reaches the designated senders. These messages would allow for a separation of context in terms of how messages are perceived in the system, allowing for important messages to be of higher priority than others.

An application like this could be highly beneficial, as it would allow for a strong data-driven approach contrary to what exists today. Many other aspects of the application must also be considered and carefully tested to ensure its usability in the field.

Bibliography

[1] Awatif Alqahtani et al. 'End-to-End Service Level Agreement Specification for IoT Applications'. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 926–935. DOI: 10.1109/HPCS.2018.00147.

[2] AVI-Networks. *What is Load Balancing?* URL: shorturl . at / oryN5. (accessed:26.02.2022).

[3] Simon Séhier Bert Hubert Jacco Geul. *Wondershaper*. URL: https://github.com/magnific0/wondershaper. (accessed: 5.05.2022).

[4] Pranav Bhardwaj. *Advantage and Disadvantage of Edge Computing*. URL: https://www.tutorialspoint.com/advantage-and-disadvantage-of-edge-computing. (accessed: 06.02.2022).

[5] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259.

[6] Max Brenner. *Kubernets Overview Diagrams*. URL: https://shipit.dev/posts/kubernetes-overview-diagrams.html. (accessed: 17.2.2022).

[7] Steve Buchanan. *What is Kubernetes, and why is its popularity exploding in the cloud?* URL: https://cloudblogs.microsoft.com/industry-blog/en-gb/technetuk/2020/09/17/why-is-kubernetes-exploding-in-the-cloud/. (accessed: 01.02.2022).

[8] Cloudflare. *What is Edge Computing?* URL: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/. (accessed: 05.02.2022).

[9] D. E. Comer et al. 'Computing as a Discipline'. In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[10] Peter J. Denning. 'Computer Science'. In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 405–419. ISBN: 0470864125.

[11] Abhinav Dubey. *All about k3s - Lightweight Kubernetes*. URL: https://dev.to/abhinavd26/all-about-k3s-lightweight-kubernetes-3ell. (accessed: 17.02.2022).

[12] Sarah Wilson Eric Sullivan. URL: https://www.techtarget.com/searchstorage/definition/block-storage. (accessed: 29.03.2022).

[13] Håvard Fossen. *5G kan erstatte DTT, Tetra og GSM-R*. URL: https://www.insidetelecom.no/artikler/5g-kan-erstatte-dtt-tetra-og-gsm-r/167868. (accessed: 10.05.2022).

[14] Cloud Native Computing Foundation. *Pods and Nodes*. URL: https://kubernetesbootcamp.github.io/kubernetes-bootcamp/3-1.html. (accessed: 26.02.2022).

[15] Kubernetes Foundation. *kube-apiserver*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/. (accessed: 17.02.2022).

[16] Kubernetes Foundation. *Kubernetes Controller Manager*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/. (accessed: 17.02.2022).

[17] Kubernetes Foundation. *Kubernetes Scheduler*. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. (accessed: 17.02.2022).

[18] Raspberry Pi Foundation. *About us*. URL: https://www.raspberrypi.org/about/. (accessed: 12.02.2022).

[19] RedHat Foundation. *What is a Kubernetes deployment?* URL: https://www.redhat.com/en/topics/containers/what-is-kubernetes-deployment. (accessed: 26.02.2022).

[20] The Eclipse Foundation. *Eclipse Mosquitto Image*. URL: https://hub.docker.com/_/eclipse-mosquitto. (accessed: 01.03.2022).

[21] The Eclipse Foundation. *Mosquitto.conf main page*. URL: https://mosquitto.org/man/mosquitto-conf-5.html. (accessed: 23.03.2022).

[22] The Linux Foundation. *netem*. URL: https://wiki.linuxfoundation.org/networking/netem#packet_loss. (accessed: 21.03.2022).

[23] Verdens Gang. *Flere hus har rast i Gjerdrum – frykter flere vil rase*. URL: https://www.vg.no/nyheter/innenriks/i/8655KQ/flere-hus-har-rast-i-gjerdrum-frykter-flere-vil-rase. (accessed: 9.05.2022).

[24] Gartner. *Edge Computing*. URL: https://www.gartner.com/en/information-technology/glossary/edge-computing. (accessed: 16.05.2022).

[25] GeeksforGeeks. *Congestion Control in Computer Networks*. URL: https://www.geeksforgeeks.org/congestion-control-in-computer-networks/. (accessed: 10.05.2022).

[26] The Tcpdump Group. *TCPDUMP and LIBPCAP*. URL: https://www.tcpdump.org/. (accessed: 10.04.2022).

[27] Anthony Wiles Hans van der Veer. *Achieving Technical Interoperability - the ETSI Approach*. URL: shorturl.at/nuzHX. (accessed: 08.01.2022).

[28] Gerhard Haßlinger and Oliver Hohlfeld. 'The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet.' In: Jan. 2008, pp. 269–286.

[29] Gerhard Hasslinger and Oliver Hohlfeld. 'The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet'. In: *14th GI/ITG Conference - Measurement, Modelling and Evalutation of Computer and Communication Systems*. 2008, pp. 1–15.

[30] Nato IST. *Federated Interoperability of Military C2 and IoT Systems*. URL: https://www.sto.nato.int/Lists/STONewsArchive/displaynewsitem.aspx?ID=564. (accessed: 08.02.2022).

[31] Mr. Vishal Jaiswa. *TRANSMISSION CONTROL PROTOCOL*. URL: https://miet.ac.in/assets/uploads/cs/TCP.pdf. (accessed: 10.05.2022).

[32] Agenda Kaupang. *Etterevaluering av TETRA Nødnettprosjektet*. URL: shorturl.at/kFZ12. (accessed: 10.05.2022).

[33] Maker.io. *Meet the New Raspberry Pi 3 Model B*. URL: https://www.digikey.no/en/maker/blogs/2018/meet-the-new-raspberry-pi-3-model-b-plus. (accessed: 12.02.2022).

[34] MatLab. *Simulate channel models for wireless systems*. URL: https://www.mathworks.com/discovery/channel-model.html. (accessed: 09.03.2022).

[35] MetalLB. *MetalLB*. URL: shorturl.at/bcuC9. (accessed: 26.02.2022).

[36] TechWorld with Nana. *Kubernetes Volumes explained | Persistent Volume, Persistent Volume Claim and Storage Class*. Youtube. URL: https://www.youtube.com/watch?v=0swOh5C3OVM&ab_channel=TechWorldwithNana. (accessed: 26.02.2022).

[37] Juniper Networks. *What is VXLAN?* URL: shorturl.at/fxKMN. (accessed: 17.02.2022).

[38] the NGINX Team. *What Is Load Balancing?* URL: https://www.nginx.com/resources/glossary/load-balancing/. (accessed:26.02.2022).

[39] NRK Nyheter. *Leirskredet i Gjerdrum*. URL: https://www.nrk.no/nyheter/leirskredet-i-gjerdrum-1.15307406. (accessed: 08.05.2022).

[40] Oasis. *MQTT Version 5.0*. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_Toc3901017. (accessed: 01.02.2022).

[41] Oracle. *What is IoT?* URL: https://www.oracle.com/internet-of-things/what-is-iot/. (accessed; 15.02.2022).

[42] World Meteorogical Organization. *Weather-related disasters increase over past 50 years, causing more damage but fewer deaths*. URL: https://public.wmo.int/en/media/press-release/weather-related-disasters-increase-over-past-50-years-causing-more-damage-fewer. (accessed: 08.01.2022).

[43] Paolo Patierno. *MQTT & IoT protocols comparsion*. URL: https://www.slideshare.net/paolopat/mqtt-iot-protocols-comparison. (accessed: 16.05.2022).

[44] Manas Pradhan. 'Federation Based on MQTT for Urban Humanitarian Assistance and Disaster Recovery Operations'. In: *IEEE Communications Magazine* 59.2 (2021), pp. 43–49. DOI: 10.1109/MCOM.001.2000937.

[45] Manas Pradhan, Marco Manso and James R. Michaelis. 'Concepts and Directions for Future IoT and C2 Interoperability'. In: *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 2021, pp. 231–236. DOI: 10.1109/MILCOM52596.2021.9653093.

[46] Manas Pradhan, Filippo Poltronieri and Mauro Tortonesi. 'Generic Architecture for Edge Computing Based on SPF for Military HADR Operations'. In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. 2019, pp. 225–230. DOI: 10.1109/WF-IoT.2019.8767181.

[47] Rancher. *Longhorn*. URL: https://longhorn.io/. (accessed: 29.03.2022).

[48] Rancher. *Rancher Docs: K3s*. URL: https://rancher.com/docs/k3s/latest/en/. (accessed: 9.3.2022).

[49] Rancher. *Volumes and Storage*. URL: https://rancher.com/docs/k3s/latest/en/storage/. (accessed: 29.03.2022).

[50] Raspian. *Welcome to Raspbian*. URL: https://www.raspbian.org/FrontPage. (accessed: 9.3.2022).

[51] Weisong Shi and Schahram Dustdar. 'The Promise of Edge Computing'. In: *Computer* 49.5 (2016), pp. 78–81. DOI: 10.1109/MC.2016.145.

[52] K3s Team. *K3s*. URL: https://k3s.io/. (accessed: 15.02.2022).

[53] Kubernetes Team. *Namespaces*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. (accessed: 17.03.2022).

[54] NetApp Team. *What are containers?* URL: shorturl.at/cmH23. (accessed: 26.02.2022).

[55] The HiveMQ Team. *Introducing the MQTT Protocol - MQTT Essentials: Part 1*. URL: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/. (accessed: 15.02.2022).

[56] The HiveMQ Team. *Publish and Subscribe - MQTT Essentials*. URL: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/. (accessed: 02.02.2022).

[57] The HiveMQ Team. *Quality of Service 0,1 and 2 - MQTT Essentials: Part 6*. URL: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/. (accessed: 04.02.2022).

[58] The Kubernetes Team. *ConfigMaps*. URL: https://kubernetes.io/docs/concepts/configuration/configmap/. (accessed: 24.03.2022).

[59] The Kubernetes Team. *Secrets*. URL: https://kubernetes.io/docs/concepts/configuration/secret/. (accessed: 23.02.2022).

[60] The Kubernetes Team. *What is Kubernetes?* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (accessed: 26.02.2022).

[61] The MatPlotLib development team. *Matplotlib*. URL: https://matplotlib.org/. (accessed: 10.05.2022).

[62] The Wireshark Team. *TShark*. URL: https://www.wireshark.org/docs/man-pages/tshark.html. (accessed: 23.04.2022).

[63] The Wireshark Team. *Wireshark*. URL: https://www.wireshark.org/. (accessed: 10.04.2022).

[64] VMWare Team. *What are Kubernetes Services?* URL: https://www.vmware.com/topics/glossary/content/kubernetes-services.html. (accessed: 27.02.2022).

[65] Nicolai Vatne. *Master Thesis Repository*. URL: https://github.com/NickVatne/master-thesis-nicolai. (accessed: 10.05.2022).

[66] Abhishek Verma et al. 'Large-scale cluster management at Google with Borg'. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[67] 5G Vinni. *5G Verticals Innovation Infrastructure*. URL: https://www.5g-vinni.eu/concept-approach/. (accessed: 19.05.2022).

[68] Rupert Wood. *Wireless Traffic Forecasts : 5G Will Make Little Difference to Long-Term trends*. URL: https://aglmediagroup.com/wireless-traffic-forecasts-5g-will-make-little-difference-to-long-term-trends/. (accessed: 08.01.2022).

[69] Konrad Wrona et al. 'Leveraging and Fusing Civil and Military Sensors to support Disaster Relief Operations in Smart Environments'. In: *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*. 2019, pp. 790–797. DOI: 10.1109/MILCOM47813.2019.9021004.

[70] Ivana Podnar Žarko et al. 'Collaboration Mechanisms for IoT Platform Federations Fostering Organizational Interoperability'. In: *2018 Global Internet of Things Summit (GIoTS)*. 2018, pp. 1–6. DOI: 10.1109/GIOTS.2018.8534547.

Appendix A

Mosquitto Deployment Manifest - TLS

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mosquitto-deployment-open
  namespace: mosquitto
spec:
  selector:
    matchLabels:
      app: mosquitto
  replicas: 1
  template:
    metadata:
      labels:
        app: mosquitto
    spec:
      containers:
      - name: mosquitto
        image: eclipse-mosquitto:1.6.15
        imagePullPolicy: Always
        ports:
        - containerPort: 1883
        volumeMounts:
          - name: mosquitto-config
        mountPath: /mosquitto/config/mosquitto.conf
            subPath: mosquitto.conf
          - name: tls-ca
            mountPath: /mosquitto/tls-ca
            readOnly: true
          - name: tls-server
            mountPath: /mosquitto/tls-server
```

```
        readOnly: true
volumes:
  - name: mosquitto-config
    configMap:
      name: mosquitto-config
  - name: tls-ca
    secret:
      secretName: mqtt-ca
  - name: tls-server
    secret:
      secretName: mqtt-tls
  - name: mosquitto-data
    persistentVolumeClaim:
      claimName: mosquitto
      emptyDir:
```

Appendix B

Mosquitto Service Manifest - TLS

```
apiVersion: v1
kind: Service
metadata:
  name: mosquitto-service-open
  namespace: mosquitto
spec:
  type: LoadBalancer
  ports:
    - name: mqtt-1883
      port: 1883
      targetPort: 1883
      protocol: TCP
    - name: mqtt-8883
      port: 8883
      targetPort: 8883
      protocol: TCP
  selector:
    app: mosquitto
```

Appendix C

Mosquitto PVC Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mosquitto
  namespace: mosquitto
  labels:
    app: mosquitto
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: local-path
```

Appendix D

MQTT Client Code

```
import time
import random
import datetime
import json
import uuid
from paho.mqtt import client as mqtt_client
import ssl

broker = "192.168.0.230"
unique_id = uuid.uuid4()
port = 8883
topic = "home/bequerel/"
deviceID = "Bequeruel-Controller-RPI" + unique_id.__str__()

def connect():
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to HADR Network")
        else:
            print("Failed
                to connect to HADR Network, Returned", rc)
    client = mqtt_client.Client(deviceID)
    client.tls_set(
            "ca.crt",
            tls_version=ssl.PROTOCOL_TLSv1_2)
    client.tls_insecure_set(True)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client

def publish(client):
    msg_count = 0
    fault = random.randint(1, 2)
    while True:
        data = {}
        data["deviceID"] = deviceID
        data["timeStamp"] = '{:%Y-%
```

```
            m-%d %H:%M:%S}'.format(datetime.datetime.now())

        if fault == "1":
            data["bequerel
                "] = round(random.uniform(30.0, 10000.0), 1)
        else:
            data["bequerel
                "] = round(random.uniform(30.0, 10000.0), 1)

        payload = json.dumps(data, ensure_ascii=False)
        print(payload)
        result = client.publish(topic, payload)
        status = result[0]
        if status == 0:
            print("Message sent")
        else:
            print("Failed to send message to topic")
        msg_count += 1
        time.sleep(30)

def run():
    client = connect()
    client.loop_start()
    publish(client)


if __name__ == '__main__':
    run()
```

Appendix E

Table of results - K3s Master Node

Mid-Band 5G	Packets lost
0% loss	51
1% loss	107
5% loss	121
10% loss	134

Table E.1: Master - Mid-Band 5G

CNR	Packets lost
0% loss	111
1% loss	396
5% loss	411
10% loss	604

Table E.2: Master - CNR

NBWF	Packets lost
0% loss	107
1% loss	546
5% loss	572
10% loss	598

Table E.3: Master - NBWF

Tactical Network	Packets lost
0% loss	59
1% loss	132
5% loss	139
10% loss	142

Table E.4: Master - Tactical Network

Appendix F

Table of results - MQTT Broker

Mid-Band 5G	Packets lost
0% loss	34
1% loss	38
5% loss	40
10% loss	43

Table F.1: MQTT broker - Mid-Band 5G

CNR	Packets lost
0% loss	123
1% loss	551
5% loss	589
10% loss	602

Table F.2: MQTT broker - CNR

NBWF	Packets lost
0% loss	107
1% loss	443
5% loss	452
10% loss	489

Table F.3: MQTT broker - NBWF

Tactical Network	Packets lost
0% loss	36
1% loss	41
5% loss	41
10% loss	44

Table F.4: MQTT broker - Tactical Network