Creating an Analysis Tool for Testing and Evaluating the MQTT Protocol

Emil Paulin Andersen

December 2021

Abstract

As more and more data is being produced by sensory devices, there has become a greater need for transporting data from these devices to a system that has better capabilities of processing it and extracting information that could be considered valuable. Difficulties arise when the networks that are responsible for transmitting the data can be considered unreliable. This is a big challenge when operations rely heavily on communications over these networks, both in military and civilian sectors. This report investigates protocols that handle the responsibility of transporting messages from one system to another, using the publish/subscribe pattern. To emulate a realistic scenario we have used several network models that have varying degrees of reliability. We have tested and analysed the performance of the MQTT protocol looking into data from both the network layer and the application layer. The most important metrics have been data rate, packet loss and transmit delay. There have previously been conducted studies that have investigated the MQTT protocol using network models with resource constraints. This study attempts to expand upon that research using other network models and test configurations. To run tests and gather data we have developed our own analysis tool. The tool consists of a test framework that takes custom test configurations and produces data for analysis. Test data is stored in a database, which is used by a frontend application that can to display information related to the tests. The application is also able to run a test using a configuration specified by the user. Findings show that the MQTT protocol performs well in tactical broadband and satellite networks, but struggles in narrower networks categorized by low bandwidth and high latency.

Contents

Chapter 1

Introduction

Today, there exist many devices that produce large quantities of data that needs to be to processed and analyzed in order to provide valuable information to some user. The amount of data and the complexity of the processes related to extracting value from the data means that the computational power needed for this is increasing, and the devices themselves are not capable nor designed to handle this workload. The Internet of Things (IoT) has increasingly become an important part of our daily lives, to help us with important and not so important tasks. Many of these devices have only the role of producing data, not storing or processing it. This work can be done by external systems that are specialized for this type of operation, leaving the smaller devices with only the simplest responsibility of producing data. There is of course a need then for a way to transfer the data from one system to another, and that is what we have investigated during this project.

One of the biggest problems when having to implement a solution for transporting data, is that the network responsible for transporting the data will not always be stable or reliable. This can especially be the case in military scenarios where communications and information sharing is vital to operations. *"Battlefield communications can be disrupted by a number of factors including environmental constraints, wireless dynamics and mobility, and both intentional and non-intentional interference. Robust, reliable, and efficient mechanisms are needed in such environments to ensure that critical data is delivered in the face of disruptions, intermittent connectivity, and low-bandwidth (DIL)."* is a quote from Scott et. al (2011) [1], which mentions challenges the military face when operations rely heavily on the networks for data communications. The quote introduces the DIL abbreviation which denotes the type of networks we have looked at in this project. DIL is not only an problem faced by the military, but can also apply to problems faced in civilian areas, e.g. search and resource operations. For this project we have looked into the process of data transportation over several network models that could be considered DIL.

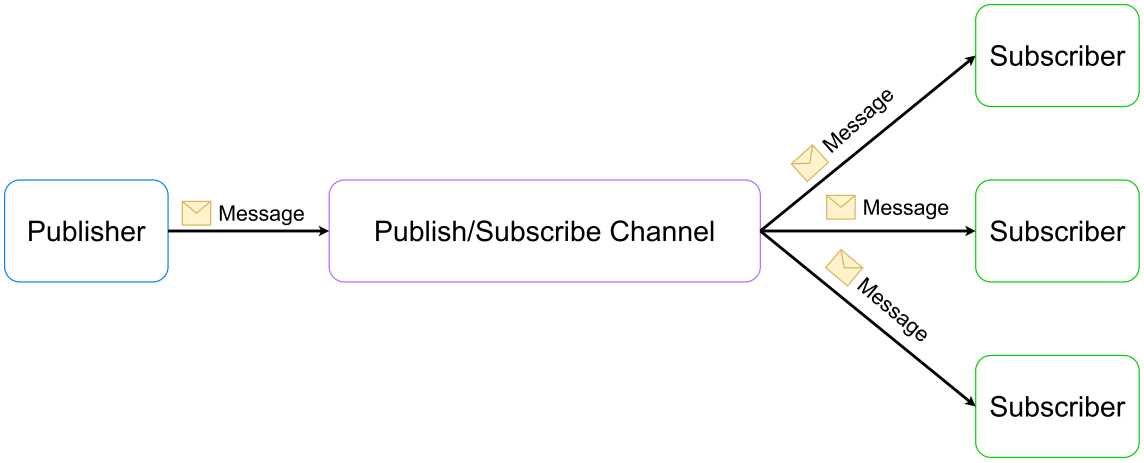

Figure 1.1: Publish/Subscribe pattern

One of the goals for this project was to investigate and measure the performance of two protocols that use the publish/subscribe pattern [2]. The protocols we have looked at are MQTT and MQTT for sensor networks (MQTT-SN) [1]. The publish/subscribe pattern consists of publishers, subscribers and a message broker, as can be seen in figure 1.1. Before the messaging process can happen the publisher and subscriber clients need to establish a connection. With MQTT the connection is with a message broker while MQTT-SN connects with a gateway. To connect, the clients need to forward some information to the broker. This varies from implementation to implementation, but most solutions forward information about the clients name, their role (subscriber or publisher) and additional information specific to the protocol. The publisher has the responsibility of packaging the message that will be sent. The subscriber is the one receiving the message sent by the publisher. This does not happen directly as the publisher and subscriber are unaware of each other. It is the responsibility of the message broker to handle the incoming messages from the publisher and send them to the correct subscriber. Filtering is used to determine where messages should be sent. In the publish/subscribe pattern, there are two ways to implement filtering, either through a topic-based system or a content-based system. Both MQTT and MQTT-SN use a topic based system. A publisher defines a topic that should be subscribed to by the subscriber. With this pattern the publisher and subscriber are only aware of the existence of the message broker at any time. This is different from the observer pattern where we have a subject that alerts other components, named observers, about an event happening. In the observer pattern the subject must have a list of dependencies, which is unlike the publish/subscribe pattern where dependencies are managed by the message broker.

The performance metrics we have investigated are both from the network layer and the applic-

[1]While the goal originally was to test and compare both protocols, due to time constraints, we only had time to run tests for the MQTT protocol. The setup needed to test the MQTT-SN protocol has been established during the time of this project and is ready for testing at a later time. We have still provided details about the MQTT-SN protocol in this report, as well as how the test setup is configured.

ation layer. On the network layer we have looked at the data-rate, the average size of messages and the number of retransmissions. On the application layer we looked at the number of messages sent, the amount of packet loss,transmission delay for messages and number of times the clients have disconnected. The MQTT protocol have been tested using different network models in order to simulate different degrees of challenging network environments. Testing the performance of publish/subscribe protocols have been researched previously [3], but not using the network models found in this study.

1.1 Scope

In order to test these metrics we have developed an analysis program that will be able to create and run different test configurations using the MQTT and MQTT-SN protocols. The program is also developed in a way that adding and testing additional protocols in the future should require few changes to the solution. The program allows for a dynamic test setup which enables many configurations to be created specifically for each protocol. Each test can be configured and started from a graphical user interface, which monitors the test as it is running. The analysis tool will also create logs during the tests, which are stored in a database. This allows us to easily retrieve data from previous tests. The first part of this project was focused on the development of the analysis tool, where the seconds part was focused on testing and experimentation using the MQTT protocol.

1.2 Outline

The remainder of the report is organized as follows: Chapter 2 provides background on previous studies related to our project, and details the MQTT and MQTT-SN protocols. Chapter 3 describes the analysis tool that we have developed for testing the protocols. Chapter 4 gives an overview of the test setup we have used, including configurations that were used for testing. Chapter 5 describes the results from testing with the MQTT protocol and provides analysis on these results. Lastly, chapter 6 provides a conclusion to our work with some recommendations based on the results from testing. The chapter also describes a few aspects of the project that can be worked on in the future.

Chapter 2

Background

2.1 Previous studies

Previous studies investigating the performance of publish/subscribe protocols have been conducted and are relevant to examine for this project. In one of these studies by Johnsen et al. (2019) [3] they investigated the performance of publish/subscribe protocols in environments where network resources are constrained. The study used network models similar to the ones tested in our research. The protocols compared in the study are the WS-Notification (WS-N) standard [4], MQTT and MQTT-SN. The study simulated a scenario based on the Anglova scenario [5], as well as a modified version of the scenario with a more challenging network topology. The most important metric to evaluate was transmission delay. The tests used one subscriber node and 23 publisher nodes, where messages were published with a 10 seconds interval. The duration of the tests were 20 minutes. The tests with the MQTT and MQTT-SN protocols used quality of service levels 0 and 1 (see section 2.2.3). Results from testing showed that MQTT and MQTT-SN had smaller message sizes, less delay and less data volume, compared to WS-N. MQTT and MQTT-SN had more packet loss than WS-N at quality of service level 0. With quality of service level 1, MQTT did not benefit much, but MQTT-SN showed an improvement with reduced packet loss. Results from all tests show that MQTT had the lowest transmission delay, thus making it more suitable than WS-N in networks with similar constraints. During testing, TCP produced many spurious retransmissions that indicate issues during transmissions over the networks used. MQTT produced many more retransmission than WS-N. This was not the case in earlier experiments that were conducted before this study. Our study tries to expand on this study by testing and evaluating MQTT and MQTT-SN with different network models, as well as testing with quality of service level 2.

Another study that is relevant for this project is a research done in 2019 by Lee et al. [6] that investigates the relation between packet sizes and performance of the MQTT protocol. The metric investigated in the study is how the packet byte size affect delay and packet loss. The study looks at all levels of quality of service. The results show that in networks that are

wired, the larger packet sizes (more than 4000 bytes) have a large negative effect on delay using quality of service level 0 and 1. Quality of service level 2 already has quite a big delay so this does not affect the delay as much. Packet loss was found to increase significantly when the packet sizes reached 8000 bytes. This happened during testing with all quality of service levels, but there was slightly less using quality of service level 2. The study also used wireless network, where the results showed that quality of service level 2 had slightly more stability in regards to packet loss over the other levels. The results from this study is helpful in determining the size of packets that can be sent during our testing, as well as help explain changes that happen when sending messages of different sizes.

2.2 Protocols

2.2.1 MQTT

MQTT is a simple and light-weight publish/subscribe protocol that allows for several levels of Quality of Service (QoS). The protocol is designed for working in environments with low network bandwidth, making it efficient and allowing a high degree of scalability. The efficiency of the protocol is due to it being very lightweight and requiring less resources compared to other protocols, making it suitable for use on small devices like microcontrollers. The protocol uses publisher clients which are responsible for sending messages from the source system, and subscriber clients that receive the messages on the end system. In order to connect the clients with each other, MQTT uses a broker which is responsible for connecting to the clients and forwarding messages from the publisher to the subscriber. Determining which messages should go where, is resolved using topics and QoS set by both the publisher and subscriber. MQTT is suitable for environments with unreliable networks where loss of connection can happen repeatedly. The support for persistent sessions minimizes the time a client spends on reconnecting. It also allows for a wide variety of data types to be sent, including binary, JavaScript object notation (JSON) objects [7], as well as images. The protocol is therefore considered data-agnostic, with the only limitation being the maximum size of a payload in a message (268435455 bytes or 256 MB) [8]. The protocol requires a network protocol that is reliable, meaning it must be ordered, have bi-directional connections, as well as lossless communication. Therefore MQTT usually relies on the TCP protocol for data transmission, but it could also use transport protocols like stream control transmission protocol (SCTP) [9] in theory. This means that there is an additional layer of ensurance for message delivery underneath what is provided by the QoS 1 and 2 settings.

2.2.2 MQTT-SN

The MQTT-SN protocol shares many of the same characteristics as MQTT, like topics and QoS levels, but unlike MQTT, MQTT-SN use unreliable protocols for transportation, removing the need for a permanent connection. Therefore MQTT-SN solutions often use UDP for transport-

ing messages. This means that MQTT-SN does not offer the same level of message delivery ensurance as MQTT, since this is not provided by UDP. Applications that use MQTT-SN are usually ones that have many low-powered sensors that are required to only transport small amounts of data at a time. One of the benefits with MQTT-SN is the reduced payload sizes, which is achieved by its own implementation of topics, which is described in section 2.2.4. Additionally the MQTT-SN protocol differentiates itself by introducing two types of gateways, the transparent gateway and the aggregate gateway. Both gateways are responsible for converting MQTT-SN messages into MQTT messages that are forwarded to a MQTT broker. The transparent gateway receives one or many MQTT-SN streams from various sources and converts them into several MQTT streams, while the aggregate gateway only produces a single MQTT stream. Clients also have the ability to discover gateways. This happens when a broker advertises using multicasting while the clients are in a process of listening on this multicast address.

A big challenge when developing a solution with MQTT-SN, is that it has not been adopted on the same scale as MQTT. Documentation is lacking and prebuilt clients and gateways do not always work optimally. This requires more work on the developer to implement their own solution, or modify existing ones so that they work as intended.

2.2.3 Quality of Service

Both MQTT and MQTT-SN provide different levels of quality of service. There is 0 (fire and forget), 1 (at least once) and 2 (only once). Publisher clients sets the QoS level when publishing messages, and it is sent with each publish message. The subscriber clients specifies the QoS they wish to have by sending a subscription message to the broker, containing the QoS level. It is important to note that if a message is published with a higher QoS than what the subscriber has subscribed to, the message will be sent from the broker to the subscriber using the QoS defined in the subscription, not what the publisher has set when sending the message. This means that if the subscriber has set QoS to 0, there is no guarantee that the message arrives in the end even if it was published using QoS 1 or 2. MQTT-SN also has an additional level, 3 or -1, which provide additional services not found in MQTT [1]. On the publisher side, they decide the QoS when publishing a message. This study will be limited to using QoS 0, 1 and 2 for both protocols.

[1]The settings is know as -1, but the flag used in messages is set to the decimal number 3. This setting reduces some of the requirements related to client connections and must use one of two specific topic types [10].

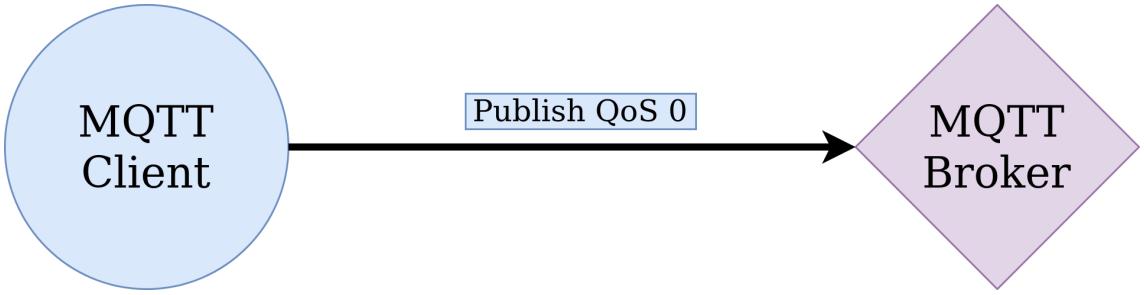

Figure 2.1: Packet sent using QoS level 0

Table 2.1 shows a message being sent using QoS 0. The publisher client is shown sending a single packet containing the message to the broker. The packet is not guaranteed to be delivered, and therefore should be used on low priority packets.

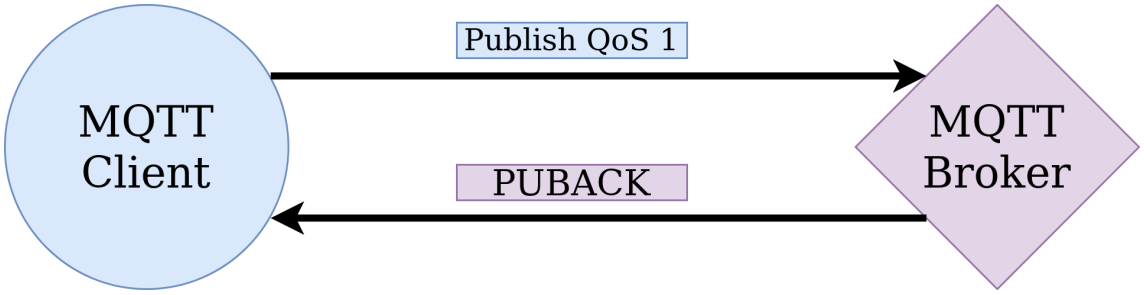

Figure 2.2: Packets sent using QoS level 1

Table 2.2 shows a message being sent using QoS 1. The publisher sends a packet with the message to the broker, which returns an acknowledgement packet confirming that the message was received. It is important to note that this level of service does not ensure that duplicate messages do not arrive in the other end. This setting is used for messages that are of some importance and must arrive, but where duplicates do not matter.

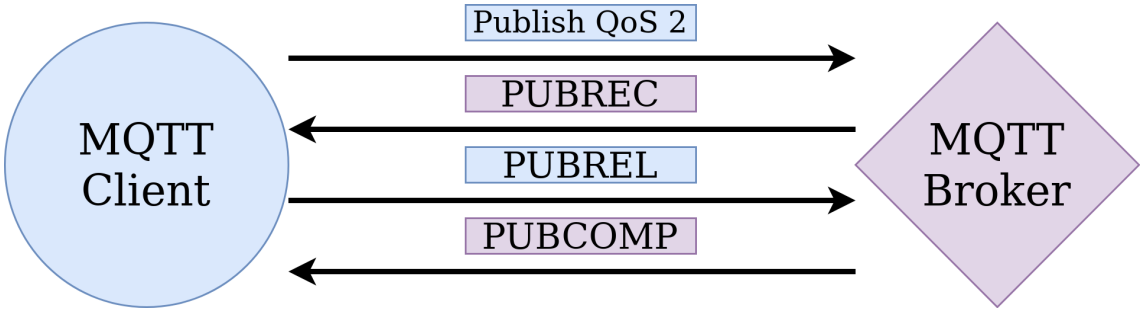

Figure 2.3: Packets sent using QoS level 2

Table 2.3 shows a message being sent using QoS 2. The publisher sends a message to the broker and receives a packet acknowledging the message. In the case that the publisher does not receive this message from the broker, it sends a new packet containing the same message. In addition to the message it also sets the duplicate flag in the packet, in order to let the broker

know that this is not a new message. When the publisher retrieves an acknowledgement, it stores the packet and discards the original message. At this time the publisher sends a new packet which in return the broker responds with a comply packet, ending the transmission. This solution does cause significantly more overhead than the other QoS settings, but does ensure that no duplicate messages arrive.

2.2.4 Topics

Topics are used by both the MQTT and the MQTT-SN protocol, and are a way for clients to share information. The format of the topics are UTF-8 strings with words that give meaning to a category and/or data source, delimited by a forward slash "/". A common example of a topic is, "*MyHouse/Basement/Temperature*", which implies that the topic is for temperature data for the basement in your house. The publishing client is responsible for registering the topic which it will send relevant data about to the broker. The subscriber is then responsible for subscribing to the corresponding topic. A subscribing client is not only limited to a single topic and can subscribe to multiple topics by the use of wildcards [11].

There are some differences in how MQTT-SN handles topics related to MQTT. In MQTT-SN the topic name is replaced by a short, two byte topic id. The purpose of this change is to reduce the strain on bandwidth as data is not published with the full topic name as is the case in MQTT. Instead a topic name is registered through a procedure where clients register a topic name with the gateway. If the registration is accepted, the gateway will assign a topic id and return it to the relevant clients. The publishing client will then use this topic id when publishing messages. There exists an implementation where we have pre-defined topics, meaning that there is no need for a registration procedure. In this case the topic id has already been mapped to a topic name that is known by both the client's and gateway. This is indicated by a flag set in the message. The subscriber still needs to subscribe to a pre-defined topic id. Lastly, there is an option to use short topic names. These names have to be a fixed length of two octets. Like with a pre-defined topic id, there is no need to register the topic in this case. Both pre-defined topics and short topics still require the client to subscribe using the correct topic id.

2.2.5 MQTT Packet format

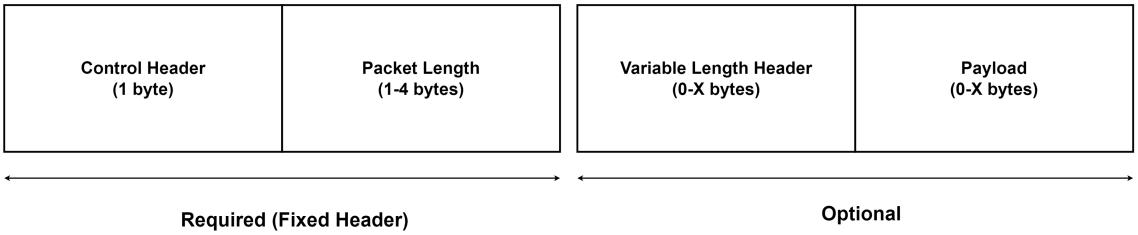

Figure 2.4: MQTT Message packet format

Figure 2.4 shows the packet format of MQTT messages. The packet header consists of a single byte control field and a remaining length field that varies from 1 to 4 bytes. These two field are the fixed header and must be included in every message. This means that the smallest size of any MQTT packet will be two bytes long, as these fields are required. In addition to the fixed header, there is also a variable header and the payload, which can be of varying lengths and only present in some packets.

Control header

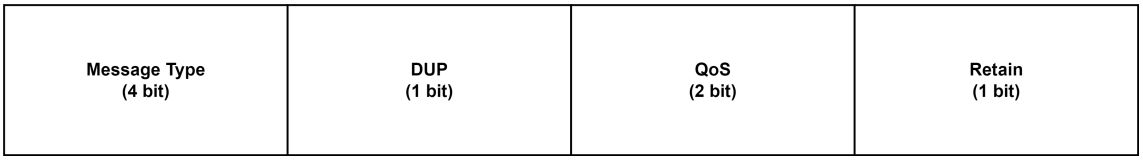

Figure 2.5: Control header

The control header contains two fields, consisting of 4 bits each. The first field represents the command type of the packet. For example a CONNECT message will contain the value of 1, telling the receiving broker that this is a connect message. The remaining 4 bits of the control header are reserved for control flags. For a publish command to the broker, the 0th bit tells the broker if the message will be retained, 1st and 2nd bit tells the broker what QoS level the message is and the 3rd bit tells the broker whether the message is a duplicate or not.

Table 2.1 shows an overview of the message types used in the MQTT protocol. There are a total of 16 message types. The value column shows the value that the message type represents in the first four bits of the control header. Direction of flow shows who can send and receive the packet. CONNACK messages can only be sent from the server, meaning the message broker. Message like PUBACK are only used with QoS 1, while PUBREC, PUBREL and PUBCOMP are used with QoS 2, which can be seen in figure 2.2 and 2.3.

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

Table 2.1: Control packet types from the MQTT version 3.1.1 specification [12]

Remaining length

This field is reserved for information about the length of the variable header as well as the size of the payload in the message. The 7th bit in each byte is reserved for a continuation flag. The continuation flag is needed to know if the remaining length field continues into another byte and is part of the remaining length field.

Variable header

The variable header contains the packet identifier. The information found in the identifier are the protocol name, the protocol level and the connection flags. Before the protocol name there is a two byte long field that represents the length of the protocol name. These connection flags can be seen in table 2.2. Will is used in the event that a client disconnects to let other clients know that the original client is currently offline. This is only used in the case that a clients

disconnect abruptly. How the will is handled can be set by the flags. The packet identifier is not required for every packet. Example of packets that require a packet identifier are PUBLISH, PUBACK, SUBSCRIBE and UNSUBSCRIBE.

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	User Name Flag	Password Flag	Will Retain	Will QoS		Will Flag	Clean Session	Reserved

Table 2.2: Connect flags in variable header

Payload

The payload contains the data that is sent with the message. This field is optional as some message have no need for data to be sent. A DISCONNECT message is an example of a packet that does not have a payload.

Packet example

Table 2.3 shows an example of how a MQTT packet looks like. The message is of type CONNECT, which can be seen from the value of the fixed header. None of the control flags are set in this example. The remaining length is 17 bytes, which includes the variable header and payload. Next comes the variable header, which first provides the length of the rest of the protocol name. Then comes the protocol name and version, before the connection flags. In this case the connection flag is set for clean session. The keep alive for this client is set to 60 seconds. Lastly there is the payload, where the first two bytes provide the length and the rest is the Client id. In this case the client id is "hello".

Byte		Hex	Value
1	Fixed Header	0x1	1
2	Remaining Length	0x11	17
3	Variable Header Length	0x0	
4		0x4	4
5	Protocol and Version	0x4d	M
6		0x51	Q
7		0x54	T
8		0x54	T
9		0x4	4
10	Connection Flags	0x2	2 (Clean Session)
11	Keep Alive	0x0	
12		0x3c	60
13	Payload Length	0x0	
14		0x5	5
15	Payload Data	0x68	H
16		0x65	E
17		0x6c	L
18		0x6c	L
19		0x6f	O

Table 2.3: Example of a packet containing a CONNECT message

Message sequence chart

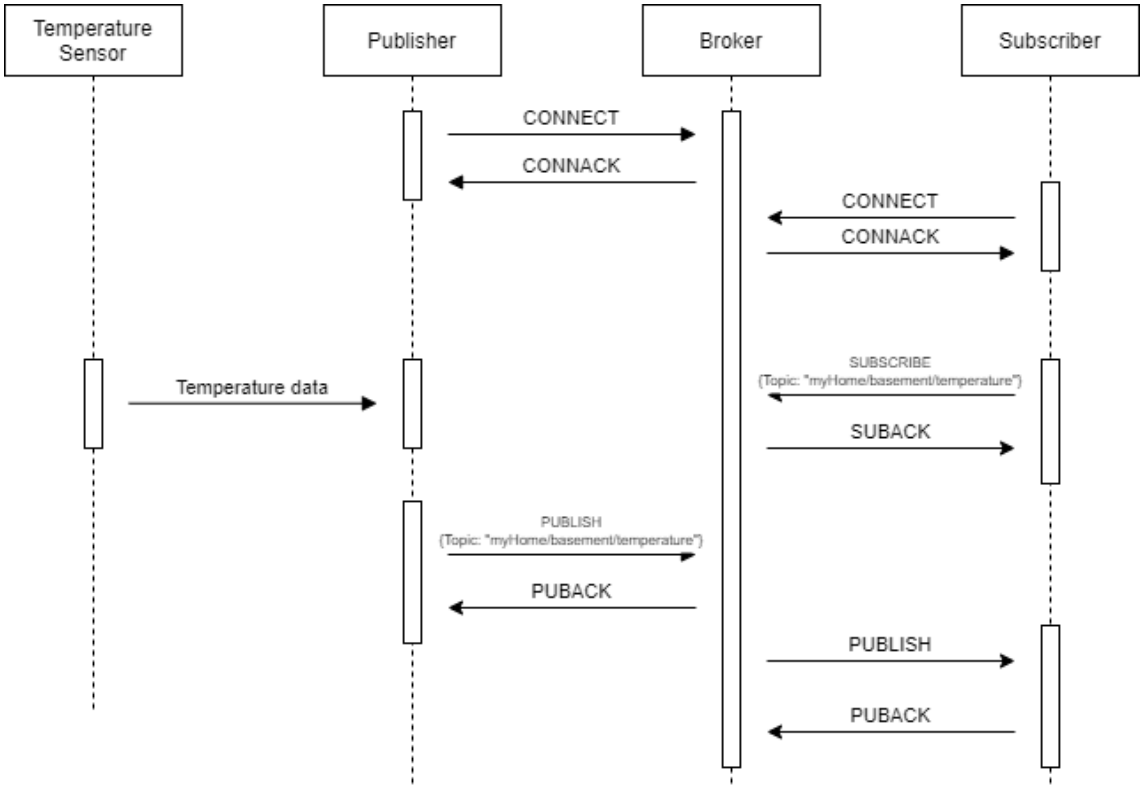

Figure 2.6: Message sequence diagram

Figure 2.6 shows a message sequence chart of a typical message transaction using MQTT with QoS level 1. Here, both clients connect to the broker and receive a reply that the connection is successful. Once the subscribing client has connected, the publisher can begin publishing messages. Since we are using QoS 1, the broker responds with a PUBACK message to acknowledge the packet has been received. When the PUBACK message has been sent, the broker can send the message to the subscriber. Since the subscriber has subscribed with QoS 1, they respond with a PUBACK message that lets the broker know that they have received the message.

2.2.6 MQTT-SN Packet Format

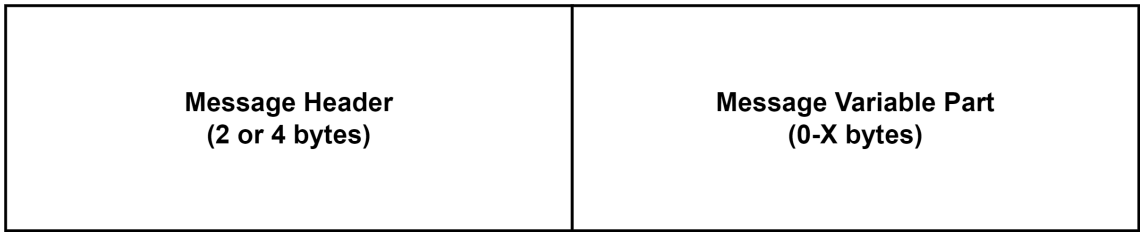

Figure 2.7: MQTT-SN Message packet format

The MQTT-SN packet format is slightly different from the MQTT format, as can be seen in figure 2.7. It consists of a message header, as well as a message variable part. The message header is either two or four bytes long and is required in every MQTT-SN message. The message variable part is not required for all messages.

Message Header

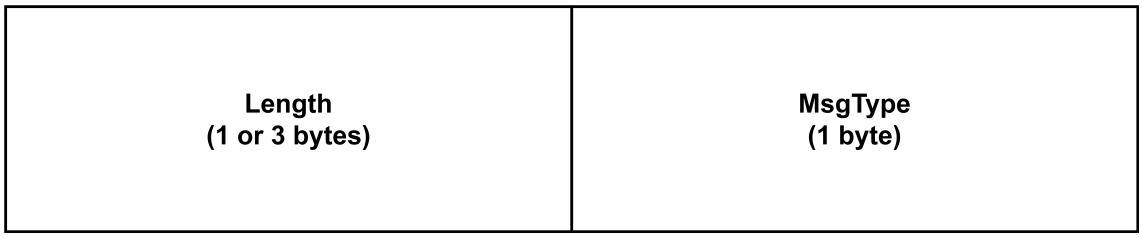

Figure 2.8: Message Header

Figure 2.8 show the format of the message header. The length field indicates the length of the of the message. If the first byte has the value *0x01*, then the length field is three bytes long, where the remaining two bytes specify the total size of the message. This means that the message has a maximum length of 65 535 bytes. If the message has a size that is smaller than 256 bytes, the first byte is used to denote the length. MQTT-SN does not support message fragmentation.

The message type field is one byte long and is reserved for the type of the messages. Table 2.4 show the message types in MQTT-SN and their value.

MsgType Field Value	MsgType	MsgType Field Value	MsgType
0x00	ADVERTISE	0x01	SEARCHGW
0x02	GWINFO	0x03	reserved
0x04	CONNECT	0x05	CONNACK
0x06	WILLTOPICREQ	0x07	WILLTOPIC
0x08	WILLMSGREQ	0x09	WILLMSG
0x0A	REGISTER	0x0B	REGACK
0x0C	PUBLISH	0x0D	PUBACK
0x0E	PUBCOMP	0x0F	PUBREC
0x10	PUBREL	0x11	reserved
0x12	SUBSCRIBE	0x13	SUBACK
0x14	UNSUBSCRIBE	0x15	UNSUBACK
0x16	PINGREQ	0x17	PINGRESP
0x18	DISCONNECT	0x19	reserved
0x1A	WILLTOPICUPD	0x1B	WILLTOPICRESP
0x1C	WILLMSGUPD	0x1D	WILLMSGRESP
0x1E-0xFD	reserved	0xFE	Encapsulated message
0xFF	reserved		

Table 2.4: MQTT-SN message types from the MQTT-SN specification [13]

Message Variable Part

The message variable part contains information that depends on the message type. Relevant information in the variable part can be client id, data, flags, duration, topic id, topic name, etc.

Packet example

Table 2.5 shows an example of a CONNECT message sent with MQTT-SN. The message is similar to the CONNECT message sent with MQTT, but the total size of the message is reduced.

Byte		Hex	Value
1	Length	0xB	11
2	MsgType	0x04	4
3	Flags	0x2	2 (Clean Session)
4	ProtocolId	0x1	1
5	Duration	0x0	
6		0x3c	60
7	ClientId	0x48	H
8		0x45	E
9		0x4c	L
10		0x4c	L
11		0x4f	O

Table 2.5: Example of a packet containing a CONNECT message

Chapter 3

Developing an analysis tool

When deciding to create the analysis tool we had to consider whether or not it was appropriate to build it ourselves or use an existing solution. Looking at existing solutions revealed that there were not many that fit the needs for this project. Many solutions that provide testing environments for the MQTT protocol can be found on GitHub, but the issue with using these solutions was that they were very specific to what the developer wanted to investigate themselves (see pymqttbench [14]). We also discovered that MQTT-SN had very little adoption, meaning that finding pre-existing tools that could measure the performance metrics that we wished to test, was difficult. Due to this, we decided that it would be better to develop our own tool. This allowed us to have more control over the framework as well as make sure that all test requirements were fulfilled.

In order to analyze the performance of the protocols in challenging networks, we have developed an analysis program running in Python (version 3.9.6). This program allows us to create and store logs which can be analyzed in order to evaluate protocol performance on the application layer. The program allows for dynamic configuration of both MQTT and MQTT-SN clients. The program configures Netem setting (see 3.3.2) for the purposes of testing different network models (see table 4.1). For some of the statistics we have used the Pandas Python library (version 1.1.5) [15], which is useful for creating comma separated values (CSV) files. The program will produce large amounts of test data, and therefore we have set up a database in MySQL that will store data related to each of the tests. To configure and run tests, we have also developed a front end application using Plotly Dash [16]. This allows us to view the test as it is running, as well as look at previously run tests.

3.1 Technology

For the analysis tool we chose to develop it in Python. Python is a programming language that is commonly used for data analysis, which is the main process that this project will focus on. The language provides a range of libraries that are well suited for the type of analysis we

wanted to perform when investigating the performance of both protocols. A large amount of data will be collected during testing, and libraries like Pandas, allow us to implement solutions quickly at every stage of the data analysis process.

Due to the large amounts of data collected from testing, we needed to have a database solution in order to organize and retrieve relevant data. The most common database models are the relational- and document models. We chose to go with the relational model and used MySQL as our database service. The relational model is more suited for the structure of the test data in this project and would allow us to more easily query and process data for analysis.

The frontend application was also developed using Python. In Python, Tkinter [17] is considered the de-facto framework for building GUI applications, but it is becoming increasingly outdated. We have instead decided to use Plotly Dash to build our frontend. Plotly Dash is a relatively new framework that is intended to be used for creating and deploying data analytics applications in the browser. The incentive behind creating a frontend application for our analysis tool was to have an intuitive and easy way to configure tests, monitor them and do post analysis on the data. Plotly already has an extensive number of pre-made graphing and visualization components that can be implemented in Dash and make our work easier. Another incentive for using a Python framework over a JavaScript framework like React and Vue, was to connect it to our test framework in order to configure and run test directly from the frontend.

3.2 Architecture

3.2.1 Test framework

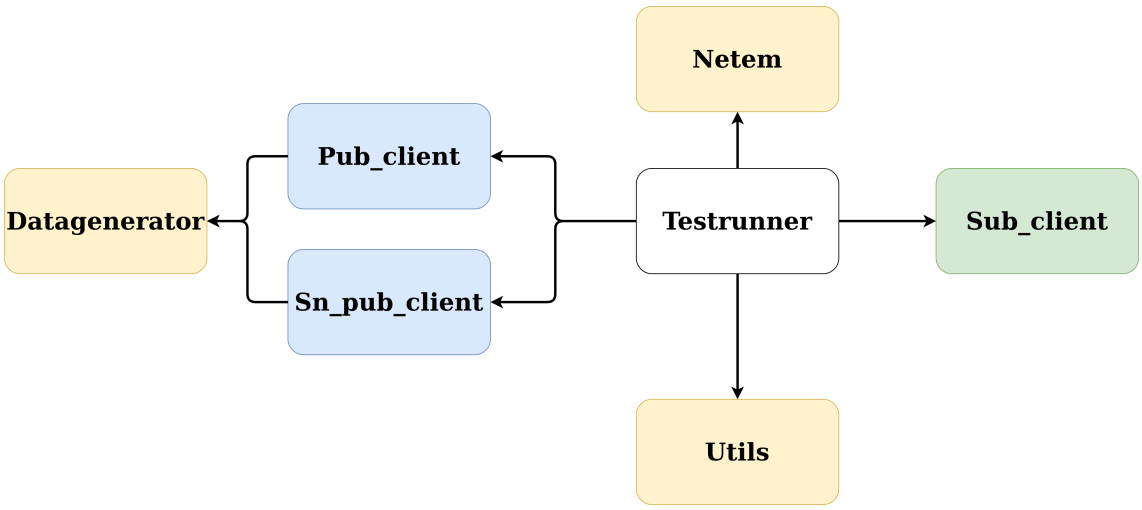

Figure 3.1: Architecture of the test framework

Figure 3.1 shows an overview of the components in the test framework. The testrunner component is responsible for running the tests, and uses the test parameters that have been configured for the specific test. It uses the Netem tool [18] to set limitations on the network based on the network model chosen. All configuration information is stored in the database. It is also responsible for configuring and initializing the client nodes that will be used for the test. To simulate the running of many nodes in parallel, the program runs each client in a separate thread, allowing the sending of multiple messages from different nodes simultaneously. The pub- and sn_pub_clients use the datagenerator component to generate data that will be published. Different types of data is available to be used as data in messages. This includes positional GPS data (lat, lon, alt) and image data (JPEG, PNG). All messages are packaged with additional metadata which can be used to determine information about messages being sent, e.g. timestamps, message id, source- and target systems. Both the pub_client and sub_client components log data about the messages that is stored in the database. In addition we also have a utils component that contains additional methods that are useful for the testrunner component.

3.2.2 Frontend application

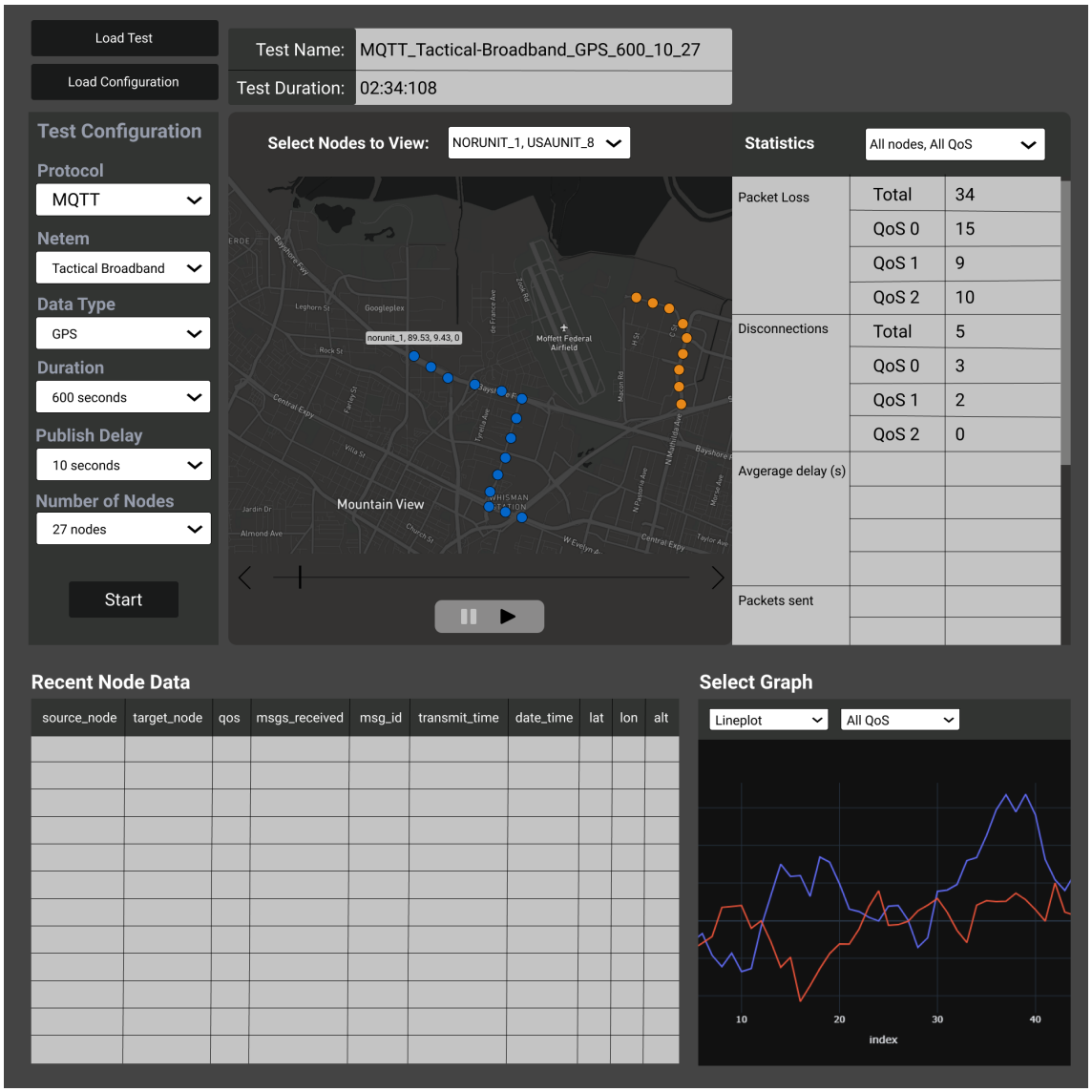

Figure 3.2: Mockup of GUI for the analytics tool

Figure 3.2 shows a mockup of the dash application. The mockup shows the features that we wanted to implemented in the program at the start of development. Not all features have been implemented during this project, but the core functionality that allows us to run tests and look at analysis are complete.

From the application the user can view tests, data and statistics that are relevant for analysis. The application allows the user to configure and run their own tests. The parameters that can be set are protocol, network model, data type, duration, publish delay and the amount of nodes. If the data used during the test is GPS data, it will be presented in a leaflet component [19] (version 0.1.23) that show the nodes on a map. The user can filter various nodes and look

at data related to each specific node by hovering over them. The table named "Recent Node Data" contains information about messages that have arrived on a subscribing node. Statistics from the test can be viewed in the Statistics table. The table show relevant statistics about messages sent, packet loss that has occurred, transmit times and how many messages were duplicates. There is also a graph component, where the user is able to view graph data about average transmit times for different QoS levels. If the user has selected a second test, the graph will display a bar chart comparing the two tests. This is useful when comparing the latency of different network models and data types.

3.2.3 Data storage

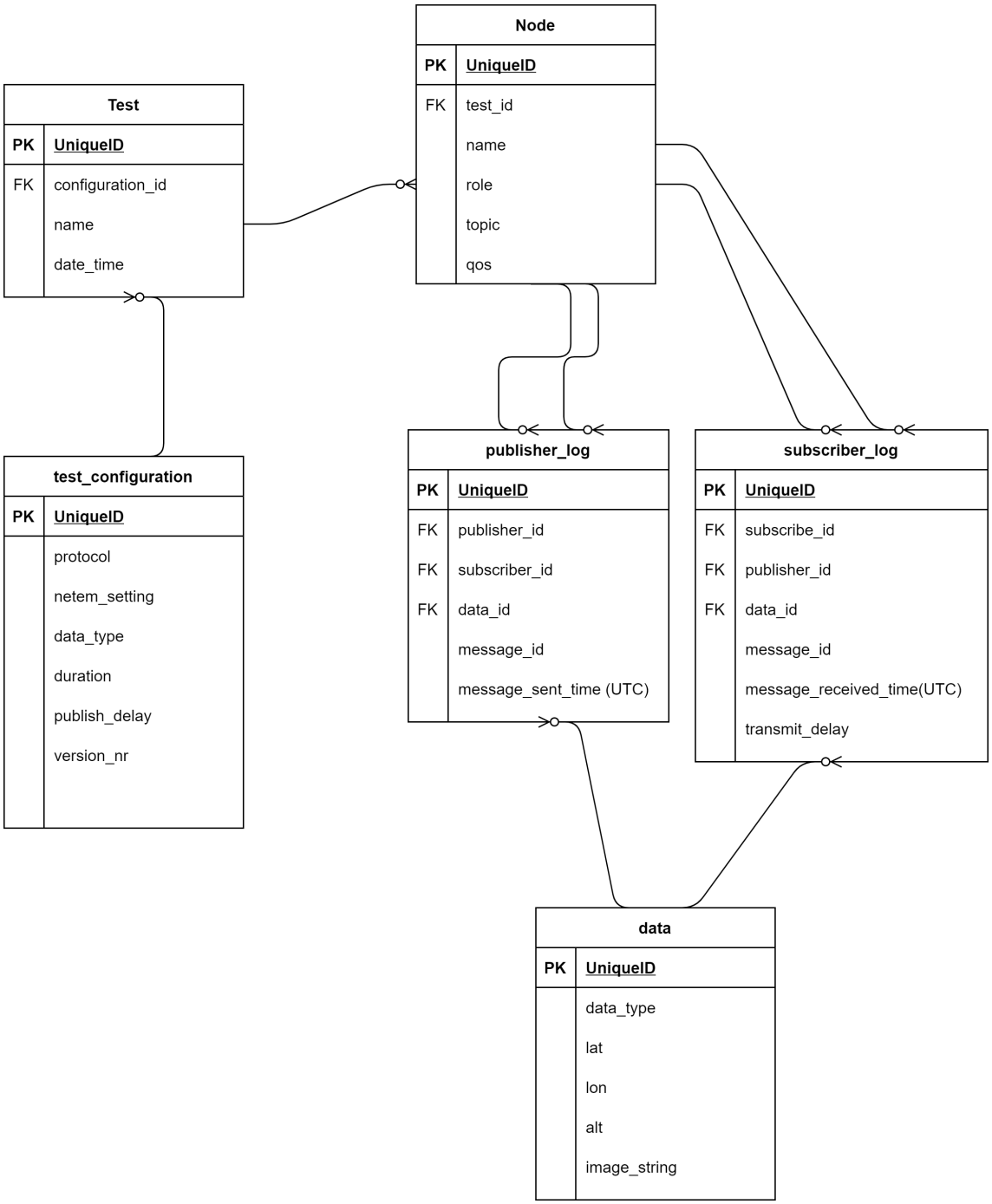

Figure 3.3: Structure of database in MySQL

Figure 3.3 gives an overview of our database solution in MySQL. The figure shows the tables, their attributes as well as the relation between the tables. Since we are using a relational model, we can perform join commands on the tables to fetch the data that is relevant for visualization in the frontend application. Each test that we run will have a test configuration

that contain the information about the test parameters. Each test will have nodes that represent the clients, both subscriber and publisher with relevant information about their role, topic and QoS. Publisher clients will have a publisher log that contains information about who they send their data to (subscribing client), the id of the message (to see the order of messages and check for duplicates) and the time the message was sent. It also contains a reference to the data that is sent which is stored in a separate table. The data table contains information about the data, depending on whether the data is of type GPS or image. The subscriber clients also have a log similar to the publishing clients. This table is slightly different from the publisher log, in that it contains information about the time the message was received and an attribute that specifies the transmit delay of the message that was received.

Database connection

In order to connect the test framework and frontend to the database, we used MySQL Connector [20] (version 8.0.26). For this we have implemented a database module, which consists of three components, DB_Connector, DB_Fetch and DB_Upload. The DB_Connector initializes and maintains the connection to the database, while DB_Fetch and DB_Upload uses the connection in order to fetch and upload data to the database. Fetching happens exclusively on the frontend application and uploading happens exclusively through the test framework. The fetch and upload components provide several methods that queries the database directly. Insert queries are simple and only creates new records in the tables, while the fetch queries are more complex as they need to filter and sort data that can be usefull in the analysis.

3.2.4 Complete Architecture

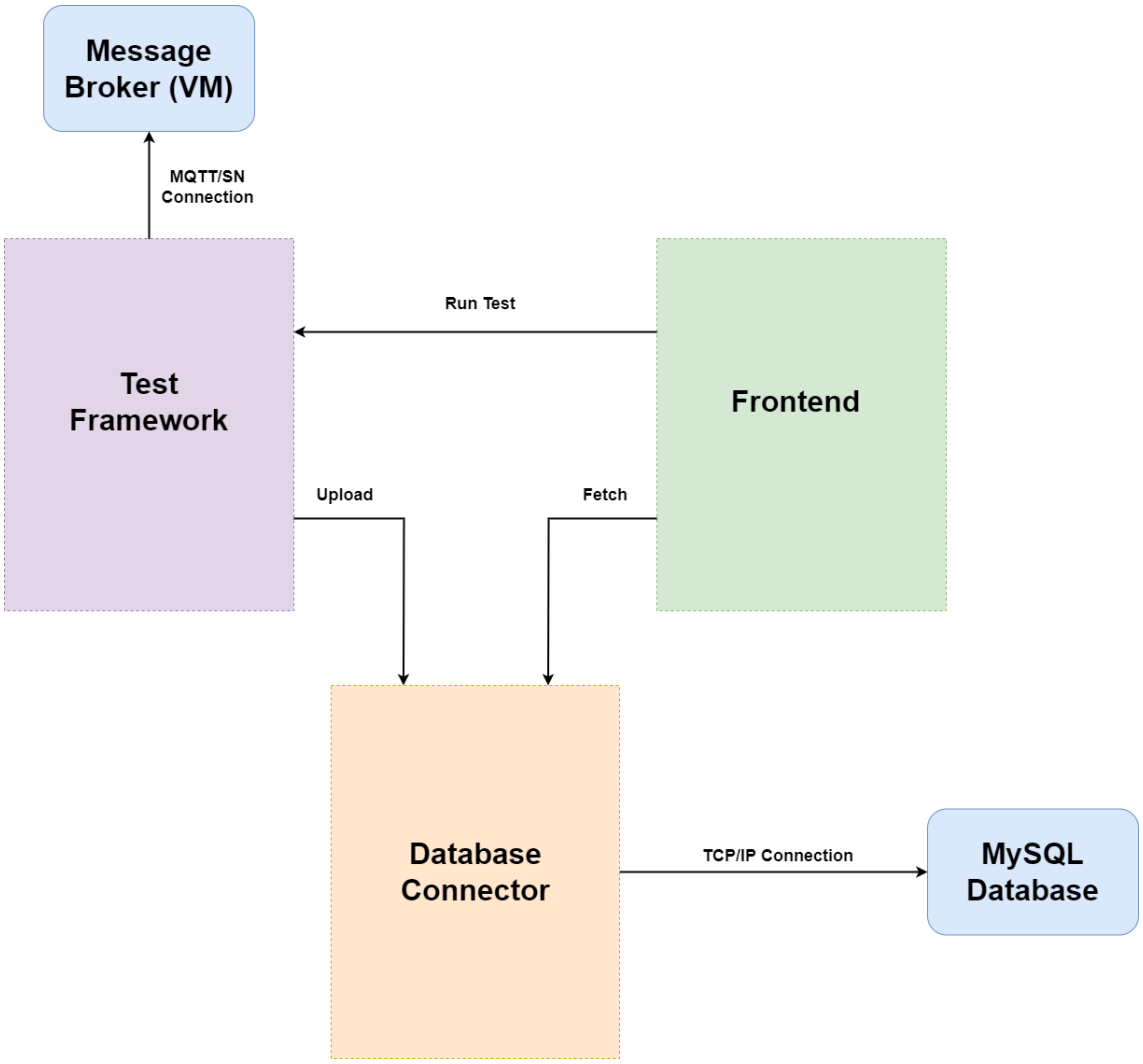

Figure 3.4: System overview

Figure 3.4 shows and overview of the entire system. There are three module contained in the analysis program, them being the test framework, frontend and database connector. The figure displays how the modules interact. The test framework contains a connection with the message broker, located on the VM. It also uses methods found in the database connector to upload data to the database. The frontend is able to run the test framework, using parameters specified in the configuration. It also uses methods from the database connector in order to fetch data from the database. The database connector maintains the connection to the MySQL database and sends queries based on input from the other modules.

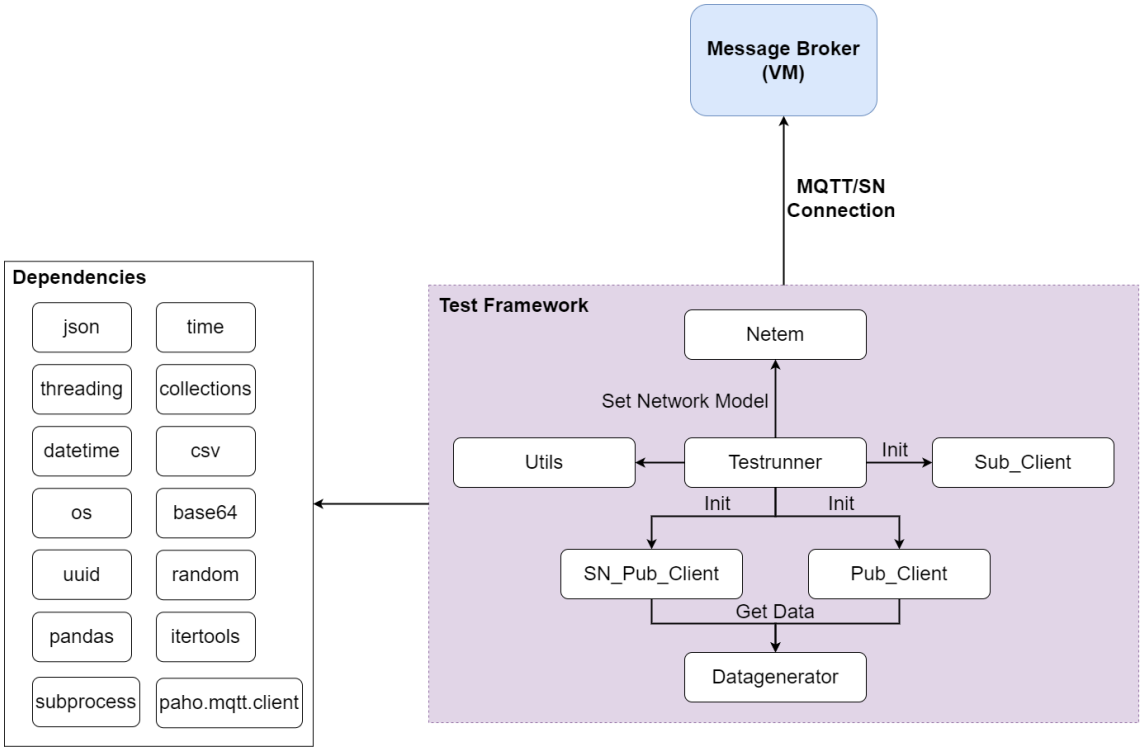

Figure 3.5: Overview of the test framework module

Figure 3.5 gives an overview of the test framework, its components as well as dependencies used in the components. Some of the dependencies are built-in in Python, while some are external (pandas, paho.mqtt.client) and requires installation via the package installer for Python (pip).

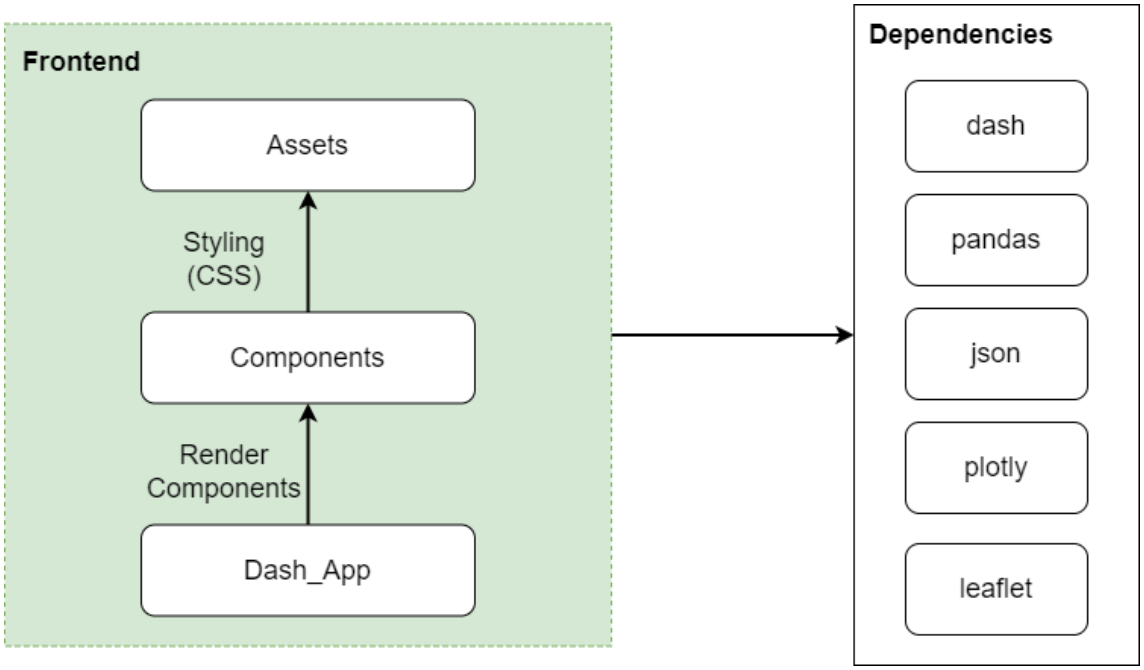

Figure 3.6: Overview of frontend module

Figure 3.6 shows the frontend module, developed using Plotly Dash. The Dash_App instantiates the application which renders the layout. It also contains the callback functions used to interact with the GUI. The layout renders the different sections of the application using the imported components. The components are split up in different files, e.g. the graph is its own components, as is the configuration panel. Assets contain the styling for each of the components. To create components the frontend module depend on dash, as well as plotly for graphs and leaflet for displaying map data. Pandas is used mostly to manage the data fetched from the database.

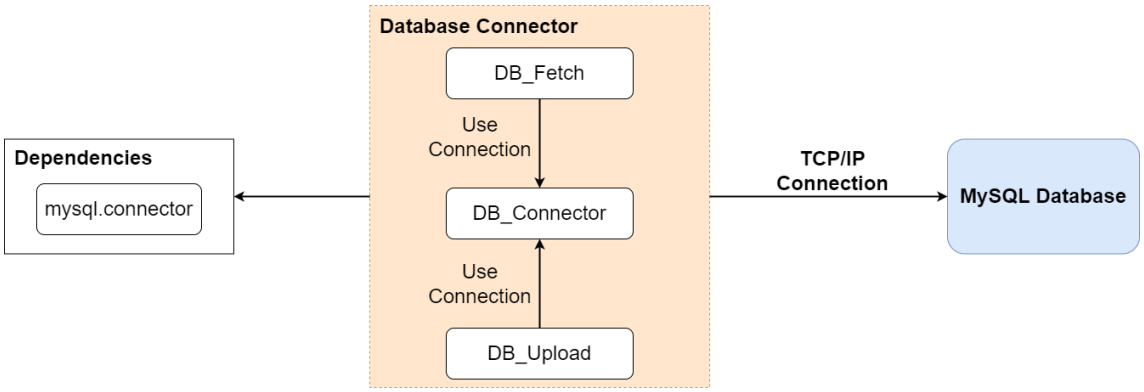

Figure 3.7: Overview of database-connector module

Figure 3.7 shows the database connector module. Both the DB_Fetch and the DB_Upload components require the connection that is maintained in the DB_Connector module to query the database. The only dependency used in this module is mysql.connector.

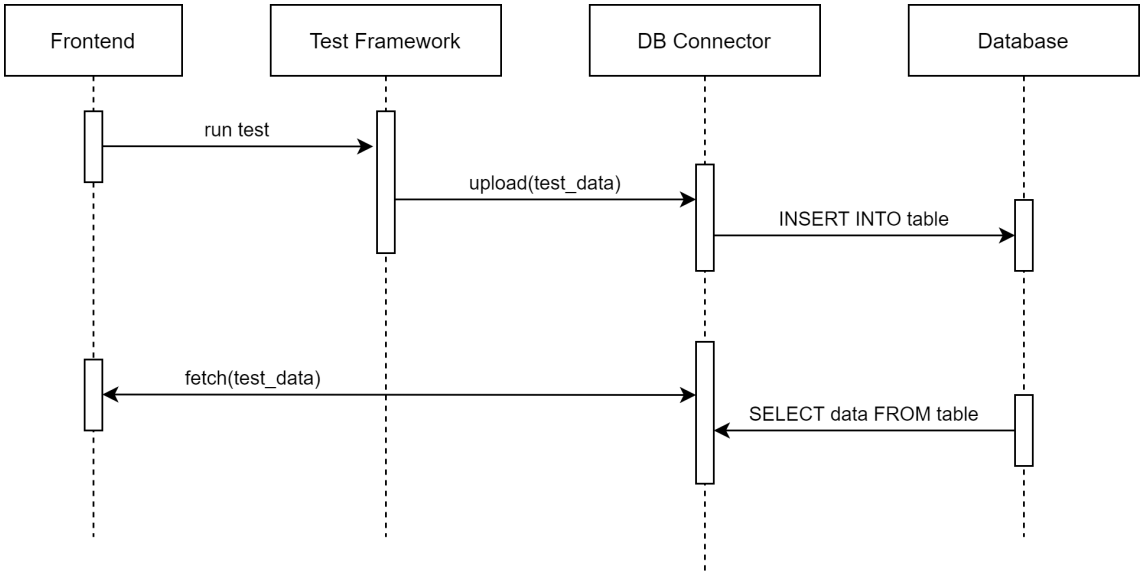

Figure 3.8: Sequence diagram showing the data flow through the modules

Figure 3.8 show the frontend and test framework modules interacting with the database connector to upload and fetch data. The frontend sends a command to the test framework to start the test, which then initializes the configuration and nodes. The information is then uploaded to the database through a method in the database connector. As the test is running, data is continuously uploaded to the database. When tables in the database is filled, the frontend can fetch this data through a fetch method in the database connector.

3.3 Additional tools

3.3.1 Wireshark

Wireshark [21] is a network analysis tool which provides the ability to monitor packets that are transported within a network. The tool was used to monitor and gather data related to the network layer. Using live capture with tcpdump [22], we can gather data during a test in a pcap file, that can be analysed after a test. Various display filters allow us to look at data that is only relevant to the protocols we have investigated.

3.3.2 Netem

Netem [18] is a network emulation tool that allows for simulation of different types of networks, with the ability to configure delay, packet-loss, available bandwidth and more. This is useful for testing various aspects related to how the protocols perform in different environments. To validate that the netem setting were correct, we used IPerf3 [23]. The tool was used in the analysis program to set the network models we wished to test through shell scripts. The network models can be found in table 4.1.

Chapter 4

Test setup and configuration

4.1 Test setup

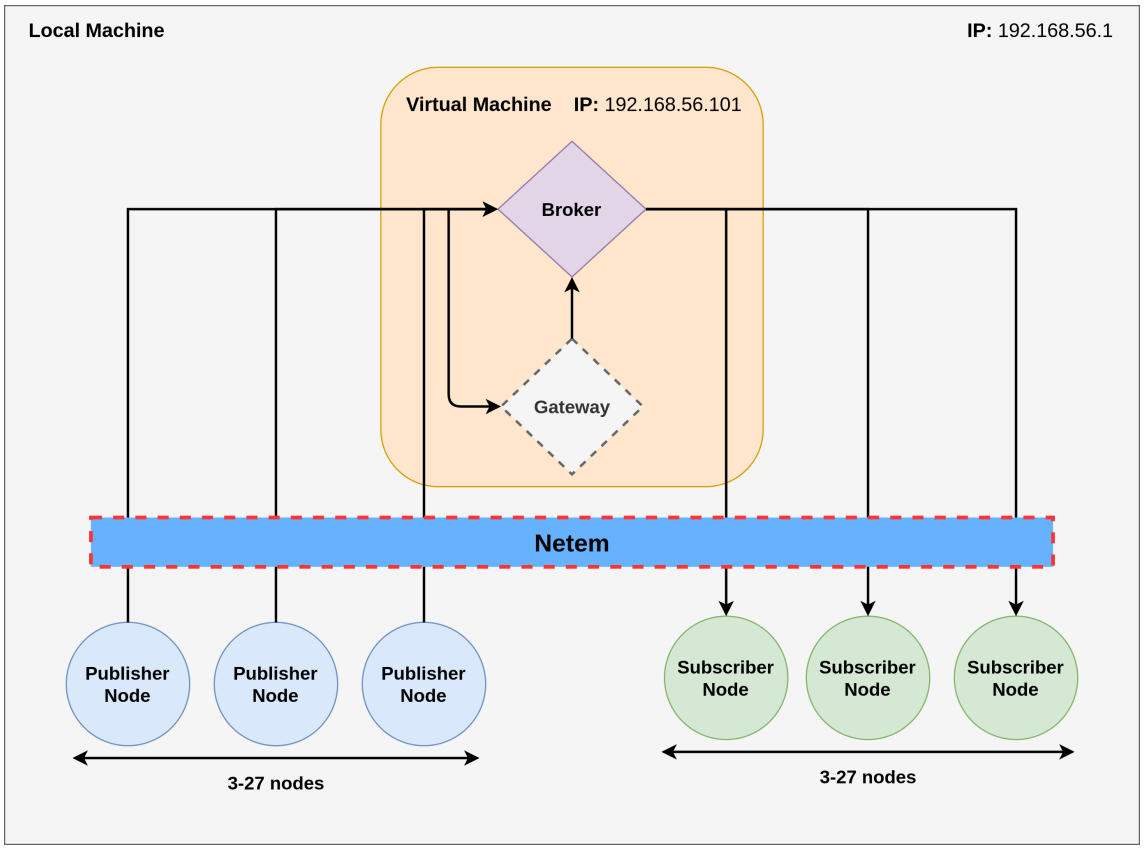

Figure 4.1: MQTT and MQTT-SN setup with publishers and subscribers, as well as a broker and gateway on the virtual machine

In order to test the capabilities of the protocols we needed to have a test setup which can be configured to benchmark the performance of each protocol. This setup must be dynamic and scale with multiple devices simulating a scenario that is close to real life and that can

present results in a structured way so that they can be analyzed and evaluated. For this we have developed an analysis tool that can be configured to be used for all protocols. In order to use the protocols, we require a broker and a gateway. Therefore we have set up a virtual Linux machine running Ubuntu (version 20.04.3 LTS), acting as a server that the clients can connect to in order to send and receive messages. By running the server in a virtual environment on the same machine we can disregard the issue of time synchronization that would occur if the setup was run on different devices. While MQTT-SN was not tested during this project, we have still included the setup as this is ready and working for use in further research.

4.1.1 MQTT setup

The broker chosen for this project is the Eclipse Mosquitto [24] message broker (version 1.6.9) which is simple and light weight. There are other brokers available that are more suited for larger projects where there are many more clients, but for this project the Mosquitto broker is found suitable. To conduct the tests we also need some clients that can publish data and receive it through subscriptions on the other end. The clients used were from the paho-mqtt-library (version 1.5.1) [25]. These clients were configured and run through the analysis tool, which can be scaled for as many clients as needed. Based on previous research we have decided to use 27 clients of each type in order to simulate a realistic scenario. To properly test the performance of the protocols there will be many clients running in parallel that are sending and receiving large amounts of data at a time. The connection between the server and the clients will be limited in order to understand how the protocols behave in networks with limited bandwidth and high loss rate which can result in severe connection issues. The tool used for simulating these environments is Netem, which we can configure to use with network models specified in table 4.1. Figure 4.1 gives an overview of the setup that was used during testing.

4.1.2 MQTT-SN setup

Testing using MQTT-SN required a slight change to the setup due to the need of a gateway component as well as a publisher that could publish MQTT-SN messages. The gateway used was an Eclipse Paho MQTT-SN transparent gateway [26]. The publisher client we used was a custom C to Python translated library [27]. At first there were some difficulties setting up the clients and having them reliably connect to the gateway, but this issue was resolved after some changes to the configuration. Since we are using a transparent gateway, the subscriber clients used were the same paho clients that were used during testing with MQTT. This means that the publishers produce MQTT-SN messages that are sent to the gateway, which are then transformed to MQTT messages that are forwarded to the broker, before reaching the subscribers. As with MQTT, we are able to use 27 clients of each type during testing, and can limit the network using Netem with the network models in table 4.1.

Network	Data rate	Latency	Loss percentage
Tactical Broadband	2 mbit/s	100 ms	1%
SATCOM	250 kbit/s	550 ms	0%
NATO Narrowband Waveform	16 kbit/s	500 ms	0%
CNR	9.6 kbit/s	100 ms	1% / 10%

Table 4.1: Tactical network models

4.1.3 Data

```
{
        "type": "Feature",
        "geometry": {
            "type": "Point",
            "coordinates": {
                "lat" : 9.321,
                "lon" : 53.154,
                "alt" : 0
            }
        },
        "properties": {
            "country": "NOR",
            "unit": "NOR-UNIT002",
            "node_name": "PUB_NOR_1",
            "node_id": 11,
            "msg_id": 12,
            "data_id" : 00c4901d-c6f2-4a69-8b76-58c58ce90211,
            "timestamp": 2021-11-30 09:32:36.435350
        }
}
```

Listing 1: Format of a message with GPS data that has been published

The messages that have been sent over the network contain data that simulates information that could be sent in real-life scenarios. This can be smaller messages like alerts or GPS data, or larger data in the form of images and reports. The data was packaged in a JSON object, using the GeoJSON [28] format for transfer with the protocols. The image data we have used during testing comes in the JPEG format with a resolution of 20 by 20 pixels and is encoded as a byte-string and packaged inside the JSON message, alongside other relevant information. This is useful for testing with larger message sizes. Listing 1 shows an example of how the data is structured. In this instance the data is of type GPS.

4.1.4 Test configuration

For each network model, two tests were ran, one with the GPS data type and one with image data. After all clients connected to the broker, the tests ran for a duration of 10 minutes before the publishing clients were disconnected. After the final messages were published there was a *cool down* period of five

minutes. This was to allow time for the subscribing clients to wait for pending messages that might be significantly delayed. After this cool down period, the subscriber clients were also disconnected. For each test, 27 publisher and 27 subscriber clients run simultaneously. Both types of clients were split in three, each using a different QoS level from another. This means that there were nine clients of each type using QoS 0, nine using QoS 1 and nine using QoS 2. This also means that all QoS levels where used at the same time. This configuration was inspired by Johnsen et al. [3] that used a similar setup. One change we have made is that the number of clients is increased to 27, up from 23, to have an even number of clients per QoS level. All publishing clients had a 10 seconds delay between each message they published. Other settings for MQTT and MQTT-SN were mostly standard client settings with no modifications.

Chapter 5

Results and analysis

This section describes the results from testing with the network models found in table 4.1. Tcpdump was used to capture a pcap file that could be used to gather metrics on the network layer. The tcpdump command was run at the start of each test, and terminated when the test concluded. The contents of the pcap file was later used in Wireshark for analysis. Data related to the application layer were analysed using the dash application. Application layer results show information related to each of the three QoS level found in the protocol.

5.1 Results using MQTT

5.1.1 Tactical Broadband

Table 5.1 shows the results from testing with tactical broadband on the network layer. The total number of packets captured was 14 157 for GPS data and 17 493 for image data. The data rate produced by GPS data was 15 kbit/s, and 32 kbit/s for image data. The average message size was 438 bytes for GPS and 962 bytes for images. With GPS data the number of retransmissions were 81 and the number of duplicate ACKs were 34. Image data produced 73 retransmissions and 35 duplicate ACKs.

Data Type	**GPS**	**Image**
Total packets	14157	17493
Data rate	15 kbit/s	32 kbit/s
Min message size	434 bytes	138 bytes
Avg message size	438 bytes	962 bytes
Max message size	440 bytes	1589 bytes
TCP retransmissions	81	73
Duplicate ACKs	34	35

Table 5.1: Results from Tactical Broadband tests on the network layer

Table 5.2 shows the results from both tests on the application layer. Both tests sent 1620 messages each and the packet loss was zero percent. For GPS data with QoS 0 and QoS 1, the average transmit delay was 0.12 seconds, while it was 0.34 seconds for QoS 2. The lowest delay found was 0.11 seconds for QoS 0 and 1, while the highest delay was 0.57 seconds using QoS 2. Using image data with QoS 0 and 1, gave an average delay of 0.12 seconds, and 0.34 seconds for QoS 2. The lowest delay found was

using QoS 0 and 1 with 0.11 seconds, while the highest delay was found using QoS 2 at 0.87 seconds. None of the clients disconnect during both tests.

Data type	QoS	Message sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
GPS	**0**	540	0%	0.11 s	0.12 s	0.46 s	0
	1	540	0%	0.11 s	0.12 s	0.45 s	0
	2	540	0%	0.31 s	0.34 s	0.80 s	0
	Total/Avg	1620	0%	0.19 s	0.19 s	0.57 s	0
Image	**0**	540	0%	0.11 s	0.12 s	0.45 s	0
	1	540	0%	0.11 s	0.12 s	0.69 s	0
	2	540	0%	0.31 s	0.34 s	0.97 s	0
	Total/Avg	1620	0%	0.19 s	0.20 s	0.70 s	0

Table 5.2: Results from Tactical Broadband tests on the application layer

The following network tests will display transmit delay data in a barchart. This was not included in tests using the tactical network model due to the low times found, making them not suitable for visual presentation and comparison.

5.1.2 SATCOM

Table 5.3 shows the results from testing using Satcom on the network layer. The total number of packets captured was 13 829 for GPS data and 17 535 for image data. The data rate from the test using GPS data is 15 kbit/s and 26 kbit/s for image data. The average message size for GPS data is 438 bytes and 856 bytes for images. The GPS test had 18 retransmissions and 27 duplicate ACKs, while the test with images had 34 retransmissions and 39 duplicate ACKs.

Data Type	GPS	Image
Total packets	13829	17535
Data rate	15 kbit/s	26 kbit/s
Min message size	433 bytes	138 bytes
Avg message size	438 bytes	856 bytes
Max message size	440 bytes	1589 bytes
TCP retransmissions	18	34
Duplicate ACKs	27	39

Table 5.3: Results from Satcom tests on the network layer

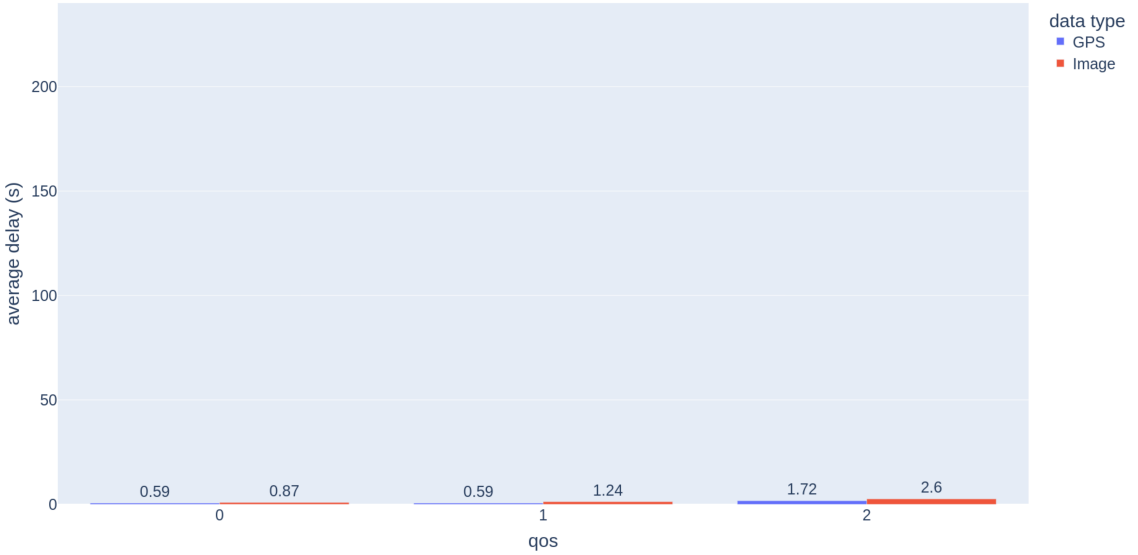

Figure 5.1: Average delay per QoS using the Satcom network model

Figure 5.1 shows that the average delay was slightly higher when using image data over GPS data. Table 5.4 shows the results from tests on the application layer with the Satcom network model. Each test had a total of 1619 messages sent and both had a packet loss of zero percent. For the test with GPS data, the average delay was 0.59 seconds using QoS 0 and 1, and 1.72 seconds using QoS 2. The lowest delay found was 0.57 seconds using QoS 0 and 1. The highest delay was found using QoS 2 and was 1.92 seconds. For the test with image data the average delay was 0.87 seconds for QoS 0, 1.24 seconds for QoS 1 and 2.6 seconds for QoS 2. The lowest delay found was 0.61 seconds using QoS 0, and the highest delay was 2.93 seconds using QoS 2. None of the clients disconnected during both of the tests.

Data type	QoS	Message sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
GPS	**0**	540	0%	0.57 s	0.59 s	1.13 s	0
	1	540	0%	0.57 s	0.59 s	0.69 s	0
	2	539	0%	1.68 s	1.72 s	1.92 s	0
	Total/Avg	1619	0%	0.94 s	2.90 s	1.25 s	0
Image	**0**	540	0%	0.61 s	0.87 s	1.99 s	0
	1	540	0%	0.73 s	1.24 s	1.66 s	0
	2	539	0%	2.37 s	2.60 s	2.93 s	0
	Total/Avg	1619	0%	1.24 s	1.57 s	2.19 s	0

Table 5.4: Results from Satcom tests on the application layer

5.1.3 NATO Narrowband Waveform

Table 5.5 shows the results from testing with the NATO Narrowband model on the network layer. The total number of packets captured was 14 561 for GPS data and 8541 for image data. The data rate for GPS data was 15 kbit/s and 10 kbit/s for images. The average message size was 438 bytes for GPS and 1343 bytes for images. Testing using GPS data had 359 retransmissions and 366 duplicate ACKs. For image data the number of retransmissions was 1347 and the number of duplicate ACKs was 619.

Data Type	**GPS**	**Image**
Total packets	14561	8541
Data rate	15 kbit/s	10 kbit/s
Min message size	433 bytes	138 bytes
Avg message size	438 bytes	1343 bytes
Max message size	443 bytes	2962 bytes
TCP retransmissions	359	1347
Duplicate ACKs	366	619

Table 5.5: Results from NATO Narrowband tests on the network layer

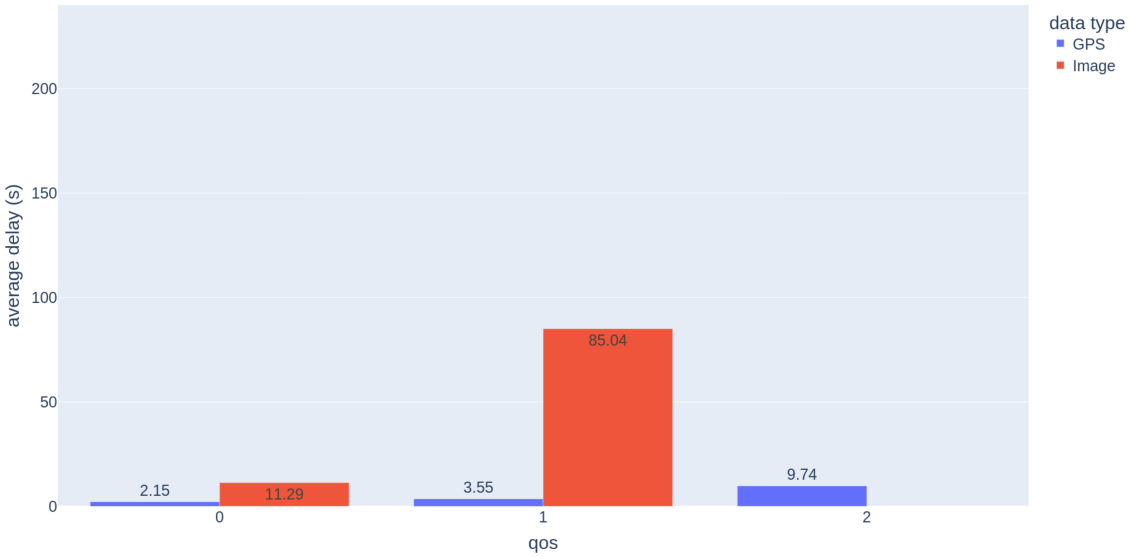

Figure 5.2: Average delay per QoS using the NATO Narrowband network model

Figure 5.2 shows that the average delay was significantly higher when using image data for QoS 1. It also shows that no messages arrived using QoS 2. Table 5.6 shows the results from the application layer. The number of messages sent using GPS data was 1619 and there was no packet loss. The average delay found was 2.15 seconds using QoS 0, 3.55 seconds using QoS 1 and 9.74 seconds using QoS 2. The smallest delay found was 0.72 seconds using QoS 0 and the highest was 32 seconds using QoS 2. No clients disconnected during the test with GPS data. For image data, a total of 672 messages were sent. QoS 0 had a packet loss of 90.22%, QoS 1 had a loss of 85.02% and QoS 2 lost all messages. The average delay was 11.29 seconds using QoS 0 and 85.04 seconds using QoS 1. The lowest delay found was 1.39 seconds using QoS 0 and the highest was 138.53 seconds using QoS 1. With QoS 0, 29 clients disconnected, with QoS 1 36 clients disconnected and QoS 2 had 42 clients disconnect.

Data type	QoS	Message sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
GPS	**0**	540	(0%)	0.72 s	2.15 s	10.85 s	0
	1	540	(0%)	1.01 s	3.55 s	7.48 s	0
	2	539	(0%)	7.73 s	9.74 s	32.00 s	0
	Total/Avg	1619	(0%)	3.15 s	5.15 s	16.78 s	0
Image	**0**	184	166 (90.22%)	1.39 s	11.29 s	21.69 s	29
	1	247	210 (85.02%)	7.98 s	85.04 s	138.53 s	36
	2	241	241 (100%)	NaN	NaN	NaN	42
	Total/Avg	672	617 (91.82%)	4.69 s	48.17 s	80.11 s	107

Table 5.6: Results from NATO Narrowband tests on the application layer

5.1.4 CNR with 1% loss

Table 5.7 shows the results on the network layer using the CNR network model with 1% loss. The total number of packets sent during testing with GPS data was 9469 and 7022 with image data. GPS data had a data rate of 8.75 kbit/s and image data had a rate of 5.70 kbit/s. The average message size was 1162 bytes using GPS data and 1269 bytes using image data. The number of retransmissions was 1640 and the number of duplicate ACKs was 804, using GPS data. For image data the number of retransmissions was 1397 and the number of duplicate ACKs was 493.

Data Type	GPS	Image
Total packets	9469	7022
Data rate	8.75 kbit/s	5.70 kbit/s
Min message size	95 bytes	138 bytes
Avg message size	1162 bytes	1269 bytes
Max message size	1514 bytes	1588 bytes
TCP retransmissions	1640	1397
Duplicate ACKs	804	493

Table 5.7: Results from CNR tests with 1% loss on the network layer

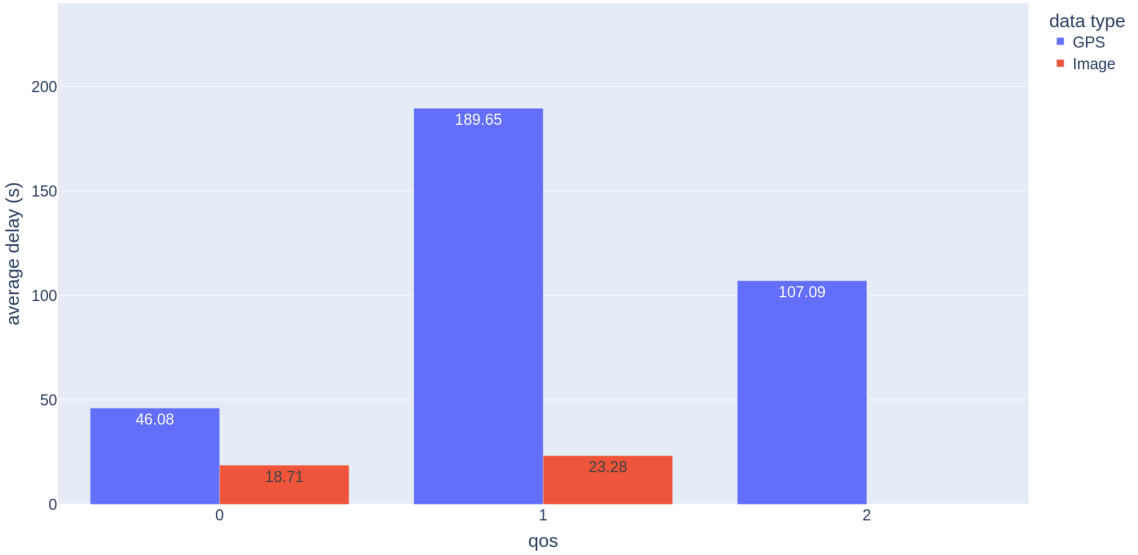

Figure 5.3: Average delay per QoS using the CNR network model with 1% loss

Figure 5.3 shows that the average delay was higher when using GPS data than image data. We can also see that no messages arrived for QoS 2 when sending image data. Table 5.8 show the results for CNR with 1% loss on the application layer. The total number of messages sent using GPS data was 560. QoS 0 had a 42.64% loss, QoS 1 had a 34.40% loss and QoS 2 had the highest loss with 89.67% loss. The average delay was 46.08 seconds using QoS 0, 189.65 seconds using QoS 1 and 107.09 seconds using QoS 2. The lowest delay recorded was 0.58 seconds using QoS 0 and the highest delay was 618.29 seconds using QoS 1. QoS 0 and QoS 2 had 44 clients disconnect, while QoS 1 had 42 clients disconnect during the test. Using image data, a total of 398 messages were sent. QoS 0 had a packet loss of 85.19%, QoS 1 had a packet loss of 93.51% and QoS 2 lost all messages. The average delay for QoS 0 was 18.71 seconds and QoS 1 had an average delay of 23.28 seconds. The lowest delay found was 1.59 seconds using QoS 0 and the highest delay was 63.66 seconds using QoS 1. QoS 0 and QoS 1 had 19 clients disconnect each, while QoS 2 had 24 clients disconnect during the test.

Data type	QoS	Message sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
GPS	**0**	129	55 (42.64%)	0.58 s	46.08 s	113.85 s	44
	1	218	75 (34.40%)	3.05 s	189.65 s	618.29 s	42
	2	213	191 (89.67%)	31.26 s	107.09 s	219.85 s	44
	Total/Avg	560	321 (57.32%)	11.63 s	114.27 s	317.33 s	130
Image	**0**	108	92 (85.19%)	1.59 s	18.71 s	38.69 s	19
	1	154	144 (93.51%)	12.56 s	23.28 s	63.66 s	19
	2	136	136 (100%)	NaN	NaN	NaN	24
	Total/Avg	398	372 (93.47%)	7.08 s	21.00 s	51.18 s	62

Table 5.8: Results from CNR tests with 1% loss tests on the application layer

5.1.5 CNR with 10% loss

Table 5.9 show the test results using CNR with 1% loss on the network layer. Using GPS data the total number of packets sent was 8509 and using image data the number was 7852. The data rate for tests with GPS data was 7.66 kbit/s and 5.96 kbit/s using image data. The average message size was 988 bytes with GPS data and 1161 bytes with image data. Using GPS data the number of retransmissions was 1771 and the number of duplicate ACKs was 779. For image data the number of retransmissions was 1487 and the number of duplicate ACKs was 830.

Data Type	**GPS**	**Image**
Total packets	8509	7852
Data rate	7.66 kbit/s	5.96 kbit/s
Min message size	92 bytes	138 bytes
Avg message size	988 bytes	1161 bytes
Max message size	1514 bytes	1662 bytes
TCP retransmissions	1771	1487
Duplicate ACKs	779	830

Table 5.9: Results from CNR tests with 10% loss on the network layer

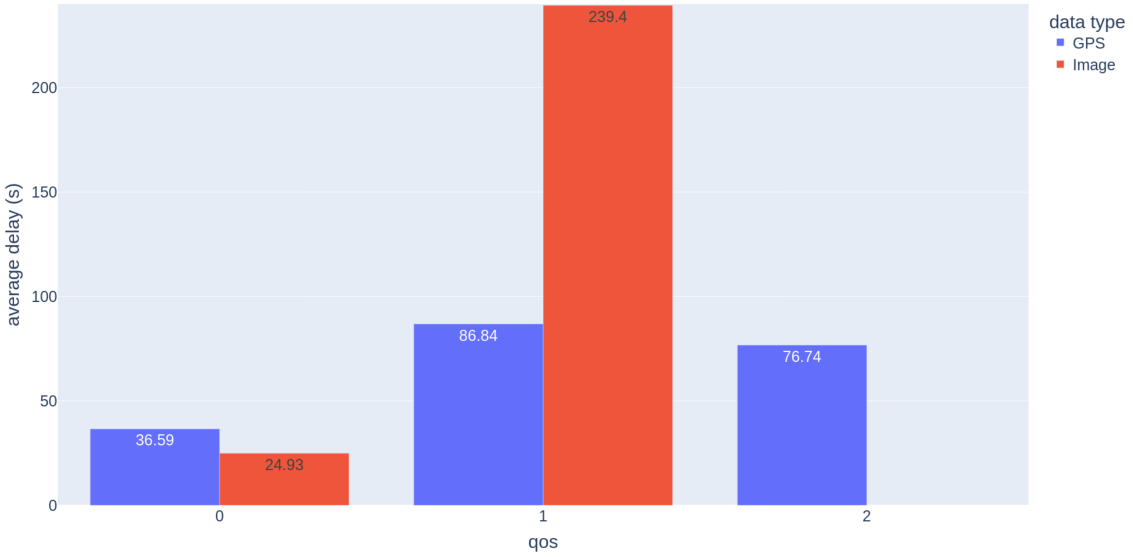

Figure 5.4: Average delay per QoS using the CNR network model with 10% loss

Figure 5.4 shows that using QoS 0, GPS data had a higher average delay over image data. Using QoS 1, the average delay was much higher when sending image data. QoS 2 had no messages arrive when sending image data. Table 5.10 show the results from the application layer. Using GPS data the number of messages sent was 638. QoS 0 had a packet loss of 73.42%, QoS 1 had a packet loss of 64.41% and QoS 2 had a packet loss of 93.84%. The average delay for QoS 0 was 36.59 seconds, for QoS 1 it was 86.84 seconds and for QoS 2 it was 76.47 seconds. The lowest delay found was 0.58 seconds using QoS 0 and the highest delay was 389.86 seconds using QoS 1. QoS 0 had 42 clients disconnect, QoS 1 had 43 clients disconnect and QoS 2 had 48 clients disconnect. The test with image data had a

total of 418 messages sent. QoS 0 had a packet loss of 86.99%, QoS 1 had a loss of 87.01% and QoS 2 lost all messages. The average delay for QoS 0 was 24.93 seconds and for QoS 1 it was 239.4 seconds. The lowest delay found was 1.62 seconds using QoS 0 and the highest delay was 581.40 seconds using QoS 1. QoS 0 had 37 clients disconnect, QoS 1 had 47 clients disconnect and QoS 2 had 49 clients disconnect during the test.

Data type	QoS	Message sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
GPS	**0**	237	174 (73.42%)	0.58 s	36.59 s	87.72 s	42
	1	295	190 (64.41%)	2.73 s	86.84 s	389.86 s	43
	2	292	274 (93.84%)	28.29 s	76.47 s	149.15 s	48
	Total/Avg	638	321 (50.03%)	10.53 s	66.63 s	208.91 s	133
Image	**0**	123	107 (86.99%)	1.62 s	24.93 s	58.82 s	37
	1	154	134 (87.01%)	9.84 s	239.40 s	581.40 s	47
	2	141	141 (100%)	NaN	NaN	NaN	49
	Total/Avg	418	382 (91.14%)	5.73 s	132.17 s	320.11 s	133

Table 5.10: Results from CNR tests with 10% loss tests on the application layer

5.2 Analysis

Table 5.11 shows the gathered results from all tests with GPS data on the network layer. From the results we have some observations:

- While using the Tactical Broadband, Satcom and NATO Narrowband network models, the data rate is stable at 15 kbit/s. The CNR network model with 1% and 10% loss shows much lower data rates, below 10 kbit/s.
- It should be noted that the data rates captured during testing is reduced slightly because of the cool down period after the inital duration of the tests. This is due to smaller ping request/reply packets that are sent because of the 60 seconds keep alive set when configuring the clients.
- The message sizes remain the same size for tests using Tactical Broadband, Satcom and NATO Narrowband. The sizes increase drastically when using CNR with 1% and 10% loss, more than doubling in sizes from the other models. From observations done, this seems to happen due to messages being packaged together and sent as one message. This only applies to messages sent using QoS 1 and QoS 2. The reason for this might be due to TCP packet batching which is used to increase the efficiency by reducing the number of packets sent over the network at a time.
- Retransmissions and duplicate ACKs happens rarely compared to the number of packets sent using the Tactical Broadband and Satcom network models. There is a slight increase using NATO Narrowband, but the most significant increase happens with the CNR models, which both have a high amount of retransmissions and duplicate ACKs.

Network model	Data rate	Avg message size	TCP retransmissions	Duplicate ACK
Tactical broadband	15 kbit/s	438 bytes	81	34
SATCOM	15 kbit/s	438 bytes	18	27
NATO narrowband	15 kbit/s	438 bytes	359	366
CNR 1% loss	8.75 kbit/s	1162 bytes	1640	804
CNR 10% loss	7.66 kbit/s	988 bytes	1771	779

Table 5.11: Results from all standard tests using GPS data on the network layer

Table 5.12 shows the gathered results from all tests with image data on the network layer. From the results we have some observations:

- The test with Tactical Broadband had the highest data rate at 32 kbit/s. The following tests showed a decline in data rates, with NATO Narrowband and CNR tests having a very low data rate.
- The message sizes vary quite significantly in the different network model tests. This variance in size might be explained by the fact that TCP splits up the messages into several packets. We observed many packets of different sizes and some were very small (138 bytes).
- Tactical Broadband and Satcom tests caused very few retransmissions and duplicate ACKs, while NATO Narrowband and CNR tests caused a large number of retransmissions and duplicate ACKs.

Network model	Data rate	Avg message size	TCP retransmissions	Duplicate ACK
Tactical broadband	32 kbit/s	962 bytes	73	35
SATCOM	26 kbit/s	856 bytes	34	39
NATO narrowband	10 kbit/s	1343 bytes	1347	619
CNR 1% loss	8.75 kbit/s	1269 bytes	1397	493
CNR 10% loss	5.96 kbit/s	1161 bytes	1487	830

Table 5.12: Results from all standard tests using image data on the network layer

Table 5.13 shows the gathered results from all tests with GPS data on the application layer. From the results we have some observations:

- The Tactical Broadband, Satcom and NATO Narrowband network models had no issue transmitting messages and recorded zero packet loss.
- QoS 0 and QoS 1 had the same or similar transmit delay using Tactical Broadband, Satcom and NATO Narrowband. There was only a slight increase in delay for QoS 1 using NATO Narrowband. QoS 2 had a much larger transmit delay with these models compared to QoS 0 and QoS 1.
- CNR with 1% and 10% loss caused issues with message transmissions for all QoS levels. QoS 1 had the lowest packet loss for both network models, while QoS 2 had very high amounts of packet loss.
- QoS 0 and 1 saw a large increase in packet loss from the CNR test with 1% loss to the test with 10% loss. They both saw an increase in about 30%.
- QoS 0 had much lower transmit delays compared to QoS 1 and QoS 2 when using the CNR network models. QoS 1 had the highest transmit delays, while QoS 2 was slightly lower but still high. The reason for the lower transmit delay for QoS 2 can be because of the low amount of

messages that were sent and received.

- The number of disconnected clients were similar for all QoS levels when testing with CNR. QoS 1 had the lowest amount of clients disconnect at 42 and QoS 2 had the highest at 48.

Network model	QoS	Messages Sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
Tactical broadband	**0**	540	0%	0.11 s	0.12 s	0.46 s	0
	1	540	0%	0.11 s	0.12 s	0.45 s	0
	2	540	0%	0.31 s	0.34 s	0.80 s	0
SATCOM	**0**	540	0%	0.57 s	0.59 s	1.13 s	0
	1	540	0%	0.57 s	0.59 s	0.69 s	0
	2	539	0%	1.68 s	1.72 s	1.92 s	0
NATO narrowband	**0**	540	0%	0.72 s	2.15 s	10.85 s	0
	1	540	0%	1.01 s	3.55 s	7.48 s	0
	2	539	0%	7.73 s	9.74 s	32.00 s	0
CNR 1% loss	**0**	129	55 (42.64%)	0.58 s	46.08 s	113.85 s	44
	1	218	75 (34.40%)	3.05 s	189.65 s	618.29 s	42
	2	213	191 (89.67%)	31.26 s	107.09 s	219.85 s	44
CNR 10% loss	**0**	237	174 (73.42%)	0.58 s	36.59 s	87.72 s	42
	1	295	190 (64.41%)	2.73 s	86.84 s	389.86 s	43
	2	292	274 (93.84%)	28.29 s	76.47 s	149.15 s	48

Table 5.13: Results from all tests using GPS data on the application layer

Table 5.14 shows the gathered results from all tests with image data on the application layer. From the results we have some observations:

- Message transmission went without any issues and there was no recorded packet loss when testing with Tactical Broadband and Satcom.
- The transmit delay was similar to what was found when testing with GPS data for Tactical Broadband and Satcom. The delay was similar for all levels of QoS, with an average delay of 0.12 seconds for QoS 0 and 1 and 0.34 seconds for QoS 2.
- The packet loss was significant for tests with NATO Narrowband and CNR with both 1% and 10% loss. Most messages were lost during these tests, where the lowest packet loss was 85.02% with QoS 1 using the NATO Narrowband network model. There were few messages sent during these tests. Using QoS 2, zero messages arrived during these tests.
- The average delay was significantly high during these test. The lowest delay was 11.29 seconds using QoS 0 with the NATO Narrowband network model and the highest was 239.40 seconds using QoS 2 and CNR with 10% loss. Comparing these tests to the tests with GPS, the average delay was significantly lower when testing using CNR with 1% loss. This result is not as expected, but could be due to the randomness of the network with the limitations set. There is also much fewer messages sent during these tests compared to the ones with GPS data.
- Tests using NATO Narrowband and CNR had a large amount of clients disconnect. Tests using CNR with 1% loss did not have as many clients disconnect, which might help explain the lower delay compared to the same test with GPS data.

Network model	QoS	Messages Sent	Packet loss	Min delay	Avg delay	Max delay	Disconnections
Tactical broadband	**0**	540	0%	0.11 s	0.12 s	0.45 s	0
	1	540	0%	0.11 s	0.12 s	0.69 s	0
	2	540	0%	0.31 s	0.34 s	0.97 s	0
SATCOM	**0**	540	0%	0.61 s	0.87 s	1.99 s	0
	1	540	0%	0.73 s	1.24 s	1.66 s	0
	2	539	0%	2.37 s	2.60 s	2.93 s	0
NATO narrowband	**0**	184	166 (90.22%)	1.39 s	11.29 s	21.69 s	29
	1	247	210 (85.02%)	7.98 s	85.04 s	138.53 s	36
	2	241	241 (100%)	NA	NA	NA	42
CNR 1% loss	**0**	108	92 (85.19%)	1.59 s	18.71 s	38.69 s	19
	1	154	144 (93.51%)	12.56 s	23.28 s	63.66 s	19
	2	136	136 (100%)	NA	NA	NA	24
CNR 10% loss	**0**	123	107 (86.99%)	1.62 s	24.93 s	58.82 s	37
	1	154	134 (87.01%)	9.84 s	239.40 s	581.40 s	47
	2	141	141 (100%)	NA	NA	NA	49

Table 5.14: Results from all tests using image data on the application layer

Summarizing the tests using the MQTT protocol, we can see that the protocol performs well with the Tactical Broadband and Satcom network models. Running 27 clients of each type and a publish delay of 10 seconds did not cause any significant performance issues, even with a varying degree of message sizes. Both network models produces similar data rates and there were very few retransmissions and duplicate ACKs, relative to the number of packets sent. We also see that QoS 0 and QoS 1 had very similar delay times, especially when sending smaller messages with GPS data.

While testing the NATO Narrowband network model, there were not many issues when sending GPS data. No message were lost and the data rate was identical to the previous models. The biggest change was the average delay, which was many times bigger than that of the previous models, but performance can still be considered good. Using image data which produced larger message sizes, the test showed that the protocol struggled regardless of QoS level. Packet loss was high and so was the delay on messages. The number of clients that disconnected was also very high.

The tests using the CNR network model with 1% and 10% loss had significant performance issues regardless of data type. Very few messages were sent compared to the tests with the other models, and packet loss was high. Sending larger messages with image data caused all packets to be lost for QoS 2 and nearly all to be lost using QoS 0 and 1. QoS 0 had the lowest delay during these test, but was still quite high. A large number of clients disconnected during these test, which can explain the low amount of messages sent. One discrepancy with the tests was that using image data and CNR with 1% loss, the number of clients that disconnected was quite low compared to other tests.

Chapter 6

Conclusion

During this project we have tested and evaluated the performance of the MQTT protocol using different network models. To do this we have created an analysis tool that has been helpful in allowing us to create custom test configurations and perform analysis on data from the tests. The conclusions we have come to in this project, came from the metrics we have gathered data on and analyzed using the analysis tool. These metrics include data from the network layer, as well as the application layer, and the most important metrics were data rate, packet loss and transmit delay. After analyzing the tests we see that the protocol performs badly in the most challenging networks, and it is therefore not suitable for use in these areas. When producing smaller, text based messages, the protocol performs well in tactical broadband, satellite and narrowband networks, but it struggles on narrower networks like CNR, and can not be considered reliable in these use cases. We have looked at larger message sizes, packaged with image data, which we have seen perform well in tactical broadband and satellite networks, but struggles significantly in the narrower networks, making it unsuitable. The problems we see occuring in the more limited networks can be attributed to the underlying TCP protocol that causes congestion on the network. This is especially the case when using QoS 1 and QoS 2.

Table 6.1 shows the network models we have tested during this project and the where we recommend the use of the MQTT protocol. Networks with wider communication channels like Tactical Broadband and Satcom are recommended for use with MQTT, while more limited networks like NATO Narrowband and CNR are not. We do recommend using MQTT for networks similar to NATO Narrowband, but only with messages of smaller sizes. We have also recommended the QoS level to use, which in most cases will be level 0 and 1, as level 2 causes too much overhead to be useful in any of the scenarios tested.

Network model	***Data type / Size***	***Recommended***	***QoS***
Tactical broadband	GPS data (∼400 bytes)	Yes	0 and 1
	Image data (∼1300 bytes)	Yes	0 and 1
SATCOM	GPS data (∼400 bytes)	Yes	0 and 1
	Image data (∼1300 bytes)	Yes	0 and 1
NATO Narrowband Waveform	GPS data (∼400 bytes)	Yes	0
	Image data (∼1300 bytes)	No	
CNR 1% loss	GPS data (∼400 bytes)	No	
	Image data (∼1300 bytes)	No	
CNR 10% loss	GPS data (∼400 bytes)	No	
	Image data (∼1300 bytes)	No	

Table 6.1: Recommendations for using MQTT with different network models and which QoS levels should be used, from a technical perspective

For future work, we want to look at testing the MQTT-SN protocol as well, to compare it to our results from the MQTT tests. The analysis tool has already been set up during this project with the necessary components needed to test the protocol. This project has therefore laid the groundwork for us to be able to compare the two protocols, which is one of the goals for the master's thesis. Another goal will be to test similar publish/subscribe protocols like ZeroMQ [29]. We should also look at testing with even higher message sizes than the ones used in this project. The MQTT protocol is capable of transmitting larger messages than the ones we have used, which can be considered to be quite small. The analysis tool we have developed is configured in such a way that implementing a test framework for a new protocol should be simple. There are still some aspects of the analysis tool that are not yet finished, and work on this should be continued in order to provide a more complete analysis of the protocols.

Bibliography

[1] K. L. Scott, T. Refaei, N. Trivedi, J. Trinh and J. P. Macker, 'Robust communications for disconnected, intermittent, low-bandwidth (dil) environments,' *2011 - MILCOM 2011 Military Communications Conference*, pp. 1009–1014, 2011.

[2] Matthew O'Riordan, *Everything you need to know about publish/subscribe*, https://ably.com/topic/pub-sub.

[3] F. T. Johnsen, T. H. Bloebaum, N. Jansen, G. Bovet, M. Manso, A. Toth and K. S. Chan, *Evaluating Publish/Subscribe Standards for Situational Awareness using Realistic Radio Models and Emulated Testbed*, 24th International Command and Control Research and Technology Symposium (ICCRTS), October 29-31 2019, Laurel, Maryland, USA.

[4] IBM, *Ws-notification: Overview*, https://www.ibm.com/docs/en/was/9.0.5?topic=notification-ws-overview.

[5] Edited by Mariann Hauge, *Anglova scenario*, https://anglova.net/.

[6] S. Lee et al, *Correlation analysis of mqtt loss and delay according to qos level*, https://www.researchgate.net/publication/261230513_Correlation_analysis_of_MQTT_loss_and_delay_according_to_QoS_level.

[7] T. Bray, 'The JavaScript Object Notation (JSON) Data Interchange Format,' Google, Inc., RFC 7159, Mar. 2014, pp. 1–15. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7159.

[8] IBM, *Mqtt restrictions and limitations*, https://www.ibm.com/docs/en/watson-iot-platform?topic=messaging-restrictions-limitations.

[9] R. R. Stewart, *Stream Control Transmission Protocol*, RFC 4960, Sep. 2007. DOI: 10.17487/RFC4960. [Online]. Available: https://rfc-editor.org/rfc/rfc4960.txt.

[10] S. Cope, *Introduction to mqtt-sn (mqtt for sensor networks)*, http://www.steves-internet-guide.com/mqtt-sn/.

[11] Packt, *Understanding wildcards*, https://subscription.packtpub.com/book/application_development/9781787287815/1/ch01lvl1sec18/understanding-wildcards.

[12] Oasis, *Mqtt version 3.1.1*, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718018.

[13] oasis, *Mqtt-sn$_s$pec$_v$*1.2, https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf.

[14] Matthew Treinish and Jeremy Stanley, *Pymqttbench*, https://github.com/mtreinish/pymqttbench.

[15] *Pandas*, https://pandas.pydata.org/.

[16] Plotly, *Plotly dash*, https://plotly.com/dash/.

[17] Python Software Foundation, *Tkinter — python interface to tcl/tk*, https://docs.python.org/3/library/tkinter.html.

[18] Linux Foundation, *Netem*, https://wiki.linuxfoundation.org/networking/netem.

[19] OpenStreetMap contributors, *Leaflet*, https://dash-leaflet.herokuapp.com/.

[20] Oracle Corporation, *Mysql connector/python developer guide*, https://dev.mysql.com/doc/connector-python/en/.

[21] Wireshark Foundation, *About wireshark*, https://www.wireshark.org/.

[22] T. T. Group, *Tcpdump and libcap*, https://tcpdump.org/.

[23] J. D. et al., *Iperf - the ultimate speed test tool for tcp, udp and sctp*, https://iperf.fr/.

[24] Eclipse Foundation, *Eclipse mosquitto™ an open source mqtt broker*, https://mosquitto.org/.

[25] Eclipse Foundation, *Paho-mqtt*, https://pypi.org/project/paho-mqtt/.

[26] Eclipse Foundation, *Mqtt-sn transparent gateway*, https://eclipse.org/paho/index.php?page=components/mqtt-sn-transparent-gateway/index.php.

[27] S. Cope, *Using the python mqtt-sn client*, http://www.steves-internet-guide.com/python-mqttsn-client/.

[28] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub and S. Hagen, *The GeoJSON Format*, RFC 7946, Aug. 2016. DOI: 10.17487/RFC7946. [Online]. Available: https://rfc-editor.org/rfc/rfc7946.txt.

[29] The ZeroMQ authors, *Zeromq*, https://zeromq.org/.