DEGREE PROJECT IN INFORMATION TECHNOLOGY, SECOND CYCLE
STOCKHOLM, SWEDEN 2016

Evaluation of communication protocols between vehicle and server

Evaluation of data transmission overhead by communication protocols

TOMAS WICKMAN

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

Evaluation of communication protocols between vehicle and server

Evaluation of data transmission overhead by communication protocols

Tomas Wickman

2016-06-29

Master's Thesis

Examiner
Gerald Q. Maguire Jr.

Academic adviser
Anders Västberg

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

This thesis project has studied a number of protocols that could be used to communicate between a vehicle and a remote server in the context of Scania's connected services. While there are many factors that are of interest to Scania (such as response time, transmission speed, and amount of data overhead for each message), this thesis will evaluate each protocol in terms of how much data overhead is introduced and how packet loss affects this overhead. The thesis begins by giving an overview of how a number of alternative protocols work and what they offer with regards to Scania's needs. Next these protocols are compared based on previous studies and each protocol's specifications to determine which protocol would be the best choice for realizing Scania's connected services. Finally, a test framework was set up using a virtual environment to simulate different networking conditions. Each of the candidate protocols were deployed in this environment and setup to send sample data. The behaviour of each protocol during these tests served as the basis for the analysis of all of these protocols. The thesis draws the conclusion that to reduce the data transmission overhead between vehicles and Scania's servers the most suitable protocol is the UDP based MQTT-SN.

Keywords

MQTT, MQTT-SN, AMQP, TCP, UDP, packet loss, header data, overhead, publish-subscribe

Sammanfattning

I den här rapporten har jag undersökt ett antal protokoll som kan användas för att kommunicera mellan server och lastbil och därmed användas för Scanias Connected Services. Då det är många faktorer som är intressanta när det kommer till kommunikation mellan lastbil och server för Scania som till exempel responstid, överföringshastighet och mängden extra data vid överföring så har jag valt att begränsa mig till att utvärdera protokollen utifrån hur mycket extra data de använder vid överföring och hur detta påverkas av paketförlust. Rapporten börjar med att ge en överblick över vilka tänkbara protokoll som kan användas och vad de kan erbjuda gällande Scanias behov. Efter det så jämförs protokollen baserat på tidigare studier och protokollens specifikationer för att avgöra vilket protokoll som är bäst lämpat att användas i Scanias Connected Services. Sists så skapas ett virtuellt ramverk för att simulera olike nätverksförhållanden. Här testas varje protokoll och får sända olike datamängder för att sedan få sin prestanda utvärderad baserat på hur mycket extra data som sändes. Dessa resultat ligger sedan till grund för den analys och slutsats angående vilket protokoll som är bäst lämpat att användas av Scania. Rapporten drar slutsatsen att baserat på den information som finns tillgänglig och de resultat som ficks av testerna så skulle den UDP baserade MQTT-SN vara bäst lämpad för att minimera mängden extra data som skickas.

Nyckelord

MQTT, MQTT-SN, AMQP, TCP, UDP, paketförlust, header data, overhead, publish-subscribe

Acknowledgments

I would like to thank Professor Gerald Q. Maguire Jr. for having written the draft thesis template to help bootstrap the writing process and for having a very helpful thesis webpage as well as providing valuable feedback which has helped immensely with the quality of the thesis.

I would also like to thank Håkan Nilsson and Ken Nordström for being very helpful supervisors that were always willing to take time to answer any questions I might have had.

My working college for this thesis project was Karl Strihagen whose thesis has the working title "Evaluation of publish/subscribe protocols".

Finally I would like to thank Scania for providing me with the opportunity to do this thesis project.

Stockholm, June 2016
Tomas Wickman

Table of contents

Abstract i
Keywords i
Sammanfattning iii
Nyckelord iii
Acknowledgments v
Table of contents vii
List of Figures xi
List of Tables xiii
List of acronyms and abbreviations xv
1 Introduction 1
1.1 Background 1
1.2 Problem definition 2
1.3 Purpose 3
1.4 Goals 3
1.5 Research Methodology 3
1.6 Network assumptions 3
1.7 Delimitations 4
1.8 Structure of the thesis 4
2 Background 5
2.1 Guarantee of delivery 5
2.1.1 At most once delivery 5
2.1.2 At least once delivery 6
2.1.3 Exactly once delivery 6
2.2 Underlying protocols 7
2.2.1 UDP 7
2.2.2 TCP 7
2.2.3 SCTP 11
2.3 Design paradigms 12
2.3.1 Publish/Subscribe 12
2.3.2 Request/Response 13
2.4 SCPv2 13
2.4.1 Protocol background 14
2.4.2 Requirement specifications 14
2.5 MQTT 14
2.5.1 Methods 15
2.5.2 Quality of service 15
2.5.3 Durable subscriptions 15
2.5.4 Keep alive packets 15
2.5.5 Last will 16
2.5.6 MQTT Messaging overhead 16
2.6 MQTT-SN 16
2.6.1 Broker gateways 17
2.6.2 MQTT-SN Messaging overhead 18

2.6.3 Topic ID registration 18
2.6.4 Keep alive message 18
2.7 CoAP 18
2.8 AMQP 19
2.8.1 AMQP Messaging overhead 20
2.8.2 AMQP v1.0 controversy 21
2.9 Related work 21
2.9.1 Broker comparisons 21
2.9.2 Comparison of MQTT and AMQP 22
2.9.3 Comparison of MQTT and CoAP 22
2.9.4 Fast and Secure Protocol (FASP) 23
2.9.5 Monitoring of other types of vehicles 23
2.9.6 SCPv2 Transmission data 23
2.10 Summary 25
3 Methodology 27
3.1 Research Process 27
3.2 Experimental Setup 27
3.2.1 Artificial Environment 27
3.3 Data Collection 30
3.3.1 Tests performed 30
3.3.2 Logging 30
3.3.3 Sample Size 31
3.3.4 Note about QoS 31
3.4 Experimental design 31
3.4.1 Test bed 31
3.5 Assessing the reliability and validity of the data to be collected 32
3.5.1 Reliability 32
3.5.2 Validity 33
4 Test setup 35
4.1 Extra header data 35
4.2 Publisher applications 36
4.2.1 AMQP publisher application 36
4.2.2 MQTT publisher application 37
4.2.3 MQTT-SN publisher application 38
4.3 Automated scripting 39
5 Analysis 41
5.1 Major results 41
5.1.1 100B payload results 41
5.1.2 1kB payload results 44
5.1.3 10kB payload results 46
5.2 Reliability and validity 48
5.3 Discussion 48
6 Conclusions and Future work 51

6.1 Conclusions 51
6.2 Limitations 52
6.3 Future work 53
6.3.1 In vehicle testing 53
6.3.2 More protocols and brokers 53
6.3.3 Encryption 53
6.3.4 Scalability tests 54
6.3.5 IPv6 and header compression 54
6.4 Required reflections 54
References 55
Appendix A: Port configurations 59
Appendix B: Automated Scripts 67

List of Figures

Figure 1-1: System Overview 2
Figure 2-1: At most once delivery 5
Figure 2-2: At least once delivery 6
Figure 2-3: Exactly once delivery 7
Figure 2-4: UDP header 7
Figure 2-5: TCP header 8
Figure 2-6: Slow start congestion window adjustment; points represent received ACKs, adapted from [10] 9
Figure 2-7: Fast Recovery Retransmission 9
Figure 2-8: CUBIC growth graph 10
Figure 2-9: SCTP Chunk layout 11
Figure 2-10: SCTP Common Header 11
Figure 2-11: Publish/Subscribe messaging pattern using a broker 12
Figure 2-12: MQTT fixed header 16
Figure 2-13: MQTT Publish variable header 16
Figure 2-14: Transparent and aggregate gateway, adapted from Figure 5 of [21, p. 5] 17
Figure 2-15: MQTT-SN header, adapted from [5, p. 7] 18
Figure 2-16: AMQP General frame format, adapted from [24, p. 33] 20
Figure 2-17: Method frame, adapted from [24, p. 34] 20
Figure 2-18: Content Header and Body frames adapted from [24, p. 36] 21
Figure 3-1: Artificial test environment 28
Figure 3-2: Testbed overview 32
Figure 4-1: Total amount of header data for each packet when transmitting a single byte of application layer payload 35
Figure 5-1: Total transmitted *data* for 1MB using 100B payloads and QoS 1 42
Figure 5-2: Number of packets transmitted from client to server, 1MB data, 100B payload, QoS 1 42
Figure 5-3: Total number of packets from server to client 43
Figure 5-4: AMQP sample traffic 43
Figure 5-5: AMQP TCP ACK flags 44
Figure 5-6: Total transmitted data for 1MB using 1kB payloads and QoS 1 45
Figure 5-7: Number of packets transmitted from client to server, 1MB data, 1kB payload, QoS 1 45
Figure 5-8: Number of packets transmitted from server to client, 1MB data, 1kB payload, QoS 1 46
Figure 5-9: Total transmitted data for 1MB using 1kB payloads and QoS 1 47
Figure 5-10: Execution time for 1MB data, 10kB payload, QoS 1 47
Figure 6-1: Execution time for 1MB data, 1kB payload, QoS 1 with 10 ms retransmission time for MQTT-SN 52

List of Tables

Table 1-1: Assumptions about network between vehicle and server gateways....4
Table 2-1: Traffic volumes for Telefonica, April 2016....24
Table 2-2: Sending rates and message sizes as specified by Scania....24
Table 3-1: Desktop computer specifications....29
Table 3-2: Virtual machine specifications....29
Table 3-3: Summary of protocols, packet loss rates, and payload sizes to be used for testing....31

List of acronyms and abbreviations

ACK	Acknowledgement (In the context of networks)
AMQP	Advanced Message Queue Protocol
BIC	Binary Increase Congestion
CAN bus	Controller Area Network
CoAP	Constrained Application Protocol
cwnd	congestion window
DNS	Domain Name System
ECU	Electronic Control Unit
FASP	Fast and Secure Protocol
FIFO	First-in-first-out
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
MQTT	Message Queue Telemetry Transport
MQTT-SN	Message Queue Telemetry Transport – Sensor Networks
M2M	Machine to Machine
MTU	Maximum Transmission Unit
QoS	Quality of Service
OASIS	Organization for the Advancement of Structured Information Standards
OTA	Over the Air
REST	Representational State Transfer
RD	Remote Diagnostics
RTO	Retransmission TimeOut
SASL	Simple Authentication and Security Layer
SCPv2	Scania Communication Protocol version 2
SCTP	Stream Control Transmission Protocol
SMS	Short Message Protocol
SSH	Secure Shell
STS	Tachograph message
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Locator
VM	Virtual Machine
WSN	Wireless Sensor Network

1 Introduction

According to predictions made in a press release from 2014, Scania estimated that in 2015 it would have around 150 000-200 000 vehicles using its services [1]. According to a statement by Volkswagen, the current owner of Scania, Scania currently has approximately 170 000 connected vehicles sending data to Scania for processing and of these, 70 000 were added in 2015 [2]. If this growth continues, the number of Scania's connected vehicles in 2020 is estimated to be around 600 000. This creates a demand for a scalable solution that can cope with the increasing amount of data that these vehicles (in the aggregate) will generate. Additionally, it is expected that the amount of data per vehicle will also increase over the amount transferred by a vehicle today. Currently, a vehicle is estimated to transmit approximately 91 kB per operating hour (see Table 2-2 on page 24.). While currently huge amounts of data are not being transferred, the increase in number of connected trucks and the expected increase in the number of services offered combined with the cost of transmission means that there is a substantial financial benefit for Scania in reducing the aggregate amount of transmitted data. Additionally, there is the limitation of data caps put in place by the different network providers. This data cap typically limits each truck to 10 MB of data transfer per day. As a result, the lower the protocol overhead, the more of this 10 MB/day that will be available for other services.

1.1 Background

Scania currently employs their own proprietary communication protocol called Scania Communication Protocol version 2 (SCPv2). This protocol is used for all communication between vehicles and Scania's servers. The rapid growth in the number of connected vehicles has prompted Scania to investigate what alternative protocols are available and how they would perform in comparison to the current solution in order to ensure that they will be able to keep up with the increasing volume of traffic.

Figure 1-1 illustrates the current system. This figure gives an overview of how the communication is done. All of the sensors in a truck are connected via a Controller Area Network (CAN bus) for internal communication. The CAN busses are connected to a C200, or in newer trucks a C300, which resides in the cab of the truck. The C200/C300 is Scania's proprietary Electronic Controller Unit (ECU). This ECU acts as a communication gateway from the truck to Scania's servers, and to the truck from the servers. The use of a communication gateway decouples the internal and external communication protocols, thus making it easy to replace any of these communication protocols. This decoupling also allows different protocols to be used for different scenarios. The decoupling of the protocols also decouples the internal and external functionality offered by the ECU, further increasing the system's flexibility. Improving the communication between these gateways and Scania's server(s) through the use of new protocols will be the focus of this thesis.

With the current growth in the number of connected devices and the emergence of the so called Internet of Things (IoT), many protocols have been introduce that profile themselves as the best option for different kinds of connected devices. The two most prominent protocols are the Message Queue Telemetry Transport (MQTT) [3] protocol and the Advanced Message Queue Protocol (AMQP) [4]. Both of these protocols are being maintained by the Organization for the Advancement of Structured Information Standards (OASIS) consortium. There is also the MQTT-SN variant of the MQTT protocol which profiles itself as a communication protocol for resource constrained network connected devices [5]. Meanwhile, the Constrained Application Protocol (CoAP) has been standardized by the Internet Engineering Task Force (IETF) [6].

Each of these protocols has its own advantages and disadvantages and has been created with different use cases in mind. The result is that each protocol is more or less suitable for use for communication between Scania made vehicles and Scania's servers. This thesis project investigates how these alternative protocols behave with regards to data transmission overhead, particularly when packet loss increases.

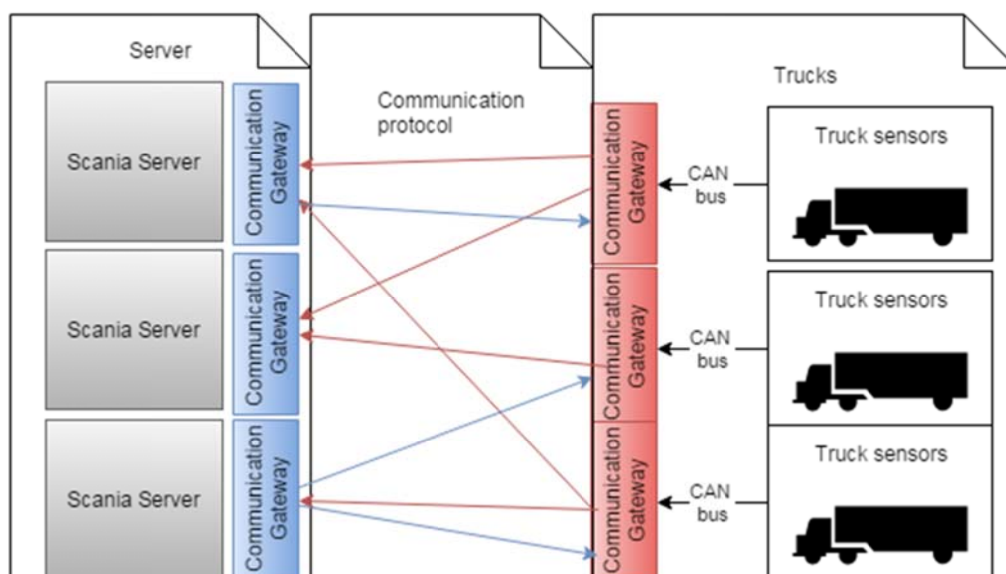

Figure 1-1: System Overview

1.2 Problem definition

The problem to be solved is: What is the best protocol with regards to Scania's desire to reduce the data transmission overhead due to the protocol used to communicate between vehicles and their servers. While Scania has expressed an interest in reducing the response time and increasing the transfer speed, their main focus was to reduce the amount of data that is transmitted. The main factors that increase the amount of data transmitted are the overhead of each protocol, the size of the payload, and how data transmission is affected due to retransmissions performed as a result of packet loss. As a result, the overhead and performance over a network with potentially high packet loss rates will be the criteria by which the alternative protocols will be evaluated.

Each protocol offers different features and levels of customizability, but the following requirements must be fulfilled by each protocol:

1. All communication must be secure; hence all of the payload data must be encrypted.

   Since all the investigated protocol operate on the application level Scania is free to encrypt the packets as they see fit. This thesis will not evaluate how the encryption of packets might affect the amount of data transmitted, but all protocols need to support encryption. A number of different encryption approaches will be discussed in Chapter 5.

2. The protocol must be able to guarantee a reliable at-least-once delivery (see Figure 2-2 on page 6).

3. The protocol must be able to handle files of up to 64 MB in size.

   This could also be supported either on a protocol level or on an application level and is thus not a strict protocol requirement. The problem with large files is that if the connection is lost during the transmission of the file, then the receiver and sender need to synchronize what portion(s) of the file were received and which were lost so that the entire file does not need to be re-sent when connectivity is re-established.

The protocols will be implemented and then evaluated in the context of Scania's communication platform. The test environment for this implementation will provide a framework in which we can vary those variables that impact the protocol's performance regarding data transmission, such as packet loss, payload size, and frequency that data is sent. An example of the most common messages and their frequencies is presented later in Table 2-2 on page 24.

1.3 Purpose

The purpose of this thesis project is to evaluate several different communication protocols in terms of how well they fit Scania's requirement regarding communication between their servers and vehicles they provide their services for. The results should allow Scania to make an informed decision about the advantages and disadvantages of alternative protocols when deciding to upgrade from their current SCPv2 protocol.

1.4 Goals

The goal of this thesis project is evaluate different communication protocols for communication between Scania's vehicles and server. This has been divided into the following two sub-goals:

1. Create a proof of concept message platform to allow communication between a virtual machine acting as server and a client using each of the investigated protocols. This will allow us to vary one variable (such as protocol used, percentage of packets dropped, and message size) at a time.
2. Evaluate each of the protocols on the basis of how they perform regarding overhead and how packet loss affects the amount of data transmitted.

1.5 Research Methodology

The original idea was that a prototype implementation of the protocol would be developed and then evaluated using a client application running inside a vehicle with a server application running on Scania's servers. However, the system that was first thought to be quite modular with regard to the protocol turned out to be quite interconnected. This meant that the SCPv2 protocol implementation could not be easily replaced by the new protocol implementation. The alternative was to have the SCPv2 protocol use the new protocol via a gateway. However, this would affect the validity of the experimental data since the evaluation would only reveal how the alternative protocols behave as a gateway protocol for SCPv2. Additionally, the SCPv2 protocol is quite large, as well as proprietary, so this would affect both the quality of the experimental data as well as the quality of the thesis.

For the above reasons, the approach used for testing was to set up a virtual environment with an emulated network connection (between the gateway in the vehicle and the gateway in front of the Scania server. This emulated network makes it possible to completely control the network's behaviour (in terms of delay, error rate, congestion, etc.), hence the protocols will be tested in this controlled environment. While this would seem to be an artificial environment which is quite far from the network conditions that the actual Scania trucks are subject to, it allows for a more exact reasoning about how different network conditions affect the protocols. The main parameter we will focus on emulating is packet loss, since packet loss can be directly controlled and it impacts the amount of data transmitted by the different protocols. This test environment enables us to draw reliable conclusions based on the experimental data about how the protocols behave as the packet loss rate varies.

1.6 Network assumptions

According to internal measurements by Telefonica, a GSM network provider for Scania, during the month of April 2016 the average packet retransmission rate was around 0-4% depending on how packet loss is counted. Based on the number of outgoing ACK/NACK from the server to the vehicle the packet loss will be around 4% NACK's, however a NACK could be due to a late packet or an indication of mismatched encryption keys and does not necessarily indicate packet loss. So if the number of packets sent from the c300 is instead compared to the number of ACK/NACKS sent from the server the packet loss rate is about one every ten millionth packet. In the end though any retransmission results in more traffic being sent over the network. So whether a retransmission is due to a late packet,

a packet loss or erroneous encryption keys does not matter for our experiments so the assumption will be that for the communication networks that Scania uses has a retransmission rate of about 4% on average. However, according to Scania the number of retransmissions is not the only network impairment that affects the communication between the vehicles and the servers. Another thing to consider is that trucks often operate in remote areas, such as mines, where the connection might be unstable or non-existent. Finally, the amount of data that can be transferred in a day is limited to 10 MB due to restrictions by the network provider. Table 1-1 summarizes the assumptions made about the network for the purposes of this thesis project. More about the SCPv2 background can be read in section 2.4.1.

Table 1-1: Assumptions about network between vehicle and server gateways

GPRS transmission
Unstable connection
High packet loss
Limited amount of data

1.7 Delimitations

There exists a multitude of protocols that are marketed as solutions for networks where limiting the amount of data transmitted is a priority. However, due to the limited time available for this project the most well-known and widely adopted protocols for connected devices have been chosen to focus on. This choice was made because extensive research and testing has already been done on these protocols and due to an expressed interest by Scania as to the potential for cloud integration. Specifically the MQTT protocol, its sensor network variant (MQTT-SN), and AMQP have been chosen. Two additional protocols were also investigated, but were not tested: CoAP and the Fast and Secure Protocol (FASP). This latter protocol was of particular interest since the transfer of larger files over unstable networks with high retransmission rates was of interest to Scania.

1.8 Structure of the thesis

Chapter 2 introduces the basic terminology and principles of the investigated network messaging protocols. It describes how the underlying protocols work and how the design choices of these protocols affect their performance. This will allow the reader to understand and follow the reasoning regarding the results presented later in the thesis.

Chapter 3 explains the experimental setup that will be used and how the protocols will be evaluated using this experimental setup. Chapter 4 describes how the experimental setup and each of the protocols were implemented. Chapter 5 looks at the experimental results and discusses them in terms of what these results might mean for Scania. Finally, Chapter 6 concludes the thesis and suggests future work.

2 Background

When evaluating each of the protocols there were three main properties that Scania is interested in: response time, transmission speed, and data transmission overhead. While all of these criteria are of interest, the main focus in this thesis project is the reduction of overhead. Moreover, the amount of overhead is directly related to the level of reliability that is desired, due to the additional transmissions that each additional level of guarantee entails. This chapter will discusses exactly how each different level of delivery guarantees affect the transmission of data.

This chapter also looks at the functionality of the underlying transport protocols (specifically, Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Stream Control Transmission Protocol (SCTP) and how the performance and reliability of protocols higher up in the protocol stack are affected by this functionality. Section 2.3 describes how the publish/subscribe paradigm and the request/response paradigm work. This is essential background knowledge since all the protocols to be considered follow one of these two paradigms.

The current Scania Communication Protocol SCPv2 will be briefly discussed in Section 2.4. Following this there will be section discussing each of the protocols that will be evaluated.

2.1 Guarantee of delivery

Guaranteeing reliable data delivery incurs additional overhead: for example, in the form of re-transmissions or using error correcting coding. A reliable protocol comes at the expense of response time and/or transfer rate, due to the overhead associated with ensuring reliability. The greater the level of reliability, the greater the data transmission overhead. When attempting to reliably deliver data, the protocol can achieve one of the following types of delivery guarantees: at most once delivery, at least once delivery, or exactly once delivery. Each of these is described below.

2.1.1 At most once delivery

At most once delivery, also known as “fire and forget”, ensures nothing in terms of actually delivering data, unless the underlying transport protocol, such a TCP or SCTP, provides a guarantee of reliability. At most once delivery simply sends data to the receiver and hopes for the best (see Figure 2-1). If the receiver is able to successfully receive this data, then this method delivers data quickly since there was no need for any additional overhead. Moreover, there is no need for the sender to wait for an acknowledgement (ACK) that data has been received. However, if the receiver is unavailable or the data is lost along the way, then we have no way of knowing that our data never reached its destination.

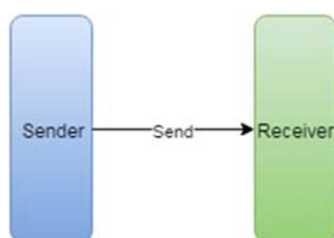

Figure 2-1: At most once delivery

“At most once” delivery could be a problem if the communication takes place over unreliable links, since data would be lost and there is no way of knowing which data was lost or even how much data was lost.

2.1.2 At least once delivery

In order to ensure "at least once" delivery there needs to be some way of knowing that the data has reached the destination at least once. This is usually done by the receiver sending an ACK to the sender to notify it that the data was received (as shown in Figure 2-2). The ACK process differs between protocols, as some protocols may send an ACK for every unit of data received, while others may send cumulative or selective ACKs.

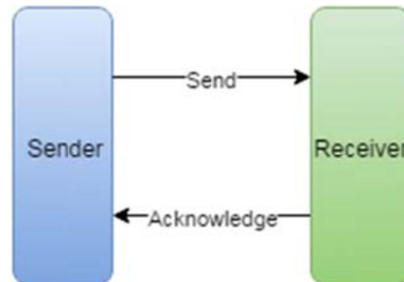

Figure 2-2: At least once delivery

There is also the question about how to determine that data was not delivered. A common solution is to have a timeout starting when the data was sent until when it is resent unless an ACK has been be received. When the timeout occurs, the sender resends the data and restarts the timer, optionally increasing the waiting time with each missed ACK.

It is important to note that the ACKs can be acknowledgements of sequential bytes, messages, or datagrams. In our later discussion of each of the protocols, we will specifically note what the ACK refers to.

2.1.3 Exactly once delivery

The highest level of reliability ensures that data is delivered exactly once. Ensuring that data is delivered exactly once requires that we deliver the data to the receiver (this can be done via a combination of timeouts and retransmissions) and that the receiver can know if it has successfully received this data previously (if so it can discard the duplicate data).

To enforce this exactly once message delivery semantics both MQTT and AMQP use a kind of "double send" with ID-tagged messages to keep track of which messages have already been received in order to filter out duplicate messages. This is a variation of "Strategy 3" described in [7]. This double send is similar to a two-phase commit in databases.

This double send (shown in Figure 2-3) works as follows:

1. When the sender sends a message it stores the message and its associated message-ID.
2. If the receiver receives the message, then it stores the message-ID and sends a received message back to the sender containing this message-ID.
3. The sender responds with a release message to the receiver which tells the receiver that the received message was seen and that the stored message-ID can be released.
4. The receiver releases the message-ID and deletes its copy of the stored message, then it sends a transfer completed message to the sender to indicate that the message transfer was successfully completed.

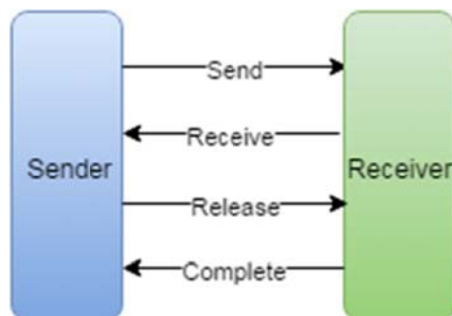

Figure 2-3: Exactly once delivery

2.2 Underlying protocols

Due to specifications from Scania all of protocols need to run on top of the Internet Protocol (IP). For our investigations we will perform all our tests using IPv4. We could have chosen either IPv4 or IPv6, but after discussing with Scania they had no plans for moving to IPv6 yet - although this transition will happen sooner or later. It should also be noted that even if the larger IPv6 header had been chosen it would not affect the comparison of the protocols since all of the protocols include an IP header. We will consider three different transport protocols: UDP, TCP, and SCTP. We will examine what effect each of these protocols has on the later tests in terms of data transmission overhead.

2.2.1 UDP

UPD is a best effort delivery, connectionless protocol that offers no guarantees regarding delivery [8]. The header of a UDP packet consists of a source port, a destination port, a length, and an optional checksum each consisting of 16 bits as shown in Figure 2-4.

Figure 2-4: UDP header

The main appeal of the UDP protocol is its simplicity. Since it includes very little additional information, apart from the data we want to send, it allows us to send a lot of data very quickly and with low overhead. The drawback is that we will have to provide reliability ourselves at a higher layer. However, since we can chose exactly what features we want to include, a UDP based solution has the potential to be optimized for a specific use case. For example, UDP is used by FASP, as described in Section 2.9.4.

2.2.2 TCP

TCP provides an ordered, reliable byte stream between two hosts. Using TCP we get a lot more features than when using the UDP protocol, but it comes at the cost of increased message size. The TCP header is shown in Figure 2-5.

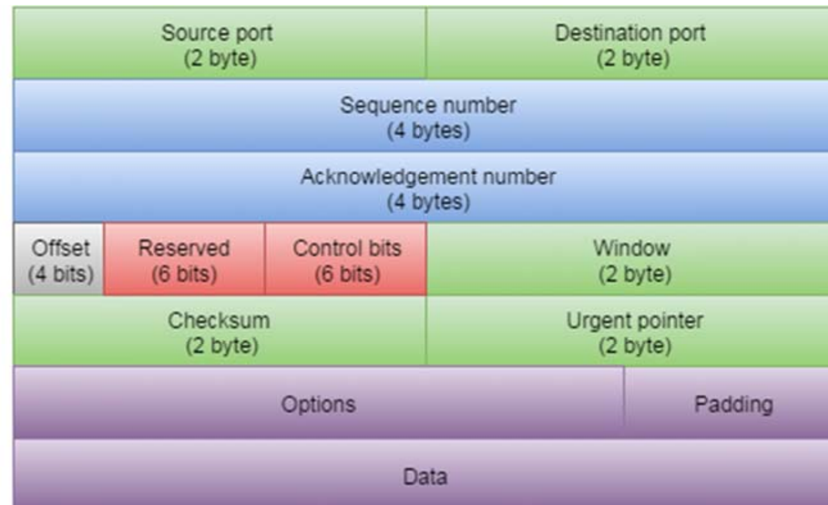

Figure 2-5: TCP header

The minimal sized, i.e., without any options, TCP header is 20 bytes. The following aspects are worth considering when using TCP in the context of this project:

1. Connection oriented protocol

   Since TCP is a connection based byte oriented protocol the bytes are ordered using a sequence number, hence the sender and receiver need to synchronize their respective randomly selected sending byte sequence numbers when a TCP connection is initialized. This is done through a three-way handshake. This handshake is designed to reduce the risk of false connections [9], that is, that someone sends a packet with the correct sequence number, but with a malicious payload. However, this three-way handshake is vulnerable to a so-called "SYN" attack, where false initial synchronization segments are set to deny service to legitimate users or to cause resource exhaustion of the device receiving these SYN segments.

2. Retransmission of missing bytes

   When TCP transmits a TCP segment it stores the segment and waits for a cumulative ACK from the receiver. This cumulative ACK indicates that this data and all of the proceeding data has been received. However, reliable ordered byte delivery is undesirable for sensor data, as it does not matter whether a previous sensor value was lost when the sensor has now reported its current value. For example, an old pressure sensor report has little value if there is a newer value. Unfortunately, TCP will wait until all previous bytes have been delivered before delivering the new data. This problem is called "head of queue blocking" and could have undesired consequences in the context of this thesis project.

3. Congestion control

   To avoid overloading the network TCP employs a congestion control algorithm together with a congestion window (cwnd). This algorithm has several phases and starts out in the "slow start phase" (as shown in Figure 2-6). In this phase the size of the congestion window is doubled on receiving each ACK. Then, either it hits a threshold which causes it to go into a congestion avoidance mode, which utilizes a linear increase of the congestion window on each ACK* until it loses a segment, or it loses a segment before reaching this threshold. In either case, when a segment loss is detected, then the congestion control threshold will be cut to half the current threshold and the congestion window will be reset and the algorithm starts over. This may result in a TCP session almost never operating at full capacity.

* The increase is by 1/cwnd + 1/8 of segment size each time an ACK is received.

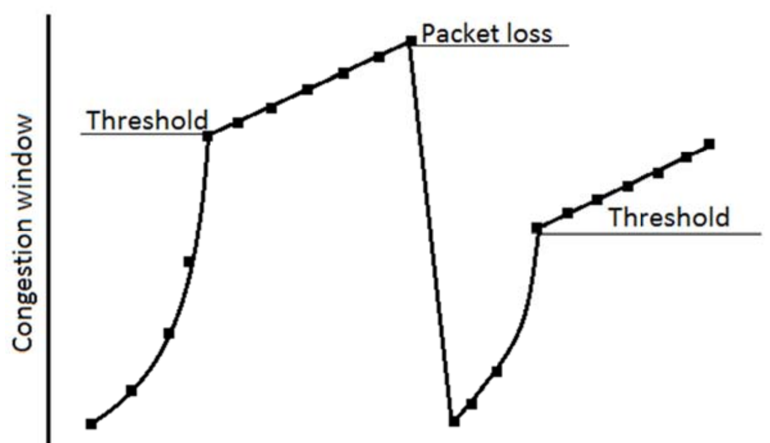

Figure 2-6: Slow start congestion window adjustment; points represent received ACKs, adapted from [10]

Another mechanism for congestion control is called "Fast retransmission recovery" (shown in Figure 2-7). TCP decides whether a segment was lost is by keeping a timer for each segment, and if the receiver does not respond with an ACK in time, then the segment is assumed to be lost. If this happens, then the congestion windows will be reset and the threshold will be halved – however, this is undesirable if the segment was lost for a reason other than congestion.

TCP avoids reducing the threshold when a segment is lost and the receiver is continuing to successfully receive other segments. In this case, the receiver will respond with an ACK indicating the sequence number of the next byte that it is expecting and it will continue to send this same value for each new segment it receives.

If the sender does not receive an ACK for a segment within a specified time, but has received three ACKs indicating that the receiver is still waiting for the same numbered next byte, the sender knows that this segment was lost for *a reason other than congestion*, so it immediately re-sends the missing segment and does **not** reduce the congestion threshold.

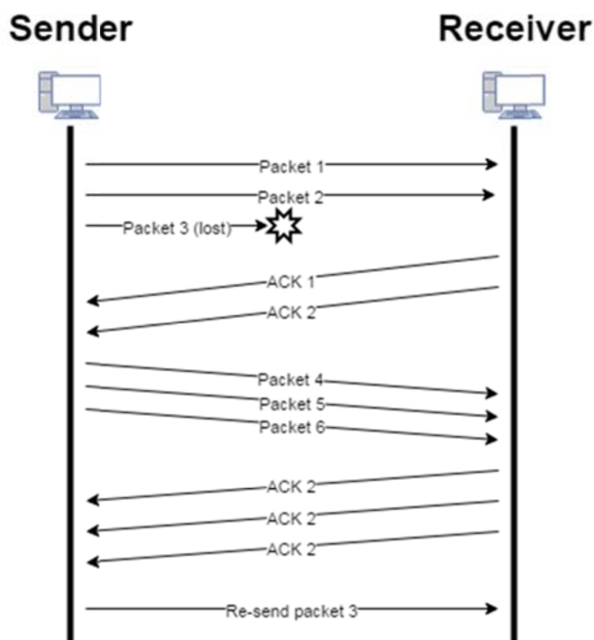

Figure 2-7: Fast Recovery Retransmission

2.2.2.1 CUBIC congestion control algorithm

There have been multiple improvements to how TCP handles congestion by using smarter congestion control algorithms. The algorithm currently used in the C300 is the CUBIC congestion control algorithm [11], an improvement of the Binary Increase Congestion (BIC) algorithm [12]. CUBIC will be the congestion control tested during all of our experiments.

The aim of CUBIC is to maintain the stability and scalability of the BIC algorithm, while simplifying the congestion window control function and increasing TCP's friendliness to other protocols. CUBIC does this by replacing the complex three-phase growth function of BIC with a simpler, two-phase cubic growth function.

In the CUBIC algorithm the algorithm has two phases when adjusting the congestion window: (1) Steady State Behaviour and (2) Probing. In the Steady State Behaviour phase the algorithm quickly increases the congestion window until it starts to approach the window size threshold (W_{max}), then it starts to rapidly decrease the growth of the congestion window. This continues until the growth reaches zero, after which the algorithm enters the second phase, i.e., the Probing phase. During the probing phase it once again starts to accelerate the congestion window growth as it moves away from W_{max}.

One of the improvements over the BIC algorithm is that CUBIC replaces multitude functions in the BIC algorithm with the CUBIC growth function described in Equation **1** :

$$W_{cubic} = C(T - K)^3 + W_{max}$$

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}}$$

Equation 1 CUBIC growth function [11, pp. 4, fig 1.2]

Here C is a constant scaling factor and T is the elapsed time since the algorithm started. When plotted, the graph of the congestion window increases as a function of time and will look similar to that shown in Figure 2-8.

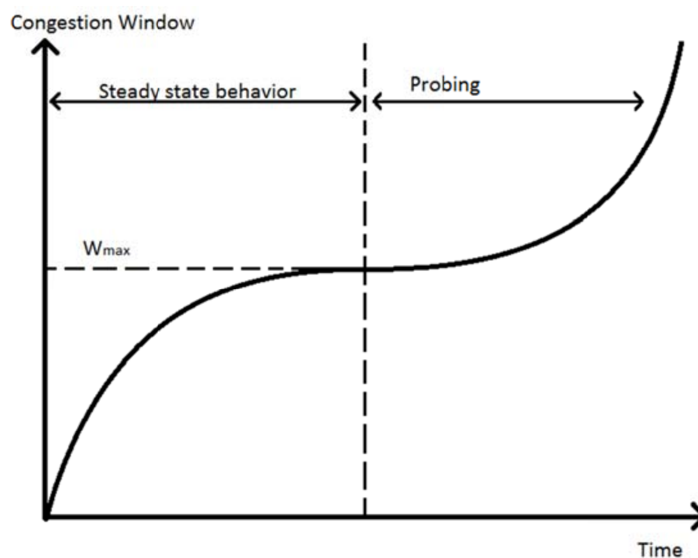

Figure 2-8: CUBIC growth graph

2.2.3 SCTP

While not utilized in any of the investigated protocols it could still be worth to look at the Stream Control Transmission protocol. Unlike UDP which works in terms of datagrams and TCP which works in terms of delivering ordered bytes, SCTP operates on messages and was designed to support multiple streams – hence avoiding head of queue blocking. SCTP supports multi-homed hosts (so that an SCTP host and client can automatically fail-over to an alternate network address if there is a problem). Additionally, SCTP supports selective acknowledgements – so that it can efficiently deal with burst errors. SCTP was designed to be used to reliably deliver messages, for example for Signalling System 7.

Figure 2-9: SCTP Chunk layout

Figure 2-10: SCTP Common Header

2.2.3.1 SCTP vs TCP and UDP

An SCTP packet consists of a common header seen in Figure 2-10 which contains the source and destination port as well as a verification tag and a check sum. After the header the protocol can transmit multiple chunks seen in Figure 2-9 which contains the actual payload as well as a type field, flags and a length field.

A list of differences between the SCTP protocol and the TCP and UDP protocol can be seen on [13] where the most relevant for this case is the SCTP avoidance of head-of-queue blocking, message oriented communication, multihoming and SYN cookies.

1. Multihoming

   For SCTP, multihoming means that the sender and receiver can specify multiple endpoints of attachment. This means that if one of the endpoints fail another one can be utilized to deliver the messages. Since this works on a transport level the application level never needs to know a different route was even used.

2. Message oriented communication

   Unlike TCP but like UDP , SCTP is a message oriented protocol. This means that if SCTP transmits X number of bytes, then the receiving application can read X number of bytes in

one read. While UDP also offers this type of transmission, it does so with no delivery guarantees. SCTP on the other hand has a guarantee of delivery.

3. Head-of-queue blocking avoidance

   TCP guarantees that the packets are reliably delivered in the order that they were sent. SCTP also offers delivery guarantees but with no guarantees about ordering. This allows it to avoid the head-of-queue blocking that can occur for TCP were the transmission has to wait for the last transmission to be ACK'd before being able to transmit the next set of bytes.

4. Four way handshake

   To establish a new connection SCTP performs a four way handshake with a signed cookie were the receiver of the connection will not reserve any resources for the incoming connection unless it can prove that it is the IP address from which it is trying to initiate a connection. This prevents a common attack against TCP known as SYN flooding [14]

2.3 Design paradigms

Two common communication patterns that are widely used today are publish/subscribe and request/response. Each of these will be described below.

2.3.1 Publish/Subscribe

In the publish/subscribe pattern senders assume the role of publisher and receivers act as subscribers. Between the publisher and the subscriber is often a broker who acts as a middleman between the publisher and the subscriber (see Figure 2-11). The subscriber subscribes for a certain topic either directly to the publisher or to the broker. A topic is a tag that the publisher attaches to its messages when it sends them. When a publisher publishes a message to the broker, the broker will forward this message to all of the subscribers who have subscribed to this topic.

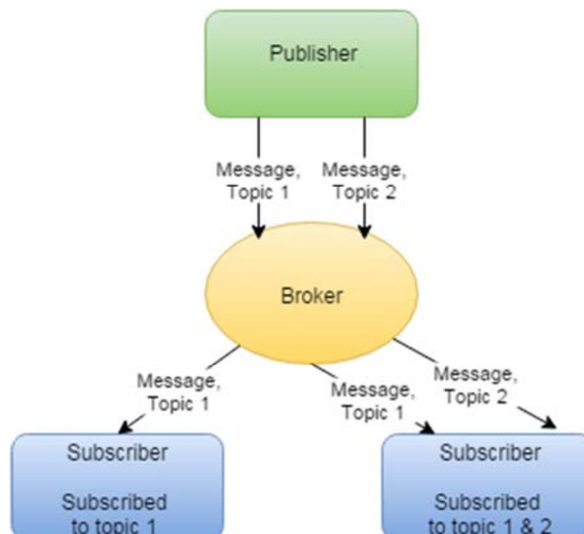

Figure 2-11: Publish/Subscribe messaging pattern using a broker

The publish/subscribe pattern has two major advantages over a traditional client/server design for disseminating data.

First, the publisher and subscriber are loosely coupled with regards to space, time, and synchronization [15]. This means that the publisher does not need to know who is listening, the subscriber and publisher do not need to be active at the same time and the publisher is not blocked while producing a new event. This is in contrast to a traditional server/client solution where the sender and receiver are tightly coupled and the server cannot send if the client is not receiving. This makes it ideal for disseminating data to subscribers via an unreliable and/or intermittent network connectivity where providing a guarantee that the other party is listening can be difficult.

Second, publish/subscribe is scalable due to its tree-based structure, since if the system needs to be scaled up more brokers can be added and the topics split between them, thus reducing the load on individual brokers.

However, there are disadvantages to using the publish/subscribe paradigm. One is that, even though it is loosely coupled regarding time, space, and synchronization; it is tightly coupled with regards to the published data. If a change is to be made in how the published data is represented, then the programmer will need to modify all the subscribers. Souleiman Hasan, Sean O'Riain, and Edward Curry are researching how to achieve decoupling with regards to data semantics by using approximate matchings [16].

2.3.2 Request/Response

The request/response (or request-reply) paradigm is a basic communication method between pairs of computers. It works by having one computer send a request to another computer. When the second computer receives the request it sends a response. The most well-known application of this design is the HTTP protocol in which a client requests a webpage from a server and the server responds with the requested document.

This paradigm is flexible in that it only requires a point-to-point communication channel over which the client sends a request to the server and the server responds to that client. However, HTTP could also use the publish/subscribe paradigm if requests and responses are sent to all interested parties [17].

The request/response model supports two modes for when the client waits for a response from the server. The first mode uses synchronous blocking, thus the sender sends a request to the server and then waits for a response. This allows a simple implementation, but can become problematic - since if a client crashes it will be problematic to re-establish the blocked thread*.

The other mode uses an asynchronous call-back where the client sends requests to the server and then sets up call-backs for the server. The client keeps one thread listening for reply messages from the server and one thread to serve the responses. This way, when a reply is received, the response invokes the call-back that was previously established. A client that uses this approach can easily recover if it were to crash by simply restarting the reply thread and continuing. This makes the asynchronous call-back more suitable than the synchronous approach when there is a need to be resilient to crashes; for example, when operating in a remote environment where a reset of the computer might be difficult to perform.

2.4 SCPv2

SCPv2 is the current communication protocol deployed by Scania for communication between the gateway in vehicles and the gateways at their server. A good understanding of the requirements of this protocol will be beneficial when comparing how other protocols perform in comparison with it.

* Note that protocols such as QUIC re-establish a communication session using cryptograhic means. (https://www.chromium.org/quic)

Unfortunately, this protocol is proprietary, hence the protocol description will be brief (and lack any specific details).

However, there is a specification that Scania wants the new protocol to fulfil (see Section 2.4.2). Each of the protocols will be evaluated against this list of requirements to ensure that there no required feature is missing.

2.4.1 Protocol background

The original Scania Communication Protocol version one only supported sending data via Global System for Mobile Communication (GSM) using the Short Message Service (SMS) protocol. SMS supports messages of up to 160 characters (including header and error detection code). However, as the amount of data being transferred between the vehicle and server grew, the need for a new protocol became evident. The two major requirements for version two of this protocol were:

1. Send a large amount of data in a single UDP packet to achieve low communication cost and
2. The choice of communication infrastructure should be *independent* of the data being sent from one application to another.

In the second version of the protocol, SMS was only used to wake up the system after which the vehicle gateway connected to one of Scania's Domain Name System (DNS) Servers to obtain an IP address of one of Scania's servers. Subsequently all communication to/from this server was done over UDP using General Packet Radio Service (GPRS).

2.4.2 Requirement specifications

Scania's requirements for version two of the protocol and all subsequent implementations are:

1. All communication must be done over a secure encrypted connection.
2. The protocol should be able to handle files of up to 64MB in size. This includes being able to resume transmission of such a large file should connectivity be lost.
3. The protocol shall guarantee at-least-once delivery.

For each protocol that will be evaluated in this thesis these requirements will be verified. It should be noted that except for requirement 3, these requirements are not necessarily provided by the protocol itself – as the functionality could be provided by a lower layer transport protocol or the application layer; however, while not a metric for our evaluation, the discussion will talk about how difficult it was to set up each of the protocols along with its respective broker.

2.5 MQTT

MQTT is a publish/subscribe protocol running on top of TCP and was originally developed in 1999. In 2013, it was turned over to the OASIS organization. The current OASIS standard version of MQTT is 3.1.1 and this version was approved on the 29th of October 2014 [18].

MQTT was designed to be a light-weight, open, simple, and easy to implement protocol which would make it ideal for use in the context of Machine to Machine (M2M) communication and IoT [3]. MQTT has been deployed for a number of real world applications. These applications, most notably Facebook messenger and Amazon Web Services IoT, show that MQTT delivers on its promises [19, 20].

2.5.1 Methods

In accordance with MQTT's aim to be easy to implement, the number of methods defined by MQTT for interacting with a specific resource are only five*:

Connect	After a network connection has been established between a client and a server the first message must be a CONNECT packet that sends identifying information about the client to the server.
Keep alive	Specifies how long the client can go without publishing a message before being disconnected. Set on connect.
Disconnect	Waits for the client to finish its work and for the TCP session to terminate.
Subscribe	Sends a SUBSCRIBE message from the client to the server to create subscriptions for one or more topics.
Unsubscribe	Sends a UNSUBSCRIBE message from the client to the server to unsubscribe from one or more topics.
Publish	Sends a PUBLISH message from client to server or from server to client to transport a message.

2.5.2 Quality of service

MQTT allows the user to specify different Quality of Service (QoS) levels depending on what is needed. The protocol defines three QoS levels that correspond to three different levels of delivery guarantees. Sending a message with a QoS of zero will result in a message with "at most once" delivery (there is no follow up of the sent message). A message sent with a QoS of one will result in a message with an "at least once" delivery guarantee with an ACK being sent for each message delivered. Finally, a QoS level of two will result in "exactly once" delivery. To offer actual "exactly once" delivery MQTT uses the message ID to filter out duplicate messages as discussed in [7] and illustrated in Figure 2-3 on page 7.

2.5.3 Durable subscriptions

It is possible for the client to specify when it connects whether its connection is a durable or a non-durable connection by setting a "Clean session" flag in the CONNECT message (setting the flag to false will enable a durable connection). If the client chooses to use a durable connection, then the broker will store undelivered messages if the client disconnects and the broker will subsequently try to deliver these saved messages as soon as the client connects again.

In contrast, for a non-durable connection the lifetime of the subscription is limited to the time the client is connected to the broker

2.5.4 Keep alive packets

When a client connects to a server it specifies a Keep Alive value. This is a 16-bit value that specifies, in seconds, how long a client can go without either publishing a message to the server or sending a PING request. If the client does not do one of these operations within one and a half times the Keep Alive value, then the server will consider the client disconnected and remove all of its subscriptions.

* The client-server terminology used is the same as specified in the MQTT 3.1.1 Specification under "Terminology" [3].

2.5.5 Last will

The client may also specify a Last Will when it connects. This consists of topic string followed by a 2-byte length-field defining the length of the message followed by the actual message. If the client has not published any message during one and a half times the Keep Alive time, then the server will automatically disconnect the client and publish its last will message on the specified topic.

2.5.6 MQTT Messaging overhead

MQTT messages have two header parts, one fixed part that is necessary for all messages and one variable part that varies depending on the message being sent.

The fixed part consists of two bytes and specifies the type of message being sent, whether it should be kept (in memory) until acknowledged, what the QoS should be for the message, and if the message should be retained after it has been delivered to its recipients. The remaining length field indicates how many octets remain of the message which includes the variable header and the payload. This fixed header is shown in Figure 2-12.

Figure 2-12: MQTT fixed header

For our tests we are mainly interested in the variable header for the publish message and the publish ACK message since these are the messages that will most frequently be sent. The variable header for the publish message contains a topic name as well as a message identifier (see Figure 2-13). The length field in the header consists of two bytes and indicates the number of octets that is required to represent the topic name that follows the two bytes. So the shorter the topic name the fewer bytes needed for each published message. The topic name is then followed by a two octet message identifier. If QoS 2 is used, there is one additional packet that will be sent a lot: the published REC packet to realize the at least-once guarantee. This packet looks identical to the published ACK message. The publish ACK message contains no additional fields beyond the fixed header besides a two octet field to indicate what message is acknowledged.

Figure 2-13: MQTT Publish variable header

2.6 MQTT-SN

MQTT-SN* is a variant of the MQTT protocol that aims to be more optimized for sensor networks through a series of changes, most notably that the underlying transport protocol is UDP rather than TCP. In 2008, Hunkeler, Truong, and Stanford-Clark listed the following design points that MQTT-SN was developed to address [21]:

1. MQTT-SN should be as close to MQTT as possible

   This is so that it requires minimal integration effort to use the MQTT-SN protocol with a pre-existing MQTT broker. As a result, the MQTT-SN protocol supports almost all functionality that MQTT supports.

* Originally, this was named MQTT-S, but was renamed to avoid any confusion that the S stood for "Security" [46].

2. Designed for resource constrained devices

   The protocol is designed to be able to run on very resource constrained devices, and whenever any complex processing needs to be done it should be done by the gateway or the broker to minimize the client side load as much as possible.

3. Designed for wireless networks

   Wireless networks are expected to have higher failure rates, lower transmission rates, and shorter maximum message payload sizes than fixed networks. For example, the IEEE 802.15.4 wireless standard provides a maximum of 128 bytes for each frame at the physical layer [21]. This includes header data for the network protocol, MAC address, security, etc., so using UDP instead of TCP means we could reduce the header overhead of each packet significantly. It should be noted that there are header compression algorithms that can be deployed to reduce the amount of header data transmitted. For example Robust Header Compression (ROHC/ROHCv2) [22, 23] for IPv4 and 6LoWPAN for IPv6 [24]. In this thesis however no compression algorithms will be investigated.

2.6.1 Broker gateways

To allow for easy integration with existing MQTT brokers, MQTT-SN utilizes gateways between the broker and the client. This gateway usually runs on the same platform as the broker and translates the incoming MQTT-SN messages from the clients to MQTT just before passing them on to the broker. This can be done in two different ways [21, p. 5] (as shown in Figure 2-14):

1. Transparent gateway

   The transparent gateway architecture connects each client to the gateway with an MQTT-SN connection. Then for each MQTT-SN connection the gateway maintains a MQTT connection to the broker.

   While this offers a straight-forward implementation where each MQTT-SN message is simply translated to a MQTT message, the number of MQTT connections that have to be maintained between the broker and the gateway creates scalability issues when a large number of clients are simultaneously connected.

2. Aggregate gateway

   In contrast, an aggregate gateway maintains one MQTT connection between the gateway and the broker. While the implementation of such a gateway is a bit more complex than a transparent gateway, the benefit is that this approach scales a lot better - since the gateway does not have to support as many MQTT connections to the broker.

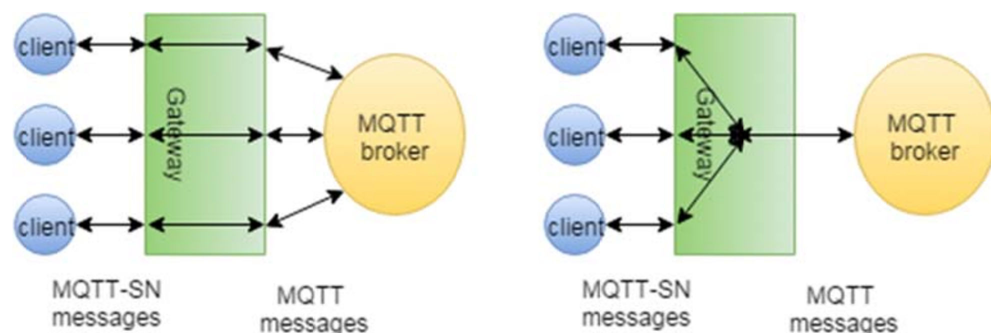

Figure 2-14: Transparent and aggregate gateway, adapted from Figure 5 of [21, p. 5]

3. Multiple gateway support

   Since MQTT-SN is designed to work well over wireless networks where connections are unreliable and have high link failure rates, MQTT-SN supports maintaining multiple connections via different gateways in order to have a fall back connection in case of failure.

This also allows the clients to spread their network traffic over multiple connections and thus avoids congestion in the gateway.

Since the gateway will translate the MQTT-SN message to a MQTT message the gateway will always operate on the message just before they are passed to the broker. This is so that we avoid transmitting the larger MQTT messages and only use the MQTT format prior to the broker where the increased size does not matter (as much).

2.6.2 MQTT-SN Messaging overhead

By design, the messaging overhead of MQTT-SN is very small. The header can consist of either 2 or 4 octets depending on how large the message. This header is shown in Figure 2-15.

Figure 2-15: MQTT-SN header, adapted from [5, p. 7]

The reason for the variable length is that MQTT-SN allows the use of 1 octet for the length if the payload size is less than 256 octets. However, for use cases the payloads will not use the shorter message length, thus the header size can be considered to always be 4 octets. The message type simply describes what type of message it is according to the protocol specification [5, p. 7].

2.6.3 Topic ID registration

One of the things that MQTT-SN does to reduce network traffic compared to the other protocols is to use a 2 octet Topic ID instead of a plain text topic string. The topic string is only used during the login process where the client sends a registration packet with the desired topic as a string. The broker then responds with a 2 octet ID number that is subsequently used to identify that topic. This way it is possible to go from a 31 octet topic string to a 2 octet topic ID. To use the topic ID instead of a topic string the sender has two options. Either performs a registration negotiation where it sends its topic string to the broker and the broker responds with the corresponding topic ID. The other way is to use pre-defined topic IDs where the broker and publisher both have a set of pre-defined topic IDs. For our tests we will only use pre-defined topic IDs since it is not unreasonable to assume that for the services Scania will be using the topic IDs will be known before transmission starts.

2.6.4 Keep alive message

Since MQTT-SN is a connectionless protocol it needs some way to keep track of whether the party receiving the messages is still there when using a QoS level of 1 or higher. Normally the receiver would respond with an ACK to tell the sender that the message was successfully delivered; however, when there is packet loss even this ACK may be lost. The way MQTT-SN solves this is by keeping a keep-alive timer that tells the sender how long it should wait before sending a keep-alive ping. The receiver of the ping will then respond with a ping ACK and then the transmissions may resume.

2.7 CoAP

CoAP is a proposed IETF standard and aims to provide a request/response protocol that allows very simple devices to communicate over the Internet [6]. CoAP is designed to allow nodes to easily be controlled via standard internet applications. This is achieved by utilizing the Representational State Transfer (REST) architectural style of the web.

Similar to HTTP, CoAP accesses content via a Uniform Resource Locator (URI) using the GET, POST, PUT, and DELETE methods. CoAP is designed to easily translate to HTTP. There are even guidelines from IETF on how to map CoAP to HTTP [25]. However, although CoAP is designed to resemble HTTP in many ways, it differs in many others due to the requirements that a constrained device puts on it.

The biggest difference between HTTP and CoAP is that CoAP is a UDP based protocol. This is to avoid the overhead that TCP entails (which a constrained device wants to avoid).

CoAP offers two options regarding delivery guarantees. These are "at-most once delivery" and "at-least once" delivery. These are referred to as "non-confirmable" messages and "confirmable" messages in the CoAP specification. Confirmation requires the receiver of a message to send an ACK when it receives the message.

2.8 AMQP

AMQP is a protocol that communicates via publish/subscribe and operates on top of TCP. It was originally developed in 2003 by John O'Hara at JPMorgan Chase and iMatix with aim of creating an interoperable message system that was non-proprietary and could be used as a standard messaging protocol for investment banks [26].

Unlike MQTT, AMQP was not designed to have a small code footprint or an easy to use interface, but rather AMQP was designed to be feature rich and high performance. Additionally, it is not simply a messaging protocol, but also defines its own type system to ensure interoperability between client and server.

Since AMQP was created to be used as a standard messaging protocol for a wide range of different users it supports a wide variety of messaging applications and communication patterns through a common interface. As a consequence, AMQP is a large protocol that is feature rich and allows for a lot of customization. However, going into detail about each feature of this protocol is outside the scope of this thesis. A short summary of the functionality that AMQP offers follows [4, p. 2]:

Types — AMQP has its own type system that defines a set of primitives that can be used to ensure interoperability between sender and receiver. These primitive values can then be associated with semantic information when sent in a message that tells the receiver how to interpret the value. For example, a string could be sent with the associated information that it is to be interpreted as a URL.

Transport — The conceptual model of an AMQP network is that it is a network of nodes connected via links. These nodes can either be sender, relays, or receivers. The link between nodes is a unidirectional communication channel which connects to the node's "terminus". This terminus can be either a source or a target depending on the role of the node.

Each node is responsible for the safe storage and delivery of messages to the next node. The link protocol between the nodes ensures that the message and responsibility are correctly transferred between nodes.

The nodes exist within containers which can, for example, be brokers or clients. For example, a broker container can consists of many queue nodes that store messages. These messages are subsequently relayed to the appropriate client container, which in turn contains a consumer node and a queue node.

Messaging — AMQP defines a standard for how messages are to be delivered. This includes how the message should be formatted with regards to header, trailers, and structure of payload, delivery states for messages, states for stored messages, how to filter messages, etc.

Transactions — AMQP uses a transactional model for message transfer. This means is that each transfer is initiated by a “declare” message and ended by a “discharge” message. This allows the transaction to coordinate independent transfers into one coordinated transaction, and ensures that if one of the transfers in a transaction fails, then all transfers in the transaction fail.

Securities — AMQP supports built-in security mechanisms for encryption and authentication. These mechanisms include the use of the Simple Authentication and Security Layer (SASL) [27] for authentication and Transport Layer Security (TLS) [28] for encryption on top of TCP.

AMQP is a large system, thus rather than going into detail about each of the features that AMQP offers - if any of these additional features are used in the experiment, then the details of these features will be described in the relevant section of this thesis.

2.8.1 AMQP Messaging overhead

Whenever a subscriber connects to a broker it must begin by sending a protocol header. The protocol header is 8-octets long and consists of the letters “AMQP” in uppercase followed by the major revision version number, major version number, and finally the protocol revision [29, p. 32]. However, as this header should only be sent when connecting it does not significantly affect the amount of data sent.

After a subscriber has connected it is able to send messages using 4 different frames namely:

1. “METHOD” - method frame
2. “HEADER” - content header frame
3. “BODY” - content body frame
4. “HEARTBEAT” - heartbeat frame

Each of these frames will be encapsulated by a general frame as shown in Figure 2-16.

Figure 2-16: AMQP General frame format, adapted from [24, p. 33]

For our purposes the heartbeat type will be set but not utilized. This is due to the fact that if we are to perform a substantial number of tests, the time waiting before performing a retransmission must be very short. This is possible since we do our testing over the loop-back interface were a response, if not dropped, should be almost instantaneous. The method type will be used to login to the broker and uses the header shown in Figure 2-17.

Figure 2-17: Method frame, adapted from [24, p. 34]

The Class-id and Method-id will be predefined AMQP variables, and the payload content will be the arguments that will be passed to the AMQP broker specifying what kind of method it is. For most messages the method will be “Publish” except for a login at the start and a disconnect at the end.

Almost all of the transmitted messages will utilize the content header and body frames to transmit the sample messages we use for benchmarking. These are illustrated in Figure 2-18.

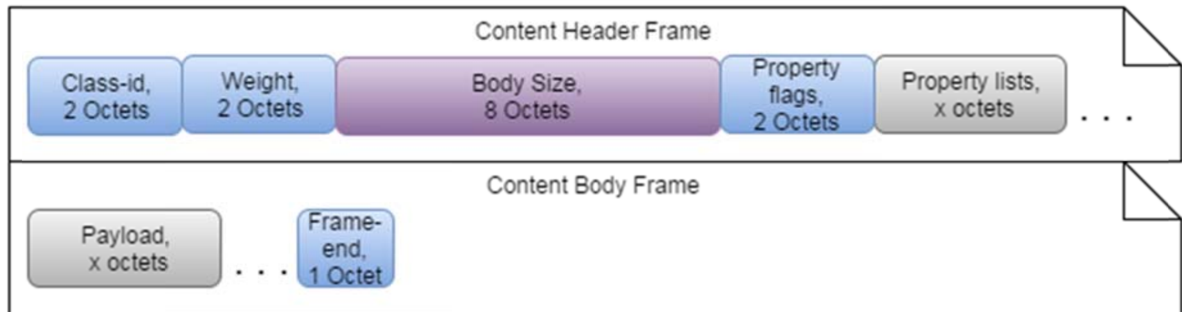

Figure 2-18: Content Header and Body frames adapted from [24, p. 36]

The Class-id is the same as for the Method frame, and the weight is unused and should be set to 0. Body size specifies the size of the content body. The property flags specify the different properties a message can have. These properties will be defined in the property list. The Content body frame contains the payload and an octet indicating the end of the frame.

2.8.2 AMQP v1.0 controversy

With the introduction of AMQP protocol version 1.0 there was a schism in the AMQP community due to the drastic changes that the version 1.0 introduced. Version 0.9.1 of the protocol specified how the client to broker protocol was supposed to operate and what messaging capabilities the broker should supply. This included how the exchanges, queues, and bindings of the broker were supposed to work, as well as what the messages between the client to broker and broker to queue would look like. However, in version 1.0 the specification became narrower and it now only specifies what the messages that will be exchanged between two endpoints are supposed to look like. This makes broker management a separate part of the protocol, thus allowing for more flexibility regarding how the creator of the broker can handle the messages.

The 0.9.1 version of the protocol has been used in the tests performed in this thesis project. However, this should not affect the results since although the protocol specifications have changed the communication between client and broker has remained largely the same.

2.9 Related work

All the investigated protocols have been available for the public to experiment with for quite some time, thus some researchers have tested and compared these different protocols. In this section some of those investigations will be surveyed.

2.9.1 Broker comparisons

When implementing a protocol that makes use of a message broker it is worth examining existing implementations of brokers. Since the broker acts as a middleman between the two communicating

parties the broker does not care very much about how the implementation of the client and server work, only that the messages transmitted to the broker are in the correct message format. For example, there is a list of MQTT broker implementations on the official MQTT GitHub repository [30]. The large number of different MQTT broker implementations would allow us to perform our tests using many different brokers, but such an investigation is out of the scope of this thesis project.

Finding unbiased and comprehensive benchmarks, i.e., benchmarks not conducted by one of the broker manufacturers, proved to be very hard to find. For this reason I settled on using RabbitMQ [31] as the message broker for AMQP and Mosquitto [32] for MQTT and MQTT-SN due to their large active user communities and ease of use. For MQTT-SN Mosquitto with a really small message broker (RSMB) as gateway is used. However, I make no claim that RabbitMQ or Mosquitto/RSMB are in fact the optimal broker/gateway choices for each protocol, thus my comparisons have to be taken for what they are.

2.9.2 Comparison of MQTT and AMQP

In 2015, Jorge E. Luzuriaga, et al. compared MQTT and AMQP in the context of mobile networks [33]. They evaluated these two protocols by creating an experimental setup where they evaluated loss, latency, and jitter. They then used these values to conclude what is the best use for each protocol and in what context it is most suitable to use a given protocol.

They evaluated common scenarios for a wireless sensor network (WSN), such as loss of connectivity for a period of time and behaviour during message bursts. They determined that apart from some peculiar behaviour from AMQP that causes it to receive messages in reversed order when a message burst occurs, they both perform well, but AMQP offers more features related to security, while MQTT is more energy efficient.

Finally, they concluded by saying that for "reliable, scalable and advanced clustering messaging infrastructures over an ideal WLAN" one should use AMQP, but for sensors operating in constrained environments, such as low-speed wireless networks, then MQTT is the better choice.

It should be noted that both MQTT and AMQP are TCP based protocols, so assuming MQTT-SN is used, the choice of MQTT is most likely even more suitable when it comes to constrained environments. However, in the context of this thesis the devices that are located in the vehicle are not constrained in terms of available power, link bandwidth, etc.

2.9.3 Comparison of MQTT and CoAP

In 2014, Thangavel, et al. carried out a comparison between CoAP and the MQTT protocol [34]. They implemented common middleware to facilitate communication between clients and a server using a common interface.

From their results they found that when a QoS level 1 for MQTT is used and a similar quality of service for CoAP using confirmable messages, then MQTT has a lower delay than CoAP when the packet loss was less than 25%. However, when the packet loss rate reached a sufficiently high percentage, then the delay for MQTT grew a lot faster than CoAP. The authors reasoned that this behaviour is due to the fact that since CoAP is UDP based and MQTT is TCP based, the retransmissions for MQTT sent a lot more data in each packet due to the overhead of TCP– hence the faster increase in delay.

Due to these different behaviours the authors argue that a good solution would be to have adaptive middleware that detects the current network conditions and then choses a suitable protocol.

2.9.4 Fast and Secure Protocol (FASP)

An alternative protocol that can be used when the objective is high speed file delivery is the proprietary application layer protocol developed by Aspera called Fast and Secure Protocol (FASP) [35]. FASP optionally offers encryption of the data.

FASP is built on top of UDP and aims to offer a protocol that will outperform TCP based protocols for file transfers, while still offering a delivery guarantee. This is achieved by using their own closed source proprietary algorithms for deciding on sending rate and retransmissions instead of TCP's congestion control and retransmission algorithms.

A 2015 Master's thesis by Patrik Hagernäs evaluated the performance of FAST over 5G, which Ericsson estimates will have a theoretical maximum throughput of 10 Gbit/s indoors and in dense areas [36]. He managed to achieve a maximum throughput of 6.3 Gbit/s [37] which he compared to the results of another Master's thesis by Victor Johansson who managed to achieve a maximum 7.3 Gbit/s throughput using pure UDP on a 10 Gbit/s link [38]. Both of these thesis projects used the same test environment.

The graphs in the analysis section of Patrik's report clearly indicate how vastly superior FASP is compared to plain TCP (with either New Reno or CUBIC) with respect to bandwidth utilization. The TCP based protocols quickly throttle themselves down when there packets are lost and then stay very consistently at low throughput even when the available bandwidth is increased. In contrast, FASP quickly achieves large bandwidth utilization and rapidly adapts to changes in available bandwidth.

2.9.5 Monitoring of other types of vehicles

While Scania has chosen to focus on real time data updates being sent via GSM to a server running either in the cloud or at a Scania facility there are other options that have been or are deployed to monitor certain engine parameters. One example is a module similar to the C300 that is mounted inside a airplane engine and then, instead of transmitting its data to a server, transmits all gathered sensor data wirelessly to a local database running inside of the plane [39]. This way, a lot more data can be gathered from the sensors since the database is local. GE focuses on gathering data as described in [40] as data lakes, where enormous amounts of data are gathered per flight and then analysed to see if something can be improved. This could be something for Scania to look into since it does not cost much to transmit data locally. Currently Scania does not log anywhere near the amount of data mentioned in the articles about GE's monitoring of their jet engines, but Scania does log small amounts of error codes and status messages to an on board chip. According to Scania engineers the biggest obstacle right now to logging larger amounts of data is that the infrastructure inside the truck is not yet able to handle large amounts of data. However, there are plans to add an Ethernet connection from the C300 or equivalent module to log CAN bus data straight to a persistent storage device for later extraction.

2.9.6 SCPv2 Transmission data

According to measurements made by Telefonica the data transmissions over their GSM networks by Scania can be summarize as in Table 1-1. In this table registration consists of truck and server initiating a secure connection. RD stands for Remote diagnostics and this used to transmit diagnostic data about the vehicle. OTA stands for over the air and this traffic is for remote software updates. STS stands for tachograph service – this service logs driving time, speed, and driver activity eliminating the needs for the driver to keep such logs. Position data is based upon a GPS receiver or other device indicating the position of the vehicle. Vehicle data consists of the current

state of the systems inside the vehicle, such as engine speed, engine temperature and other sensor data values. Geofencing triggers an alarm when a vehicle enters or leaves a defined geographical zone.

Table 2-1: Traffic volumes for Telefonica, April 2016

Traffic type	Delivered bytes	Protocol bytes	%of total bytes
Registration	117 586	340 927	0.04
RD	20 048 644	13 441 380	1.44
OTA updates	0	0	0
STS	5542	5315	~0
Position data	170 467 835	205 865 722	22.1
Vehicle data	613 665 115	711 055 980	76.32
Geofence	641 693	976 031	0.1
Total	804 946 415	931 685 355	100

The delivered bytes column indicates total amount of payload (in bytes) being transmitted while protocol bytes indicate the overhead introduced by the protocol. As can be seen about 86.4% of the transmitted data was payload data, which means that **the overhead introduced by the SCPv2 protocol is around 14%**. **However,** this overhead only includes the application level overhead introduced by the SCPv2 protocol and so it does not take into account the IP and Ethernet header that will need to be prepended. This is for the entire month of April so no conclusions can be drawn about how the protocol behaves during different conditions, but the average packets loss rate during this period of time is around 0-4% (depending on how you count a packet loss, see section 1.6) according to an internal document by Scania network providers. Another interesting fact to note is that over 98% of the traffic is generated by the position data and vehicle data messages. These correspond to the Positioning and Current Status messages described in Table 2-2. It is evident from these two tables that having a protocol that performs well on small (around 1kB) and frequent (about every minute) messages will provide Scania with the greatest benefit.

Table 2-2: Sending rates and message sizes as specified by Scania

Name	Size	Frequency
Vehicle data	~1 kB	10 per min
Positioning	~1 kB	1 per min
Tachograph file	~100 kB	4 per h
OTA Software update	~10 MB	Sporadic, very uncommon

2.10 Summary

From the earlier investigated protocols we can now summarize the protocols to be evaluated in the context of our implementation:

MQTT
: A light-weight protocol built on TCP that prioritizes ease of implementation and small code footprint.

MQTT-SN
: The sensor network variant of the MQTT protocol is built on the same principles as MQTT. However, MQTT-SN uses UDP as its transport layer protocol, thus it is more light-weight than MQTT with regard to the overhead of each packet. Additionally, it has features that might be helpful when used in a WSN.

CoAP
: CoAP is the only one of the listed protocols made for WSN that does not follow the publish/subscribe paradigm, but rather was designed to be similar to HTTP. This choice was made to facilitate easy integration of WSN nodes with the web. However, as evident by comparisons with other protocols, it is obvious that CoAP also performs well in non-web contexts - due to its low packet overhead.

AMQP
: AMQP is the most complex of the protocols investigated due to its huge feature set and wide range of customizability.

FASP
: A proprietary UDP based protocol that offers high available bandwidth utilization while still offering delivery guarantees. This protocol is particularly useful for delivering large files.

3 Methodology

This chapter provides an overview of the research methodology used in this thesis project. Section 3.1 explains the process used to conduct the research. Section 3.2 explains the different experimental setups. Section 3.3 explains what data was collected and why. In Section 3.4 the experimental setup is explained, along with why it was chosen. Section 3.5 discusses the reliability and the validity of the experiments.

3.1 Research Process

The research was originally planned to be performed in two stages. In the first stage the different protocols would be run in an artificial environment where the client and server each will be running inside a Virtual Machine (VM). In the second stage the protocols were planned to be run inside the C300 module and would communicate with a Scania server. However, due to timing constraints the implementation scope meant that second stage would not be possible and if attempted, would not produce valid data. Instead the implementation was done inside a virtual environment to collect simulated data in a controlled environment. This data will then be evaluated based on the volume of data overhead introduced by each of the different protocols.

The VM for the artificial environment will be running Linux with iptables[*] setup to emulate different network conditions. While there are other tools that offer more functionality regarding network emulation, such as Netem[†], there were a lot of problems getting Netem to work properly when setting up different networking conditions on multiple ports. Since the only emulation we needed was packet loss and iptables directly provides this and since this approach worked immediately the implementation used iptables instead of another alternative. Using iptables provides an environment with as little external interference as possible, while allowing us to change one network parameter at a time. The VM was configured to simulate different scenarios ,such as packet loss and protocol used, to see how these factors affect the amount of data transmitted.

3.2 Experimental Setup

This description of the experimental setup describes how the artificial environment was set up. The description describes how the environment was created and gives an overview of the system used for testing. This section also describes how the publisher and broker were set up and connected together with what operating system and hardware were used for the network emulations.

3.2.1 Artificial Environment

The artificial environment allows the experiments to be performed in a completely controlled environment where minimal interference is present. This makes it easier to reason about how the protocols behave when conditions vary since we can adjust the conditions exactly as we want them without any uncontrolled external factors influencing the results.

---

[*] http://linux.die.net/man/8/iptables

[†] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

3.2.1.1 Overview

For the first experiment, a virtual machine will be running on the desktop computer described in Section 3.2.1.2. This configuration facilitates communication between the publisher and broker by using the loop-back interface. Oracle's VirtualBox software is running on the test computer. VirtualBox provides a virtualized execution environment the VM. The test environment for the experiment is shown in Figure 3-1.

The host machine connects to the VM through a secure shell (SSH) connection. Through this connection each set of tests can be automated started by the host machine. First the host machine starts the brokers on the VM in order to prepare them to receive messages from the publisher applications. The server has one broker running on each utilized port and for each port iptables is configured to provide each of different network emulations in order to realize the specific network conditions for the relevant test. Utilizing the port configuration this way makes it easy to differentiate the packets when logging the experimental data. Next the publisher application for the first test will be started. The host machine will tell the client to commence sending packets of different sizes on the different ports and logs how quickly it is able to complete the set of tests and how much data was sent in order to transmit the desired data. This continues until all the tests for all the configurations have been performed.

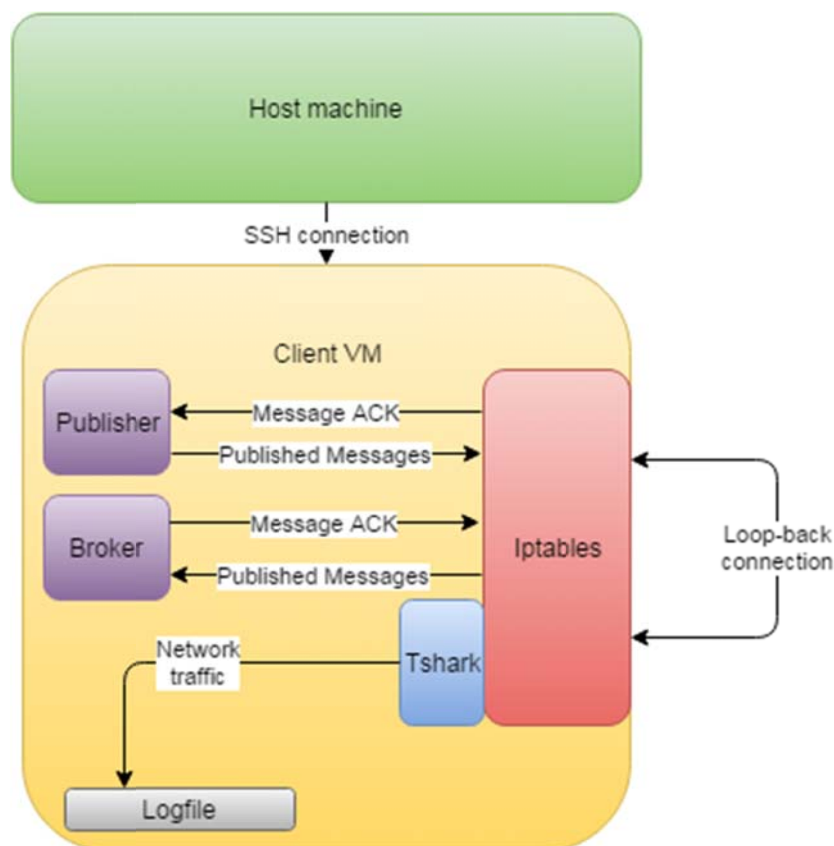

Figure 3-1: Artificial test environment

3.2.1.2 Test machine

The computer on which the tests will be run is a Dell Precision M2800 whose specifications are shown in Table 3-1.

Table 3-1: Desktop computer specifications

Processor	Memory	Storage	Operating System
Intel Core i7-4810MQ @ 2.80 GHz with 4 CPU cores	16 GB DDR3 SDRAM @ 1600 MHz	LCS-256L9S-11 (256GB Solid State Drive)	Windows 7 Enterprise 64-bit, Service Pack 1

To simulate various conditions under which the protocols are to be evaluated iptables is used on this desktop. Iptables allows the user to set up rules for packet filtering including stochastic rules. Rules were written for each port to have different probabilities to drop packets. This made it easy to compare the protocols since it allows us to change only one variable at a time.

3.2.1.3 Virtual Machine

As noted earlier, the VM will run on top of Oracle's VirtualBox environment. This VirtualBox environment is in turn run on the computer specified in Table 3-1 and has access to the hardware resources specified in Table 3-2.

Table 3-2: Virtual machine specifications

Processor cores	Memory	(Guest) Operating System
1	8 GB	Debian 1.3.14.1

3.2.1.4 Client

The client will be a publisher application running inside the VM. This application is invoked via SSH to start sending packets to a specified address (in our case the loopback address) and port. The clients are small programs built specifically for this test. Each client has the same interface for each of the protocols that are to be tested. When invoked, the user can specify the destination IP address, port number, number of iterations (how many times the program will be run), burst size (how many packets will be sent in a burst = one iteration), time between bursts (how long, in μs, that the program will wait between each iteration), payload size (the size in bytes of each payload) as well as a QoS level. Since the interface is the same for all the tested protocols we can easily automate the testing procedure for quick and easy testing.

3.2.1.5 Server

The server consists of a broker and a small custom C program that when activated will subscribe to the topic that the client publishes messages on. As soon as a message of that topic is published the program will consume it and, depending on the QoS level, the broker may send an ACK. This procedure will be the same for all protocols. For AMQP the librabbit-c [41] library will be used, while for the MQTT and MQTT-SN protocols the Paho libraries will be used [42]. The only task for the server is to consume messages and send ACKs so that we can see how these protocols behave when sending data.

As described in Section 2.9.1, two different brokers and one gateway will be used when the tests are conducted. For the AMQP protocol the RabbitMQ version 3.6.1 broker will be used, for MQTT the Mosquitto version 1.4.8 broker will be used, and for MQTT-SN Really Small Message broker version 1.3.0.2 (RSMB) will be used as a gateway.

3.2.1.6 Link properties

Since the tests utilize the loopback-interface there is no link *per-se*. However, since the criteria that the protocols will be evaluated on is the amount of transferred data and not transfer speed we want the transfer speed to be as fast as possible to shorten the time required to run all of the tests. However, the MTU of the loop-back interface will affect how much data can be placed into each IP packet and as such will impact the amount of data transferred. The MTU is set to 1500 bytes since this is the same as the MTU for a GPRS connection that Scania utilizes on the C300 module. This choice will produce more valid data than selecting another MTU size would.

3.3 Data Collection

Scania has specified that their main points of interest are: (1) how much data is sent over the network when sending the desired data, (2) how quickly the protocols are able to transmit messages, and (3) what is the response time for the requests. The reason for the first question is that Scania pays for each kB sent over the GSM network at a fee set by the network provider. Scania is allowed a maximum of 10MB per day per truck. This limit is set by the network provider, thus it is in Scania's best interest to transmit below 10MB a day per truck.

The focus of this thesis will be to evaluate each of the protocols as to how much overhead they introduce for each test. Tshark[*] will be used to log all the network traffic to a local file on the VM. These files will then be in a format that is easy to analyse using Wireshark's graphical interface.

3.3.1 Tests performed

Each test will consist of a 1MB data transfer from the publisher to the broker. The tests will be conducted using a QoS level of 1 and an application payload size, excluding headers, of 100B, 1kB, and 10kB. This set of tests will be done for each protocol and for each percentage of packet loss (from 0% to 30%). By performing the tests this way we will collect a lot of data about how the protocol behaves when transferring the most common packet sizes[†] as well as how they perform when transferring a larger amount of data converted into multiple smaller packets. The exception to testing with each of the three packet sizes occurs for the MQTT-SN variant as the broker did not accept the larger sized payload. Thus except for the 100B test, each of the tests with MQTT-SN will have its payload split into payloads of 255B on the application level. This of means that the MQTT-SN protocol will transmit more header data than the other protocols in these tests were (application layer) fragmentation is used, hence this will be addressed in the analysis.

3.3.2 Logging

To log all incoming and outgoing network the terminal version of Wireshark, Tshark, is used on the VM. This allows logging of all packets that are sent over the network *including* the TCP handshakes and other data related to the transmission. Since we need to be able to filter the received packets based on which test they belonged to we ran each test on a different port and then group the logged files by ports. Having the possibility to filter on ports also allows us to filter out the SSH packets used to control the tests. The result is that we can tell exactly how much data was sent for each test for a given emulated network condition.

---

[*] https://www.wireshark.org/docs/man-pages/tshark.html

[†] The three different sizes were selected to be representative of the typical amounts of payload generated by the different sources expected from the current C300 software and a representative colection of attached sensors.

3.3.3 Sample Size

For each protocol the amount of data transmitted will be investigated as a function of the packet loss rate. The percentage of packet loss will vary from 0% to 30% in steps of 1%, and for each percentage and for each protocol 1MB of data will be transferred using three different payload sizes. These sizes are 100B, 1kB, and 10kB. These payload sizes were chosen to capture certain attributes of the protocols. The 100B payload size is used to illustrate how MQTT-SN compares to the other protocols when the payload is fragmented into the same number of packets as for the other protocols. This is so we can understand how this protocol would compare to the other protocols, especially when a larger payload size of 1kB is used which would be the case normally for the protocols when deployed by Scania. The 1kB payload size is used to see how the protocols would behave when transmitting the most common of the expected payload sizes, while 10kB is meant to reveal how the protocols behave when they make use of transport layer fragmentation to transport large payloads. Each test will be conducted with a QoS level of 1. Table 3-3 summarizes the different protocols, packet loss rates, and payload sizes to be used for testing.

Table 3-3: Summary of protocols, packet loss rates, and payload sizes to be used for testing

Protocols	Packet loss	Payload sizes
AMQP, MQTT, MQTT-SN	0%, 1%, 2%, ..., 29%, 30%	100B, 1kB, 10kB

3.3.4 Note about QoS

Although each protocol offers multiple QoS levels they offer it in slightly different ways. For example, AMQP allows for more precise control of the service level than simply setting the QoS level for messages. As one can specify exactly how many un-ACKd messages the broker should be allowed to send to the client before it starts to wait for ACKs. However, we need to be able to guarantee at-least once delivery for all of our messages. This corresponds to a QoS level of 1 for MQTT and MQTT-SN. For the tests of the AMQP protocol, the protocol will be configured to either assume a QoS of 0, that is retain none of the messages, or it will be configured to retain all messages until ACKd.

3.4 Experimental design

This section describes how the experiment was set up and how each test was conducted as well as how many tests were performed and with what parameters. It will also talk about the reliability and validity of the data that the tests are expected to produce.

3.4.1 Test bed

The test bed is set up to allow for easy testing of the protocol together with logging data in order that the test results will be simple to analyse. To achieve this there was a need to quickly test lots of different network conditions. Additionally, it should be easy to tell to which test each packet belongs. This was done was by configuring iptables to emulate the different network conditions on different ports. Then we have one broker for each port ready and listening for traffic on that port as shown in Figure 3-2. Having multiple brokers running on the same server at first seemed as if it would influence the tests by taking system resources; however, for our tests we have around 100 brokers running and an idle Mosquitto broker is shown as using 0.0% of the CPU and 0.0% memory

and a RabbitMQ broker as 0.0% CPU and 0.5% of memory when monitored using htop 1.0.2*. By setting the tests up in this way we do not have to worry about starting and re-starting the brokers on the same port. Additionally, because we start the next test immediately after finishing the previous test, there is a chance that we might receive a late packet during a new test or miss packets from previous tests.

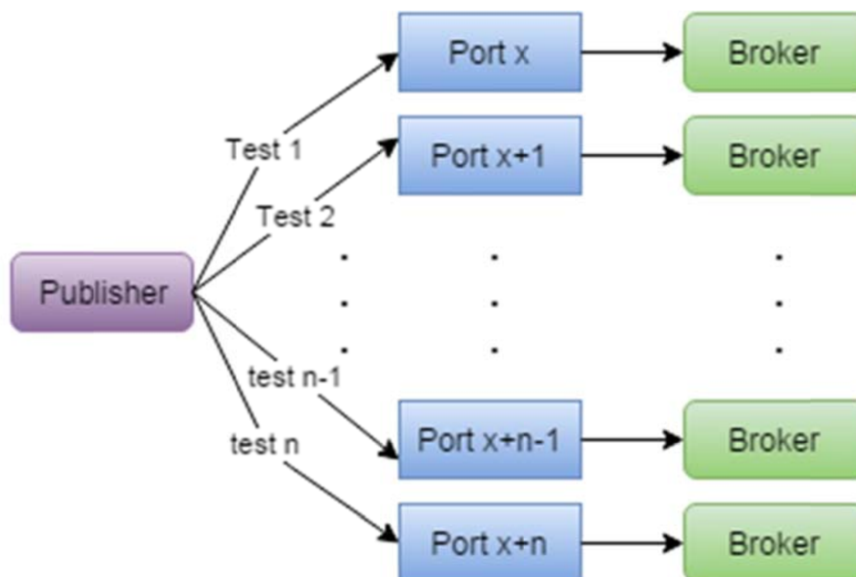

Figure 3-2: Testbed overview

For each test a total amount of 1MB is transferred from the publisher to the broker. The data is sent in chunks of 100B, 1kB or 10kB or each packet loss percentage between 0 and 30 percent packets as described in Section 3.3.3 and shown in

Protocols	Packet loss	Payload sizes
AMQP, MQTT, MQTT-SN	0%, 1%, 2%, ..., 29%, 30%	100B, 1kB, 10kB

. As a result we have a total of 279 ports open with 279 brokers listening to one port each for each protocol tested. Each port is configured as shown in Appendix A. It should be noted that Figure 3-2 only illustrates the outgoing port, but the brokers will need to be connected to a port as well when responding to the publisher.

3.5 Assessing the reliability and validity of the data to be collected

A lot of thought went into designing tests that would produce both reliable and valid results. However, there are some practical factors that need to be addressed that will affect the validity of the results that it was not possible to solve within the time frame of this thesis project. These factors are described in the following subsections.

3.5.1 Reliability

For the artificial environment we are able to produce very reliable results. This is due to the fact that given the parameters to configure iptables, the programs used to send the data, the data we send,

* http://linux.die.net/man/1/htop

and the virtual machines we used someone else should be able to set up the same tests, perform the experiment, and obtain almost exactly the same results. This is due to the fact that in our artificial environment there is nothing interfering with the testing, hence the results should be easily reproducible.

3.5.2 Validity

With regard to validity there are some problems. The purpose of the experiments is to determine how well suited the protocols are for Scania's purposes. This is done by testing how the protocols behave when different network conditions are emulated. Since we can eliminate nearly all interference with the experiments we are able to produce very valid data about how each protocol behaves with regard to packet loss. However, the C300 module will never operate in a vacuum and there will always be interference affecting the results which means that the results of our testing will not be completely valid in the context of one of Scania's vehicles.

4 Test setup

The main focus of the tests is to determine how each of the different emulated network conditions affects the amount of data transmitted. For these experiments the MTU will be kept at 1500 (the same as that of a GPRS link) and the transfer speed will be as fast as the loop-back interface allows. While GSM would theoretically only allow a throughput of ~100kb/sec, this thesis focuses on the amount of data transferred rather than the transfer time. Thus we chose not to limit the transfer speed in order to able to perform the tests as quickly as possible. Note also that the actual throughput of a GPRS link depend upon both the radio conditions and whether GMSK or 8PSK modulation is used, what type of coding is used, and how many slots are allocated to the device.

4.1 Extra header data

For every packet sent, both an application level header and transport level header need to be prepended. While this certainly is not be the case for every application using TCP or UDP it will be for these tests due to the fact the application will issue a push call for every TCP transmission, and for the UDP the fragmentation will occur on the application level so no IP fragmentation will be used. The impact of this will be discussed in section 5.2.

The application level headers will be the headers of the protocol we are investigating, while the transport layer header will be either that of TCP or UDP. Additionally an IPv4 header of 20 octets and, for our simulated tests, an Ethernet II header of 14 octets will be transmitted. The total amount of header data is shown in Figure 4-1. This data was generated by publishing a one character message to the broker and then using Wireshark to investigate the size of the packets that were sent.

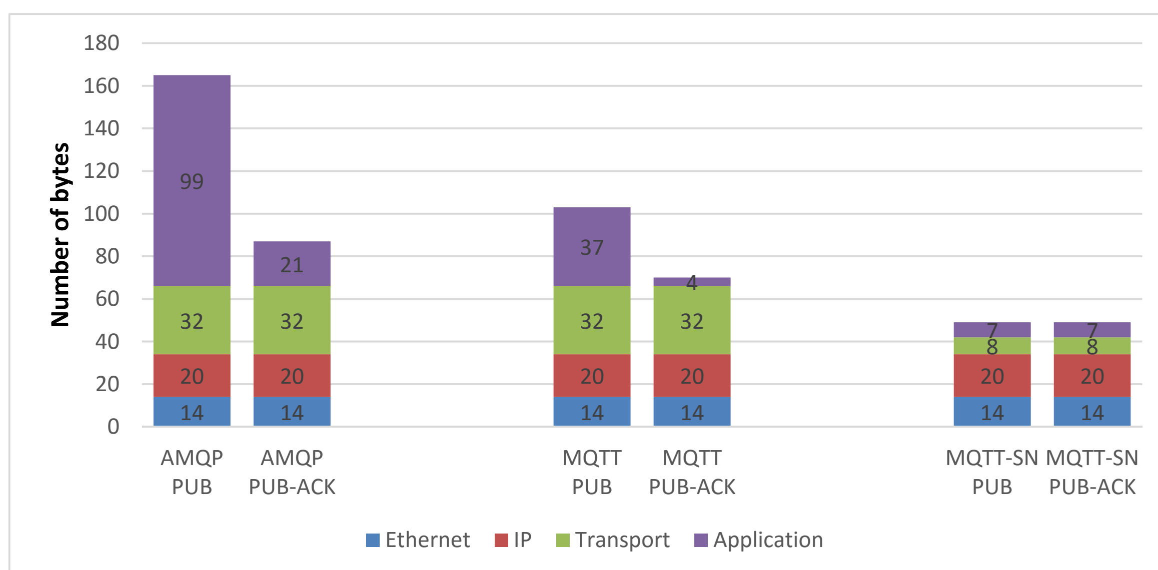

Figure 4-1: **Total amount of header data for each packet when transmitting a single byte of application layer payload**

Since the application layer headers vary depending on the message being sent, the most common message header was chosen for these measurements. For the AMQP protocol this message is the PUB and PUB-ACK messages – as they contain the general frame plus the method, header, and body frame (the header and body are only used for the PUB message) along with two property fields as will be explained in Section 4.2.1. Except for the initial login message and the final disconnect message to the broker, PUB and PUB ACK messages will only be sent once assuming

none of them are lost. The MQTT setup is similar. Here the PUB message will include the fixed length header along with a variable length PUB header. Finally, for MQTT-SN since the payload was only one octet the size the length field is only one octet rather than three. The total application layer header is 7 octets. One of the reasons that the MQTT-SN protocol header is so much smaller than the other protocols is that it does not include the topic string, but instead uses the registered topic ID (encoded as two octets). This means that for the tests performed in Figure 4-1. MQTT included a topic ID as a string **31 octets** in size, while AMQP for this same topic together with an exchange string results in an extra **40 octets** of payload. However, it should be noted that although MQTT-SN supports 64kB payloads during our tests if the longer size field is used, then the broker in our tests refused packets that used this larger size. As a result we could only test MQTT-SN with 255 octet payloads. This more header data was transferred in our tests than would occur if the longer size field had been supported by the broker. As noted earlier this will be addressed in the analysis section.

It should also be noted that this test only considered the data sent in one application level PUB and received in one application level PUB-ACK. For the MQTT protocol there will be an additional **66 octets** transferred in the form of a TCP ACK from the client to the server to ACK the application level ACK. This will not be the case for the AMQP protocol, despite running over TCP for the reasons to be discussed in Section 5.1.

4.2 Publisher applications

To measure how each protocol performs with regard to the amount of data overhead for each scenario, a set of applications were created that publish a fixed amount of data to a broker. These applications take as input the amount of data that is to be sent and how many times each item is to be sent. The programs operate as described in the following subsections.

For the applications, no modifications to the socket buffer sizes were made and the default receive and send socket buffer size were the following[*].

```
rmem_default = 212992
rmem_max = 212992
wmem_default = 212992
wmem_max = 212992
message_cost = 5
netdev_max_backlog = 1000
optmem_max = 20480
```

4.2.1 AMQP publisher application

For the AMQP publisher application the first thing we do, apart from reading from standard input (stdin), is to set the exchange and routing key. An exchange is a construct inside the broker that maintains a number of queues. These queues are indexed by the routing key when publishing or subscribing to a topic. Consider the topic “test.topic.first”with the exchange key “amq.topic”. This can be defined as:

```
char const *exchange = "amq.topic";
char const *routingkey = "scania.truck.test.system.sensor";
```

Next, the message to be sent is composed. This is done by creating a C-style string, which is a null terminated array of characters. The size of this message is taken from stdin via the automated

[*] http://man7.org/linux/man-pages/man7/socket.7.html (socket setting variables explained in section “/proc interfaces”)

script and depends on what payload size will be used. The message will be created in the same way for all the publisher applications.

```
char* messagebody = (char*)malloc(payloadsize + 1);
memset(messagebody, 'a', payloadsize);
messagebody[payloadsize] = '\0';
```

The login to the broker is initiated using the following parameters

```
amqp_login(conn, "/", 0, AMQP_DEFAULT_FRAME_SIZE, 0, AMQP_SASL_METHOD_PLAIN,
"guest", "guest");
```

This function call tells the broker to allow for any number of channels, although only one will be used in our case. The maximum frame size is set to the default of 128 KB (the value of `AMQP_DEFAULT_FRAME_SIZE`). Additionally, heartbeats are disabled. The `AMQP_SASL_METHOD_PLAIN` parameter tells the broker to expect two additional arguments: the name and password. In this case both are simply set to the string “guest”. The broker was configured to have a user with this name and password.

Finally, we set the Content Header frames property flag and property list fields as described in Figure 2-18. The flag fields are set as follows:

```
props._flags = AMQP_BASIC_CONTENT_TYPE_FLAG | AMQP_BASIC_DELIVERY_MODE_FLAG;
```

This sets bits 12 and 15 of the property flags field and tells the broker that the property list will contain the content property and the delivery property. These are set to:

```
props.content_type = amqp_cstring_bytes("text/plain");
props.delivery_mode = qos;
```

The above tell the broker that the payload will be a null terminated C string and that the QoS level will be as specified, in our case 1.

4.2.2 MQTT publisher application

For the MQTT application a client object is first created to initiate the connection. The connection is then created by calling the create function with these settings:

```
MQTTClient_create(&client, address, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE,
NULL);
```

This call passes, a handle to the client object to be created, the IP address of the server, and the ID of the client (which in this case is the default ID “ExampleClientPub”). The `MQTTCLIENT_PERSISTENCE_NONE` flag indicates that the client does not request any persistence for the messages. This means that these messages, assuming a QoS level 1 or higher, will be stored in volatile memory, hence if and only if the client dies they will be lost.

The only additional connection options are the keep alive interval and the clean session setting:

```
conn_opts.keepAliveInterval = 20;
conn_opts.cleansession = 1;
```

This above sets the keep-alive timer to 20 seconds and enables a clean session (i.e., tells the server not to keep any information about the client when it disconnects).

Finally the message to be published is set with:

```
pubmsg.payload = payload;
pubmsg.payloadlen = strlen(payload);
pubmsg.qos = qos;
pubmsg.retained = 0;
```

In this case payload is the message (as a null terminated string) to be sent. The bit fields for QoS are set by the user and the retained property is set to false (telling the server not to keep a copy of the message).

4.2.3 MQTT-SN publisher application

The MQTT-SN publisher is a bit different as MQTT-SN utilizes UDP rather than TCP. This means we simply assign the application a port, an IP address, and a topic and then we can start sending messages.

When initiating communication from an MQTT-SN application to a broker the user can chose to perform either a registration handshake where the client uses a long topic ID to register a unique shorter ID to be used for all conversations using that topic or the client can use a pre-registered two octet long short ID, which is what will be done for this client. This short ID barely impacts the amount of data transmitted and it is reasonable to assume that Scania will use sufficiently few topics that pre-registering them will be possible. The chosen topic name has no significance an could be any two character combination.

```
topic_id = "tt";
```

A UDP socket is created and initiates a connection to the broker. This was a bit problematic since the client would sometimes crash when the connection failed. This was solved by having the automated script described in Section 4.3 handle the problem by re-starting the client if the connect packet was lost.

```
sock = mqtt_sn_create_socket(mqtt_sn_host, mqtt_sn_port);
mqtt_sn_send_connect(sock, client_id, keep_alive, TRUE);
```

After that the client is ready to start transmitting data by looping over a publish and receive pair:

```
for(i = iterations; i > 0; i--){
   usleep(tbb);
   for (n = burstsize;n > 0; n--){
      mqtt_sn_send_publish(sock, topic_id, topic_id_type, payload, qos, retain);
         while(!mqtt_sn_simple_pub_ack_wait(sock)){
            mqtt_sn_send_publish(sock, topic_id, topic_id_type, payload, qos, retain);
         }
      }
   }
```

Here the client will publish the pre-defined message to the broker and then wait for an ACK. The “`mqtt_sn_simple_pub_ack_wait`“ function will cause the client to wait 10 ms from when it publishes the message when it checks whether an ACK has been received before performing a retransmission. Initially, 1 second was used ensure that the retransmission were not done erroneously, but waiting 1 second between retransmissions takes very long time when sending 4000 packets per test with lots of packet losses. Waiting 10 ms is plenty of time when running on the loopback interface which is confirmed by pinging the loopback address which gives an average RTT of 0.06ms

```
--- 127.0.0.1 ping statistics ---
101 packets transmitted, 101 received, 0% packet loss, time 99998ms

rtt min/avg/max/mdev = 0.024/0.060/0.121/0.014 ms
```

4.3 Automated scripting

To be able to perform all the tests for all the different configurations a script was used to set up the scenarios and run the tests. The scripts were written in Bash script and executed using the GNU Bash shell version 4.3.30(1). These scripts are included in Appendix B. To perform all the tests the script used three stages to ensure everything worked properly. First the setup phase was run. Here the script generated all the configurations for the scenarios that would be run when using the script and output them to a text file. This included rules for iptables and parameters to the publisher applications. After the setup phase was done all of the brokers were started and set to listen to the ports to be used. This is followed by a delay of 20 seconds to ensure that all of the brokers have completed their startup before continuing. The script then starts to iterate through the test scenarios generated in the setup phase. Each iteration begins by first starting tshark to log all of the network traffic over the loopback interface to a file on the virtual drive inside the VM, then the iptables are configuring according to the requirements for scenario and lastly the publisher application is run with the parameters from the scenario file. This is repeated for all of the scenarios in the scenario file. It should be noted that when the publisher connects to the broker the programs would sometimes crash if the connect packet was lost. For this reason the publisher applications were set to exit with an error code that is non-zero which the script can check for. If this occurs, then the script will restart the publisher application and continue to do so until a connection is successfully established. The order that the script will perform the tests in is that all packet loss configurations for a certain protocol and payload size are runs with first AMQP, then MQTT, and finally MQTT-SN. After this iteration is done, then the payload size is increased and the tests are done over again in the same order.

5 Analysis

In this chapter experimental results will be analysed. In section 5.1 the experimental data from the test will be presented and 5.2 will talk in more detail about how the data was transmitted between server and client. Section 5.3 will discuss the reliability and validity of the obtained data and finally Section 5.4 will talk about the set up and performance of the tests.

5.1 Major results

Looking at the graphs Figure 5-1, Figure 5-2, and Figure 5-3 we can see the results for the different test scenarios for the first payload size. It should be noted that for some of the tests with higher packet losses the protocols would sometimes break down or the logging of the traffic was corrupted or erroneous. This means that some of the plots contain more data and some less, but dots on the line represents a successfully completed and logged scenario. The most common protocol to breakdown was MQTT-SN which is most likely due to the fact that the protocol is very minimal and offers little functionality in terms of automatic connection recovery. This behavior could probably have been fixed had more time gone into the implementations and configurations of the brokers, but as will be discussed in Section 5.3 there were some limitations that made running multiple tests difficult.

5.1.1 100B payload results

The purpose of the 100B payload test was to see how the protocols compared when the 1MB of data to be sent was broken up into a large number of small payloads to be transmitted by each protocol. This test was necessary since, as mentioned in Section 4.1, the broker would not accept MQTT-SN payloads using the larger size field; hence the maximum payload size for the MQTT-SN protocol was 255 octets. In order to prevent the MQTT-SN protocol from having to break the payload up into multiple smaller transmissions the 100B payload was used. The results can be seen in Figure 5-1 where the total number of kB's transferred for each protocol as a function of packet loss is plotted. The plots contain some expected and some unexpected results. The fact that the MQTT-SN protocol outperforms the more data heavy and TCP based protocols with regards to the total amount of data transmitted is unsurprising. Unfortunately, the UDP based protocol broke down after reaching 19% packet loss, hence there is no test data available for higher packet loss rates. However, based on how MQTT-SN behaves in the 1kB tests shown in Figure 5-6, there is no reason to believe that the behaviour for higher packet loss rates in the case of 100B payloads would be drastically different than it was for 0 to 18%.

The more interesting result is that for the AMQP and MQTT protocols, as shown in Figure 5-1, they do not differ that much despite AMQP being a much heavier protocol with regards to data transmission overhead.

However, when looking at the number of packets transmitted between the server and the client, as well as the contents of those packets it becomes clears as to why this is. When comparing the number of packets going from the client to the server in Figure 5-2 and from server to client in Figure 5-3 we see that the number of packets from the server to the client is about the same for all the protocols, but the number of packets sent from the client to the server for the MQTT protocol is about twice as many packets as the other two protocols.

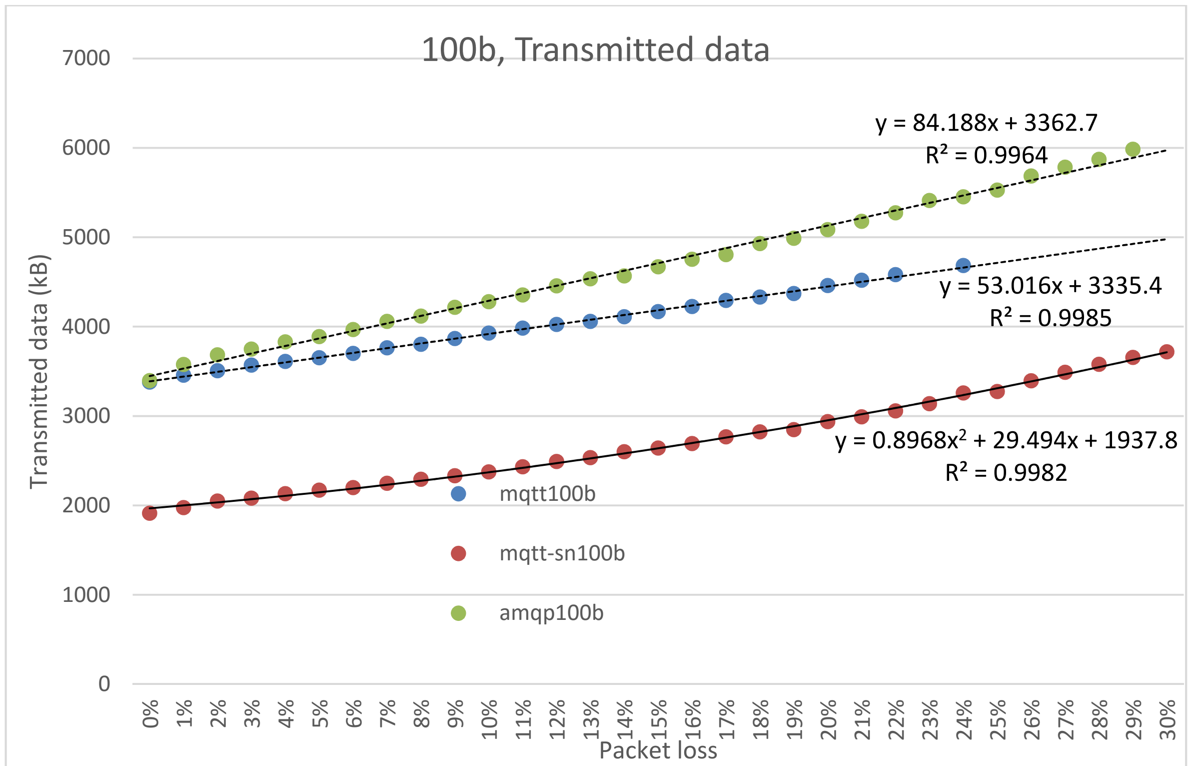

Figure 5-1: **Total transmitted *data* for 1MB using 100B payloads and QoS 1**

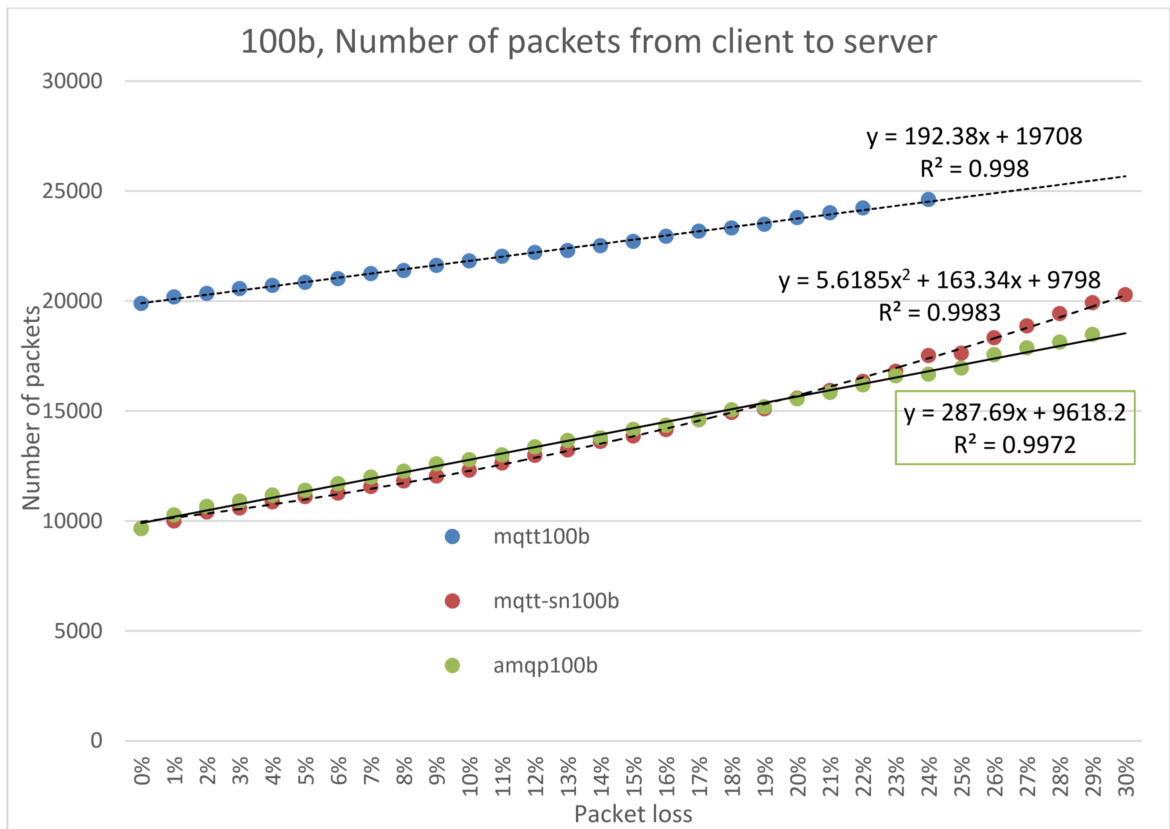

Figure 5-2: **Number of packets transmitted from client to server, 1MB data, 100B payload, QoS 1**

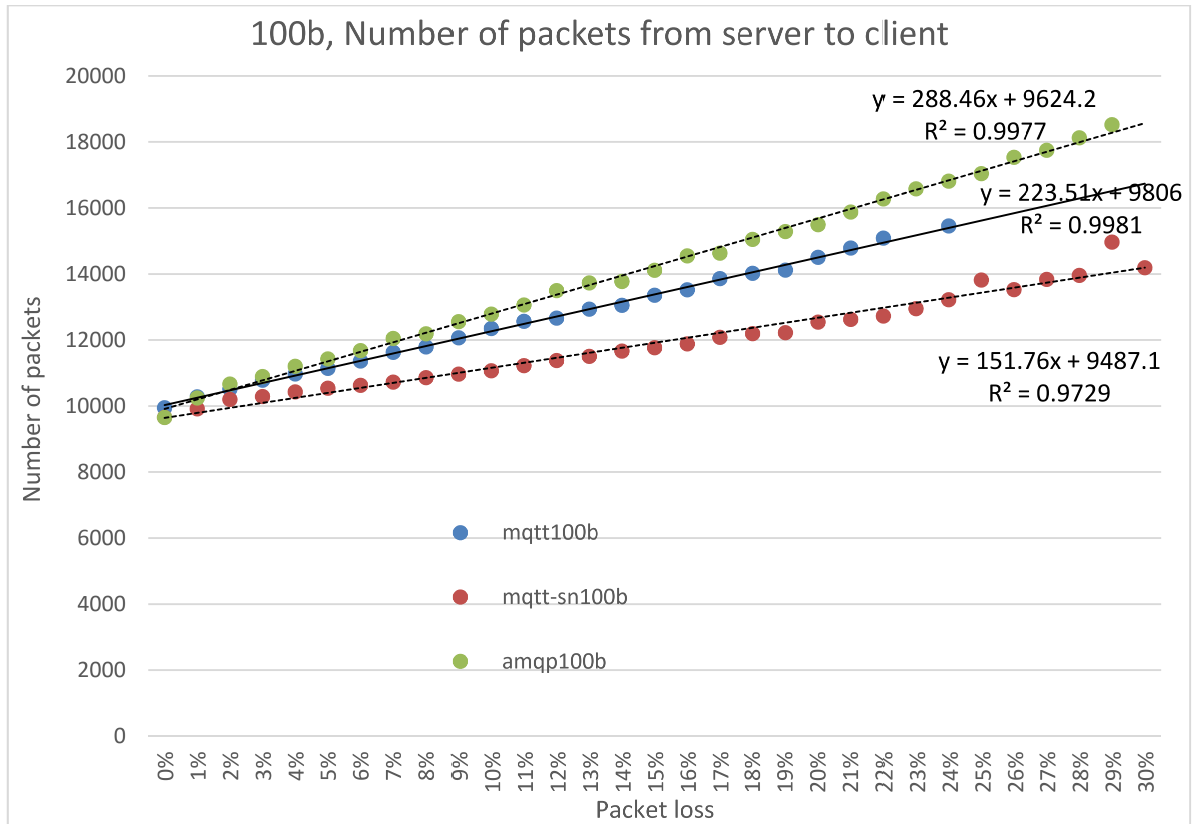

Figure 5-3: Total number of packets from server to client

The reason for this behaviour is, as mentioned in Section 4.1, because MQTT sends an extra TCP ACK compared to AMQP. This ACK is sent from the client to the server to ACK that the application layer ACK was delivered. In contrast, AMQP solves this in a different way. Figure 5-4 shows the output of Wireshark when looking at the transmissions between the client and server. Here it looks as if AMQP never performs a transport layer ACK, but instead only does application level ACKs. The way AMQP gets away with this is by setting the ACK flag for each packet it transmits, as shown in Figure 5-5, to ACK the previous packet. This way, each PUB is a TCP ACK for the last application layer ACK, and each application layer ACK is a TCP ACK for the last PUB.

No.	Time	Source	Destination	Protoc	Length	Info
13	0.232486	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack
14	0.232568	127.0.0.1	127.0.0.1	AMQP	1165	Basic.Publish
15	0.232647	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack
16	0.232717	127.0.0.1	127.0.0.1	AMQP	1165	Basic.Publish
17	0.232793	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack
18	0.232896	127.0.0.1	127.0.0.1	AMQP	1165	Basic.Publish
19	0.232977	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack
20	0.233078	127.0.0.1	127.0.0.1	AMQP	1165	Basic.Publish
21	0.233134	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack
22	0.233239	127.0.0.1	127.0.0.1	AMQP	1165	Basic.Publish
23	0.233294	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack
24	0.233403	127.0.0.1	127.0.0.1	AMQP	1165	Basic.Publish
25	0.233464	127.0.0.1	127.0.0.1	AMQP	87	Basic.Ack

Figure 5-4: AMQP sample traffic

```
Acknowledgment number: 190    (relative ack number)
Header Length: 32 bytes
Flags: 0x018 (PSH, ACK)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR): Not set
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...1 .... = Acknowledgment: Set
  .... .... 1... = Push: Set
  .... .... .0.. = Reset: Not set
  .... .... ..0. = Syn: Not set
  .... .... ...0 = Fin: Not set
```

Figure 5-5: AMQP TCP ACK flags

Since we transmit a lot of packets that result in a TCP ACK this makes AMQP competitive with MQTT. However, as the packet loss rate increases the difference in the number of packets sent by AMQP as compared to MQTT shrinks and the amount of data transmitted for each AMQP scenario increases more quickly than for MQTT. This indicates that AMQP is more sensitive to high packet loss rates than MQTT with regards to the increased amount of data transmission. It should be noted though that the MQTT test was not able to complete some of the tests when the loss rate became high, perhaps indicating that AMQP is more stable. However, no conclusions will be drawn about the stability of each protocol due to the fact that there might have been a number of factors that made these test fail (such as the client's implementation, the broker's configuration, the VM, or simply the logging program corrupting the log file).

5.1.2 1kB payload results

For the 1kB payload tests the graph in Figure 5-6 looks quite as expected. First, the MQTT-SN performance is a lot worse now when compared to the 100B tests due to the fact that it has to transmit 4 IP packets for every AMQP or MQTT payload transmitted. Thus it will transmit 4 times as much IP header bytes than it would have if the larger packet size was used (almost 4 times since an extra octet would be needed for the larger payload size field). However, despite this the MQTT-SN protocol manages to almost be however on par with the other protocols with regards to the total number of bytes transmitted data at lower packet loss rates. But as the packet loss rate grows so does the number of transmitted bytes and it does so at a much more rapid pace for MQTT-SN than for the other protocols. Looking at the number of transmitted packets to and from the server we can see that the number of transmitted packets for the MQTT-SN protocol in Figure 5-7 and Figure 5-8 are much higher than for the other two protocols, especially when compared to the 100B tests. It should also be noted that the reason why the number of packets sent from the client to the server is higher than from server to client is due to the fact that the retransmission wait time is set to be smaller than the keep alive timer. Therefore, if an ACK is not transmitted within 10 ms then the client performs a retransmission. This means that there will always be a retransmission when an ACK is lost, but no ACK will be sent when a PUB is lost - instead a PUB retransmission will occur after 10 ms. As a result there will be more messages set from the client to the server than in the reverse direction.

For the AMQP and MQTT the larger payload size results in the overhead from the header being smaller in comparison to each other than when the payload size was 100B. The number of transmitted packets for each of the TCP based protocols maintains largely the same proportion to each other as was to be expected.

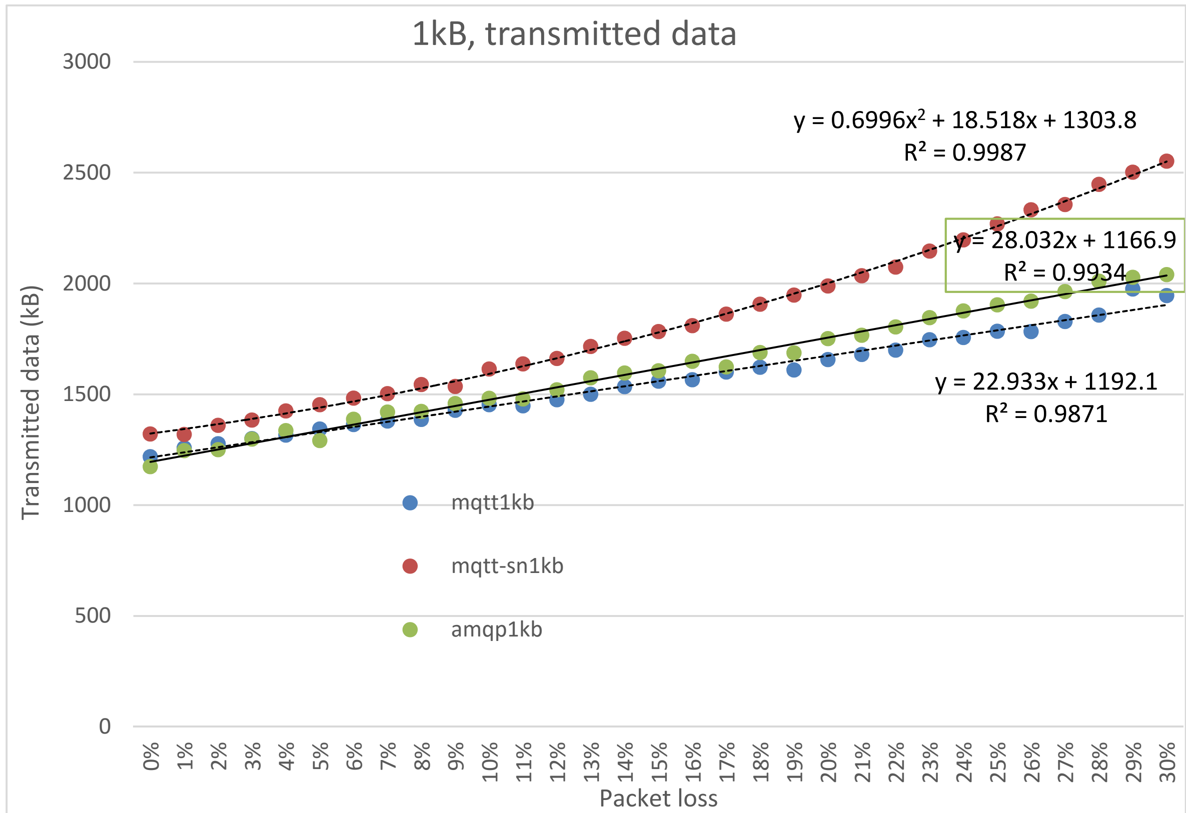

Figure 5-6: **Total transmitted data for 1MB using 1kB payloads and QoS 1**

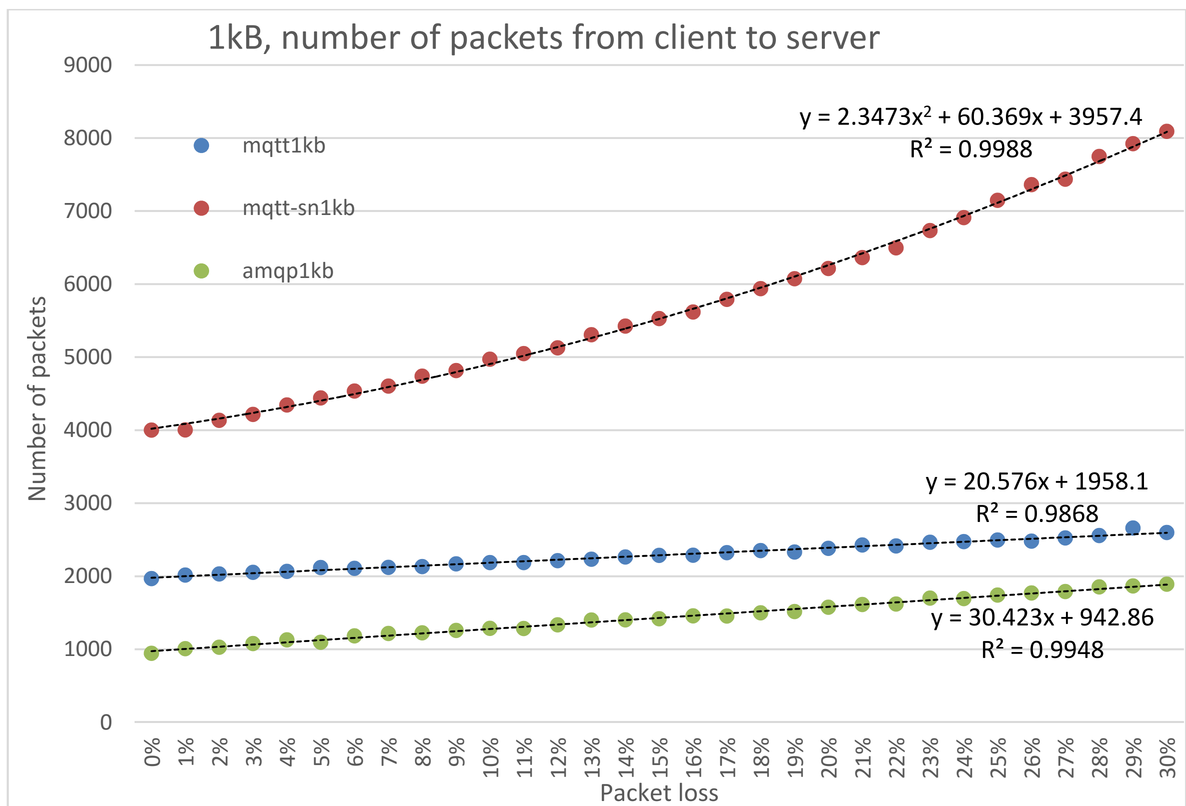

Figure 5-7: **Number of packets transmitted from client to server, 1MB data, 1kB payload, QoS 1**

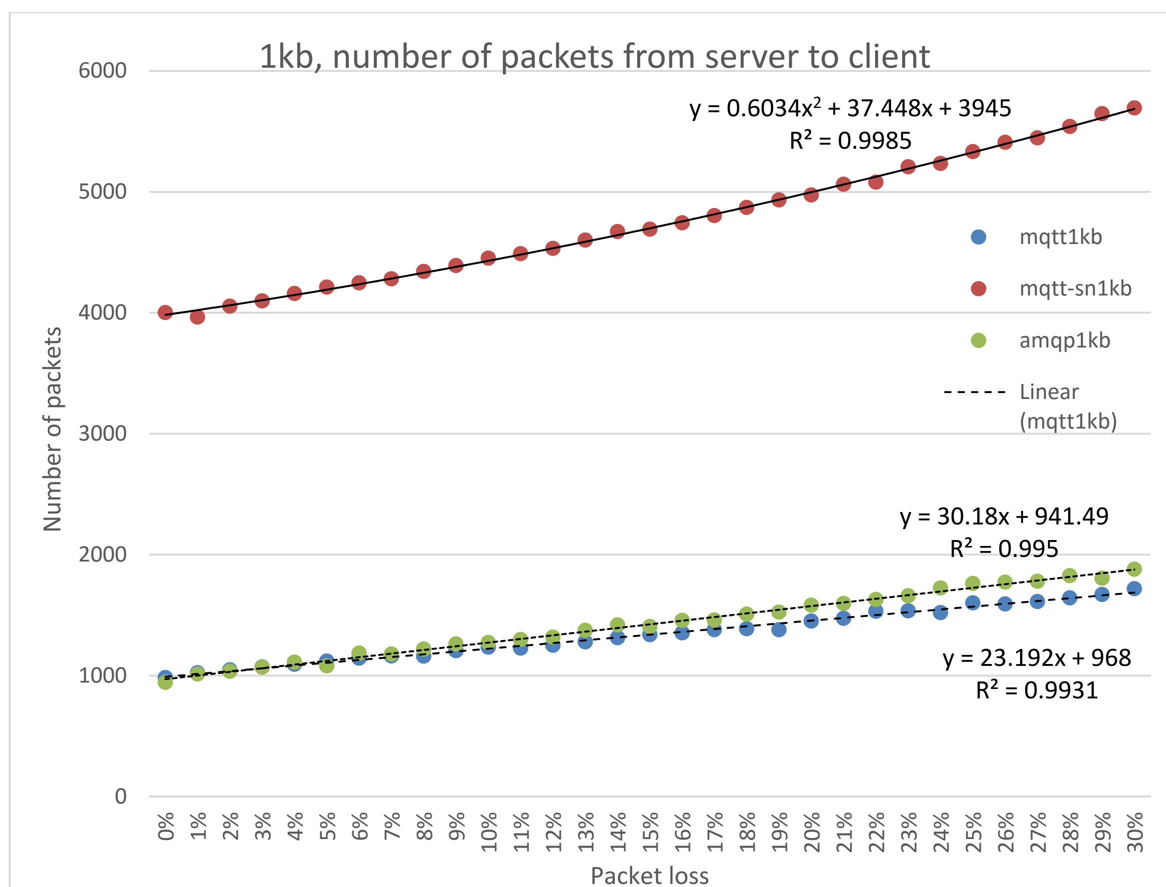

Figure 5-8: Number of packets transmitted from server to client, 1MB data, 1kB payload, QoS 1

5.1.3 10kB payload results

For the 10kB payload test the results were also expected although with greater variation for the tests due to fewer packets being sent over the network. Interestingly, the MQTT-SN variant refused to work for this scenario despite being the one that should not be affected by the average size since it will be fragmented on the application level anyway. For AMQP and MQTT the results were also quite unstable with often erroneous packet logs. The logs would show that despite a total payload of 1 MB that the total delivered data was less than that. No indication of the logging program failing were found since tshark properly logged each packet. For the results that were had however it can be clearly seen in Figure 5-9 that the larger payload size reduces the difference the header size makes due to much fewer packets transmitted.

The results for the 10kB transmission should however not be considered very trustworthy. Mainly due to the instability and error proneness of the tests, but also because of sporadic behaviour that can be seen for example in Figure 5-10 which lists the time to complete a transmission. The behaviour of AMQP is especially interesting since it seems to indicate that for large payloads the protocol is very sensitive to packet loss with regards to transmission time.

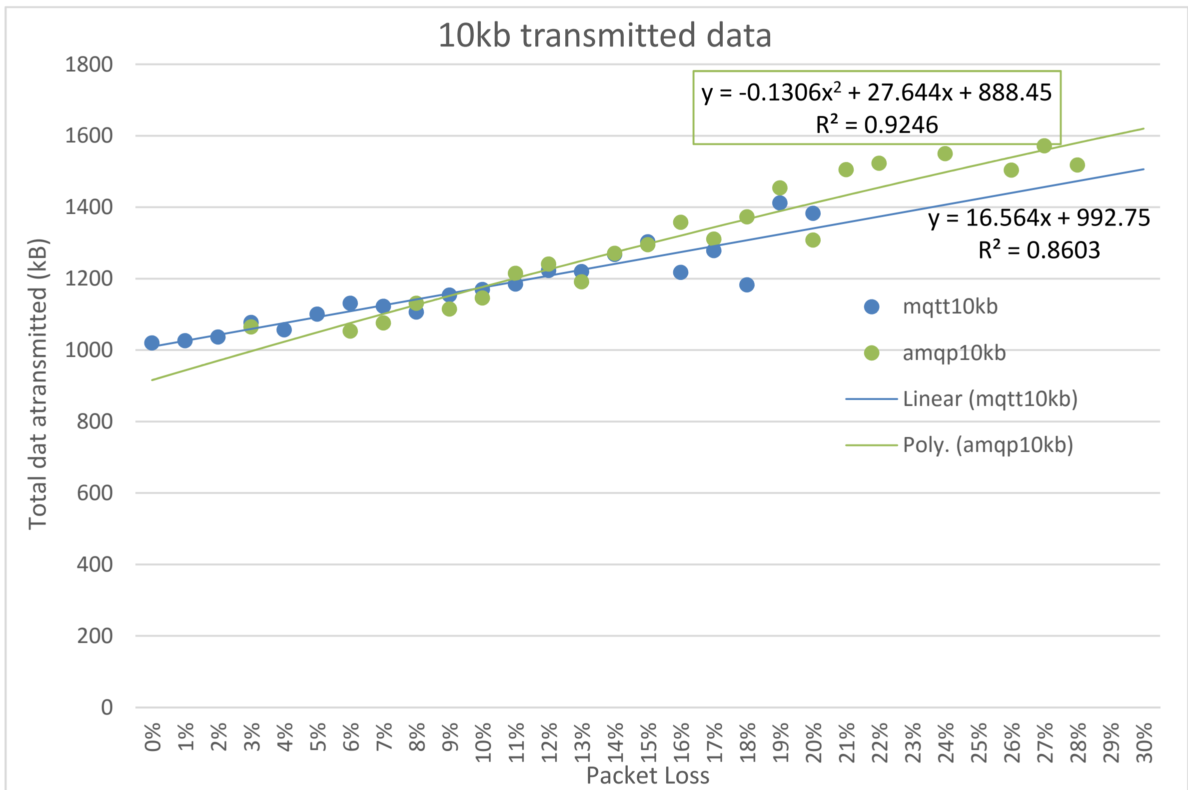

Figure 5-9: **Total transmitted data for 1MB using 1kB payloads and QoS 1**

Figure 5-10: **Execution time for 1MB data, 10kB payload, QoS 1**

5.2 Reliability and validity

As discussed in Section 3.5.1 the reliability of the tests are very good since the conditions can be controlled very precisely. However, the validity suffers for the same reasons that the reliability is good. Since we have a virtual environment communicating over the loop-back interface it is quite different from the case when using the C300 module communicating over a GPRS link. Unfortunately, we want to evaluate the protocols suitability for the C300 module, but these tests sacrificed validity for reliability. On the other hand, these provide Scania with better data about how the protocols behave when the packet loss rate varies. It would have been difficult to precisely control the packet loss rate when running from a real truck as the packet loss rate would vary with both the position and the velocity of the truck. Another factor that influences the validity of the tests is that the testing frameworks are custom built. Despite being small programs it is out of the scope for this thesis project to tweak and test the programs to see if there are settings and functionality that the protocols provide that could improve their performance with a different test program. This is especially true for the MQTT-SN protocol where the documentation was scarce and the programmer has to set up a lot of the functionality on his own.

Another thing that impacts the validity of the results is the way that the transmissions are done with regards to packet fragmentation and pushing. For TCP, every transmission issues a PUSH request to immediately transmit the packet. While this might not be an unreasonable thing to do when assuming that the payload is small enough to fit inside the MTU and the user want the message to be transmitted as soon as possible (as would be the case for the 1kB payloads in 24) it would not be the same if the payload was 10kB. This payload would need to be fragmented, but it would not need a new TCP header for every fragment. Thus, if we assume that the 1MB payload sent in the tests is 1000 packets with a payload of 1kB that would have been pushed, then the results are valid. If we however assume that the 1MB payload sent would have been 10 messages with a payload of 100kB then the results are not entirely accurate since in a real world scenario multiple packets would have been able to fit under the same TCP header. This holds true for UDP as well where a single UDP header could have been used to transmit a 10kB payload fragmented into multiple packets.

5.3 Discussion

Most of the results were pretty unsurprising and as expected. The result that was not expected was that the AMQP protocol was able to keep up as well as it did, despite having a lot larger headers even when the payload size was small.

It was unfortunate that the larger packet size did not work for the case of MQTT-SN, since that skewed the results for the MQTT-SN protocol on the 1kB test. Moreover, 1 kB is by far the most common packet size for the Scania traffic. However, I believe that it is evident from the 100B tests that MQTT-SN most likely would have outperformed the other protocols with regard to the total number of bytes transmitted if the larger payload sizes could have been utilized. However, as with the other two protocols the advantage of smaller headers diminishes as the payload size grows; thus while it would have been better, this still would not have been as large a difference as it was for the 100B tests.

An interesting point to look at to determine how well these protocols are suited for Scania is to see how they perform under the average conditions discussed in Section 2.9.6, i.e. when the packet loss rate is about 4% and the packet overhead is about 14%. Since the most commonly transmitted payload is 1kB (about 98% of the payloads) the corresponding overhead for each of these protocols would be most interesting. However, as mentioned multiple times before, the MQTT-SN protocol experiences a lot more overhead due to the fragmentation at the application layer. As the results currently stand AMQP has an overhead, when transmitting 1MB data as 1kB payloads, of about 34%

and MQTT has an overhead of about 32% according to the data from Figure 5-6. According to the same data, MQTT-SN, has an overhead of 43%, which is a lot higher than the other two protocols. However, when comparing the 100B data payload, AMQP introduces about 389% overhead, MQTT around 366%, and MQTT-SN about 217%. In this test MQTT-SN does a lot better than the other two protocols which leads me to believe that if the larger packet size had been used the UDP based MQTT-SN would have beaten the other protocols by a wide margin with regards to the total number of transmitted bytes.

Another way to look at this is that if the larger packet size could have been used, then the percentage of overhead can be calculated with the following formula

$$Overhead = 1 - \frac{TotalData}{\left(\frac{PacketsFromClient}{4}\right) * (PayloadSize + HeaderSize + 1) + \left(\frac{PacketsFromServer}{4}\right) * (ACKSize)}$$

Equation 2 MQTT-SN data overhead calculation formula

Since the header size is the same for both directions except for an extra octet for the length field, and the 1kB payload would fit both with regards to the application layer protocol and the MTU, then the formula enables us to calculate the overhead. Using the data from Figure 5-8 and Figure 5-9 along with the header size from Figure 4-1 for MQTT-SN the corrected overhead data for 4% packet loss and 1kB payloads with have a total byte overhead of approximately 193kB, which is roughly 19% for a 1MB transfer. This is a better than both the other tested protocols and also a better than the SCPv2 protocol considering that the 14% overhead mentioned in Section 4.1 only takes the application level headers into account.

6 Conclusions and Future work

This chapter states the conclusions that I have drawn based on the analysis performed in Chapter 5. The limitations that were faced will be addressed in Section 6.2, while Section 6.3 describes both work that was left undone and work that should/could be done as future work. The chapter concludes some reflections about the impact this work may have.

6.1 Conclusions

While the final goals for this thesis project were met, these goals changed a bit along the way. Therefore, it is a bit of a stretch to state that the goals met. Originally, the idea was to first create a virtual environment and test the protocols there. Then utilize these results as a benchmark for tests under real conditions and running on the real hardware (i.e., a client for each protocol running on Scania's C300 module) while communicating with a broker running on one of Scania's servers. Moreover, it was desired that the protocols be evaluated in terms of their transmission speed, response time, and the total amount of data sent over the network. While this idea at first seemed plausible, the more time that was spent on developing the applications the more out of scope this original idea for testing seemed. So after discussing the situation with my supervisors, it was concluded that the most interesting investigation was how to reduce the number of bytes transmitted, hence the scope of the project was changed and the goals updated.

With respect to achieving these updated goals the biggest problem faced was that the specification for AMQP is so huge that getting an overview of all the parameters that might influence the results was difficult. At the same time the MQTT-SN protocol had very scarce documentation. Moreover, the official IBM/Mosquitto supplied broker only supported QoS levels of -1 and 0 which resulted in the need to set up a custom PUB/ACK procedure for the client to achieve QoS 1.

Getting the stochastic packet filtering to work was also a bit of a challenge. The intuitive choice was to use netem to emulate different conditions and then log each protocol's behaviour. However, the netem utility would randomly crash or fail to set up the rules properly. This lead to the use of iptables to set up random packet drops. While iptables is not traditionally used to emulate lossy networks, it did supply the requisite functionality to randomly drop packets. Since this was what we needed and it worked well, this solution ended up being s used.

Since lots of time was spent doing superfluous things, such as C300 implementations of each of the protocols the time spent implementing the actual test applications was less than would have been desirable. The time would have been better spent verifying the correctness of the applications, speeding them up, and performing more tests. So if the thesis project were to be done over again the focus would be set earlier and be narrower than it was to begin with in this project.

From the results that were collected, despite the poor performance on the tests were the payload had to be split, MQTT-SN appears to have the potential to be the lightest protocol (of the three) for transmitting data. However, since the MQTT-SN protocol is far from being plug-and-play the number of bytes transmitted depends upon the developer handling connections, registers, and ACKs properly; otherwise, the end result could be worse than the other protocols (as exemplified when the smaller payload size was used). MQTT-SN also has the potential to be integrated with existing cloud solutions which Scania has expressed interest in, since MQTT-SN could be feed through a gateway making it into MQTT messages.

6.2 Limitations

The biggest limitation faced regarding the implementations was without a doubt the complexity of the system that the protocols were to be evaluated for. Underestimating the effort required for a standalone protocol implementation meant that a lot of time was spent on setting up and investigating solutions that were never tested. This combined with the fact that the documentation for the MQTT-SN variant was very limited and that the scope of the AMQP protocol was quite huge meant that a lot of effort went into developing the test applications instead of running the tests.

With regard to results, one of the major limitations was that no data was generated from an actual Scania vehicle. While the generated data is reliable, having some data generated by transmissions from a Scania vehicle to a server would have been nice to complement the experimental data from the emulated test environment. As of now, running over the loop-back interface of the virtual machine it becomes obvious what effect the packet loss rate has on the performance of the protocols; however, for Scania it would have been interesting to know how the protocols behave in the context of a C300 module transmitting data to a Scania server.

The time it took to perform the tests was also a limiting factor. For the first round of tests a 1 second wait time was used to ensure no erroneous retransmissions occurred. This was OK for the TCP based protocols where TCP seemed to take care to the retransmissions, thus the 1 second wait time was rarely needed. However, for the UDP based MQTT-SN this proved to be an extremely long time to wait when you actually need to perform a lot of packet retransmissions. This was amplified by the fact that the smaller packet size resulted in even more packets. Fortunately, this testing was running while other work was being done, thus it did not block any other tasks from being completed. However, when these tests finished it became apparent that some scenarios had failed, so the time between retransmissions was reduced to 10 ms to allow for quicker testing. While the MQTT-SN tests did go about 20% faster they were still way to slow to perform a lot of tests with. The total transmission times for sending 1 MB of data using a 1 kB payload (with 10 ms retransmission time for MQTT-SN) as a function of packet loss rate are shown in Figure 6-1.

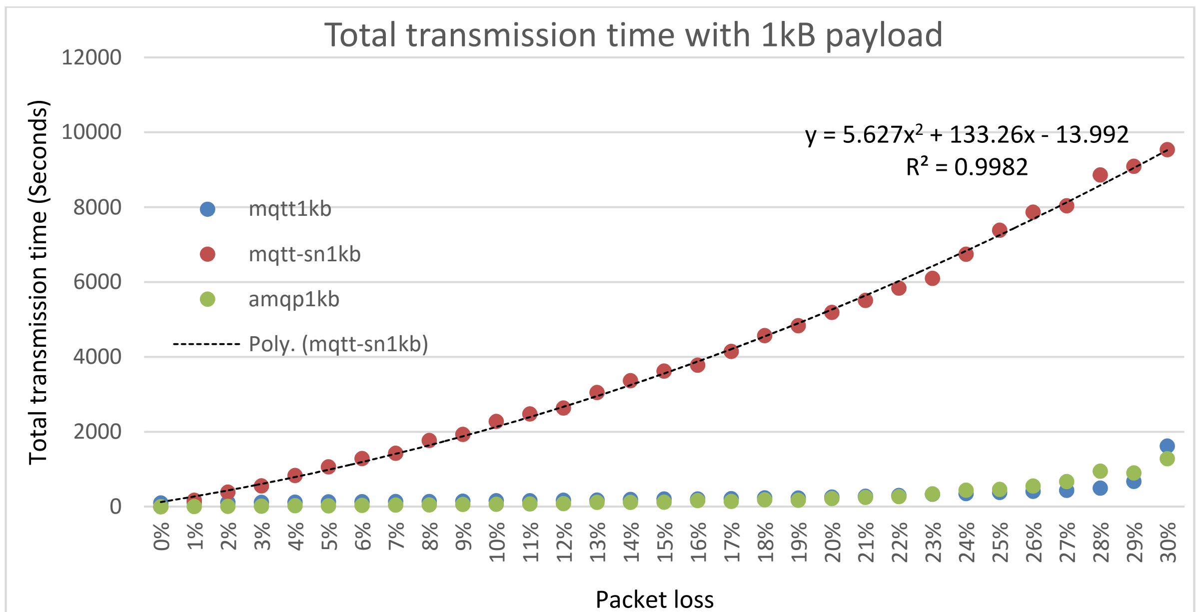

Figure 6-1: **Execution time for 1MB data, 1kB payload, QoS 1 with 10 ms retransmission time for MQTT-SN**

It should be noted that the reason why the TCP based are so much faster is most likely due to the fact that since we are running on the loopback interface the response time is extremely low. This will cause TCP to configure the RTO to a very low value resulting in very quick retransmissions when the packet is dropped. The UDP based MQTT-SN on the other hand gets no such protocol configurations and thus needs to wait for the application layer to perform the resend.

6.3 Future work

The suggested future work ranges from tasks that were planned, but left undone, to suggestions for further research that would build upon insights gained doing this thesis project.

6.3.1 In vehicle testing

The most obvious thing that was left undone and what would be the natural next step is to test the protocols from inside an actual Scania vehicle. This would provide relevant data for Scania to base their protocol implementation decisions on. To perform this test the current communication stack of the C300 module would need to be replaced and the Scania server would need to run the broker corresponding to the protocol used.

6.3.2 More protocols and brokers

While the protocols that Scania has expressed the most interest in have been evaluated in this thesis project there are many additional protocols that would/could offer features that might be of interest to Scania. One example is the FASP protocol discussed in Section 2.9.4. While the experiments of this thesis focused mainly on multiple smaller transfers over a unreliable link (because these transfer currently they make up the bulk of the traffic), there will still need to be OTA updates. These updates can often be multiple MB in size and will need to be transferred over the same lossy network as the other communication between the vehicle and Scania's server. FASP would be an ideal solution since it is designed to transfer large file over networks with high packet loss rates.

Another protocol that was discussed is CoAP. Since this protocol was also designed to transmit very little data over the network it could very well prove to be a nice fit for Scania. This protocol is also designed to easily translate to HTTP which means it could be integrated with existing web solutions if the need for that would arise.

Finally, evaluating additional brokers could be of interest. However, since the protocols specify how the transmitted data needs to be packetized a change in broker would not affect the number of bytes transmitted (other than in the case of the broken MQTT-SN broker). However, other brokers could be of interest for other metrics which Scania is interested in, such as response time and transmission speed.

6.3.3 Encryption

One of the things that a protocol developer will need to look into is the type of encryption that is used. It would be interesting to evaluate how well different types of encryption would work in networks with very high packet loss rates. For example, TLS would need to do handshaking procedures to negotiate a certificate, but with high packet loss rates this might take a very large number of retransmissions to successfully complete. An obvious topic to look at is stream based ciphers as opposed to block ciphers, for example the Espresso cipher for IoT devices [43] as this would negate the need to introduce unnecessary encryption data overhead.

6.3.4 Scalability tests

For this evaluation we had one publisher running with one subscriber connected. This is not a very realistic scenario and it would have been interesting to see how the protocols behave as the number of connections per broker increases. However, this is would be as much a broker and publisher implementation evaluation as it would be a protocol evaluation, hence for this thesis such testing was out of scope.

6.3.5 IPv6 and header compression

For the current scenarios no form of compression was used. While more exact data about the payload would be needed to evaluate how much of an impact header compression has on the overhead. One example of an interesting compression form that could be investigated in the future when IPv6 is adopted is the 6LoWPAN compression or the ROHC/ROHCv2 algorithms described in section 2.6.

6.4 Required reflections

As the fleet of Scania vehicles grows so will the need for reducing the number of bytes transmitted. While the amount of data being sent right does not require a new protocol, a new protocol will be needed to handle the aggregated date of an increased number of trucks, especially as this number is expected to continue to grow. Not only is the number of vehicles increasing but so is the number of services that are offered. When coupled with the advent of autonomous vehicles this creates real economic incentives to minimize the amount of data traffic in order to allow for the greatest possible scalability. A scalable system will also mean that more data can be transmitted and acted on with low delay as compared to if the data would need to be offloaded when the vehicles are connected to a static network. This could allow for real time detection of failing systems, information about obstacles in the traffic and weather conditions among other things.

Being able to offer more services might also mean more efficient vehicles and more efficient vehicles means less carbon (and other) emissions from the engines. Since both the environment and the customer benefit from reduced fuel consumption this might be a win for all parties involved.

References

[1] Scania, "www.scania.se," 23 December 2014. [Online]. Available: http://www.scania.se/Images/100_000_uppkopplade_lastbilar_tcm85-454469.pdf. [Accessed 09 02 2016].

[2] Volkswagen, "Scania has 170,000 connected vehicles," 26 January 2016. [Online]. Available: http://www.volkswagenag.com/content/vwcorp/info_center/en/news/2016/01/connected_vehicles.html. [Accessed 09 02 2016].

[3] OASIS, "MQTT Version 3.1.1.," 29 October 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. [Accessed 18 February 2016].

[4] OASIS, "AMQP 1.0 specification," 29 October 2012. [Online]. Available: http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf. [Accessed 15 February 2016].

[5] A. Standford-Clark, H. L. Truong, "MQTT-SN 1.2 protocol specification," International Business Machines Corporation (IBM), 14 November 2013. [Online]. Available: http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf. [Accessed 4 May 2016].

[6] Z. Shelby, K. Hartke, C. Bormann, *The Constrained Application Protocol (CoAP),* vol. RFC 7252 (Standards Track), 2014.

[7] H. Garcia-Molina, Y. Huang, "Exactly-once semantics in a replicated messaging system," in *17th International Conference on Data Engineering, 2001. Proceedings*, 2001.

[8] J. Postel, *User Datagram Protocol,* vol. RFC 768 (Internet Standard), 1980.

[9] J. Postel, *Transmission Control Protocol,* vol. RFC 793 (Internet Standard), 1981.

[10] G. Bochmann, [Online]. Available: http://www.site.uottawa.ca/~bochmann/CourseModules/NetworkQoS/TCP-congestion-control.jpg. [Accessed 16 February 2016].

[11] S. Ha, I. Rhee, L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS Operating System Review,* vol. 42, no. 5, pp. 64-74, July 2008.

[12] K. Harfoush, I. Rhee, L. Xu, "Binary Increase Congestion Control for Fast, Long Distance Networks," in *Proceedings of IEEE INFOCOM*, Hong Kong, March, 2004.

[13] P. D. Amer, R. Stewart, "Why is SCTP needed given TCP and UDP are widely available?," The Internet Society, June 2004. [Online]. Available: https://www.isoc.org/briefings/017/briefing17.pdf. [Accessed 17 June 2016].

[14] F. Baker, P. Savola, *Ingress Filtering for Multihomed Networks,* vol. RFC 3704 (Best Current Practice), 2004.

[15] P. TH. Eugster, P. A. Felber, R. Guerraoui, A. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys (CSUR),* vol. 35, no. 2, pp. 114-131, June, 2003.

[16] E. Curry, S. Hasan , S. O'Riain, , "Approximate Semantic Matching of Heterogeneous Events," in *6th ACM International Conference on Distributed (DEBS 2012)*, Berlin, 2012.

[17] G. Hohpe, in *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, p. 147.

[18] OASIS, “MQTT version 3.1.1 becomes an OASIS standard,” 29 October 2014. [Online]. Available: https://www.oasis-open.org/news/announcements/mqtt-version-3-1-1-becomes-an-oasis-standard. [Accessed 18 February 2016].

[19] J. Barr, “AWS IoT - Cloud Services for Connected Devices,” Amazon.com, Inc., 8 October 2015. [Online]. Available: https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/. [Accessed 18 February 2016].

[20] L. Zhang , “Building Facebook messenger,” Facebook, Inc., 12 August 2011. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920. [Accessed 18 February 2016].

[21] U. Hunkeler, H. L. Truong, A. Stanford-Clark, “MQTT-S: A publish/subscribe protocol for Wireless Sensor Networks,” in *in 3rd International Conference on Communication Systems Software and Middleware and Workshops*, 2008.

[22] L.-E Jonsson, G. Pelletier, K. Sandlund, M. West, *RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP),* vol. RFC 6846 (Proposed Standard), 2013.

[23] G. Pelletier, K. Sandlund, *RObust Header Compression Version 2 (ROHCv2): Profiles for RTP, UDP, IP, ESP and UDP-Lite,* vol. RFC 5225 (Proposed Standard), 2008.

[24] C. Bormann, *6LoWPAN-GHC: Generic Header Compression for IPv6,* vol. RFC 7400 (Standards Track), 2014.

[25] A. Castellani, E. Dijk, T. Fossati, S. Loreto, A. Rahman, “Guideline for HTTP-CoAP Mapping Implementations,” CoRE Working Group, 3 July 2015. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-core-http-mapping-07.txt. [Accessed 23 February 2016].

[26] J O'Hara, “Toward a Commodity Enterprise Middleware,” *ACM Queue,* vol. 5, no. 4, pp. 48-55, May/June 2007.

[27] A. Melnikov, K. Zeilenga, *Simple Authentication and Security Layer (SASL),* vol. RFC4422 (Standards Track), 2006.

[28] T. Dierks, E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2,* vol. RFC 5246 (Standards Track), 2008.

[29] AGPM Standard, Advanced Message Queuing Protocol Specification v0-9-1, 2008.

[30] OASIS, “MQTT servers and brokers,” 28 February 2016. [Online]. Available: https://github.com/mqtt/mqtt.github.io/wiki/servers. [Accessed 1 March 2016].

[31] Pivotal Software, Inc., “RabbitMQ,” 2016. [Online]. Available: https://www.rabbitmq.com/. [Accessed 1 March 2016].

[32] The Eclipse Foundation, “Mosquitto, An open source MQTT broker,” [Online]. Available: http://www.mosquitto.com. [Accessed 22 April 2016].

[33] P. Boronat, C. Calafate, J. C. Cano, J. E. Luzuriaga, P. Manzoni, M. Perez, , “A comparative evaluation of AMQP and MQTT over unstable and mobile networks,” in *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, Las Vegas, NV, 2015.

[34] X. Ma, A. Valera, H. Tan, D. Thangavel , “Performance evaluation of MQTT and CoAP via a common middleware,” in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, Singapore, 21-24 April 2014.

[35] Aspera, Inc., “FASP overview,” Aspera, [Online]. Available: http://asperasoft.com/technology/transport/fasp/#overview-464. [Accessed 2 March 2016].

[36] E. Dahlman, G. Mildh, S. Parkvall, J. Peisa, J. Sachs, Y. Selén, *5G radio access,* Vols. 91, 2014, D. P. Doyle, Ed., U f Ewaldsson, Ericsson Review, 18 June 2014, p. 2, https://www.ericsson.com/res/thecompany/docs/publications/ericsson_review/2014/er-5g-radio-access.pdf .

[37] P. Hagernäs , "5G user satisfaction enabled by FASP : Evaluating the performance of Aspera's FASP," Master's thesis, KTH Royal Institute of Technology, Stockholm, TRITA-ICT-EX-2015:199, August 2015, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-172760 .

[38] V. Johansson, "Enhancing user satisfaction in 5G networks using Network Coding," Master's thesis, KTH Royal Institute of Technology, Stockholm, TRITA-ICT-EX-2015:178, July 2015, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-171210 ..

[39] J. J. Ziarno, "Wireless engine monitoring system with multiple hop aircraft communications capability and on-board processing of engine data". United States Patent US20140114549 A1, 24 April 2014.

[40 D. Tweney, *Here comes the industrial Internet — and enormous amounts of data,* VentureBeat, 2015.

[41] A. Antonuk, "rabbitmq-c repository," Pivotal Software, [Online]. Available: https://github.com/alanxz/rabbitmq-c. [Accessed 22 March 2016].

[42] The Eclipse Foundation, "Paho project," [Online]. Available: http://www.eclipse.org/paho/. [Accessed 22 March 2016].

[43] E. Dubrova, M. Hell, "Espresso: A Stream Cipher for 5G Wireless," Cryptology ePrint Archive, 2015.

[44] Pivotal Software, Inc., "Which protocols does RabbitMQ support?," 2016. [Online]. Available: https://www.rabbitmq.com/protocols.html. [Accessed 1 March 2016].

[45] B. Moyer, "All About Messaging Protocols," Techfocus media, inc., 20 April 2015. [Online]. Available: http://www.eejournal.com/archives/articles/20150420-protocols/. [Accessed 23 February 2016].

[46] A. Piper, "MQTT for Sensor Networks – MQTT-SN," 2 December 2013. [Online]. Available: http://mqtt.org/2013/12/mqtt-for-sensor-networks-mqtt-sn.

Appendix A: Port configurations

Port number	Packet loss
10001	0%
10002	1%
10003	2%
10004	3%
10005	4%
10006	5%
10007	6%
10008	7%
10009	8%
10010	9%
10011	10%
10012	11%
10013	12%
10014	13%
10015	14%
10016	15%
10017	16%
10018	17%
10019	18%
10020	19%
10021	20%
10022	21%
10023	22%
10024	23%
10025	24%
10026	25%
10027	26%
10028	27%
10029	28%
10030	29%
10031	30%
10032	0%
10033	1%
10034	2%
10035	3%
10036	4%
10037	5%
10038	6%
10039	7%
10040	8%
10041	9%
10042	10%

10043	11%
10044	12%
10045	13%
10046	14%
10047	15%
10048	16%
10049	17%
10050	18%
10051	19%
10052	20%
10053	21%
10054	22%
10055	23%
10056	24%
10057	25%
10058	26%
10059	27%
10060	28%
10061	29%
10062	30%
10063	0%
10064	1%
10065	2%
10066	3%
10067	4%
10068	5%
10069	6%
10070	7%
10071	8%
10072	9%
10073	10%
10074	11%
10075	12%
10076	13%
10077	14%
10078	15%
10079	16%
10080	17%
10081	18%
10082	19%
10083	20%
10084	21%
10085	22%
10086	23%
10087	24%

10088	25%
10089	26%
10090	27%
10091	28%
10092	29%
10093	30%
10094	0%
10095	1%
10096	2%
10097	3%
10098	4%
10099	5%
10100	6%
10101	7%
10102	8%
10103	9%
10104	10%
10105	11%
10106	12%
10107	13%
10108	14%
10109	15%
10110	16%
10111	17%
10112	18%
10113	19%
10114	20%
10115	21%
10116	22%
10117	23%
10118	24%
10119	25%
10120	26%
10121	27%
10122	28%
10123	29%
10124	30%
10125	0%
10126	1%
10127	2%
10128	3%
10129	4%
10130	5%
10131	6%
10132	7%

10133	8%
10134	9%
10135	10%
10136	11%
10137	12%
10138	13%
10139	14%
10140	15%
10141	16%
10142	17%
10143	18%
10144	19%
10145	20%
10146	21%
10147	22%
10148	23%
10149	24%
10150	25%
10151	26%
10152	27%
10153	28%
10154	29%
10155	30%
10156	0%
10157	1%
10158	2%
10159	3%
10160	4%
10161	5%
10162	6%
10163	7%
10164	8%
10165	9%
10166	10%
10167	11%
10168	12%
10169	13%
10170	14%
10171	15%
10172	16%
10173	17%
10174	18%
10175	19%
10176	20%
10177	21%

10178	22%
10179	23%
10180	24%
10181	25%
10182	26%
10183	27%
10184	28%
10185	29%
10186	30%
10187	0%
10188	1%
10189	2%
10190	3%
10191	4%
10192	5%
10193	6%
10194	7%
10195	8%
10196	9%
10197	10%
10198	11%
10199	12%
10200	13%
10201	14%
10202	15%
10203	16%
10204	17%
10205	18%
10206	19%
10207	20%
10208	21%
10209	22%
10210	23%
10211	24%
10212	25%
10213	26%
10214	27%
10215	28%
10216	29%
10217	30%
10218	0%
10219	1%
10220	2%
10221	3%
10222	4%

10223	5%
10224	6%
10225	7%
10226	8%
10227	9%
10228	10%
10229	11%
10230	12%
10231	13%
10232	14%
10233	15%
10234	16%
10235	17%
10236	18%
10237	19%
10238	20%
10239	21%
10240	22%
10241	23%
10242	24%
10243	25%
10244	26%
10245	27%
10246	28%
10247	29%
10248	30%
10249	0%
10250	1%
10251	2%
10252	3%
10253	4%
10254	5%
10255	6%
10256	7%
10257	8%
10258	9%
10259	10%
10260	11%
10261	12%
10262	13%
10263	14%
10264	15%
10265	16%
10266	17%
10267	18%

10268	19%
10269	20%
10270	21%
10271	22%
10272	23%
10273	24%
10274	25%
10275	26%
10276	27%
10277	28%
10278	29%
10279	30%

Appendix B: Automated Scripts

Scenario creator script

```
#!/bin/bash

base_dir=$1

outfile_cli=$base_dir/cli_scen
outfile_srv=$base_dir/srv_scen

cli_prog=( $base_dir/testframework/producers/ThesisMQTT
$base_dir/testframework/producers/ThesisMQTTSN
$base_dir/testframework/producers/ThesisAMQP ) #"$base_dir/programs/test_hello_cli" )
#cli_prog=($base_dir/testframework/producers/ThesisMQTTSN ) #"$base_dir/programs/
srv_prog=( "MQTT" "MQTTSN" "AMQP") #"$base_dir/programs/test_hello_srv" )  #("MQTT"
"AMQP" "COAP" "SCPv2")
#srv_prog=("MQTTSN") #"$base_dir/programs/test_hello_srv" )  #("MQTT" "AMQP"
type=("a" ) #"pingpong")

for i in ${programs[@]};
do
       echo "$i"
done

echo


packet_loss_scale=0.5
packet_loss="."
#for i in $(seq -10 2 -2);
for i in $(seq 0 1 30)
do
       echo $i
       #packet_loss+="$(echo "1.0/(1+e($i*-1))" | bc -l ) "
       packet_loss+="$(echo "$i/100" | bc -l ) "

done
latency_scale=0.5
latency="0 "
for i in $(seq -1 0.5 4);
do
       echo $i
       latency+="$(echo "(e($i))" | bc -l ) "
done
size_scale=0.5
size=""
for i in $(seq 2 1 4);
do
       echo $i
       size+="$(echo 10^"(($i))" | bc -l ) "
done

echo ${programs[@]}
echo ${packet_loss[@]}
echo ${latency[@]}

rm $outfile_cli
for p in ${cli_prog[@]}
do
       for t in ${type[@]}
       do
```

```
        for x in $packet_loss
        do
            for s in $size
            do
                echo "$p $t $x 0 $s" >> $outfile_cli
            done
        done
    done
done

rm $outfile_srv
for p in ${srv_prog[@]}
do
    for t in ${type[@]}
    do
        for x in $packet_loss
        do
            for s in $size
            do
                echo "$p $t $x 0 $s" >> $outfile_srv
            done
        done
    done
done
```

client run script

```
#!/bin/bash
echo $1 $2 $3


interface=lo

#$1 scenario_file  $2 other node
srv_addr="127.0.0.1"
k=1
base_dir=$1
echo "hello $2"
echo $PATH
echo $USER
filters=
fields="-T fields -e frame.time_relative -e frame.len  -e ipv6.dst -e ip.dst -e
tcp.dstport -e udp.dstport -e icmpv6.echo.identifier -e ipv6.src -e ip.src -e
tcp.srcport -e udp.srcport -e icmpv6.echo.sequence_number -e tcp"

ip link set dev $interface mtu 1500

export LD_LIBRARY_PATH=export
LD_LIBRARY_PATH=/home/karlstri/programing/exjobb/testframework/producers

/sbin/tc qdisc replace dev $interface root netem loss 0
#start tshark
#tshark -i $interface $fields $filters > $base_dir/packetlog/$3/p &

sleep 20
while read -r proto typ loss latency size file
do
	tshark -i $interface -w $base_dir/packetlog/cli_log$k.pcap &
	tshark_pid=$!
	((portno=k+10000))
	percent_loss=0"$(echo "$loss*100" | bc -l ) "   #convert prob to procent
	#set netem
	chain_name=PROBDROP${loss:0:5}

	iptables -L $chain_name > /dev/null

	if [ $? -eq 1 ];
	then
	echo -e "no chain, adding chain $loss\n"
	iptables -N $chain_name
	iptables -A $chain_name -m statistic --mode random --probability $loss  -j DROP

	fi


	iptables -A INPUT -p tcp --source-port $portno -j $chain_name
	iptables -A INPUT -p udp --source-port $portno -j $chain_name

	#/sbin/tc qdisc replace dev $interface root netem loss $percent_loss delay
"$latency"ms
	#sleep 1
	maxsize=1000000
	total_size=1000000
	reruns="$(echo "$total_size"/"$size"  | bc ) "
	num_pack=1
	pack_size=$size
	if [ "$proto" = "$base_dir/testframework/producers/ThesisMQTTSN" ]
	then
```

```
		maxsize=255
		if (( $size > $maxsize ))
		then
			#echo -e "\tspliting\t$size\t$maxsize\n"
			num_pack="$(echo "$size"/"$maxsize" +1 | bc ) "
			#echo -e "\tspliting\t$num_pack\n"
			pack_size="$(echo "$size"/"$num_pack" | bc ) "
			#echo -e "\tspliting\t$pack_size\n"
		fi
	fi
	echo $k
	for i in $(seq 1 1 1)
	do
		exit_code=1
		while [ $exit_code -ne 0 ]
		do
			#syntax: prog host portno mode
#			echo "$proto $srv_addr $portno $reruns $num_pack 10 $size 1 "
			$proto $srv_addr $portno $reruns $num_pack 10 $size 1   >
/dev/null
			exit_code=$?
		done



	done
	#sleep 0.3
	kill  $tshark_pid
	((k+=1)) #incremetn counter
#tc qdisc delete dev $interface root netem
done <$2
sleep 2
#clean
$base_dir/programs/test_hello_cli ::1 9999 > /dev/null

/sbin/tc qdisc delete dev $interface root netem
$base_dir/packet_separator.sh $base_dir $base_dir/packetlog/$3/p $3
```

Execution script

```
#!/bin/bash
#set vars
usr=karlstri
node1=::1
node2=::1

base_dir="/home/$usr/programing/exjobb"

ctl_node=::1
node1_ctl=10.0.2.15
node2_ctl=10.0.2.15

scenario_file_cli="$base_dir/cli_scen"
scenario_file_srv="$base_dir/srv_scen"
#init
reset
echo -e "\t$base_dir\t"

$base_dir/scenariocreator.sh $base_dir
sudo /sbin/tc qdisc replace dev lo root netem loss 0 delay 0ms

#make each node have the requierd stuff

./resource_copyier.sh $base_dir root@$node1 $base_dir/temp
./resource_copyier.sh $base_dir root@$node2 $base_dir/temp

#clean previus tests

ssh root@$node1 -f  "$base_dir/prep_srv.sh" $base_dir $node2
ssh root@$node2 -f  "$base_dir/prep_cli.sh" $base_dir $node1

#run the test

ssh root@$node1_ctl -f $base_dir/test_srv.sh $base_dir $scenario_file_srv srv $node2
ssh root@$node2_ctl -f $base_dir/test_cli.sh $base_dir $scenario_file_cli cli $node1
```

Server run script

```
#!/bin/bash
echo $1 $2 $3




iptables -F
iptables -X

interface=lo

#$1 scenario_file  $2 other node
k=1
base_dir=$1
echo "hello $2"
echo $PATH
echo $USER
filters=
fields="-T fields -e frame.time_relative -e frame.len  -e ipv6.dst -e ip.dst -e
tcp.dstport -e udp.dstport -e icmpv6.echo.identifier -e ipv6.src -e ip.src -e
tcp.srcport -e udp.srcport -e icmpv6.echo.sequence_number -e tcp"

AMQP_broker=rabbitmq-server
MQTT_broker=mosquitto
MQTTSN_broker=$base_dir/org.eclipse.mosquitto.rsmb/rsmb/src/broker_mqtts


export LD_LIBRARY_PATH=export
LD_LIBRARY_PATH=/home/karlstri/programing/exjobb/testframework/producers

ip link set dev $interface mtu 1500

killall test_hello_srv

#start tshark
#tshark -i $interface $fields $filters > $base_dir/packetlog/$3/p &
tshark -i $interface -w $base_dir/packetlog/srv_log.pcap &
tshark_pid=$!

       #export RABBITMQ_NODENAME=rabbit
       #export RABBITMQ_NODE_PORT=5469
       #rabbitmq-server
       #pid=$!
       #echo -e "$pid\n" > $base_dir/pidfile

while read -r proto typ loss latency file
do
       percent_loss="$(echo "$loss*100" | bc -l ) "    #convert prob to procent
       ((portno=k+10000))
       echo $k
       chain_name=PROBDROP${loss:0:5}

       iptables -L $chain_name > /dev/null

       if [ $? -eq 1 ];
       then
       echo -e "no chain, adding chain $loss\n"
       iptables -N $chain_name
       iptables -A $chain_name -m statistic --mode random --probability $loss  -j DROP

       fi
```

```
        iptables -A INPUT -p tcp --destination-port $portno -j $chain_name
        iptables -A INPUT -p udp --destination-port $portno -j $chain_name



        echo $proto
        if [ "$proto" = "AMQP" ]
        then
               echo "AMQP"
               export RABBITMQ_NODE_PORT=$portno
               export RABBITMQ_NODENAME=node$portno
               $AMQP_broker  -detached > /dev/null &
               pid=$!
               echo -e "$pid" >> $base_dir/pidfile
        fi
        if [ "$proto" = "MQTT" ]
        then
               echo "MQTT"
               $MQTT_broker -p $portno  > /dev/null &
               pid=$!
               echo -e "$pid" >> $base_dir/pidfile
        fi
        #mqttsn (rsmb)
        if [ "$proto" = "MQTTSN" ]
        then
               echo "MQTT-sn"
               rsmb_conf="$base_dir/confs/m$k.conf"
               echo -e "listener $portno INADDR_ANY mqtts " > $rsmb_conf
               $MQTTSN_broker $rsmb_conf  > /dev/null &
               pid=$!
               echo -e "$pid" >> $base_dir/pidfile
        fi

        ((k+=1)) #incremetn  counter
#tc qdisc delete dev $interface root netem
done <$2
#sleep 20
#clean
echo -e "-cleaning up"
$base_dir/programs/test_hello_srv ::1 9999 > /dev/null
kill  $tshark_pid
#ip6tables -L
iptables -F
iptables -X

$base_dir/packet_separator.sh $base_dir $base_dir/packetlog/$3/p $3
#rabbitmqctl stop
#killall rabbitmq-server -s9
#killall beam -s9
#killall inet_gethost -s9
sleep 10
.base_dir/abort.sh
```

Abort script

```
#!/bin/bash

sudo killall test_srv.sh
sudo killall test_cli.sh
sudo killall test_hello_srv
sudo killall test_hello_cli
rm -f pidfile
```

Packet separator script

```
#!/bin/bash

#assume packetes are udp/tcp
#assume port 22 are not used by anything but ssh
#assume each test have a unique set of src and dst port

echo "separating"

packet_file=$2
k=1
sshport=22
sshpactes=0
outfile=$1/packetlog/$3/s

#echo $1 $2 $3 $outfile
declare -A map;
map[22]="ssh";

old_src_port=0
old_dst_port=0
while read -r  ts size dst dst_port src src_port flags  file #read from logfile
do
	if [ $dst_port  -eq 22 ] || [ $src_port -eq 22 ] #ignore ssh
	then
		continue;
	fi;

	if [ $dst_port  -eq 4369 ] || [ $src_port -eq 4369 ] #ignore erlang port maper
	then
		continue;
	fi;

	if ( test "${map[$dst_port]+isset}"  &&  test "${map[$src_port]+isset}" ) #if
previud know combo
	then
		#echo "yes $dst_port";
		:
	else #if combo is unkown make combo know
		if [ $dst_port  -eq $old_dst_port ] || [ $src_port -eq $old_dst_port ]
||[ $dst_port  -eq $old_src_port ] || [ $src_port -eq $old_src_port ]
		then
			echo "">"$outfile""$k"
			echo "restart"
			map[$dst_port]=$k
			map[$src_port]=$k
		else
			map[$dst_port]=$k
			map[$src_port]=$k
			((k++))
		fi


	fi;
	#do the loging
	#echo "$outfile""${map[$dst_port]}"
	echo $ts $size $dst $dst_port $src $src_port>>"$outfile""${map[$dst_port]}"

	old_dst_port=$dst_port
	old_src_port=$src_port
	#echo $ts $size $dst $dst_port $src $src_port #>>"$outfile""$k"
```

```
done <$packet_file
echo "done , $k files"
```

Client preparation script

```
#!/bin/bash

#
mkdir -p $1/cli_log
mkdir -p $1/packetlog/cli

export LD_LIBRARY_PATH=export
LD_LIBRARY_PATH=/home/karlstri/programing/exjobb/testframework/producers/:$LD_LIBRARY_
PATH


rm $1/cli_log/*
rm $1/packetlog/cli/*

make -C programs -B

killall local_run.sh
killall tshark
killall ping6
#sudo rabbitmqctl stop
```

```
#!/bin/bash

mkdir -p $1/srv_log
mkdir -p $1/packetlog/srv

export LD_LIBRARY_PATH=export
LD_LIBRARY_PATH=/home/karlstri/programing/exjobb/testframework/producers/:$LD_LIBRARY_
PATH


rm $1/srv_log/*
rm $1/packetlog/srv/*

make -C programs -B

killall local_run.sh
killall tshark
killall ping6
#sudo rabbitmqctl stop
```

TRITA-ICT-EX-2016:87

www.kth.se