NTNU | Faculty of Information Technology and Electrical Engineering
Department of Computer Science

FFI Norwegian Defence Research Establishment

Middleware for Constrained Networking in Military Operations

IT2901 - Informatics Project II

Bachelor's Thesis in Informatics

FFI2_middleware

Authors
Simon Doksrød
Ola Vanni Flaata
Ola Horg Jacobsen
Thorbjørn Stephan Håkon Lundin
Tobias Ringdalen Thrane

May 2023

It's the job that's never started as takes longest to finish.

J. R. R. Tolkien – *The Fellowship of the Ring*

Table of Contents

List of Figures

List of Tables

Abstract

Effective communication is crucial in military operations, involving text, telemetry, and voice and video calls for rapid information exchange. However, tactical networks face challenges that are rare in civil networks, such as low data rates, fragmented networks and potentially high packet loss.

In this project a prototype middleware that acts as an application programming interface for networking applications to communicate over sockets that was developed in collaboration with the customer. The resulting product is a Python module which aims to be a plug-and-play replacement for the standard socket library, which should allow for easy integration in existing services.

The product was tested, evaluated and tuned for specific emulated network configurations and shows higher performance, especially at low bandwidth.

While a number of challenges remain to be solved for the middleware to be viable in real situations, our work represents a step towards solving this challenge faced in tactical military networks.

Finally, the process of the development is discussed, as well as challenges and problems faced and how they were resolved.

1 Introduction

1.1 Motivation

Communication in military settings is essential for almost every aspect of tactical decision making. A commander needs to communicate about evolving situations in the field, and get updates from their soldiers. This entails using text, telemetry, and real-time voice and video calls for fast information interchange. The networks used for this communication can be unreliable and potentially low bandwidth. This creates a need for a way to communicate rapidly, in a way that efficiently utilizes the network resources while minimizing loss of information (Andersen 2022). It was with this understanding that the customer, FFI, wanted to collaborate on this project in creating a middleware service that sits between the application and the underlying transport layer protocols. This would serve as an API to be used by developers of networking applications to communicate over sockets in a configurable and predictable way.

The main reason for wanting to incorporate configurability is due to limitations of the commonly used Transmission Control Protocol (TCP). TCP is designed for reliable communication between two hosts on a packet-switched network. Its goal is to make sure data is recovered from being damaged, lost, duplicated or delivered out of order. For this it has many tools, however this project was especially concerned with how TCP used acknowledgements and retransmissions, so-called congestion control (*Transmission Control Protocol* 1981).

Acknowledgements (ACKs) are small packets that are sent to the other host to confirm that data has been received and verified to be in good shape. These are positive ACKs, there are also negative ACKs which can be used, but shall not be further discussed here. To ensure correct ordering of the delivered data, TCP uses a sequence number. The ACK confirms received good packets up to a certain sequence number. If the ACKs sent from the receiver are delayed by enough time, the transmitter will assume lost data, and retransmit (*Transmission Control Protocol* 1981).

It was from this interaction between ACKs, wait timers and retransmissions that the main problem tackled in this project originated from. In civil computer networks one may always assume a route between two endpoints exists (at all) or with a certain connection speed or latency. In military networking environments, this may not be true. Sub-networks may be disconnected altogether for a certain amount of time, as troops move in and out of communication coverage or radio shadows. Transmission equipment may be slow and/or unreliable, and total network bandwidth may be limited as in the case of narrow-band radio communication. When these network statistics fluctuate wildly or change in a short time span, one cannot guarantee that TCP will behave properly. TCP may expect a certain latency for its ACKs, but if the network situation has changed, this latency may be much larger, leading to retransmissions, when in reality it was just a delayed ACK; the data made it through just fine.

To showcase some of these problems with TCP, and confirm FFI's previous experiments, a simple test was constructed to use a dummy chat service to send approximately 100 kB of text through an unconfigured TCP stream. This test involved two computers, a Dell Latitude 3480 as a chat client (A), and an Apple MacBook Pro as the chat host (B). These computers were connected through the same router using a wireless access point. The host MacBook Pro was set up for one of the tests to emulate a slow network connection, using the Apple developer tool *Network Link Conditioner*[1]. See table 1 for results and network configuration. The first test was on the unlimited network. The second test was ran on a network limited to a *10kbps* bandwidth, a *1%* packet loss and a *100ms* delay to emulate military network conditions – a configuration close to the CNR scenario in Table 2 – as per information from FFI.

Network	**Duration**	**Data** $A \to B$	**Data** $B \to A$	**Data rate** $A \to B$	**Data rate** $B \to A$
Unlimited	$0.055s$	99.23 kB	0.823 kB	14 433 kbps	119.71 kbps
Limited	$36.86s$	30.37 kB	0.416 kB	6.591 kbps	0.090 kbps

Table 1: Results from TCP connections

[1] https://developer.apple.com/downloads/

The test on the network with unlimited connection ended up having no retransmissions or duplicate ACKs. It completed in under a tenth of a second. The test with the limited network was more interesting however. After about 30% of the data transfer being finished, the connection was cut due to a reset packet being sent from the client from a time-out. Out of the 22 packets sent, 5 were retransmitted, a retransmit rate of 22%. Compared to the network packet loss configured for 1%, this was high, leading to unnecessary network traffic. From the 7 packets sent from the host to the client, 3 were normal, and 4 were retransmissions for one of the ACKs. The team believed this to have been caused by the client trying to push 100 kB of data through the network, congesting it completely and choking the host's ability to respond in time. The client waited for too long, tried retransmitting and gave up, sending a reset packet to the host. This was a problem, as the host never went offline, it was waiting for the data through the slow link.

The project was limited to mainly examining five different network configurations that emulated existing military networking technologies. This was to keep the project highly relevant for the real world applications faced by FFI, and build upon previous work. The five network configurations are shown in Table 2.

Network	**Data Rate**	**Latency**	**Packet Loss**
CNR	10 kbps	$100ms$	5%
NATO narrow-band waveform	16 kbps	$500ms$	0%
SATCOM	250 kbps	$550ms$	0%
Tactical broadband	2 Mbps	$100ms$	1%
Low-band 5G	100 Mbps	$20ms$	0%

Table 2: Summary of the networks for which the middleware will be tuned for. The chosen configurations mirror those outlined by Andersen (2022), except that 10 kbps is used instead of 9.6 kbps and 5% packet loss instead of 1%/10% for the CNR configuration. This was done to reduce the complexity of the testing.

As students, the team members were limited by various factors such as time, other courses, and lack of experience. Due to the short time frame of a semester, the project scope was constrained, and tasks that were essential to achieving the project objectives had to be prioritized. In addition to time constraints, the members had to balance the project work with other university courses that required their attention. As this project was outside the scope of prior university courses, extra effort needed to be invested to develop the necessary skills and knowledge. Moreover, report writing could have become a hindrance to product development efforts, so the team needed to find a way to manage this effectively in order to ensure that a satisfactory product, as well as a good report, was produced.

In summary, this is an important issue in the world of military networking. Society pushes for technology where optimizations are made for speed and the focus lies on the larger scale networking of the Internet as a whole, creating rigid solutions that might not map well to every case, especially those networks that lie completely outside the Internet. Security, stability and reliability of networking in Norway's defence sector, as well as for those of our allies, are paramount in a world of continuous technological development and ever-changing warfare. The middleware that was developed as part of this project is an attempt to give a solution to this problem by taking a stance in the direction of multi-network configurable and adaptable software. The middleware was built upon decades of networking knowledge, making it easier for developers, such as FFI, to configure how the underlying networking protocols should behave, without showing the complexity of low-level networking to the end service running on-top.

1.2 Scope

This project was meant to be a proof-of-concept for a middleware system to communicate with a router that was being developed by the defence sector in collaboration with FFI. It was not meant to be a complete system ready for deployment; its purpose was to serve as a basis for research and development, for the purpose of understanding what makes a good complete system (protocols, hardware and services) to be used by militaries. The middleware will be further developed and tested by FFI after the end of the project. FFI acquired the ownership of the middleware at the end of the project. Nonetheless, no part of this project was under a non-disclosure agreement, and the source code was made public and freely available.

From the initial project description, FFI was always open about setting a realistic scope for the project, given the number of students and time span of the project. There were originally plans for two groups of students, one group working on the middleware technology itself, and one group developing services to run on-top the middleware. Not enough students applied for the projects to justify two individual groups, so both projects were combined together with a focus on the middleware technology. The development of the prototype and initial releases required some service and testing to verify functionality, and such the needed applications to run on-top the middleware needed to be developed as well.

As for this project's deliverables, there were multiple releases throughout the span of the project to aid in agile development and keep FFI updated on the direction. The prototype was developed quickly and with regular demonstration of functionality together with FFI to get feedback and help guide attention towards the main work. Most of this work was planned to lead up to a first release around the project's midpoint, to aid FFI in testing and migrating their existing solutions to the new communication platform that the middleware would provide.

Further into the project, the main deliverables consisted of revised parameters for tuning the machine and middleware itself. This was based on the principle to implement most of the must-have features such that they would satisfy FFI requests, and then moving focus over to testing and tuning to get the middleware more performant. This can be seen as a shift in focus to the research part of the project.

1.3 Norwegian Defence Research Establishment (FFI)

The project's main stakeholder, the *Norwegian Defence Research Establishment* (FFI), is a governmental research institution, focused on interdisciplinary research related to the needs of the defence sector (FFI 2023). FFI is the chief adviser on defence-related science and technology to the Norwegian Ministry of Defence and the Norwegian Armed Forces' military organization. FFI partners up with universities and students to teach and supervise in projects, like this one.

2 Product

2.1 Pre-study

During the initial stages of the project, there was just over a week dedicated for a pre-study phase to do research and find previous work that had been done on this topic. Unreliable communication was never a big problem, and there was a broad consensus from the get-go that the industry standard User Datagram Protocol (UDP) would be the protocol of choice for unreliable communication. UDP ticked all the boxes needed while being simple, efficient and well studied. Of course, all of these protocols build upon the Internet Protocol (IP).

From this initial stance on using UDP, the possibility of basing the reliable communication portion of the project on UDP as well was examined. This would combine the underlying protocols used for the communication channels into one, simplifying the system as a whole, which can be seen in figure 1. The entire system would be based on a novel approach and implementation, letting the team have full control over the different aspects of communication.

Work started immediately on discovering previous work to build reliable protocols on-top of UDP. With guidance from FFI and the work of Suri et al. (2021), some candidates were uncovered. *ZeroMQ*[2], *CoAP* (Shelby et al. 2014), *RUDP* (Bova and Krivoruchka 1999) and *Quic* (Iyengar and Thomson 2021; Thomson and Turner 2021) were initially shown to be promising. The team decided to go forward with researching these protocols in particular, and did not spend more time examining previous work other than these protocols. Table 3 shows the evaluation of each protocol's fitness for this project after the pre-study phase. In addition to the protocols based on UDP, the configurability of TCP was looked at as an option, and added to the list of candidate protocols to be examined for this project. This was, in part, due to conversations with the team's networking correspondent[3].

In testing, ZeroMQ has performed poorly on simulated military networks, discovered by the work of Andersen (2022). From this conclusion, with support from FFI, it was decided to not further explore ZeroMQ as a candidate for this project.

CoAP is favorable for its light-weight, resource-constrained design with focus on machines and networks with low capacity. However, CoAP has a weaker model for reliability than the other protocols. It only prioritizes the receiver, and does not acquire the mechanisms to adequately ensure reliable delivery. In addition, CoAP does not have easily configurable congestion control parameters, making it more difficult to fine-tune if problems were to arise. From these factors, the team decided to not further explore CoAP as a candidate for this project, as flexibility and reliability was key for FFI.

RUDP provides for many of the requirements needed in this project. Messaged-based communication, good reliability and configurability, low overhead, error detection and the possibility for secure transmission. Some questions could be raised about its telephony signaling approach and focus, however the other factors made RUDP a good candidate for this project, and was selected as a candidate.

QUIC is a protocol a with heavy focus on security. As described in Thomson and Turner (2021), QUIC assumes the responsibility for confidentiality and integrity protection for its packets. This is useful in the context of confidential data, however it was not a requirement by FFI and the services. There are instances where confidentiality is not strictly required, but reliable transport is. Due to the overhead required by the confidentiality implementation in QUIC on every connection, it was not desirable for this project. Thus, it was not selected as a candidate.

The research ended up yielding two protocols that could be used for this project, RUDP and TCP. RUDP has many strengths going for it, and could have worked well for this project, but the networking correspondent's arguments weighed strongly in on the decision-making: TCP is built upon decades of networking knowledge and has been built for edge cases that possibly would not

[2]https://zeromq.org/
[3]See 5 Acknowledgements

Protocol	Evaluation	References
ZeroMQ	Performed poorly in testing	Zeromq, Andersen (2022)
CoAP	Good for resource-limited situations. Bad reliability & configurability	Shelby et al. (2014)
RUDP	Good initial evaluation; candidate protocol	Bova and Krivoruchka (1999)
QUIC	Unnecessary cryptographic overhead	Iyengar and Thomson (2021) and Thomson and Turner (2021)
TCP	Performs poorly in testing with default configuration; candidate protocol with tuning	Eddy (2022) and *Transmission Control Protocol* (1981)

Table 3: Summary of existing networking protocols

be considered until encountered in the wild after deployment, leading to instability. TCP was more configurable than expected; system calls can configure the way TCP behaves on the host machine. It is possible to tune the different wait timers, parameters and internal variables that configure TCP's congestion control and behaviour to better fit military networks. Taking this into account, the team conversed with FFI on changing the initial draft of the architecture, instead using TCP as the transport layer protocol of choice.

After choosing to utilize TCP for reliable communication, there was also the question of whether or not to remove UDP entirely, and base both reliable and unreliable communication on TCP. This was motivated by a fundamental problem with UDP, namely a lack of congestion control. The main problem FFI encountered with TCP during testing was early retransmissions putting more pressure on the network, which lead to congestive collapse. When tuned properly, TCP could not only avoid causing congestive collapse by unnecessary retransmissions, but also avoid other causes of collapse with the built in congestion control. This is not a feature of UDP, and the team suspected that the congestive collapse FFI aims to avoid might be more prevalent without it.

Using TCP for both reliable and unreliable communication would, however, prove troublesome, as multiple TCP configurations on the same network stack is problematic. It was therefore decided that UDP would still be used for unreliable communication, and adding a custom congestion control implementation on top would be a possible optimization if time would permit.

Some other important terms to have covered to understand this report, is Maximum Transmission Unit (MTU), Type-of-Service (ToS) and IP fragmentation. The MTU is the size of largest unit of data supported by a protocol or system in a single communication. It is important to understand the implication this has: there may be a need to send more data at once than the MTU supports, which demands a strategy to deal with with fragmentation – the process of splitting a large packet into smaller packets, in such a way that they can be reassembled at the receiving end. ToS is the name of a field in the header of the packets sent which is important for FFI – it is used to indicate the service from which the packet originates. IP fragmentation is packet fragmentation which occurs at the IP-level.

2.2 Requirements

Requirements were formulated together with FFI to get a grasp on the scope of the project, and to get a clear understanding of the application that was to be developed. The basis of the project was that the middleware would provide an *Application Programming Interface* (API) for services and applications to use. It would handle communication over the network for services that would want to use it, concealing the low-level inner workings of handling network traffic.

Requirements assigned as high priority were must-have requirements that were required for a minimal viable product for FFI. Requirements were the basis on the mutual understanding of what entailed as the product between the team and FFI. What follows is a description for the requirements that were *not* high priority, and the table with all requirements.

ID	Name	User story	Requirement	Pri.	Acceptance criteria
R1	Send and receive	As a developer I want the middleware to be able to send and receive packets from my service application, so that it can communicate across a network.	The middleware must be able to send and receive packets from services over a network.	5/5	Two clients on the same network, using the middleware, must be able to communicate.
R2	Value configuration	As a developer I want the middleware to be able to set the Type-of-Service field of the IP header, so that my application gets the proper priority and treatment by the network hardware.	The middleware must be able to set the Type-of-Service field of the IP header to give information to FFI's router.	5/5	The Type-of-Service value must set in all the packets sent by an application, as configured by said application.
R3	Service configuration	As a developer I want the middleware to be able to configure the value of the Type-of-Service field, so that I know my application gets the correct priority by the router.	Services must be able to configure the value of the Type-of-Service field through the API.	3/5	The Type-of-Service value must set in all the packets sent by an application, as configured by said application.
R4	Fragmentation	As a developer I want the middleware to be able to fragment and reassemble packets larger than the MTU, because otherwise it would be handled by IP fragmentation, which is not always supported on military networks a	The middleware must be able to fragment and reassemble packets when packets are larger than the MTU.	5/5	No packets should be transmitted over the network which are larger than the MTU which is configured in the middleware.
R5	Network environments	As an end-user I want the middleware to be configurable for different network environments, so that I can be sure the network is always utilized to its full extent.	The middleware must be configurable for different network environments such as latency, packet loss and MTU.	3/5	The middleware must have configuration options to allow for changing its behaviour according to the network tuning.
R6	Reliable transmission	As an end-user I want the middleware to support reliable transmission, so that when I send an important message I can be sure it is received.	The middleware must support a protocol for reliable transmission of data.	5/5	The middleware must provide reliable transmission, even in bad networks with high latency, dropped packets and low bandwidth.
R7	Multicast	As an end-user I want the middleware to support multicast regardless of IP multicast support, as I want to easily communicate with multiple people and the military networks do not always support IP multicast.	The middleware must support multicast on all networks (with and without IP multicast support).	1/5	The middleware must implement its own multicast support which does not rely on underlying network features. Otherwise, IP multicast must be supported.

ID	Name	User story	Requirement	Pri.	Acceptance criteria
R8	No single point of failure	As an end-user I want the middleware to not constitute a single point of failure, because in military environments the network characteristics are often poor and available communication channels are essential.	The middleware must not constitute a single point of failure. All instances of the middleware must be able to operate without support by other instances.	5/5	The middleware must not rely on a certain machine, resource or client to be available. If it does, and such a dependency is removed, it must not fail and should resume operation in some way.
R9	Network independence	As an end-user I want the middleware to operate fully without an Internet connection, as the Internet is not always available in military situations.	The middleware must be able to operate fully disconnected from the Internet.	5/5	All features and functions of the middleware must be available when there is no Internet connection.
R10	Deployable to Linux	As a developer I want the middleware to be easily deployable to Linux, so that I can easily develop applications with it.	The middleware must be easily deployed to Linux-based machines.	5/5	The middleware must have clear and comprehensive installation instructions for Linux that can be easily followed by a developer.

Table 4: Project requirements

As for **R3**, a central point from FFI was the ability for services to set the ToS-value, as outlined in the high-priority **R2** requirement. During further discourse with FFI the team settled on an approach where the ToS-value should be set to a known value to signify that the middleware was the one communicating. This was good enough for a proof-of-concept system to show that ToS-values could be set. Unforeseen development time on the core functionality of the middleware, and roadblocks with configuring TCP, solidified this decision. The requirement was moved to the later part of the project, and it was therefore decided that this requirement was of moderate priority.

R7 was deemed to be needed by FFI, however not as important as the other requirements. For the middleware to be functional enough for FFI to inherit the code and use it as a basis for further research, *peer-to-peer* (P2P) functionality was deemed absolutely necessary. To support multicast on networks where one may not have the underlying ability to use IP multicast was deemed important for the middleware and its use-cases, but was the least important of the requirements. Multicast communication could be exchanged for a P2P-based communication system as required under **R1**. The clients would still get their data, although it may not have be as efficient.

As described previously, military networks are not as predictable as civilian networking environments. **R5** was a requirement which outlined the need for the middleware to be adaptable and use configurations to operate efficiently under different network environments. Like the previous two requirements which were not high priority, it was deemed together with FFI that the *configurability* of the middleware was not as important as the other requirements. It would be acceptable if the middleware used a hard-coded approach for trying to achieve efficient communication in these sub-optimal conditions while the other core functionality was being developed.

2.3 Technology

For the language used to develop the middleware, the team came to a consensus on using either C or Python. Whereas C was likely to produce a more performant product, Python was easier for prototyping and development velocity. Both languages had good support for handling socket and network programming and none of the requirements from the customer indicated that performance was of utmost importance. Therefore it was concluded that velocity would be of greater benefit to the project and customer, resulting in the choice of Python for the project. Additionally, this resulted in the use of a language which all team members had previous experience in.

Python had great support and many different libraries available. The team was aware, however, that a pure Python application could have performance issues which might have to be worked around. As Python provided relatively easy ways to write performance critical code in lower level languages (such as C or C++[4]) the team thought that prototyping in a garbage-collected language would be more efficient, while allowing for easily crossing the performance-bridge once it was reached.

Scapy[5] was quite heavily looked into for creating the packets for the middleware. Scapy is a packet manipulation program and integrates with Python. It was put forth for the project due to its ease of setting values in packets (e.g. ToS-value). At the time the team were unsure if the socket library in Python could directly manipulate such values, as testing on Windows showed that this did not happen in practice. The team's minds were eased when further testing showed that this could be done on Linux-based machines, giving no reason to continue trying to use Scapy as an underlying technology of the project, except in testing. The standard Python socket library was used instead, as it was simple and closely resembled the widely spread Unix socket API.

[4]https://docs.python.org/3/extending/extending.html
[5]https://scapy.net/

2.4 System Architecture

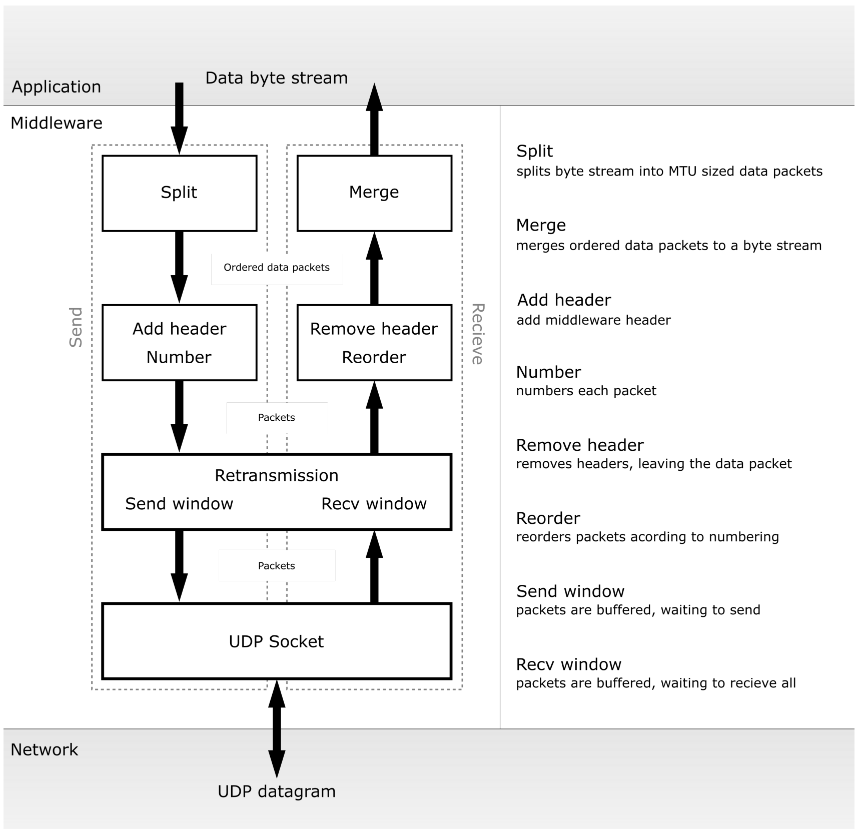


Figure 1: First system architecture draft

As per the requirements and customer recommendation, the first draft of system architecture (shown in Figure 1) was based on an underlying UDP socket. The main objective of this draft was to outline how data from the application would be split into packets, and furthermore how these packets would be sent as UDP datagrams. To mimic how applications interact with UDP, the middleware would expose an API which enables sending and receiving a byte stream (shown at the top of Figure 1). This byte stream would then be divided into an ordered list of "data packets" (fragments) to fit within the Maximum Transmission Unit (MTU) of the network (shown as "Split" and "Merge" in Figure 1). IP fragmentation would normally take care of this, however, military networks do not always support fragmentation, and the customer therefore required that the middleware should handle this.

To reassemble the fragmented packets at the receiving end, a header is prepended with a unique identifier when sending, and packets are reordered using this information when receiving (shown as "Add header, Number" and "Remove header, Reorder"). The final part of the diagram, before packets are sent to the UDP socket, shows how packet buffering was planned for both sending and receiving. This was to enable more control over when packets are sent, and how long they are kept before reassembly. This was also the "module" that handled retransmission for when the middleware was configured for reliable communication. With the exception of "Retransmission", the above explanation and architecture diagram (Figure 1) holds true for the middleware when configured for unreliable communication.

As explained in Section 2.7.4, due to reliable communication not needing manual fragmentation, the system architecture was reworked. The new system architecture is shown is Figure 2, and is centered around the idea of a "Middleware API" that supports both unreliable and reliable communication. The diagram can be split into three parts: unreliable communication (left), configuration (middle) and reliable communication (right). Unreliable communication is realized by allowing the user to instantiate instances of `MiddlewareUnreliable` sockets. These sockets utilize an underlying UDP socket to communicate over the network and implements fragmentation and

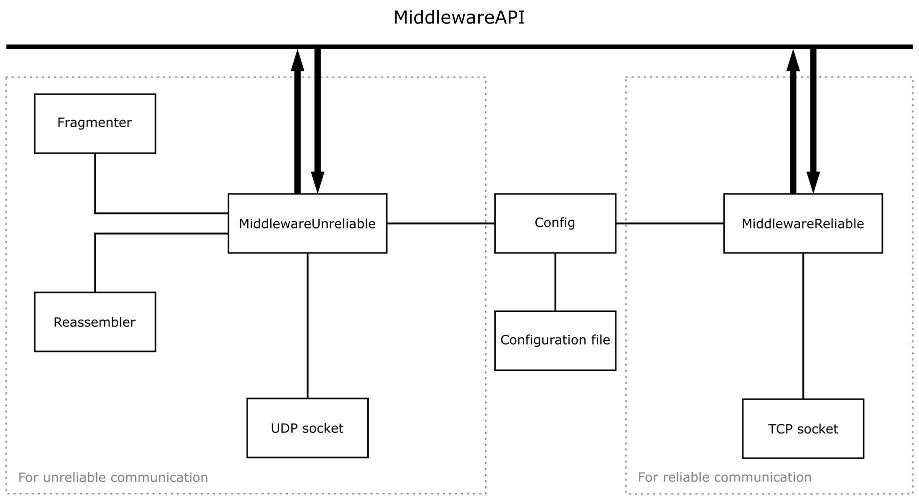


Figure 2: System architecture

defragmentation. The design and actual implementation of fragmentation is discussed in Section 2.7. The middle part of the diagram depicts configuration, which is responsible for allowing the user to adapt the middleware to different network environments. This is done by editing a configuration file, which is then loaded by the middleware at startup and used by both unreliable and reliable communication. The design of this system is described in Section 2.8. The final part of the diagram is reliable communication, which is realized by configuring an underlying TCP socket. The specifics of this configuration are described in Section 2.7.5, which deals with the implementation of unreliable and reliable fragmentation.

With this new system architecture, the most interesting part is unreliable communication. An updated architecture diagram for unreliable communication is shown in Figure 3. The diagram is somewhat similar to the draft presented in Figure 1, with some notable changes. Firstly, the diagram depicts a message oriented communication API, as opposed to the byte stream oriented API presented in the draft. This is an important change from the draft, as it allows unreliable communication to more closely reflect how UDP works, as well as unburdening reliable communication from having to emulate a message oriented API on top of TCP. This is dicussed in more detail in Section 2.7.4. Secondly, the send window is removed, as UDP has no concept of congestion control, and therefore does not use a send windows, as opposed to TCP. Lastly the language has been changed to more closely fit the language used in Section 2.7.4 and the implementation, so as to avoid unnecessary confusion.

2.5 Middleware API

The interface to other programs comes in the form of a Python module. Other Python programs can import this module and use the functions within to establish connections to other middleware instances. Once you have imported the module, you can choose between using a reliable or unreliable connection by creating either a `MiddlewareReliable` object or a `MiddlewareUnreliable` object. These objects are modeled after the Python socket library, and will have many of the same functions as a Python socket, such as bind, send, close etc. This was done to make the middleware more easily integratable with existing services, since many of the functions used are the same.

As for a description and documentation of the middleware API itself, this is contained within the GitHub repository.[6] In addition to this, the code is documented through the use of `docstring`s in the Python code. This was the chosen method of documentation as it would be the easiest to update, as well as convenient for developers using the API.

[6]https://github.com/Kurumiiw/Middleware

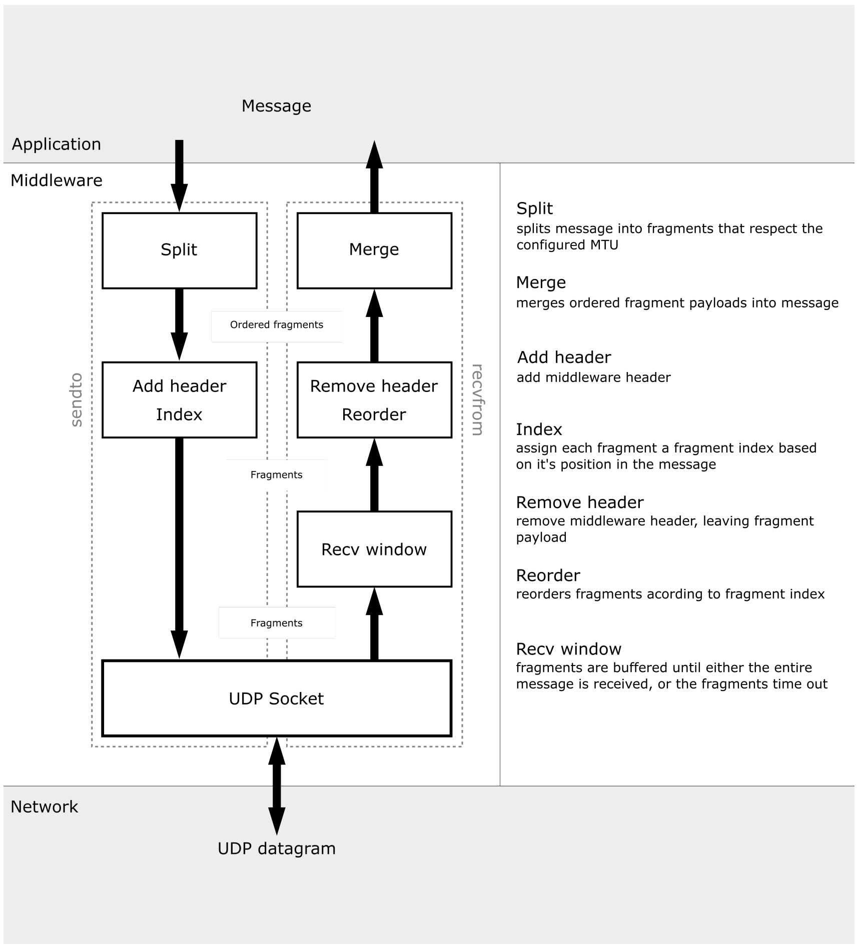

Message
Application
Middleware
sendto
recvfrom
Split
Merge
Ordered fragments
Add header
Index
Remove header
Reorder
Fragments
Recv window
Fragments
UDP Socket
Network
UDP datagram
Split
splits message into fragments that respect the configured MTU
Merge
merges ordered fragment payloads into message
Add header
add middleware header
Index
assign each fragment a fragment index based on it's position in the message
Remove header
remove middleware header, leaving fragment payload
Reorder
reorders fragments acording to fragment index
Recv window
fragments are buffered until either the entire message is received, or the fragments time out

Figure 3: Unreliable communication architecture

2.6 Example services

It was proposed by FFI that the team should also develop a few simple services which would utilize the middleware to communicate. What was made was a simple chat service where users would connect to a host and chat in the terminal, and a test service which could easily be configured to send at different rates and would store all the data received with a timestamp.

The chat service only uses reliable connections to communicate. One instance is set up as the host and will allow others to connect to them as clients, and the connections will be kept alive until the client leaves the chatroom. The host service is responsible for keeping track of each connection and distributing messages from one client to all others.

The test service consists of a controller and a worker. The controller has both an unreliable and a reliable middleware socket and is only responsible for receiving data and recording metrics such as how much data was received and at what time. The worker would read a JSON configuration file and load different sending configurations. The configuration can set different parameters for the worker such as configured MTU, ToS-value, burst size, packet size, burst interval etc. The main function of the worker program can easily be edited to change which of the configurations are run. A single worker program can run multiple configurations simultaneously, and a controller can receive from many workers at the same time. This allows the middleware to be tested with multiple different styles of transmissions by emulating different services, such as video streaming or file transfers.

2.7 Fragmentation

One requirement for the project was that the middleware should support fragmenting at the middleware layer (above TCP/UDP) with the goal that this would prevent undesired fragmentation at the IP layer. This was beneficial as the middleware would be designed with the specific military networks in mind, and therefore could fragment in a way that is more adapted to those particular networks, compared to the more general solution. Additionally, IP fragmentation is considered somewhat fragile, and has security concerns (Bonica et al. 2020).

In order to fragment packets a thin header on top of UDP/TCP was developed. This header went through several iterations, as more was learned about fragmentation in networks, and about performance in the tests.

2.7.1 Initial design

The initial design is shown in Figure 4.

This was a naive approach that was functional but quite inefficient. The team implemented a 4 byte identification field. If the identification has a non-zero value, it indicates that the packet is a fragment in a sequence and needs to be reassembled using other received fragments sharing the indicated identification. This is done using the fragment id and final fragment id. The reassembly of the fragments occurs by removing the header, then appending them in order of fragment id.

On the other hand if identification has a value of 0, it indicates that the packet is not fragmented and can be used as is. In this case, the header can be stripped, and passed on to the application immediately.

This approach was quick to put together and allowed the team to quickly demonstrate a working prototype of fragmentation to the customer. However, a number of concerns quickly became evident:

The inclusion of the entire final sequence number in the header was inefficient, as it had to be included for every fragment. Considering that the aim was to support MTU values down to 100 bytes, any reduction in header size would save significant overhead.

Furthermore, the identification field would have to be unique on the receiving end for fragments to be reassembled correctly. This could not be guaranteed without extensive added complexity if a host received data from multiple sources simultaneously. While this would not be a problem for TCP, this is a use case that is supported by UDP.

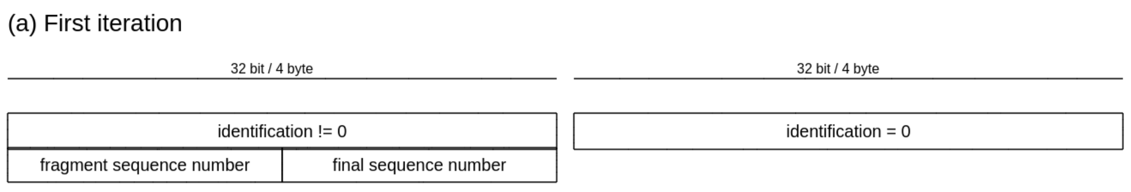


Figure 4: The first iteration of the header format, shown both for fragmented (left) and unfragmented packets (right).

2.7.2 Limiting IP fragmentation

It was found that the problem could be solved by imitating IP fragmentation (*Internet Protocol* 1981), by indicating the final package with a single bit flag instead of including the sequence number of the final packet. With this change the new header looked as shown in Figure 5.

During the work on this iteration it had been decided that traffic would be received on separate ports based on configuration and destination service. The team there thus no longer had to have identification be globally unique amongst all received fragments; instead it would only have to be unique among all received fragments from a given source IP address and port combination – which can be guaranteed by the sending instance of the middleware, thus sidestepping some complexity. As a consequence the identification field size was reduced the to three bytes.

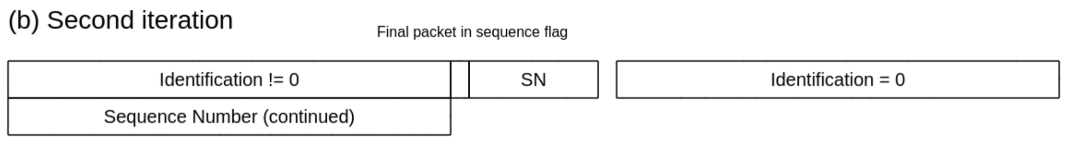


Figure 5: The second iteration of the header format.

2.7.3 Introducing a length field

The first two iterations had only been tested within unit tests, and had worked well under those conditions. As fragmentation was integrated into the service API, another issue was quickly encountered, however. The Python socket library only provides a byte stream of received data, that is, it does not return the packets. This means that the middleware would have to extract the individual packets from the byte stream itself, something the unit tests so far had just assumed to happen.

Multiple options were considered to make packets separable, but it was concluded that the easiest solution would be to include a length field in the header from which the receiver could then determine at what position in the byte stream the next packet header would be found. The the identification field size was also further reduced to two bytes, matching that of IP fragmentation.

The resulting design is shown in Figure 6, and resulted in 7 bytes of header overhead for fragments and 4 bytes of overhead for unfragmented packets.

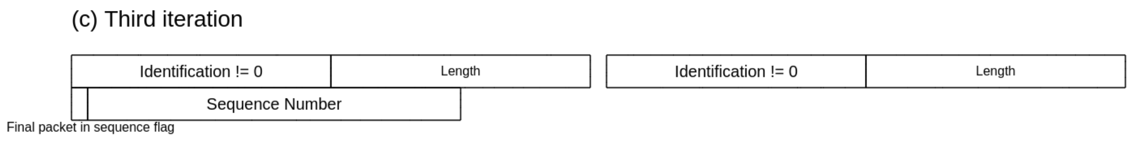


Figure 6: The third iteration of the header format.

2.7.4 Final design

A major problem encountered in iteration 3 was how packets were received as a byte stream, and this made reassembly more difficult. TCP, by design, does not delineate between data received from separate segments. For messages to be separable over a TCP connection, the start and end has to be encoded in the message data, which requires either specifying the message length before the data, or using a terminator sequence (e.g. the null terminator) to mark the end of messages. Additionally, a related problem with the current approach was discovered: data sent with TCP is buffered. This means that even though the application might send the data of each fragment separately, TCP might buffer these operations and send them in a batch. This is normally not a problem, but if IP fragmentation is to be avoided, using a fixed MTU, buffering by TCP with no restrictions on the buffer size will not respect the set MTU.

Buffering can be avoided for socket send operations by disabling Nagle's algorithm (*Congestion Control in IP/TCP Internetworks* 1984), but retransmissions will still be sent in bulk. TCP does however provide an option for setting the Maximum Segment Size, which determines the maximum payload a TCP segment can deliver. This applies both to individual send operations, as well as retransmissions. Setting this before establishing a connection will make TCP correctly fragment the buffered data for sending, respecting the set MTU. However, since TCP buffers the data, and correctly fragments it for sending, as well as guaranteeing that fragments arrive, and are reassembled, in order, there is no need for fragmenting in the middleware. This does however require that the middleware API for reliable transmissions is based on a byte stream, and not message oriented.

Therefore, by changing the API to more closely reflect how TCP works, adding headers, fragmenting and reassembling packets is not required by the middleware, which both lowers complexity and network overhead.

Furthermore, fragmentation in unreliable transmissions can also be simplified, while simultaneously making it more robust. This can be shown by first analyzing what the system has to handle, and then extrapolating the necessary information that has to be exchanged for correct reassembly on the receiving end. Firstly, on the sending side messages to be sent over the network have to be divided into fragments, such that the resulting IP packets respect the configured MTU. Secondly, on the receiving end, the fragments have to be reassembled to the original message. Since IP packets can be dropped, for reassembly, the receiver must know both how many fragments constitute a message, and which message they belong to. Information identifying which message a fragment belongs to has to be included in every fragment from said message, while information about the total number of fragments can be transferred in a multitude of ways. Additionally, since IP packets can arrive out of order, each fragment must include information relating them in a total order.

The information that has to flow from the sender the receiver has now been specified. However, since there are no bounds on e.g how large messages can be or how small fragments should get, there is also no bound on the amount of overhead in bits that have to be transferred to carry this information. The minimum MTU fragments have to respect can easily be set to 68 bytes, since "[e]very internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet header may be up to 60 octets, and the minimum fragment is 8 octets" (*Internet Protocol* 1981). The minimum payload size can then be set to $68 - udp\_ip\_header\_size - middleware\_header\_size$ bytes. To reduce overhead, IP options can be forced off, which reduces $udp\_ip\_header\_size$ to 28 bytes. Additionally, since this is the minimum payload size, setting maximum $middleware\_header\_size$ to 8 bytes will simplify calculations and add some leeway, so long as the header size never exceeds this value. These adjustments will then result in a minimum payload size of 32 bytes. Furthermore, since the unreliable API is made to mirror UDP, it makes sense to choose the same maximum payload size, which is 64 KiB (KiB = 2^{10} bytes, in comparison to kB = 10^3 bytes.) It is dubbed "message length" in the middleware to avoid confusion with UDP and fragment payload. Since an encoding of a total ordering of fragments require enough bits to describe the maximum number of fragments, the overhead will then be $\left\lceil \log_2\left(\frac{maximum\_message\_size}{minimum\_payload}\right)\right\rceil = \lceil \log_2\left(\frac{64\text{KiB}}{32\text{B}}\right)\rceil = \lceil \log_2(2048)\rceil = 11$ bits. The middleware header will then need to include at least 11 bits for what will now be called the "fragment index".

With this information the receiver is able to reorder packets, but still not determine if a message is complete or not. This requires the sender to also relay information about how many fragments there are. One way of doing this is to include one bit (a "fin" bit) in the header which signifies whether or not the fragment is the last fragment in the message. The number of fragments can then be inferred by the fragment index of the fragment with the fin bit set. For small messages, this is ideal. However for larger messages (with more than 11 fragments), this will waste one bit per fragment. An approach that reduces this waste is to drop using any bits in the header and instead prepend 11 bits to the payload, describing how many fragments are sent. Since the first 11 bits are guaranteed to fit in the first fragment's 32 byte payload, the receiver can retrieve the number of sent fragments from the first fragment's payload. This is much more efficient for larger messages (saving $2^{11} - 11 = 2048 - 11 = 2037$ bits = circa 255 bytes), however it also penalizes smaller messages (adding 11 bits of overhead even to single fragment messages). Since unreliable communication is often used for telemetry, smaller messages are expected to occur more often. This makes the fin bit approach more advantageous. Additionally, since UDP can put a lot of pressure on the network when sending lots of datagrams (due to having no congestion control to throttle down), large payloads are not recommended to be sent unreliably. The fin bit approach was therefore chosen, which adds 1 bit to the middleware header.

Now, the receiver can both reorder fragments, and determine when a message is complete. However, there is currently no way to distinguish between fragments that belong to different messages. This is not a problem for e.g. a bus protocol, but when the internet is involved, and packets can both be reordered and selectively delayed, this is a major problem. The obvious answer is to just add some bits to the header, and using this to identify individual messages. Which is also probably the only sane solution. However, this alone doesn't solve the problem completely. Since the number of bits identifying each message is finite, it is impossible to uniquely identify messages over a certain number. When talking about binary arithmetic, this is often referred to as overflow, and will determine how many messages can be "in flight" at the same time. With connection oriented protocols like TCP, this overflow is not a problem, since the receiver acknowledges what has been received, making it possible for the sender to wait in order to avoid duplicate IDs. However, since UDP, and by extension the middleware unreliable API, is less connection oriented and more "fire and forget", it is not possible for the sender to throttle down to avoid such situations dynamically. There are however some solutions to this problem, with differing trade offs.

One solution is to include enough bits for the message id to realistically never overflow. This is however very waste full. Another solution is to include enough bits to permit a certain sending rate for a certain amount of time, and make sure the sender does not exceed this. Which encounters problems during bad network conditions and short quick bursts. The problem both of these solutions face is the receiver keeping fragments long enough for the message id to overflow, and messages to be mangled. Choosing a good strategy to timeout fragments before this happens makes both solutions viable.

One strategy is to timeout fragments after a certain amount of time has passed since they were received. This is however highly susceptible to network delay. Theoretically, a message could be sent, with the first fragment being rerouted to a highly delayed path. The whole message, except for the first fragment is discarded. When the sender then sends a new message with the same id, the first fragment might be dropped. When the new message is received, the highly delayed fragment is also received. Assembling the message will then yield corrupt data. IPv4 fragmentation is based on the second approach (including enough bits for a certain send rate) and encounters this problem at high data rates (Bonica et al. 2020).

An approach that partially solves this is to add timing information in the fragment header. This allows the receiver to know when a fragment was sent, instead of only when it was received. In combination with a fragment id, the receiver can forcibly time out an older message when a newer message with duplicate id arrives. This solves the problem of reassembling corrupt messages in some cases, but not all. There is no easy way of separating two messages with identical id using only timing information, since the receiver does not know how much time has elapsed between messages. One way of solving this problem in some cases is to use the timing information from successive ids to separate messages with duplicate ids by time. However this adds a lot of latency and does not solve the problem in all cases (e.g. all successive messages might be dropped).

A compromise has to be made, since none of the solutions actually solve the problem in all cases. The approach chosen for the middleware unreliable communication protocol ended up being to include enough bits to allow a certain sending rate while aggressively timing out messages. This is a similar approach to IPv4 fragmentation, except that the fragment timeout is much lower. Since the communication channel is unreliable and dropping messages is acceptable, it was decided that a fragment timeout as low as 5 seconds (as apposed to 30s for IPv4) would be acceptable. The lower fragment timeout will then allow much higher sending rates for the same amount of identification bits, except in the situation of highly delayed fragments. In the military networks the middleware is designed for, delay can be considerably larger than regular network conditions. To counteract this the identification field length was set to 28 bits (almost twice the IPv4 length). Additionally the timeout strategy is to drop messages which have not received new fragments in

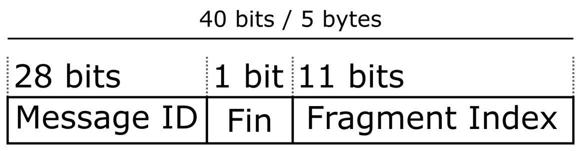


Figure 7: The final header format for unreliable messages.

a certain amount of time. This was chosen over timing out based on the first fragment's arrival, since it avoids dropping messages on spurious delays which might not cause duplicate id problems. The new middleware unreliable fragment header is shown in Figure 7.

There are however some problems with this approach, since it is a compromise, not a full solution. The receiver can easily be overwhelmed and forced to produce corrupt messages. If the sender uses the entire bandwidth, sending short mostly short messages, followed by larger messages, any packet loss will eventually lead to incorrect assembly of the larger messages. This is because the time it takes for message ids to overflow is in the order of milliseconds or less when the data rate is on the order of Mbps (depicted in Table 5). A way of mitigating this is to include a checksum in the message payload, corrupt messages can then be discarded by comparing the assembled content to the payload checksum. However, as outlined in Bonica et al. (2020) section 3.6, the checksums of both TCP and UDP are insufficient for catching such fragmentation errors. Adding a checksum that is robust enough to avoid corrupt assembly entirely will add considerable overhead to messages, which in turn will put more pressure on the network. Since these errors happen per sender, the problems can be instead mitigated by reducing send rate to something more reasonable (and not the entire network's bandwidth).

	5 Gbps	**100 Mbps**	**2 Mbps**	**250 kbps**	**16 kbps**
Overflows in	$125\mu s$	$6.40ms$	$320ms$	$2.62s$	$50.0s$

Table 5: Time until message ID overflow at max data rate when sending 8B messages

2.7.5 Implementation

The implementation of reliable fragmentation is based on configuring the TCP stack of the machine the middleware is running on. A middleware reliable socket is then functionally just a wrapper around a POSIX TCP socket. To ensure the configured network MTU is respected, the socket option TCP_MAXSEG is set with a Maximum Segment Size (MSS). This restricts all TCP segments sent to the specified size. Which solves the fragmentation problem, except the segment size relates to the TCP payload, not the total IP packet size. To respect the configured MTU, the IP packets that contain the TCP segments have to fit within said MTU. There is however no one-to-one mapping between segment size and IP packet size, since both the IP header and TCP header are variable length. However, the IP header can be forced to a fixed length by disabling IP options (which is not needed by the middleware), and TCP options are rarely used in payload carrying segments. Furthermore, testing has shown that disabling IP options and setting TCP_MAXSEG to $MTU - IP\_HEADER\_SIZE - TCP\_HEADER\_SIZE$ yield IP packets that fit the configured MTU. This is also the case when TCP options are used in payload carrying segments, which seems to imply that either TCP options are regarded as part of the segment payload or that the MSS is adjusted to take into account the increased header size. Whether or not this is the case is unclear, as the specifics of TCP options are not well documented, with most options being obsolete, experimental or not standardized.

Unreliable fragmentation is however not so simple. The implementation can be split into two parts: the fragmenter, and the reassembler. The fragmenter is responsible for taking a message, dividing it into appropriate sizes and preprending a fragment header containing the necessary information for reassembly. The reassembler is the responsible for taking received fragments and organizing them into partial received messages, assembling partial- to complete messages, and discarding old fragments.

Fragmentation work by first taking the message to be sent determining how many fragments are to be sent. This is found by taking the ceiled result of the total message size divided by the maximum payload a fragment can carry with the configured MTU. The integers less than this fragment count is then used as fragment indices. To perform the actual fragmentation the message is divided into chunks that fit the maximum fragment payload, and for each chunk, a fragment is created by combining the fragment header with the payload chunk. The fragment header is created from the fragment index, the message id and the fin bit (determined by checking the fragment index against the fragment count), which are then bitshifted into place in a 5 byte buffer to be preprended to the payload chunk.

Reassembly is a bit more complicated. The process starts by first timing out old messages. This is done first to avoid unnecessary message id collisions in the event of the process being blocked for an extended period of time when receiving UDP datagrams. The time out logic is to simply iterate over the received partial messages, and delete every message that has not received a new fragment within a certain time interval. With the time interval being the fragment timeout value specified in the loaded configuration file. When old messages have been removed, the new datagram can then be added to a partial message. This is done by first stripping off and decoding the middleware header from the UDP datagram payload. The sender's address combined with the message id from the decoded header is then used to either find an already existing, or create a new, partial message. A partial message is a map of fragments combined with a boolean indicating whether the map contains the final fragment, an expected fragment count, and a timestamp of when the last fragment was added. The timestamp is used to time out messages, while the fin boolean in conjunction with the expected fragment count and fragment map are used to determine whether a message is complete, and assemble it. Then after timing out old messages, stripping and decoding the middleware header and finding, or creating, a partial message, the remaining UDP datagram payload is added to the fragment map of the partial message with the fragment index as key, and remainding payload as data. Since the partial message has received a new fragment, it's timestamp is updated to the current time. Finally, if the fin bit was set in the middleware header of the received UDP datagram, the partial message's fin boolean is set to true, and the exptected fragment count is set to the fragment index plus one (which is ensured in the fragmentation stage to equal the number of fragements constituating the message). The final step of reassembly is to search through the received partial messages and check to see if any of them are completed. This is done by checking whether or not the fin boolean is set, along with whether the fragment map contains as many fragments as the expected fragment count foretells. The first message where this is the case is then assembled by sorting the fragments by their fragment index and combining the data into a resulting message with a corresponding message id and sender address.

2.7.6 Performance

To evaluate the performance of the proposed fragmentation scheme, multiple tests where run with equal amount of data transfer, but differing fragmentation. A baseline with regular UDP and IP fragmentation was established for 1 000 000 datagrams of size 146 bytes, 100 000 datagrams of size 1 460 bytes and 10 000 datagrams of size 14 600 bytes. The same tests were run with the middleware configured for unreliable communication with the MTU set to 500 bytes, 1 000 bytes and 1 500 bytes. Table 6 shows the time taken to send 146 MB under the different configurations between a Linux virtual machine running on a windows host and a Linux sever connected to the windows host via a router. Table 7 shows how much slower the middleware is compared to IP fragmentation when measuring IP packets transferred per second. This shows that the middleware is slower than IP fragmentation by 7% to 8% in most configurations. This slowdown does does not seem to be caused by fragment time out and checking for completed messages, as testing has shown these speed up transfer by only 0.3%. The major culprit seems to be memory allocation, as changing the receive buffer size for UDP datagrams to a larger number seems to significantly impact performance. In the test case of sending 1 460 bytes with a configured MTU of 1 500 bytes, changing the receive buffer size from the configured MTU to 2 048 bytes resulted in an additional 5% slowdown. An additional 548 bytes of memory allocation per UDP datagram therefore almost doubled the slowdown compared to regular IP fragmentation. An obvious solution to this would be to reuse the memory instead of allocating for every receive call. This can be done by using an alternative receive function for Python sockets that overwrites an input array, up to a specified size, instead of allocating it's own byte array and returning the result. The problem with this is that the received UDP datagrams need to be stored in order to be reassembled, and reusing the receive buffer memory will therefore only make the copy explicit, instead of implicit.

This is however not the only cause for slowdown, as can be seen in Figure 8. The graph shows raw throughput over the network, and as can be seen, there are multiple dips in throughput which grow larger over the runtime of the test. The dips seem to happen at a regular interval, roughly one second apart. This interval is much greater than the interval between each successive message reassembly (which is on the order of milliseconds). The cause of these dips could again be garbage collection, as it is a process which is in many languages running on a timed interrupt on the order of seconds. However, this would likely not explain why the dips grow greater as time progresses. This is due to the dips increasing significantly 15 seconds into the test (circa 9 seconds into transmission) which is after roughly 75% of the data has been sent. If the dips were purely caused by garbage collection overhead, they would likely start growing earlier. This is however pure speculation and would need a full memory trace to prove one way or another. Future iterations of fragmentation reassembly would therefore likely benefit from focusing more on memory usage, and e.g. use an object pool to not put as much pressure on the garbage collector.

Configured MTU	$10^6 \times 146$ **bytes**	$10^5 \times 1460$ **bytes**	$10^4 \times 14600$ **bytes**
IP fragmentation only	$111.87s$	$11.358s$	$10.403s$
1 500 byte, middleware	$121.227s$	$12.207s$	$11.409s$
1 000 byte, middleware	$121.27s$	$23.581s$	$18.185s$
500 byte, middleware	$121.13s$	$46.027s$	$36.067s$

Table 6: Time taken to send 146 MB divided into messages of differing length, under different configured MTUs.

Configured MTU	$10^6 \times 146$ **bytes**	$10^5 \times 1460$ **bytes**	$10^4 \times 14600$ **bytes**
1 500 byte, middleware	7.719%	6.955%	8.818%
1 000 byte, middleware	7.751%	3.668%	8.470%
500 byte, middleware	7.645%	1.293%	7.701%

Table 7: Relative slowdown of the middleware fragmentation compared to IP fragmentation.

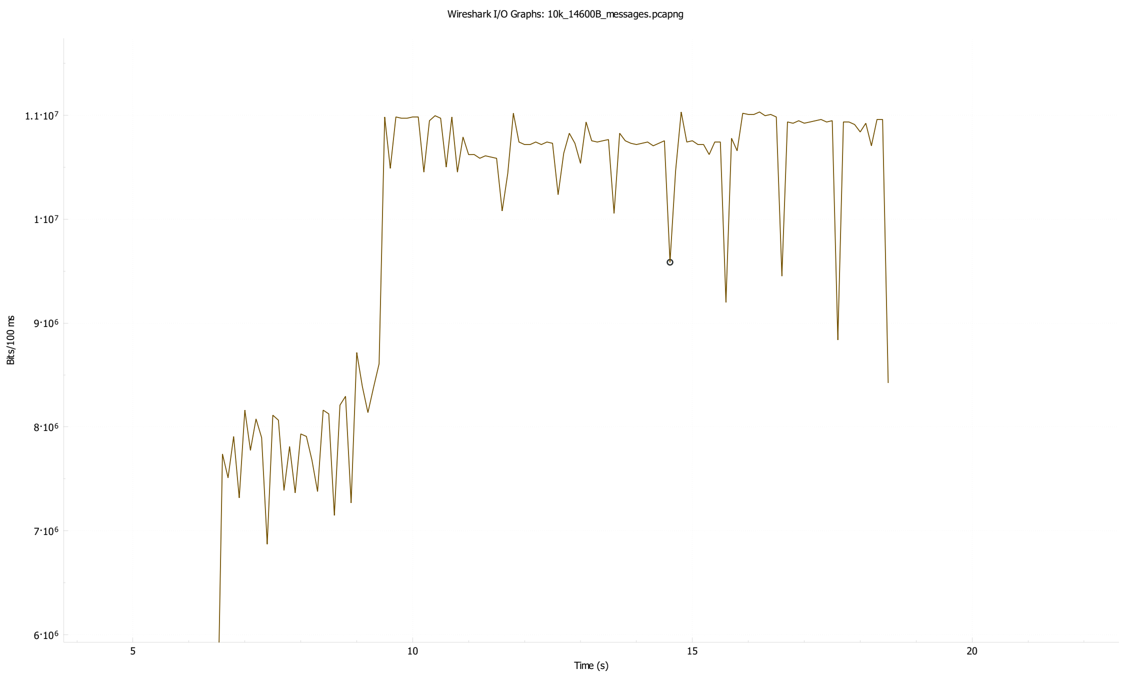


Figure 8: Number of bits sent over the network under one of the test cases in Section 2.7.6 where the message length was 14 600 byte with and MTU of 1 500 bytes (which equals 10 fragments per message).

2.8 Configuration

As per the requirement **R5** the middleware has to be configurable for different network environments. FFI noted this would ideally be done via a configuration file that the middleware loaded upon startup, and optionally through an API. It was therefore decided that the middleware would be configurable through an INI configuration file, which would be loaded upon module initialization. To make the system easily extendable, the serialization and deserialization was generated through metaprogramming from a class definition. This would allow adding new configuration options by adding an entry in the INI file and a variable in the class definition. A global singleton is instantiated of the class, which would then be made globally available, such that all subsystems of the middleware could access configuration options.

Initially both configuration of the middleware and TCP stack would be done in this way, and the configuration system would generate system calls to apply system related configuration options. However, this turned out to be a poor choice, as the whole application would have to be run with elevated privileges for the system calls to work. The configuration of the system's TCP stack would therefore be split of into a separate configuration file, which would be applied to the system via a script.

Generating serialization and deserialization seemed at first like a good idea. However, it added unnecessary complexity to the application. This complexity would only be worth it if there were frequent changes to the configuration options. Since the configuration system ended up both rarely being used, and only updated with new variables once, the complexity of the system did not yet pay off. Nonetheless, it is built to be easily configurable for the future, and could pay dividends in the future.

2.9 Congestion control analysis

During the initial testing phase to tune TCP, it was decided to start by looking at the problem which is easiest to investigate: TCP congestion algorithms. Using the testing service, the team ran tests on the networks outlined in Andersen (2022). For the choices of congestion algorithms, only TCP Vegas, TCP Westwood and TCP Cubic were deemed relevant to look into. This is partly due to limitations with the Linux kernel, as only common algorithms which could be used without much hassle were investigated. Another point is the fact that testing all TCP congestion algorithms under the sun would not yield productive, as there was other options to investigate too. However, due to TCP Hybla's focus on satellite communication networks, it was deemed relevant enough to warrant investigating on that network scenario. With this, the testing results are as shown in table 8.

Network type	**TCP Vegas**	**TCP Westwood**	**TCP Cubic**	**TCP Hybla**
CNR	65.896 kbps	75.64 kbps	**76.92 kbps**	N/A
NATO narrow-band w.	119.72 kbps	**126.10 kbps**	124.18 kbps	N/A
SATCOM	1 612.5 kbps	1 571.4 kbps	1 462.2 kbps	**1 769.2 kbps**
Tactical broadband	4 081.6 kbps	6 457.6 kbps	**13 068 kbps**	N/A
Low-band 5G	12 816 kbps	**14 744 kbps**	14 736 kbps	N/A

Table 8: Comparison of effective bandwidth for the emulated network configurations from Table 2, for each of the tested TCP congestion algorithms. The highest value for each configuration is highlighted in bold. N/A means the congestion algorithm was not tested for that environment.

It should be noted that due to suspected limitations with the testing service, the high-bandwidth networks like those of tactical broadband and 5G specification are likely limited by how quickly the service itself can send data. However, this should not be much cause for concern, as at the point where one is using such high-speed and stable networks, it should still give an adequate representation.

An example of the resulting data from the testing can be seen in Figure 9 and Figure 11. This is taken from the testing of the congestion algorithm TCP Vegas. Firstly, please notice the initial spike

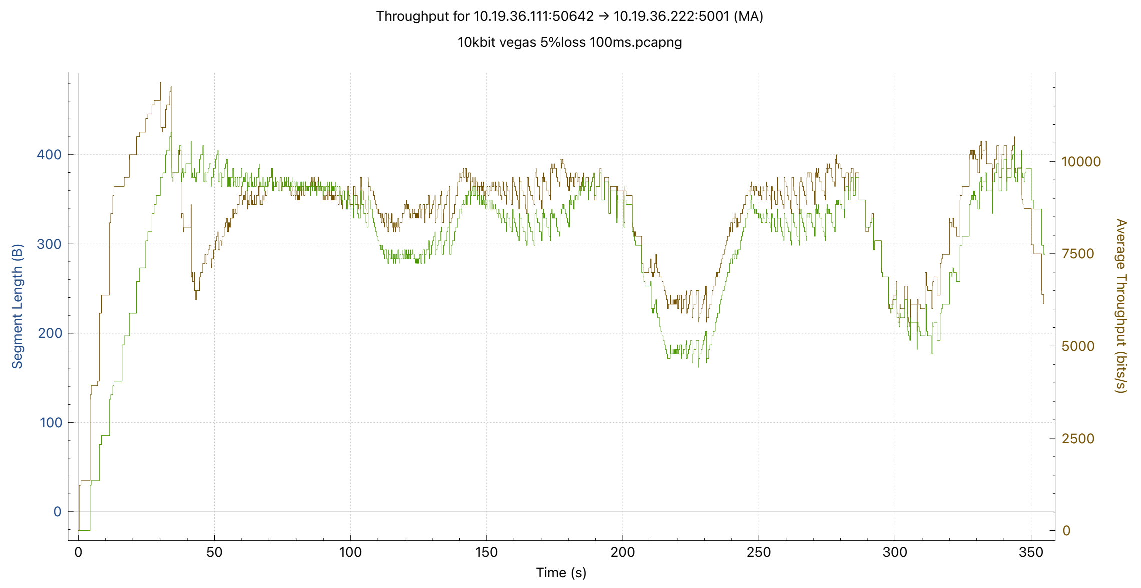


Figure 9: TCP Vegas throughput over a CNR-like network. Brown represents throughput – total amount of data sent. Green represents 'goodput' – total amount of ACKed data.

in transmit rate. This is believed to be caused by excessive eagerness by the congestion algorithm, ending up having to do on-the-fly tweaking of its behaviour to match the limited network. The proceeding dip in transmission rate is this correction, as can be seen the goodput is fairly constant at the network bandwidth limit. The dips at approx. $200s$ and $300s$ shows to come from packet drops and/or duplicate ACK packets, and TCP needing to retransmit data accordingly.

In Figure 10, one can see the time taken for communication to happen; Round Trip Time (RTT). Measured in ms, this is the time taken from when the data is sent by one client, to a response is received. For most of the communication stream, this was under $5 \times 10^3 ms$. Notably, spikes can be observed of up to $1.5 \times 10^4 ms$, a three-fold increase in the RTT. These spikes coincide with the timings of the drops in throughput as discussed above. This is another representation of a loss event, where packets were lost, and the algorithms need to retransmit old data to catch up. This leaves the sender waiting on their newest data, as old data is still being responded to, which in turn increases the RTT as shown.

Choosing congestion algorithms which only retransmit when absolutely necessary is fundamental to optimizing the network usage, as more of the network's capacity is taken up to verifying already transmitted data. As can be seen by the first spike in both of these figures, if the algorithm is too eager. If the congestion algorithm is not calibrated for the network environment which may be unstable and slow, unnecessary retransmits could occur; if the network has unforgivably strict performance limits, there may be a disproportionate drop in availability and service quality for clients on the network.

For use of a counter-example of these restricted networks, provided is Figure 11. This is a representation of TCP Cubic operating in an essentially unrestricted low-band 5G-like environment. What is noteworthy is the "pulsing" of the data, with ever-increasing throughput. As seen with TCP Vegas, if the congestion algorithm over-stepped what is possible on a given network, it will tune its transmit rate to be sustainable. If there is headroom, the transmit rate is increased. This can be seen happening with the 5G-like network, as each time TCP sends data, it realises there is more available bandwidth, and thus increases its transmit rate in steps. It is important to also have congestion algorithms which can effectively take use of good networks.

This process of analysis was accordingly done for the rest of the congestion algorithms, with a heavy emphasis on retransmit rate and duplicate ACKs. As good network utilization is important for the customer, it was decided to not go after congestion algorithms which would give higher bandwidth with lots of retransmissions clogging the network. Rather, a balance was struck between efficiency and speed. The final results can be seen in table 9.

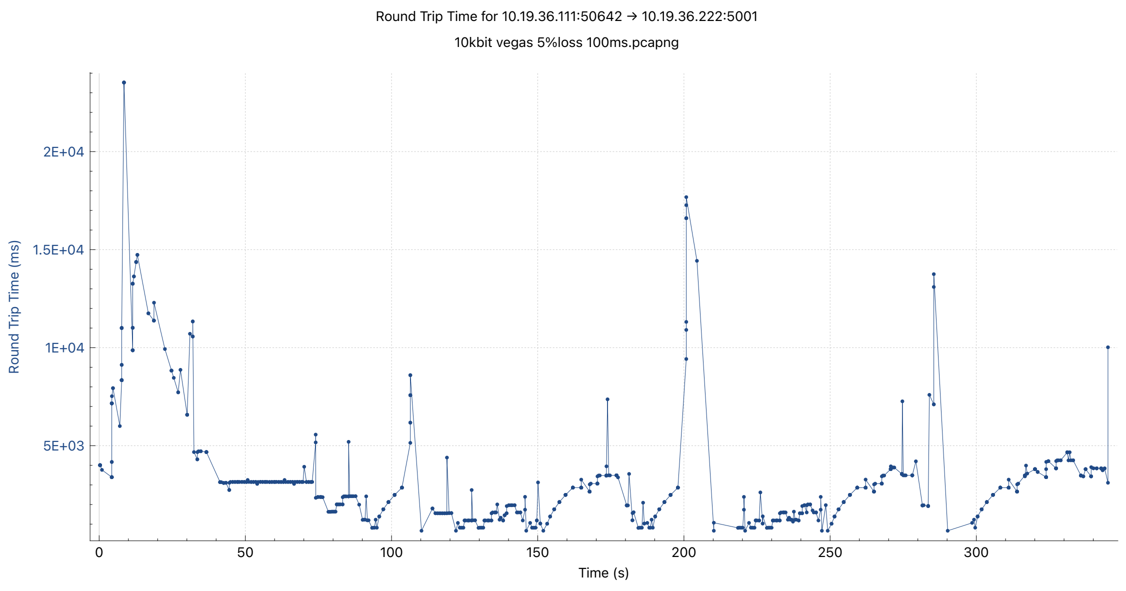


Figure 10: TCP Vegas Round Trip Time (RTT) over a CNR-like network. RTT is the time from one client sends a packet to the time it receives a response.

Network type	Congestion algorithm
CNR	TCP Westwood
NATO narrow-band waveform	TCP Vegas
SATCOM	TCP Vegas
Tactical broadband	TCP Cubic
Low-band 5G	TCP Cubic

Table 9: TCP congestion algorithm tuning results

The team believes these choices have many benefits. Firstly, all of these congestion algorithms are relatively standard, and implemented as modules in the latest Linux kernels.[7] This makes deployment and usability easy for most users. Moreover, for the networks with higher bandwidth and minimal packet loss, TCP Cubic was found to be a reasonable choice. TCP Cubic – together with TCP Reno – is usually used as the standard implementation of the TCP congestion algorithm in the Linux kernel, which means for these network types, no change is needed and the default configuration can be used.

2.10 Limitations

In its current state, there are a number of limitations, that reduce the usability of the middleware. Here, these issues will be briefly discussed.

First, the middleware configuration is not isolated from the rest of the system. Other applications that are not programmed with the middleware in mind will be affected by the middleware's configuration of congestion algorithm. This issue is difficult to solve within the constraints of this project. Likely a custom implementation of a novel transport layer protocol would be required, as multiple instances of the TCP stack are unable to run in parallel on the same system.

Second, all applications that use the middleware on a host system must share the same overall configuration. This problem – which relates to the former issue – is not necessarily desirable. For some services, aggressive retransmission may be worthwhile in order to reduce latency, despite the increase in networking overhead. On the other hand, other services may want to minimize the overhead of their traffic. This is a use case that is currently not supported by the middleware.

[7]As implemented in the Linux kernel source repository, under linux/net/ipv4.

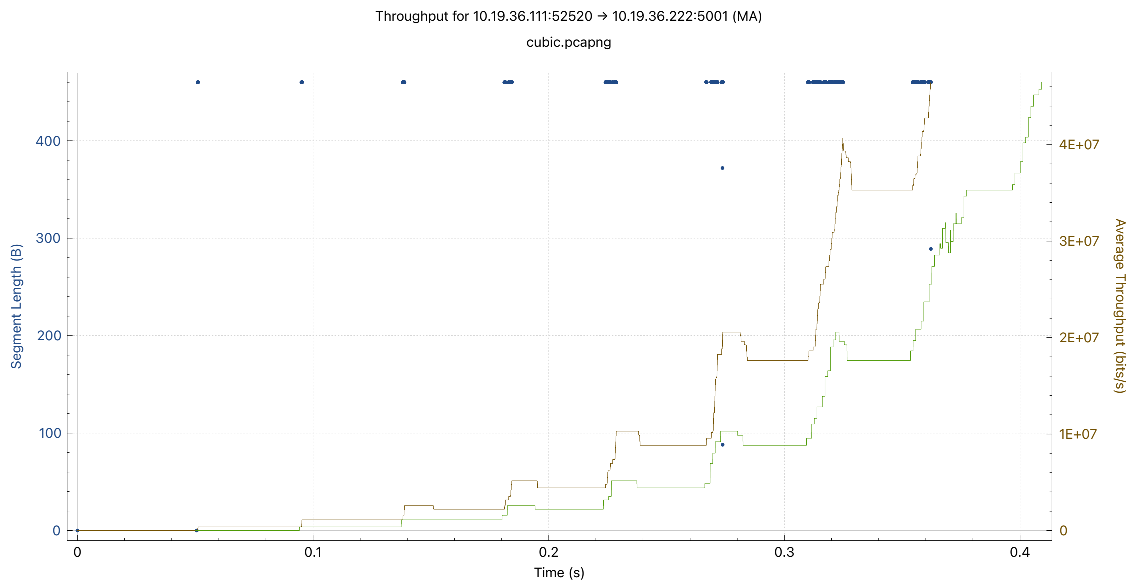


Figure 11: TCP Cubic over a 5G-like network. Note the progressive increases in throughput.

3 Process

In this section of the report, the team will look back on the process during this project, and reflect about issues and problems the team encountered during the project which are not directly related to the technical aspects of the product. First, there will be a short introduction of the team members, followed by a description and reflection of the most important events that occurred, in chronological order.

3.1 Team & Competencies

The team consisted of five third-year students taking their degree in Informatics. As a consequence the team shared a lot of experience, due to the courses in the study program being similar. The team members all had experience from *Communication - Services and Networks*, as well as a number of programming courses (*Information Technology, Introduction*; *Object-Oriented programming*; *Software Engineering*; *Informatics, Project I*).

In addition, a number of group members brought additional experience to the table.

Ola Horg Jacobsen has previously taken the Applied Cryptography and Network Security course at NTNU, in addition to the required networking and computer science courses. He has been at *Orbit NTNU* as an embedded software engineer on the student satellite FramSat project. From this, he has some extra experience working with various communication protocols and system-level network communication in unfavourable environments. For this project, he mainly took on the role of setting up tests for examining the behaviour of the communication protocols in use. Moreover, he was the main scrum master for the first part of the project.

Ola Vanni Flaata is a member of the student rocketry team *Propulse NTNU* in which he has been responsible for telemetry for two years. Through this he has gained experience with using ipv4-based communication over links with intermittent connectivity, and ensuring data integrity and smooth operation in case of data loss or sudden disconnection. For this project, he has been focusing on the API provided to the services, as well as assisting with fragmentation and general debugging. He was the main scrum master for the latter part of the project.

Tobias Ringdalen Thrane has previously taken the Applied Cryptography and Network Security course at NTNU, as well as the Operating Systems course in which he developed a multi-threaded web-server. He is also a member of the student rocketry team *Propulse NTNU* in which he has been developing and maintaining the website. Through this he has gained experience with deploying services over a network. For this project he been working on the chat service utilized in the testing of the middleware, as well as assisting with the API.

Thorbjørn Lundin has previously taken the the course in *Applied Cryptography and Network Security.* His main contribution to this project has been implementing the fragmentation functionality. Furthermore, he was Scrum master for the second part of the project.

Simon Doksrød has been appointed group leader for this project, and has mostly been working on configuration of TCP. As he has a good understanding of the inner workings of operating systems like Linux, he took on the role to investigate system-level configuration options.

3.2 Project Planning phase

The first part of the project – which spanned a period of roughly two to three weeks – entailed planning the project. This phase involved clarifying and coming to an agreement on the requirements of the product with the customer, as well as agreeing on which tools to use and how to structure the development process. In addition, a rough project roadmap leading up to the midterm was created and a risk assessment was performed.

3.2.1 Product Requirements & Learning About the Problem Domain

After the team contacted the customer, it was agreed to meet once a week. During the initial phase of the project these meetings focused on discussing the customer's requirements for the project, and developing a clear, shared understanding of the requirements.

In addition, these meetings were focused on getting the team introduced to the problem domain; while there was some understanding of networks from the previously mentioned courses, there had no experience with the particulars of networking in tactical military networks – or similar high latency, low bandwidth networks.

3.2.2 Development tools

One of the first decisions that was made was related to which tools that were to be used. The tools were divided into two main groups: development tools and communication tools.

One of the first decisions made was to use a Revision Control Software (RCS). An RCS is of paramount importance when developing complex software as a team. It allows developers to keep their work isolated in separate branches, postponing the merging – and inexorable merge conflicts – of functionality to a later date. RCS has been a staple in software development since Software Code Control System (SCCS), developed in 1972 by early Unix developers (Rochkind 1975).

There are a number of current RCSs that are used. *Mercurial*, *Subversion* and *Git* were all possible candidates. It was decided that going with the familiar – that is Git – was the way to go, which also happens to be the popular behemoth in the space. All of the team members had experience with Git from earlier projects, both as part of studies and hobbyist projects, and it was not considered that experimentation with a new RCS would be beneficial to the project or stakeholders, even though it could be an interesting and beneficial learning experience.

Once the RCS of choice had been determined it was necessary to decide on a server-side hosting for the repository. The likely candidates were *Gitlab* and *GitHub.* The team was more familiar with Gitlab, due to other projects earlier in the team's studies having been hosted there, but it was decided to go with GitHub for two reasons:

First, the team knew that the customer would like the option to make the repository public at the end of the project, in order to facilitate further development outside the limited scope of the bachelor thesis. GitHub would make this possible without having the complexity of moving the repository off of the NTNU-hosted Gitlab instance, which is only accessible through a VPN or the university network.

Second, all of the team members already had an account on GitHub, saving the team some administrative hassle by using it.

Python project management was a bit of a mess. There are many ways of managing dependencies, virtual environments and packaging of dependencies in python, but after considering the options it was decided to use *Poetry*[8], a relatively recent development in the Python packaging ecosystem. Poetry uses a similar model to the node.js ecosystem. Packages are tracked using package file ("project.toml") and a lock file ("poetry.lock") file, the equivalent to "package.json" and "package-lock.json". As there was familiarity with that way of handling dependencies and the benefits it entails – in the form of frozen dependency versions. After experimentation, the team thought it would be the best approach for the project.

Additionally, Poetry bundles most other functionality that could be required, while many other solutions require different packages for different aspects of the project management and deployment process (e.g. `venv` for virtual environments, `pip` for dependencies, etc).

It was also desirable to use a code formatter, and this was quickly chosen to be *Black*[9], a PEP-8 compliant formatter with few configuration options. It served the team well in being out of the way, while keeping the code nicely formatted and diffs small, without requiring extensive configuration.

Finally, as quality was important to the stakeholders, it was decided to use *Pytest*[10] for extensive unit testing. Later in the project (roughly around the time of release 1) this would be expanded with a test coverage requirement, where a certain proportion of the code base would have to be covered in tests.

GitHub Actions was used to run a simple CI pipeline that would check that code was formatted properly, and that the tests would succeed before the changes could be merged.

Wireshark has been one of the most essential tools during the development, testing and verification of the middleware. Wireshark is a network analysis tool which allows you to inspect the traffic across a network interface. It can provide many valuable insights such as viewing what data was sent at what time, the values of different header fields, and even built-in analysis tools which can help identify things like spurious retransmission or duplicate acknowledgements.

The team has used this tool actively both during development and testing. It has notably been used to verify that the values that were configured, such as MTU and the ToS field, are actually adhered to. Among other things it has helped us confirm that the ToS field is not set in Windows, which led us to take action and discuss deployment with the customer, which in turn contributed to the formulation of requirement **R10**. It was also vital in measuring the efficacy of different congestion control algorithms, and has produced the graphs seen in Figure 9, 10 and 11.

Though Wireshark is a powerful tool for network traffic inspection, it does have some limitations. The team encountered one such limitation when trying to verify that the maximum segment size that was set for TCP translated into smaller packet sizes that could fit within the given MTU. Even when configured with a segment size of 512, we observed segments upwards of 7000 bytes in size with Wireshark. After some investigation, we learned about the TCP offload engine (TOE). TOE is implemented by some network interface controllers and allows the operating system to offload certain aspects of the TCP/IP stack to the network interface. Two of the functions of the TOE are the TCP Segmentation Offload (TSO) and the Generic Receive Offload (GRO). As indicated by the name, TSO offloads segmentation of the TCP data stream to the network interface controller, while GRO aggregates segments together to reduce processing overhead for the CPU. However, since Wireshark is a program run by the OS it can not detect segmentation and aggregation that happens on the NIC. To overcome this limitation we disabled GRO on the receiving computer to verify that the segmentation done by the NIC adhered to the segment size set by the OS.

[8] https://python-poetry.org/
[9] https://pypi.org/project/black/
[10] https://docs.pytest.org

3.2.3 Communication tools

The team used a multitude of communication tools during the project. *Microsoft Teams* was used for meetings with the customer throughout the development processes.

The team decided to set up a *Google Drive* for keeping track of administrative writing such as progress reports, meeting notes, time tracking and more. The main report was written in *Overleaf*, a tool for writing LaTeX documents collaboratively.

For discussions related to the development process, the team aimed to keep it mainly in the comments of the issues and pull requests on GitHub. The messaging service *Discord* was used for online meetings and work sessions within the team, and *Facebook Messenger* for administrative communication such as planning meetings, work, etc.

3.2.4 Development process

At the start of the process the team discussed how the development process would be structured. There was a consensus within the team that an agile process (Mike Beedle et al. 2023) would be followed, and as the team was familiar with Scrum from previous courses and projects, there was little discussion and broad agreement.

The team also wanted to have quick sprints of only a week, as that could be used to force quick merging of new functionality into the main development branch. At the same time, doing an extended sprint planning phase and retrospective as described in Kniberg (2015). It would simply cause too much overhead to be able to complete any meaningful work on the project while having other courses to tend to as well.

Due to limited time for completing the project the team knew that doing a full sprint planning session, and retrospective after each sprint would be futile, especially since the team aimed to have one sprint each week. Instead of doing planning for each sprint individually, the team therefore set out to get a roadmap (consisting of sprint 3-7) for the entire period leading up to the midterm deadline (8th of March) with goals for each sprint. This roadmap is shown in Figure 13.

The team would then do one big retrospective after the midterm deadline, in order to reflect about the development process, and make changes if necessary.

In addition to the CI pipeline described earlier, there was also a requirement added that a code review needed to take place before merging. At least one person who had not been working on the issue had to approve before changes could be merged, ensuring an additional level of quality control. This ensured that missed functional requirements, code quality, and implemented test cases would be held to a higher standard.

3.2.5 Risk Assessment

At the onset to a new project – especially if it is a project of significant scope – it is important to get an idea of the risks involved, as unmitigated risks can stop a project in its tracks. A useful approach to get an idea of the risks is to perform a risk assessment in which risks are identified, and assigned probabilities and consequences qualitatively, in order to determine the risk.

The team performed a risk assessment at the start of the project, which the team continuously updated as the risks changed and the project progressed. Once risks have been analysed the team determined risk mitigation measures to reduce the overall risk preemptively.

This analysis was performed using a template developed by the *Norwegian Digitalisation Agency* (Digdir, formerly Difi), and entailed qualitatively identifying the probability and consequences of various risks, as shown in Table 10.

To get a more visual impression of the risks, the results were visualized as a matrix plot, shown in Figure 12.

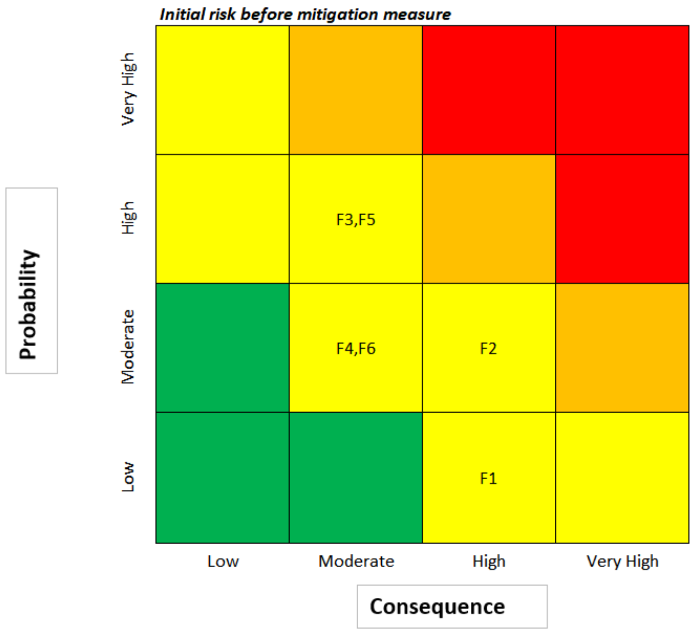
The team originally identified four risks that felt the relevant to the project (i.e. F1 - F4). These were chosen because of their probability and potential impact on the project, but admittedly also because the mitigation strategies were already agreed upon and obvious beforehand, which may have reduced the amount of actual reflection and strategizing during the analysis.

After feedback from the customer two more risks were added to the analysis (i.e. F5 & F6). Wrong time estimates during sprint planning is a common issue, thus the team felt the risk was certainly worth adding. As a mitigation strategy for this risk, the team agreed that planning poker could be used to reduce the probability of wrong estimates. Even though this may have been the only mitigation strategy that was not already planned beforehand – thus an actual product of the analysis – it was not put into practice.

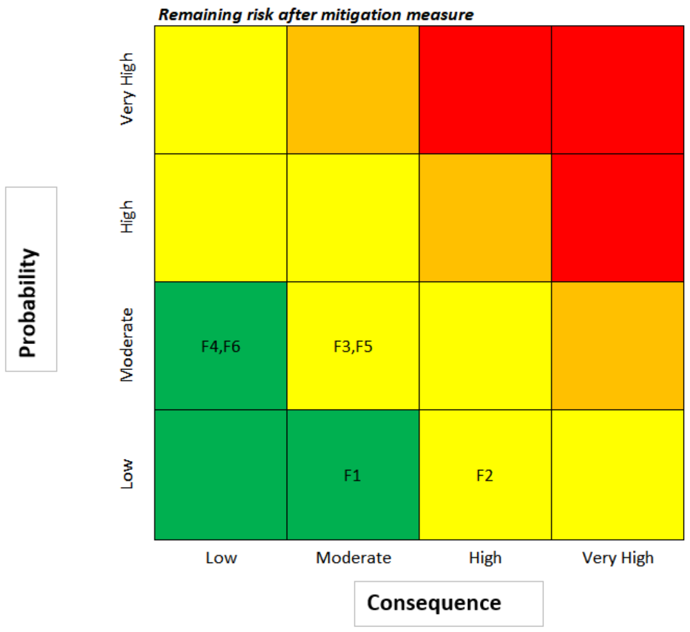
This is indicative of a larger issue with the team's relation to the risk analysis: it was hardly considered outside of the bi-weekly status reports sent to the team's supervisor. Though meant to be a tool planning ahead and mitigating risk, it felt more like an unnecessary chore for the team when strategizing could be done more efficiently and flexibly in less structured group discussions.

ID	Risk description	Justification	Consequence	Prob.	Prob. justification	Init. risk	Risk mitigation	Consequence	Prob.	Final risk
F1	Team member quits	An substantial increase in work for the rest of the team, and the team may lose valuable competencies	H	L	It is expected that all members will want to finish their bachelor thesis	L	Shared repository to ensure work is not lost, and the members regularly explaining their work to the others	M	L	L
F2	Changing requirements and expectations	Customer will not receive desired product, and unnecessary time spent on development	H	M	It is easy for misunderstandings to occur	M	Focus on good communication with the customer in order to ensure a common understanding of the project and its limits	H	L	M
F3	Lack of a representative network to test the product	Difficult to ensure quality without a representative network	M	H	It is difficult to get access to authentic military networks	M	The team will emulate network configurations using network emulation tools	M	M	M
F4	Uneven distribution of work	Could lead to a bad report and members could experience burnout	M	M	It will be neccessary to prioritize other courses at times	M	Time tracking and focus on good communication in the team	L	M	L
F5	Wrong time estimates during sprint planning	Could be unable to complete tasks according to plan, having negative effects later in development	M	H	Accurate time estimates are difficult	M	Planning poker (Kniberg 2015, p. 38-40)	M	M	M
F6	Someone in the team becomes ill	The member may not be able to work as much for some period of time	M	M	According to FHI, infectious disease occurence is back to pre-covid levels	M	Sick members stay at home and join the meetings digitally, in order to avoid more sick students	L	M	L

Table 10: Identified risk factors during our risk assessment phase. L, M and H represents *Low*, *Moderate* and *High*, respectively.



(a) Before mitigation measures



(b) After mitigation measures

Figure 12: Risk assessment matrix based on identified risk factors from Table 10, before and after performing risk mitigation measures.

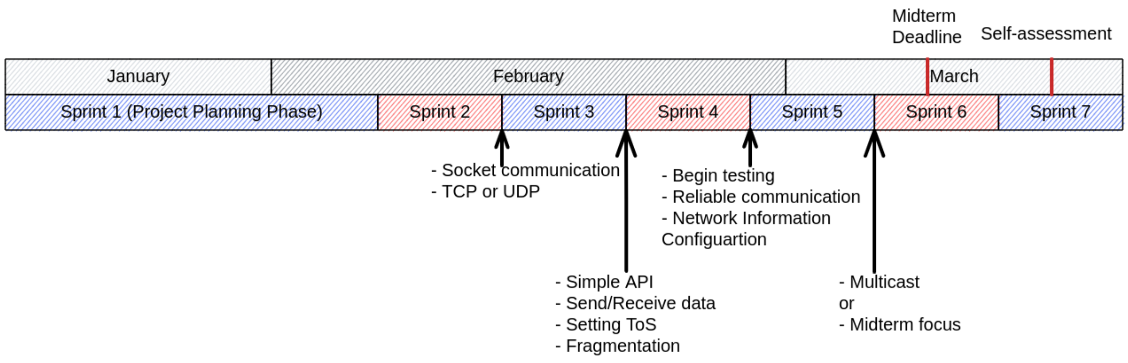


Figure 13: The pre-midterm roadmap. It was created before sprint 2, and was an aggressive timeline which would have much of the essential product functionality ready for the midterm, leaving the remaining time for testing, quality assurance, and additional functionality. The two main deadlines during this period—the midterm and self-assessment— are highlighted in red.

3.2.6 Roadmap

The final result of the planning work resulted in a roadmap which outlined what the team wanted to accomplish in developing the product up until the midterm deadline. The resulting roadmap is shown in Figure 13.

One issue which was encountered was that the roadmap was constructed somewhat haphazardly. No formal structured process was performed to do the planning, and there was only focus on making a roadmap that seemed optimistic in implementing the desired requirements. This point will be returned to when discussing the pre-midterm period, as this may be part of some of the issues encountered with being behind schedule during the pre-midterm development sprints.

3.3 Pre-midterm work

After the project planning phase was completed, work started on implementing the desired functionality in accordance to the road map. The period leading up to the midterm deadline encompassed a period of roughly four-and-a-half weeks. That is, four sprints.

Initially, good progress was made on the product. The team had quickly put together a service which could send some data over the network, and an initial draft of how fragmentation could work – even though this ended up with substantial rework later on in the project.

As the midterm deadline approached the team realized that the report had been somewhat neglected. This would be a trend throughout the rest of the project. Adding to that that some functionality on the road map which required solving unforeseen challenges, the consensus was that the project as a whole was behind schedule. Reflecting on how this occurred, it seemed that many of the team members found the problem outlined by the customer exciting to work on, and would rather develop the product – which in itself is good – rather than the report.

One problem could have been that the team never underwent a more structured planning process. For instance, as part of Scrum it is recommended to do time estimates for issues through activities such as planning poker, in which team members will collectively estimate 'story points' (time unit) to each 'story' (issue) (Kniberg 2015, pp. 38-40). This was never done, and so that may have resulted in unrealistic expectations with regards to progress. In the end, this lead to negotiations taking place with the customer outlining the new speed of development, and subsequently changing the order and priorities of the requirements. In retrospect, using something like planning poker could substantially have helped the team understand the scope of the planned work, though of course unplanned time sinks like unforeseen technical problems comes on top.

Furthermore, planning multiple sprints in advance removes some of the inherent adaptability in agile development processes. When following Scrum, a constant process of re-prioritizing and

Duration	Description
5 min	Explanation of retrospective
5 min	Write 'likes'
5 min	Write 'dislikes'
5 min	Write 'start doing'
5 min	Write 'stop doing'
10 min	In turn each team member reveals and explains their sticky notes
5 min	The sticky notes are categorized in groups of related notes
5 min	Voting to determine importance of notes
10 min	Collective writing of action point notes, to address issues

Table 11: Midterm retrospective schedule.

adapting to changing circumstances occurs, which the team lacked during the initial phase of development. As Kniberg (2015, p. 101) writes: "Reality will not adapt itself to a plan, so it must be the other way around."

3.4 Midterm retrospective

As the team had not been doing much reflection about what was working well and what was not working well in the initial part of the project, it was decided that the team would set aside some time after the midterm, in order to take stock of the situation and do a retrospective of the previous sprints. After all, as Kniberg et al. (2011, p. 99) write: "A great process isn't designed; it is *evolved.*"

The team followed the guidance of Kniberg (2015, pp. 84-90) on how to perform an effective retrospective. Kniberg suggests setting aside between 1 and 3 hours, depending on the amount of discussion anticipated. There was potential for lots of discussion in this project – since the retrospective was for multiple week-long sprints – but at the same time it is good to limit derailments of discussions by having an aggressive time schedule: This can help keep focus high, and force minor or trivial issues to be ignored, thereby isolating the major problems. It was decided to set aside one hour for the retrospective, and as the team has limited experience with retrospectives, it was decided that the suggested retrospective plan close to that proposed by Kniberg would be sufficient. A digital tool was used to perform retrospectives (*Metroretro*)[11], which is pretty close to how Kniberg describes their process, except that "Improvements" is split into two sections: "Start doing" and "Stop doing".

This resulted in the retrospective plan shown in Table 11.

The team decided upon a number of action points in order to address the problems that were encountered. The goal was for these to be specific and actionable in order to hopefully address the underlying issues. The team settled on the following seven action points:

1. Enforce work evenings with standup on Wednesday (Starting 16:00).
2. Reschedule Monday standup to Tuesday 16:00.
3. Add a third standup Friday 11:45 before the FFI meeting.
4. Planning meeting after FFI meeting.
5. Create meeting note documents in advance.

[11] https://metroretro.io/

6. Retrospectives every other week from now on.
7. Rotating the role of scrum master after each retrospective

One issue identified was that many of the standups did not result in much happening. For example the team had a standup at noon on Mondays, which was right after the weekend. Usually that particular standup resulted in everyone saying that they had not been working on anything over the weekend. This was addressed by moving the Monday standup to Tuesday, while introducing a third standup right before the meeting with customer on Friday. This would also have the positive effect of getting everyone up to speed on the state of the project before the meeting with the customer, allowing for less time spent on discussions between group members during the meeting, and more time having productive discussions with the customer.

Most of the team had felt that a non-productive meeting in the middle of the day – which didn't serve a clear purpose – was an unnecessary distraction and so all were happy to make these changes. As a consequence this action point was perhaps one of the more successfully implemented action points, though that may also have been due to changing schedules being trivial in comparison to the effort required for collective behavioural change.

Another issue identified was that the physical meetings – where each team member would meet at the university campus once a week to work together – did not work particularly well. Here multiple conflicting sticky notes arose: On one hand there was a suggestion to meet physically more often. On the other hand there were complaints that such meetings had "little rigor" and were "unstructured". There were also members who disliked "people not showing up to planned physical meetings". As can be seen in the remainder of the report this issue would return being an ongoing issue of contention which the team would revisit in later retrospectives. As an action point to address this the team decided to "enforce work evening[s] on wednesdays", with the understanding that these would be physical.

Additionally, some further administrative issues had been identified. The team frequently experienced that it was unclear who would be taking notes during the meetings with the customer, as well as constructing an agenda of desired discussion topics. In theory this role would be performed on a rolling basis, but in practice it was often a last minute decision who was responsible.

The team decided to include a planning meeting which would occur directly after the meeting with the customer each week. This meeting would have the purpose of ensuring that a meeting document for next week's meeting would be created and that someone would be assigned to take notes. This was shown to be highly beneficial, as there now was a predetermined place to write down any questions to discuss, preventing them from disappearing into the ether.

On the other hand, it was also discussed that this meeting would be used to assign issues and make sure that all members had something to work on for the next week. This detail was quickly forgotten, which resulted in members continuing starting new work weeks not quite sure of what needed to be done. In hindsight, this would have been a good place for some kind of Scrum inspired planning activities, such as backlog refinement, sprint planning and planning poker.

Finally, the team had collectively identified that the frequency of retrospectives was too low. A number of the issues that had been identified and discussed during this retrospective had been ongoing for a long time. Had there been an earlier retrospective it is likely that these issues could have been rectified at a much earlier stage of the project. It was therefore determined that it would be imperative to have a higher frequency of retrospectives during the remainder of the project. Another action point was therefore to have retrospectives every other week in the future. It is as Kniberg (2015, p. 84) writes: "[T]he more stressed you are, the more badly you need the retrospective", a quote the team strove to keep in mind in the future.

3.5 Release v0.1.0

The team's next focus after the retrospective was to get a first release delivered to the customer. The goal for this release was to deliver something that implemented a working API, without necessarily delivering on the more tuning specific requirements. While it would have been nice to have a product that delivered at least some network scenario specific tuning for TCP, a test service would have been needed to systematically test various configurations in order to do this properly. This would have extended the time required for development, and an agreement with the customer was reached. This means the team could focus more on having an iterative – and agile – approach for deliveries. The tuning was therefore postponed to the second release.

The team received some positive feedback from the customer on this release. In particular, the customer liked that the API mirrored that of the Python socket library, making it easy to integrate in pre-existing applications built with the Python socket library.

The customer performed their own testing by implementing the middleware in a *Simple Video Service*[12], that would stream a webcam feed from one device – using the middleware – to another device, both using unreliable and reliable communication. As part of this testing, an issue was detected – where using the 'reliable middleware' resulted in an 'uneven' video feed. FFI requested that this issue would be resolved by the next release.

Overall, the team was satisfied with this release. The most important aspect was to get the product in the hands of the customer, and let them give feedback on the work done so far. The team concluded during the wrap-up in preparation for the second retrospective that the goals and expectations the customer had were met to a satisfying degree. An important takeaway from this product release was the continuous dialog with the customer on the progress and situation with the product leading up to the release.

3.6 Second Retrospective

The second retrospective occurred roughly two-and-a-half weeks after the first. It was postponed slightly compared to the originally scheduled time due to the agreed upon first demo release having its deadline the day after the retrospective was initially scheduled.

The second retrospective occurred on March 28th, and followed the same structure described for the midterm retrospective (Table 11).

Again number of issues were identified, and agreed on a number of action points the team would use to remedy the situation:

1. Have physical meetings of fixed time Wednesday 16-18.
2. Create and agree upon a group contract.
3. Dedicate Tuesday evening to focus on report writing.
4. Limit memes and humor to dedicated channels.
5. Set tuning and testing as main focus for the product development moving forward.

The retrospective started with revisiting the issue of physical meetings, which were agreed to be obligatory after last retrospective. This did not work out that well in practice, with the physical attendance at these meetings being consistently poor, and the meetings being significantly more unfocused than the digital meetings.

The attendance could in some instances be explained by some testing requiring tools that were more easily available at home, but some members argued that the lack of focus during these meetings were detrimental to actual work being done, and that they would rather join the meetings digitally.

[12]https://github.com/EPA1/simple_video_service

On the other hand, other members appreciated the physical meetings despite the lack of focus, feeling that they created an environment where free discussion was easier and that seeing each other physically improved morale.

As a compromise, the team agreed to keep the obligatory physical meetings, but in an attempt to remedy the problem of a lack of focus, the meetings would be held strictly within the designated time and everyone would do their best to focus on project work during those hours. There was no guarantee that this would solve the problem, but the team didn't feel there was any good way of forcing focus without limiting communication.

During the retrospective the team came to the conclusion that a group contract – which had not been made until this point – would be necessary in order to clear up issues related to attendance that had occurred. From about the start of the project, the team agreed to have a system of giving 'marks' to members who did not show up on time to meetings. The rules for these marks were never written and not made totally clear, which resulted in a few members getting marks which they themselves thought were illegitimate. This resulted in an agreement that an proposal to a group contract would be made by the next meeting, describing the rules of attendance and marks in detail, and requiring all the members to sign.

In practice these rules worked great, with a lot less marks being given out, and no complaints about the ones given. They did however compromise the previous action point – about the obligatory physical meetings – by allowing members to avoid these meetings without consequence by simply giving notice the day before. Thus the physical aspect of these meetings became rather optional, which dismayed some members, but forced them to accept that people work best in different environments.

Also related to focus and professionalism, the team decided to reel in the humor and memes after some members felt it had gotten out of hand. Jokes had at certain points gotten into channels with the customer as well as into the report, making the team look unprofessional as well as taking the focus away from the work. This action point worked out well, and the jokes were kept to the appropriate channels and situations, without any further incidents.

It was also agreed that the scheduled work hours on Tuesday evenings should be dedicated fully to writing the report, the reason for this being that most of the work on the product was finished, with only tuning and a few bugs left, while an substantial amount of work was left on the report. By requiring that all work during the dedicated time slot would be on the report, it ensured that the views of each team member would be represented, and that at least a minimum amount of progress would be made to the report each week. This might not have worked perfectly in practise however. Members who already had assigned themselves to report writing in general had no issue continuing this for the writing session, but others who were deeper into the testing and tuning of the middleware putting that aside.

Finally, the team decided to make an action point for having a focus on testing and tuning the product. This decision was made mainly as a formality in order to make sure the team was aware of how the remaining development time should be spent, and made it clear that if one was unable to contribute directly to this goal for the product, that one should rather focus on the report.

This ended up being a well executed action point, as most work leading up to the final release was split along this line; people were either writing the report, or working on tuning the middleware.

3.7 Release v0.2.0

The main goal of the second – and final – release of the middleware was to finally have tuned TCP to different network configurations. In order to do so, testing different middleware configurations with different emulated network configurations was necessary. It was considered beneficial to be able to do this automatically – or at least semi-automatically – and therefore there arose a need to develop a test service that could be used to test a specific configuration of the middleware over a connection by sending data in a configurable pattern of packets sent (e.g. packet size, burstiness, etc.), and reporting the results.

As a part of this release, such a service was developed, and used to create pre-made configuration files for the five network configurations provided in Table 2. This would be the main source of tuning provided to the customer in this release.

This new functionality was not easily integrated into the chat service, and therefore a new service was developed. This test service would need to be able so send data at different rates to emulate the transmission of different services. This testing service also needed to keep track of some statistics to help with the tuning process.

The second goal of the release was to fix the issue discovered by FFI using the middleware with their video service. After testing, it was discovered that the issue was not due to packet loss or en error with the middleware itself, but rather was a quirk of TCP. The *Simple Video Service* generated packets up to around 7000 bytes, but the default fragmentation settings split the packet into segments of size 512. When the receiver calls the receive function, it reads however many bytes has been received thus far from the receiving buffer. This resulted in the receiver being able to read an incomplete packet from the buffer and try to parse that as an image.

To solve this issue, it was decided to modify the service instead of patching the middleware. The reason for this is that by modifying the middleware to make it work for this issue, it would have to make the reliable communication part of the middleware message oriented instead of stream oriented, which in turn would make it deviate from how TCP normally functions. This would introduce unnecessary overhead for all reliable traffic sent via the middleware. In order for the reliable middleware to be easily integrated into existing applications, it is important that the functionality it delivers is similar to the already implemented functions.

The modifications made to the service was put in a separate fork[13] and consisted of changing the encoding from base64 to JPEG, and checking if the last two bytes of the data received matched the JPEG trailer, to try and always construct a valid image.

The feedback for the second release of the middleware was received to to be positive. During initial testing by the customer it was found that the configuration files tuned for each of the emulated network scenarios (Table 2), performed better than the default TCP configuration in Linux. Additionally, no major bugs were found which allowed us to focus on the report and documentation for the remainder of the project duration. In agreement with the customer, it was determined that the product at this point was in a satisfactory state, and the main focus for the team for the rest of the project duration was to document and work on the report.

Overall, the team was once again satisfied with the results, though it should be noted that the scope of the testing was mediocre at best. In part, this fault came from bad time management by the group, and negligence for meetings, as discussed in 3.6.

[13] https://github.com/olavfla/simple_video_service

3.8 Third Retrospective

A third retrospective was originally planned to happen around the time of the second release, but this ended up not happening for practical reasons. Mainly due to having to focus on the product release, as well as other school work and deliveries taking up most of the team members' time.

It was discussed that having the retro would be beneficial, and a way to discuss the path forward now that the development part of the project was over. Yet, all team members agreed that the time spent on going through the formal process of a retrospective, evaluating the development process since the last retro, would be time wasted, and that the time would be better spent on report writing, and the path forward could rather be discussed more effectively in general meetings.

Additionally, the team considered the benefit of a retrospective to be proportional to how much longer one will work on the project, and the project was at a stage where not much benefit would be gained from doing a retrospective, due to the small amount of development work remaining. Kniberg (2015, p. 84) likens not doing retrospectives due to stress with cutting down trees without sharpening your saw, however as the project was winding down, the team decided that it would ultimately be worth to skip out this time.

3.9 Final Delivery

With the release of v0.2.0, the product development part of the project was over, and the team could now fully focus on finishing the report. At this point there were many remaining issues that needed to be fixed. The biggest part missing at this point was in the product section, which had fallen behind compared to the product development progress, and process writing.

Fortunately, due to many of the other courses finishing their main assignments and projects – as well as the product being considered done – the tempo of the report work quickly increased to an all-time high.

The second remaining task was to create a five minute video to give an overview of the project. As a consequence of the effort the team made to document the product while it was being developed, there was little to be done at this stage in the project; only some tweaking and updating to incorporate the customer's feedback.

3.10 Time sheet

A summary of the hours spent on the project for each team member can be seen in Appendix A. As can be seen in Figure 14, the work amount reported by each team member is quite evenly distributed throughout the project, as the team members often worked together.

Overall, the amount of time spent on the project by the team members can be divided into two broad parts; the work done before the Easter break, and the work done afterwards, as can be seen in Figure 15. From the data, it seems that either morale was lower after the break, and/or the team members had more work to do in other courses and projects. Nonetheless, the discussion around the reported time was minimal, and usually as a consequence of having to deliver the bi-weekly status updates.

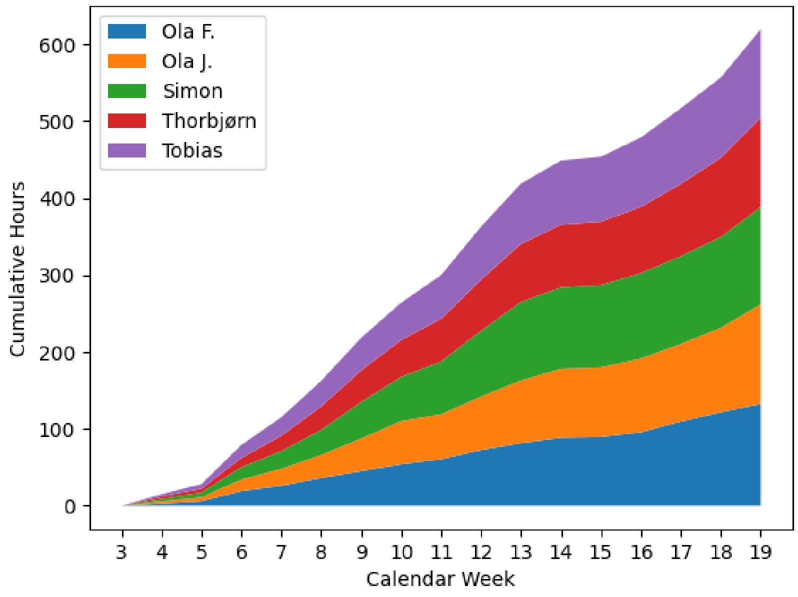


Figure 14: Graph of cumulative hours spent on the project by week

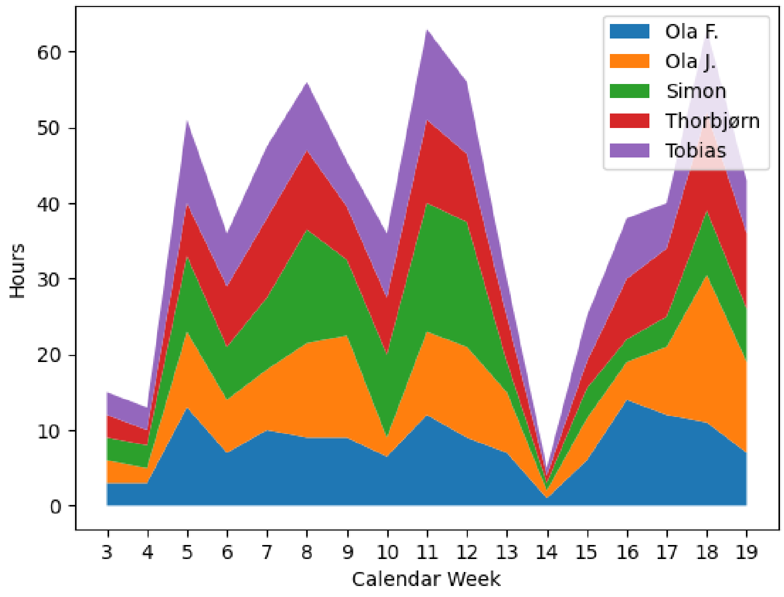


Figure 15: Graph of hours spent on the project by week

4 Conclusion

During the course of this project, an application programming interface was developed to let developers communicate over sockets. This software went through a couple iterations, and through feedback and continuous development together with the customer, reached a state where both parties were satisfied.

The development process was however not flawless, as the team did not succeed fully to follow Scrum as outlined by Kniberg (2015). While this worked to some extent – the team implemented many of the core aspects of Scrum, such as sprints, standups and retrospectives – others were lacking, such as sprint planning, backlog refinement and estimation. However, it became apparent that this was not the best method for a team in this situation to deploy. No-shows were common and the structure of the work week was not optimal, as was found out in the two successfully performed retrospectives during the project. It should be noted however, that even though the team did not successfully implement Scrum, the team managed to stay agile, most likely in great part due to good cooperation with the customer throughout the entirety of the project.

However, it is not all cut and dry. The first half of the project constituted none of the releases to the customer. This is definitely not agile, but in part it could be argued that weekly meetings with the customer and discussions about findings, development choices and problems made up for this lack of feedback.

The final product managed to deliver on every requirement except one, which was deemed low-priority. This shows that the team made an effort to effectively use the limited time with the project in a meaningful way. A good scope for the project which was not overly ambitious also contributed to this, as the customer was understanding of the limited time available for the project. This can also be expanded to the testing portion of the project, which was limited in scope to only include some environments.

As for future work which could be done to improve upon the middleware and this approach, it is important to realise that the middleware as-is does not automatically adapt an approach to tackle changing network environments. Currently, this has to be done manually, but work should be done to introduce a "smart" way to dynamically change e.g. congestion algorithm based on the current network throughput, latency, and other factors.

One of the ways this could be achieved is to make the multiple middleware instances aware of each other and able to communicate status messages, such as their configurations and 'negotiate' what configuration to use. This would have multiple other benefits such as making the middleware able to be remotely configured, and it would also make it easier to find neighboring middleware instances which would lay the groundwork for building an efficient application layer multicast system for improving sending data to multiple recipients.

Another point of improvement identified is to make the middleware API language agnostic. So far the "API" is a python module which naturally only provides support for python programs. It has been discussed within the team that it might be better to implement the middleware as a service running on a port in which an application would send a request over the loopback adapter to connect to another client. This could make it easier to implement some of the functionality mentioned earlier, but will also make it possible to run the middleware in docker which would make it more easily deployable.

It is also possible to improve performance by develop a novel congestion control algorithm which would make it easier to tailor its behaviour to the specified networks based on the given characteristics (latency, packet loss, bandwidth). Another way of further improving this could be to develop a whole new transport layer protocol, but this would likely be an arduous task and would require further analysis to see if the effort would be worth it.

5 Acknowledgements

The team would like to take this opportunity to express their gratitude to Steve Parker, for their valuable advice during the project. Their knowledge helped immensely on deciding how to shape the product, especially in regards to how reliable communication should be implemented.

Bibliography

Andersen, Emil Paulin (2022). 'Evaluating Publish/Subscribe Protocols for use in Constrained Networks'. In: *Master's thesis in Informatics: Software Engineering*, p. 6.

Bonica, Ron et al. (Sept. 2020). *IP Fragmentation Considered Fragile*. Request for Comments RFC 8900. Internet Engineering Task Force. DOI: 10.17487/RFC8900. (Visited on 30th Apr. 2023).

Bova, Tom and Ted Krivoruchka (Feb. 1999). *Reliable UDP Protocol*. Internet Draft draft-ietf-sigtran-reliable-udp-00. Internet Engineering Task Force. (Visited on 29th Apr. 2023).

Congestion Control in IP/TCP Internetworks (Jan. 1984). Request for Comments RFC 896. Internet Engineering Task Force. DOI: 10.17487/RFC0896. (Visited on 29th Apr. 2023).

Eddy, Wesley (Aug. 2022). *Transmission Control Protocol (TCP)*. Request for Comments RFC 9293. Internet Engineering Task Force. DOI: 10.17487/RFC9293. (Visited on 29th Apr. 2023).

FFI, Norwegian Defence Research Establishment (2023). *Information about FFI*. URL: https://www.ffi.no/en/about-ffi (visited on 8th Mar. 2023).

Internet Protocol (Sept. 1981). Request for Comments RFC 791. Internet Engineering Task Force. DOI: 10.17487/RFC0791. (Visited on 29th Apr. 2023).

Iyengar, Jana and Martin Thomson (May 2021). *QUIC: A UDP-Based Multiplexed and Secure Transport*. Request for Comments RFC 9000. Internet Engineering Task Force. DOI: 10.17487/RFC9000. (Visited on 29th Apr. 2023).

Kniberg, Henrik (2015). *Scrum and XP from the Trenches*. C4media.

Kniberg, Henrik, Kent Beck and Kay Keppler (2011). *Lean from the Trenches: Managing Large-Scale Projects with Kanban*. Pragmatic Programmers. Dallas, Tex: Pragmatic Bookshelf. ISBN: 978-1-934356-85-2.

Mike Beedle et al. (2023). *Manifesto for Agile Software Development*. https://agilemanifesto.org/. (Visited on 29th Apr. 2023).

Rochkind, Marc J. (Dec. 1975). 'The Source Code Control System'. In: *IEEE Transactions on Software Engineering* SE-1.4, pp. 364–370. ISSN: 0098-5589. DOI: 10.1109/TSE.1975.6312866. (Visited on 29th Apr. 2023).

Shelby, Zach, Klaus Hartke and Carsten Bormann (June 2014). *The Constrained Application Protocol (CoAP)*. Request for Comments RFC 7252. Internet Engineering Task Force. DOI: 10.17487/RFC7252. (Visited on 29th Apr. 2023).

Suri, Niranjan et al. (2021). 'Evaluating the Scalability of Group Communication Protocols over Synchronized Cooperative Broadcast'. In: *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, pp. 1–9. DOI: 10.1109/ICMCIS52405.2021.9486407.

Thomson, Martin and Sean Turner (May 2021). *Using TLS to Secure QUIC*. Request for Comments RFC 9001. Internet Engineering Task Force. DOI: 10.17487/RFC9001. (Visited on 29th Apr. 2023).

Transmission Control Protocol (Sept. 1981). Tech. rep. 793. 91 pp. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793.

Appendix

A Time sheet

Week	**Ola Vanni Flaata**	**Ola Horg Jacobsen**	**Simon Doksrød**	**Thorbjørn Lundin**	**Tobias Ringdalen Thrane**
3	3	3	3	3	3
4	3	2	3	2	3
5	13	10	10	7	11
6	7	7	7	8	7
7	10	8	9.5	10.5	9.5
8	9	12.5	15	10.5	9
9	9	13.5	10	7	6
10	6.5	2.5	11	7.5	8.5
11	12	11	17	11	12
12	9	12	16.5	9	9.5
13	7	8	4	6	5
14	1	1	1	1	1
15	6	5.5	4	3.5	6
16	14	5	3	8	8
17	12	9	4	9	6
18	11	19.5	8.5	13	13
19	10	12	10	10	10
Tot.	142.5	141.5	136.5	126	127.5

Table 12: Weekly number of hours invested into the project by each group member