Network protocol for distribution and handling of data from JAS 39 Gripen

by

Jonathan Karlsson

2015-09-10
Network protocol for distribution and handling of data from JAS 39 Gripen

Jonathan Karlsson
10 September 2015

Examiner: Simin Nadjm-Tehrani
Technical supervisor: Mats Svensson, Saab
Supervisor: Mikael Asplund
Student reviewer: Marcus Högberg

*Linköping University*

*Department of computer and information science*
Abstract

On board the aircraft JAS 39 Gripen a measuring system, Data Acquisition System (DAS), is sending sensor data to a server on the ground. In this master thesis, a unified API for distribution and handling of the sensor data is designed and implemented. The work has been carried out at Saab Aeronautics, Linköping during, 2014.

During flights with the aircraft the engineers at Saab need to monitor different sensors in the aircraft, including the exact commands of the pilots. All that data is serialized and sent via radio link to a server at Saab. The current data distribution solution includes several clients that need to connect to the server. Each client has its own connection protocol, making the system complex and difficult to maintain. An API is needed in order to make the clients connect in a unified manner. This would also enable future clients to implement the API and start receiving sensor data from the server.

The research conducted in the thesis project was centered on the different choices that exist for designing such an API. The question that needed answering was; how can an existing complex system can be replaced by a publish-subscribe system and what the benefits would be in terms of latency and flexibility of the system? The design would have to be flexible enough to support multiple clients. The investigated research question was answered with a design utilizing ZMQ, pthreads and a design pattern. The result is a flexible system that was sufficiently fast for the requirements set at Saab and open to future extensions.

The thesis work also included designing a unified API with requirements on latency and functionality. The resulting API was designed using the publish-subscribe design pattern, the network library Zero Message Queue (ZMQ) and the threading library pthreads. The resulting system supports multiple coexisting servers and clients that request sensor data. A new feature is that the clients can start sending calculations performed on samples to other clients.

To demonstrate that the solution provides a unified framework, two existing clients and the server were developed with the proposed API. To test the latency requirements, tests were performed in the control room at Saab.
Acknowledgements

By this thesis, I will conclude my studies in computer engineering at Linköping University, Sweden, with a degree in Master of Science in Computer Engineering.

I would like to give a special thanks to my supervisor Mats Svensson at Saab Aeronautics, examiner Professor Simin Nadjm-Tehrani and supervisor Mikael Asplund at Linköping University for the commitment and support during the entire project. Further I would like to thank my life companion for her support and patience.

Lastly, I would also like to thank all those who have helped and supported me during my academic years at Linköping University: classmates, friends, family and teachers. All of you have supported and helped me to perform my best.

Jonathan Karlsson

Linköping, September 2015
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>Boost</td>
<td>Collection of libraries with multiple functionalities and features</td>
</tr>
<tr>
<td>Boost Asio</td>
<td>Network library within Boost</td>
</tr>
<tr>
<td>C++</td>
<td>Programming language</td>
</tr>
<tr>
<td>CVT</td>
<td>Current Value Table</td>
</tr>
<tr>
<td>DAS</td>
<td>Data Acquisition System, measuring system on board the aircraft.</td>
</tr>
<tr>
<td>IP</td>
<td>Internet protocol address, used by the TCP and UDP to calculate paths through the network to the destination host.</td>
</tr>
<tr>
<td>JAS 39 Gripen</td>
<td>Swedish fighter aircraft.</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notification, text-based format used to exchange data.</td>
</tr>
<tr>
<td>Matlab</td>
<td>Advanced calculation software, developed by MathWorks.</td>
</tr>
<tr>
<td>Pthreads</td>
<td>POSIX (Portable Operating System Interface) Threads, unified cross platform threading library</td>
</tr>
<tr>
<td>RSS</td>
<td>Rich Site Summary, publication technology</td>
</tr>
<tr>
<td>Saab</td>
<td>The company that produces the JAS 39 Gripen</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer, cryptography layer added the network communication</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol, a protocol used for network communication</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol, a protocol used for network communication</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language, used to model the design of software</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>ZMQ</td>
<td>Zero Message Queue, network transport protocol</td>
</tr>
</tbody>
</table>
# Table of Contents

Abstract ................................................................................................................. iii
Acknowledgements ................................................................................................. iv
Abbreviation list ....................................................................................................... v

Chapter 1  Introduction ............................................................................................ 1
  1.1  Background .................................................................................................... 1
  1.2  Problem description...................................................................................... 3
     1.2.1  System description .............................................................................. 3
     1.2.2  Requirement specification ................................................................. 5
     1.2.3  Research question ............................................................................. 6
  1.3  Approach ........................................................................................................ 6
     1.3.1  Implementation .................................................................................. 6
     1.3.2  Measurement ..................................................................................... 7
     1.3.3  Limitations ....................................................................................... 8
  1.4  Thesis outline ................................................................................................ 8

Chapter 2  Available technologies .......................................................................... 11
  2.1  Publish subscribe systems .......................................................................... 11
  2.2  Network libraries ......................................................................................... 12
     2.2.1  C++ Network library ...................................................................... 12
     2.2.2  Boost Asio......................................................................................... 12
     2.2.3  Zero Message Queue ................................................................. 13
  2.3  Serialization of messages ............................................................................ 14
  2.4  Threading library ........................................................................................ 14
     2.4.1  Windows threading ........................................................................... 15
     2.4.2  Zero Message Queue threads ......................................................... 15
     2.4.3  Pthreads ......................................................................................... 16

Chapter 3  Publish-subscribe library definitions ..................................................... 17
  3.1  Design choices .............................................................................................. 17
     3.1.1  Choice of transport protocol ............................................................ 17
     3.1.2  Choice of network library ................................................................. 18
     3.1.3  Choice of message serializer ............................................................ 18
     3.1.4  Choice of threading library ............................................................ 19
  3.2  Behavior of the system ............................................................................... 19
     3.2.1  Events ............................................................................................. 21
     3.2.2  Data transmission ........................................................................... 22
     3.2.3  Network commands ...................................................................... 24
     3.2.4  Workflow of publisher and subscriber clients .................................... 27
  3.3  Interface to the network library .................................................................... 29
     3.3.1  Initial setup ...................................................................................... 29
     3.3.2  Publish subscribe connections .......................................................... 29
     3.3.3  Messaging ....................................................................................... 30
  3.4  Solution architecture .................................................................................... 30

Chapter 4  Integration with existing system ......................................................... 35
  4.1  Implementation details ............................................................................... 35
     4.1.1  Broker ............................................................................................. 36
     4.1.2  Publisher and subscriber ................................................................. 36
  4.2  Problems encountered .................................................................................. 37
     4.2.1  Message serializer ........................................................................... 37
     4.2.2  Network system ............................................................................... 37
     4.2.3  Telemetry system ............................................................................. 38
     4.2.4  Matlab client .................................................................................... 38
Chapter 5  System evaluation ......................................................................................................................... 41
  5.1  Test methodology ................................................................................................................................. 41
  5.2  Scalability requirement ......................................................................................................................... 43
  5.3  Latency requirement ............................................................................................................................... 43
    5.3.1  Verification and results .................................................................................................................... 44
    5.3.2  Requirement verification ................................................................................................................ 47
  5.4  Test conclusion ....................................................................................................................................... 48
Chapter 6  Conclusions ....................................................................................................................................... 49
  6.1  Summary of the work ............................................................................................................................. 49
  6.2  Future work .......................................................................................................................................... 49
References ......................................................................................................................................................... 51
Chapter 1

Introduction

As an introduction to this master thesis work the background behind the network protocol for distribution and handling of data from JAS 39 Gripen will be described. The current data distribution solution has problems with the connection protocols and this is described below.

1.1 Background

Figure 1.1 General picture of the system. The aircraft has a measuring system on board that sends encrypted, compressed data to a server at Saab, via radio link. At Saab the data can be displayed in the control room.

Figure 1.1 visualizes the system used to monitor different sensor data from the aircraft JAS 39 Gripen. On the aircraft there is a Data Acquisition System (DAS) that collects the data and compresses and encrypts it before sending the transmission via radio link. On the
ground at Saab a server receives the compressed sensor data from the DAS to be decompressed into individual sensor data. The transmission contains sensor data, bus communication, speed, height and similar attributes. At the recipient server, the telemetry server, all different types of data variables are stored in a Current Value Table (CVT). Connecting clients are able to choose between all the variables and receive packets of the chosen data sent through the network. When a new transmission arrives from the aircraft, the unpacked variables are distributed to all connected clients that have chosen to receive those variables.

There are multiple types of clients that each has its own different connection protocol and parameters of interest. This makes the server communication complicated due to the need for separate handling of each protocol. All parameters in the CVT are given an index that is used in the transmission between the server and the clients. When transmitting samples in the system, the packet includes a timestamp, the index and the actual data. The index corresponds to a name in the CVT and is used by the clients to map the sample to the correct parameter. The DAS also creates the timestamp when the sample was produced and is needed to verify the validity of the samples. The sensor data received by the client cannot be too old since that would disable the test controllers to give commands to the pilot sufficiently fast. It will take some time for the data to arrive at the client for analysis of the parameter. If the sample indicates an error with the aircraft it needs to be detected and adjusted, this puts requirements on the responsiveness of the system.

The system that exists today is complex and difficult to maintain. It consists of the aircraft connected to a telemetry server together with four different clients. Each client, including the aircraft, has its unique protocol used to connect and communicate to the telemetry server. Even the way the transmissions of samples are handled between the server and the clients differ. The number of parameters sent from the aircraft can be as high as 20 000 that is sent at regular intervals. There are different versions of the protocols that create different indexes of the same parameter, which adds to the complexity of the system. The server code has been developed over many years, which has caused it to become complex and full of dead code.

Below follows a description of the most significant clients and their protocols for communicating with the server on the ground. These clients will be referred to in the later chapters.

The telemetry server, or simply, as the server is the most central piece of the system and handles both the data and all connection communication between the aircraft and the clients.

The telemetry client runs a program that displays multiple Windows with the telemetry data from the CVT. This is the most utilized client to view the telemetry data from the aircraft.

The aircraft client uses a protocol to tightly pack the samples before transmission to the server, this to reduce packet size. All the data at the server is unpacked into the CVT. There is a one to one mapping between the aircraft and the server. There can however
be several servers in the network simultaneously where each handles its own separate aircraft.

The client running matlab is used to make advanced calculations on the sensor data. It is run from a Java program to enable running the matlab scripts.

1.2 Problem description

The current solution is very complicated due to the high number of different connections. Additionally some important and interesting functionality is missing. For example, the client utilizing matlab cannot transmit the advanced calculations to the telemetry server, enabling all clients to receive them along with the other parameters. In addition, if a future application wishes to connect to the server they would have to implement a new protocol, which would just add more complexity to the system.

This work aims to investigate the possibilities for a unified API (application protocol interface) that all the above-mentioned clients are able to use instead of the existing solution. The existing clients would have to implement the unified API rather than the current protocols. The idea with the unified API is also that a future application could connect to the server using the API and be treated in the same way as the current clients. The question is how such an implementation can be designed and how latency measurements can be performed. The latency of sending samples from the server and viewing them at the clients needs to be below 200 milliseconds for the data to be relevant for the clients. There can be multiple servers each with a large number of connected clients.

1.2.1 System description

For the remainder of the thesis work the system will be described as a so-called publish-subscribe system. The server corresponds to a broker, clients receiving data are subscribers and clients sending data are publishers. Publish-subscribe systems are pretty common usage for distribution systems, where users have similar responsibilities as described for the problem (Fateri, Ni, Taylor, Panchadcharam & Pisica, 2012; Patel, Rivière, Gupta & Kermarrec, 2009). Publish subscribe systems are described further in section 2.1.
Figure 1.2 describes a sample diagram of the current solution, the boxes indicate a client or a program running on a client connected to the server in the middle box, ovals indicate the protocol used for the connection, and the arrows indicate the current data flow. The parentheses in the boxes indicate the terminology to be used by the new unified API.

Figure 1.3 displays a diagram of how the solution should look like. The aircraft protocol still exists, this because it is optimized to handle encryption and compression of the transmissions. The other protocols have been upgraded to the new unified API. The boxes indicate a subscriber or a publisher connected to the broker in the middle boxes, ovals indicate the protocol used for the connection, and the arrows indicate data flow.

Figure 1.2 shows the architecture of the current system where all clients are described with their current connection protocol to display the complexity of the system. Figure 1.3 shows how the API has replaced all protocols and the broker only has subscribers and publishers to keep track of. The broker can disregard the different types of clients and the previous protocols that used to be followed. If a client wants to contribute parameters, it can connect as a publisher and add parameters to the CVT of the broker. At that event a
notification of the additional parameters should be sent to the subscribers. With the existing clients, only the matlab and the simulator client will contribute parameters using the API, but it might also be relevant for future applications to send data to the network. Those clients will be both subscriber and publisher simultaneously. The aircraft cannot implement the new unified API since it is optimized to send compressed and encrypted data and should not be modified unnecessarily.

1.2.2 Requirement specification

The developed system is expected to fulfill the requirements listed below.

1. Clients must be able to connect as both a subscriber and a publisher and gain access to functionality of both roles if needed.

2. Network monitoring must be possible, thereby enabling a latency measurement of the network.

3. Multiple publishers must be able to send data to the broker. This extra data should be added as new parameters with incrementing index to the CVT.

4. Brokers must listen to a specific broadcast from subscribers and reply with a message identifying themselves as brokers. Thereby the requesting subscriber knows where to connect to receive data. Connecting subscribers detect brokers using this functionality.

5. The broker must have another value table for the hardcoded, unpacked data stream, containing for example sound and video. Subscribers should be able to request data from that CVT to begin receiving streams of selected parameters. With this feature the need to hardcode this stream is removed.

6. In the existing solution all parameters are stored and sent as the data type double instead of the actual data type. For example all Booleans are stored as doubles even though they only require one bit. The API must identify what parameters could be stored as other data types than double and send that information in the initial setup ensuring the subscribers know what types to receive. The thesis work should implement this feature for at least two parameters. The goal is to show how the feature works and that it saves communication bits. The remaining parameters are open for the extended functionality when needed.

Requirements 1-6 are mandatory for completion of the API within this thesis work. Requirements 5-6 are available for implementation if the time allows. If possible due to project considerations at Saab, the thesis will fully implement the API to all of the current clients.

Requirement 2 is necessary when testing different solutions for optimization purposes. The monitoring function will be a helpful tool for comparison. A large system can consist of several brokers, each connected to their own aircraft and with over 50 connected publishers and subscribers. That big of a network can cause issues with latency, thus the
measurements are important when adding functionality to make sure the system scales up. This thesis work will use one broker with two subscribers and two publishers connected to test the functionality of the network. Large scale testing will verify the latency parts of the requirements.

1.2.3 Research question

The research question this work intends to investigate is how an existing network system can be adapted to a publish-subscribe system. The research include tests on the latency of such a solution and discussion on what the benefits would be in terms of flexibility and feature improvements.

1.3 Approach

To answer the research question, existing libraries that handle certain key aspects of the system will be compared to form a theoretical background. The goal is to create and implement a new solution to the described problem that will add flexibility and maintainability to the system. Together with this, measurements are to be taken in order to verify that the solution meets the requirements on latency. The first key aspect that will be considered is the network library since that component is very central to this type of system. The second aspect is the serialization of messages since that can potentially be a bottleneck of the system that can save a lot of latency on the network. The third and final aspect to be considered is the threading library since that has a big impact on the system design.

Flow charts will be created that define how the system is expected to behave in certain situations. The findings from the existing libraries and solutions should provide a solution to the flow charts, to be implemented in the system. UML (unified modeling language) diagrams will be used to describe the solution library that is to be created.

1.3.1 Implementation

In order to perform the necessary measurements and to verify the system design, the libraries will be implemented into the current solution at Saab. The first step of the implementation is to create a dummy broker, subscriber and publisher that utilized the designed architecture on a local computer. This will be the starting point of the system and will also be used during the remainder of the implementation phase for testing and debugging. The dummies are as simple as possible and only perform the tasks necessary to verify the behavior of the specific part of the system.

To ensure realistic testing the broker must be implemented into the telemetry server and the subscriber into the telemetry client. This enables testing of the functionalities of the broker and the subscriber parts as well as network communication with samples from the aircraft.
To add another subscriber to the testing as well as a publisher, the client utilizing matlab will be adapted to utilize the functionalities of both subscriber and publisher. This will enable the telemetry subscriber to receive data from the publisher as well as the broker.

1.3.2 Measurement

To fully answer the latency part of the work, a number of tests produced data for graphs displaying the latency with different system setups. If the latency scale with increasing size of the system and the latency is still below the requirement for systems with at least a total of \( N = 20 \) nodes, the API can be considered a good solution to the problem. If \( N = 20 \) is not a feasible number, the test will show a feasible number \( N \) for the architecture. The graphs should show the number of connected publishers and subscribers, data size, average latency as well as minimum and maximum latency and be used for comparisons to one another. Table 1.1 shows the different setups that will be tested in the thesis work to ensure that the latency requirement is fulfilled. In the tests, all publishers and subscribers will be set to send and receive the same amount of data each second to simplify the comparison between graphs.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>One broker, one publisher and one subscriber</td>
<td>Minimum network should mean very low latency.</td>
</tr>
<tr>
<td>One broker, one publisher and two subscribers</td>
<td>Should be about one third more data flowing due to one extra node, meaning an increase in latency. Here the publish-subscribe system should be beneficial and reduce the latency.</td>
</tr>
<tr>
<td>One broker, two publishers and one subscriber</td>
<td>Should be slightly higher latency compared to the above case. This case will not be as beneficial as the above case since the increased amount of traffic cannot benefit from the distribution system.</td>
</tr>
<tr>
<td>One broker, two publishers and two subscribers</td>
<td>Should be double the latency compared to the first test.</td>
</tr>
<tr>
<td>One broker, one publisher and at least ( N ) subscribers</td>
<td>A test designed to see how many subscribers could connect to the network before the latency overflows.</td>
</tr>
</tbody>
</table>
Table 1.1 setup of tests that will ensure the latency requirement, $N = 20$, if feasible and if not decide what is the feasible number for $N$ with the chosen design.

### 1.3.3 Limitations

To limit the scope of this work, not all clients in the existing architecture will be implemented with the new API. In order to test the unification of the API as well as the latency of the system, this thesis work implemented the API to two of the current clients. One client will act as both a subscriber and a publisher and the other as subscriber and broker. The API is deemed sufficiently unified when those clients are successfully implement into the API. This requirement is deemed sufficient to support the rest of the clients as well as future applications that would need to access the system. The telemetry server has to be updated to act as a broker for the new API system to work. Several brokers must also be able to coexist in the system to enable the support for multiple aircrafts. The clients that will be modified to use the API are:

- The telemetry client because it is the most central one and will be using the subscriber part of the API.
- A subscriber client running matlab because it is a central part of the system. A different protocol for initialization is used compared to the telemetry client. If the API can be used together with both subscribers it is likely it will work with other clients as well.
- A client running matlab as a publisher since it is a feature that is useful and today only the simulator and matlab client are relevant for sending data simulator is a complex program and will not be included in the thesis work.

The remaining current clients are scoped out of this work.

### 1.4 Thesis outline

- Chapter 2 covers the interesting technologies that were considered when designing the system. It includes the different choices and some discussion why one was preferred over another.
- Chapter 3 describes the workflows mentioned in section 1.3 as well as UML diagrams of relevant parts of the solution architecture.
- Chapter 4 describes the integration of the libraries developed in this thesis, into the current clients and the encountered problems.
- Chapter 5 includes the measurements and conclusions about the measurements. This allowed the confirmation of the requirements.
• Chapter 6 contains notes on the benefits of the system as well as some future improvements.
Chapter 2

Available technologies

This chapter gives a theoretical background to some of the aspects that needs to be considered when designing complex network systems with multiple clients. The chapter also contains discussions about available technologies and how they can be utilized to create a solution. For the problem described in Chapter 1, the relevant parts to look at is how to design an API with low latency.

2.1 Publish subscribe systems

In section 1.2.1, the publish-subscribe system was introduced. An example of a publish-subscribe system is the Rich Site Summary (RSS) system that is often used for exchange of news. With a publish-subscribe system users can monitor RSS feeds and register subscriptions and as soon as a match is found, get a notification of the new message. The request can be issued with certain condition to filter out certain news ensuring the users only get news of specified interests.

It is important to understand the problem in order to create a model that is suitable to the solution (Aries, Banerjee, Brittan, Dillon, Kowalik & Lixvar 2002). It is further important to use the model during tests. There is a design pattern called publish-subscribe that can be used to design such systems. This can be used as a starting point during the design of the architecture. This is well suited to be used when creating the solution API to Saab’s existing system. The design pattern gives the system relevant classes and functionality with loose coupling and scalability. When designing distributed architectures it is good to develop it in different service layers (Cavage, 2013). By using the design pattern it is easier for new developers to include new functionality to the system and such layers can be achieved. Since the system is built around a broker in the middle, neither the publishers nor the subscribers need to know about each other. All communication can go through the broker, which gives a simplified system in terms of creation of networking sockets. With the broker as a hub the subscribers and publishers are only required to know the IP address of said broker. The main advantage on a high design level of the publish-subscribe design pattern is the network scalability as well as a more dynamic topology (Yaxiong & Jie, 2013). The drawbacks of the broker topology include the workload for processing all requests, but that is comparable to the workload of the server topology. Compared to
serving the subscribers directly, the publishers will have an easier task with the broker topology. The reason for this is that the publishers are significantly simpler compared to the regular servers and have less responsibilities compared to the broker (Bastião, Costa & Oliveira 2013).

A drawback is that all parties must know the data structure of the publication feeds. Since the data structure needs to include some way to subscribe to certain headers the message size is increased compared to other network systems. This could be problematic for the requirement of latency. This means an increase in complexity of the solution, but this also adds structure to the messages, which is a good thing.

2.2 Network libraries

Two very important transport protocols are UDP and TCP. These connection protocols are utilized on the Internet today. UDP is a best effort protocol where bits are streamed through the network in the simplest way possible. Speed and simplicity is prioritized over reliability and procedures to guarantee delivery and order of messages.

TCP uses a few steps to establish a connection, sends acknowledgment messages for each packets sent and has steps to guarantee the delivery of messages, making it a safer but slower protocol. Additionally the TCP protocol makes sure that the messages arrive in the order they were sent, something that is not done by the UDP protocol. For every network system implementing low layer network communication, a protocol needs to be chosen. The system needs to be built around that choice since it makes the system handle messages very differently (Kurose & Ross, 2009).

2.2.1 C++ Network library

C++ has its own library for network programming, cpp-netlib, where the developer has to setup the sockets manually. That includes how the listening and receiving is to be done as well as create the network layer headers for the messages that are to be sent. The headers include IP address of sender and receiver, number of bits in message, some flags for setting different statuses, etc. For the C++ network library, the user needs to define the headers that are to be created, IP addresses and lots of other settings that can be quite troublesome. The gain is full control of the network as well as room for optimization (Hall, 2011).

Most programs use several sockets for receiving different types of messages. A common praxis is to create new threads with its own sockets for each new client that connects enabling the setup to extend to several threads (Kurose & Ross, 2009).

2.2.2 Boost Asio

There exist several libraries that add layers of abstraction to the socket programing and hides parts of the headers and the setup process for the sockets as well. The libraries have their own functions for sending and receiving messages that are usually built upon the C++
library. In addition to making the program easier to use, the library handles the allocation process and socket setup (Demming & Duffy, 2010; Schäling, 2011).

Boost is a big library pack containing several libraries where one of them is called Asio and handles network communication. Performance tests utilizing this library show that it is sufficiently fast, from the comparisons of (Bush, 2010). Some advantages of Asio are that it is cross platform, supports TCP and UDP, has asynchronous operations and supports SSL (Secure Socket Layer) connections.

### 2.2.3 Zero Message Queue

Zero Message Queue (ZMQ) has a higher number of functionalities compared to Boost Asio and is also cross platform with support for SSL connections (Akgul, 2013; Hintjens, 2013). ZMQ is built upon TCP but has some features that are similar to UDP when choosing between different socket types that are implemented. An example of such a feature is the publish-subscribe socket that are described below and in some sense comparable to the broadcast signaling. The sockets have different setup with different properties and use cases. All traffic utilizes TCP and graphs in the documentation show that it is fast compared to other networking systems. The system will have to be modified to fit into the design of ZMQ since the whole architecture is a bit different compared to the regular client server approach. Since ZMQ has a setup that is similar to the publish-subscribe design pattern this is not considered to be negative (Akgul, 2013; Hintjens, 2013).

By initializing the message structure, certain functions for sending and receiving can utilize the message for automatically sending it through connected sockets, either on different threads or through the network. ZMQ doesn’t know what type of data is transmitted thus it sends the number of bits assigned by the user with no knowledge about the content. It is the software that owns the socket to de-serialize the message (Akgul, 2013; Hintjens, 2013).

The different types of sockets are publish-subscribe, request-reply, push-pull and router-dealer as well as some combinations of the different sockets (Ashish, 2014). The choice of socket depends on the needed functionality and level of complexity. Thesimplest sockets to use are the request-reply pattern, where a server takes request from client that needs a reply. This makes for a regular client-server connection and ZMQ handles all the headers needed for establishing the connection. Flexibility is an issue, for each request that comes to the server must respond with a reply since ZMQ keeps all information about the client until a reply has ben sent. Moreover, there is no way for the server to spontaneously send information to the client. One limitation is that the client can only connect to one server on one socket. This makes the use case of sending requests to multiple servers include either re-connects or multiple request-sockets.

All messages for the publish-subscribe pattern of ZMQ include a header consisting of an identity string and an empty string as separator. For messages to arrive at the subscribers, a subscription must be made to the subscribe socket. The subscription call includes the specific headers of the publication message. Now the pattern of ZMQ automatically filters out messages where the header does not match. ZMQ allows the clients to subscribe to
multiple headers from multiple publishers adding rules to the filter and receiving more messages. For the client to send information to the server a request-reply pattern could be used in parallel to the publish-subscribe pattern as it only goes in one direction (Akgul, 2013).

The router-dealer pattern is a more advanced version of the request-reply pattern. It adds a requirement for the same header system described for the publish-subscribe pattern. There is no requirement for each message to have a reply making the pattern is more flexible and have a wider range of use-cases compared to request-reply. Some drawbacks are the need for knowing whom to address with the certain headers as well as the added complexity that needs to be addressed and handled in a correct manor for the system to work properly (Ashish, 2014).

The push-pull pattern is used to distribute work tasks to multiple consumers in a pipeline. The consumers can simultaneously perform the tasks and a final collector can collect the results (Ashish, 2014). As this is not considered relevant for this thesis it will not be discussed further.

### 2.3 Serialization of messages

To minimize the network load, messages are serialized and sent in one chunk of data that needs to be de-serialized before the message can be read. The simplest solution is to copy all the data into a large buffer and send the amount of data copied. For complex data structures containing multiple data types, packing the samples tightly can save a lot of data to be sent through the network. For example, a Boolean can be stored using one bit rather than storing an entire integer, as often is the case. This will save 31 bits and still send the same amount of information. By removing irrelevant bits the message size decreases and thereby both latency and throughput are gained in the network.

All message that are packed need to be unpacked before they can be read by the software and later the user. This can create an overhead for complex serialization techniques, but for large messages that time is often gained in terms of latency.

The MessagePack library is efficient when it comes to compressing data to be sent through the network (McAnlis & Lubbers, 2014). It saves 63 percent more data compared to plain JSON (JavaScript Object Notification). The reason for this is that JSON uses a data structure that is directly readable to humans, whereas MessagePack compresses the data into binary information and thus compromises that readability. For large data sizes this is far superior and can save a lot of network performance.

### 2.4 Threading library

Threads in the computing world are parts of programs with a specific task to perform. The operating system kernel frequently switches the thread that is allowed to run and the illusion of parallelism is created enabling several program parts to run simultaneously. The conventional design is that the server creates a new thread for each new connected client
and let that thread handle the requests of the client (Hall, 2011). This design gives the main thread the responsibility of listening to new connections and creates new threads that handle the client. The threads only communicate with the client and have no dependencies towards other threads. The operating system will allocate time for each thread to run and thus multiple clients get equal amount of time for their request (Carver & Tai, 2006; Duffy, 2008; Larsen, Sarangam, Huggahalli & Kulkarni, 2009). The solution proposed in the thesis should utilize threads since it can add flexibility and be used to group common functionalities from brokers, subscribers and publishers. Different threads, depending on responsibility can handle the different parts of the system. This gives more flexibility for modification on those parts without affecting other parts of the system.

2.4.1 Windows threading

In the current solution, the server program compiles a list of parameters that all clients listen to. Every time the server sends data it fills a buffer with data and then uses a UDP broadcast to all clients. The clients then receive the entire buffer and have to search for the parameters they need. The main thread in the current networking library basically consists of one thread that in a while loop checks for Windows events and handles all events separately, thus utilizing a single thread for the network.

The current system is based on the Windows threads and utilizes the built-in event system of Windows to signal what is happening between different libraries that are included in the system. A library that handles the user input, from the user interface, sets an event for the new command and a different library can handle the command by reading the event and catch the new command. The biggest concern about the Windows threading system is that it isn’t compatible with Unix systems and thus isn’t portable to future implementations.

Threads in Windows are easy to understand and use. It is easy to pass variables to the thread and Windows create parameters such as the thread ID automatically. Windows also comes with special events, such as interrupts and timers, which threads can wait for. The thread will sleep until the events occur and not occupy kernel time rather than idling. Other threads then trigger the event and wake up the sleeping thread ensuring that it can handle events appropriately (Duffy, 2008).

2.4.2 Zero Message Queue threads

Zero Message Queue (ZMQ) has its own threading system that is very different from other threading libraries. It basically adds a layer of abstraction above such things as semaphores and mutex locks and instead uses the built-in network message protocol to send messages between threads and can thus distribute workload and events through both network and local sockets. The advantages for the programmer are the absence of mutexes that can be quite troublesome and cause strange errors. This causes the system design to look quite different from what threading used to look like and it requires a bit of redesigning (Hintjens, 2013).
Since ZMQ is cross platform, the threading is also cross platform, which is good compared to the Windows threads. There exist few examples and references of how to implement ZMQ threads (Akgul, 2013). Additionally the different system design could mean it is more difficult to extend the API in the future.

2.4.3 Pthreads

Pthreads (POSIX threads) is a cross platform library for creating threads (Carver & Tai, 2006). The library includes mutex locks, semaphores as well as methods for synchronization between threads. There is no specific event functionality like the one in Windows threads, but the threads can wait for a condition to be met. Using a mutex lock and a shared variable, a different thread can set the condition. By setting different values to the shared variable, multiple events can be issued with the condition functionality.
Chapter 3

Publish-subscribe library definitions

This chapter describes the chosen solution libraries and intentions of how they are to be integrated into the system. The chapter describes the workflows of the three libraries to give a basic idea of their differences and similarities and their relevant events and responses to certain events. The second part contains a more detailed explanation of how external library functionality is utilized, as this is very central to the solution. The last part of the chapter includes UML diagrams of the solution and describes the structure of the implementation using class diagrams.

3.1 Design choices

In this section, the research conducted in Chapter 2 is discussed and design choices for the different parts of the system are made.

3.1.1 Choice of transport protocol

In Saab’s existing system both the TCP and the UDP protocols are used. TCP is used to set up the system and UDP is used for distributing the samples across the network. The UDP protocol includes a broadcast functionality that is utilized by the system. This makes every device on the network receive the broadcasted data, regardless of whether it is a receiving client or not. The broadcast system makes a simpler solution compared to packing the specific samples and sending the messages individually to every client. It is however relevant to investigate the possibility of sending the parameters using the TCP protocol instead of UDP, because TCP is more reliable and includes congestion control. Additionally the broadcast system in UDP could be replaced by a system sending only the requested data to each client. This forms a solution to one of the most relevant parts of designing the unified API, the requirement on latency.

With the help of Wireshark (Shimonski, 2013) it was shown that during the setup process of the current system there is a high demand for communication through the network. Most importantly are the demand for the CVT but also the requests for different parameters as well as basic connection messages. Both publishers and subscribers need to contact the brokers in order to setup the connection and request the parameters. In the request the transmission status of the parameter is included, receiving for subscribers and
sending for publishers. The part of the system with the highest requirement on bandwidth consists of sending the samples from the aircraft through the broker, but also the additional data that the publishers produce. Because of that reason the choice of library for handling the network communication becomes central for the system and can lead to relevant changes in the rest of the design.

All this leads to the choice and usage of the network library. The functionality needed for network communication is implemented in the network library. This raises the abstraction level of the system and makes it easier to develop and hides the lower levels of the network system. If future needs require changes on a network level, it is fairly simple to upgrade to a new network library that meets the required changes. There exist multiple libraries that handle network communication with different advantages. The choice should have good documentation, fulfill the requirements discussed above and fit into the design of the API.

### 3.1.2 Choice of network library

ZMQ was the chosen network library since it had many features that fit well with the expected behavior of the system. TCP is utilized by ZMQ, which is the chosen network protocol. The publish-subscribe pattern is to be used to fulfill the goal of implementing the design pattern described in section 1.2.1. By utilizing the router-dealer pattern the communication from the subscribers and the publishers to the broker can be achieved. Furthermore, the ZMQ seems to be sufficiently fast for the requirements (Ashish, 2014).

### 3.1.3 Choice of message serializer

The simplest way to pack a message, which is already utilized in the current solution made by Saab, is to copy all the content into a large array and then send the whole array through the network. The array is not compressed in any way, making the solution take up maximum amount of bytes and thus hurt both throughput and latency.

This made the initial choice of message serializer the MessagePack. It seemed easy to use and seemed to make an important upgrade since the number of transmitted bits would be heavily reduced. A test program was written that implemented MessagePack to serialize a sample transmission in order to test how MessagePack was working. The resulting program could successfully serialize and de-serialize the data structure of the transmission.

This choice was redefined to use a large array in section 4.2.1. The motivation for this is that when the implementation was integrated into the API, it turned out that MessagePack saved a few bits in the buffer that was used during de-serialization. Those bits were not part of the network communication of ZMQ. This caused the subscribers to be unable to de-serialize the transmissions. After a session of debugging it was determined that the gain of using MessagePack as a serializer was not worth the effort of implementing it. The current solution is not using any serializer and is still sufficiently fast. It was thus concluded that the new API would also use the large array solution when sending the transmissions. A future optimization would however be to implement a serializer such as MessagePack.
Another serializer that was considered was the one included in the Boost library pack. However, it was concluded that it was not worth to spend more time to investigate the possibilities of the serializer during the thesis work.

### 3.1.4 Choice of threading library

Windows threading is utilized by the previous solution at Saab where incoming data, user input and timers each trigger different events. A main thread that distributes the events to the correct handlers receives all events (Carver & Tai, 2006). The reason why Windows threads are not chosen are due to not being cross platform compatible which would mean a serious limitation to the API.

The unified API uses pthreads as the threading library due to the fact that it can be used cross platforms in comparison to Windows threads. Additionally there are more documentation on pthreads compared to both Windows and ZMQ threading. Although some of the benefits from fully using the ZMQ library are lost by using pthreads the solution was easier to get started with since pthreads have more examples and got more advanced features to be utilized. Combining ZMQ network sockets with pthreads proved to be a smooth and powerful solution that solved the problem efficiently.

### 3.2 Behavior of the system

As discussed in the previous chapters, the existing solution at Saab has been redefined using a publish–subscribe system description. The new API will implement ZMQ with the publish–subscribe pattern as network layer. The ZMQ functionality will be built into a common class used by the entire system. This will make it easy to change the network implementation. Pthreads is to be used to separate the API from the main part of the program, but also within the networking functionality to ensure that the networking system is not blocked or delayed.
A complete overview of the system is described in Figure 3.1. At the top of the figure the main program is defined with three components. All main programs that are to implement the API need to be modified to utilize the functions described in the interface parts. The arrows indicate communication to the interfaces of the certain libraries. Listed under each interface are the functions that are accessed by the main program. The network manager spans across all library classes as it acts as a superclass for all the network functionality. The interface will create an instance of the broker, subscriber or publisher, depending on which interface is used. This chapter will describe the interactions of the library modules, focusing on the common functionality of the modules. The figure precedes Figure 3.11, where a more detailed view of the Broker-, Subscriber-, and publisher implementation level is described.
3.2.1 Events

A precondition of each broker, subscriber and publisher client is that each respective initialize function of the interface is called at startup. The main part of each client consists of a loop that calls the respective wait for event function, brokerWaitForEvents, subWaitForEvents and pubWaitForEvents. The functionality of each event is described using workflows in the remainder of section 3.2. The workflows show how the system is expected to behave in certain key situations and forms the basic system design. The basis for the workflows rests in both the requirements and the functionalities defined in section 1.2.2, as well as the design choices made in section 3.1. Following the standard UML design, for all the workflows listed and described below the arrows indicate the next procedure. The diamonds are used to show different choices and the boxes indicate the next state or action to take. The boxes with rounded corners are described in other figures.

![Workflow Diagram](image)

**Figure 3.2** Workflow of the broker and the different events that occur. The transmit data event and the handle subscriber are returned to the broker client by the brokerWaitForEvents function of Figure 3.1. The transmit data box is described further in section 3.2.2. The handle subscriber and handle publisher boxes are described in Figure 3.6 and Figure 3.7. The handle command box is described in section 3.2.3. The unknown event box is returned directly to the main program to be handled by the user.

Each library utilizes a thread that listens to ZMQ sockets where different messages arrive. A function for handling the messages is called that will determine the command. For the network commands the most significant one is the broker identification system, where subscribers and publishers can send a command to all brokers to identify themselves. By using the identification system from ZMQ, the brokers can subscribe to the string “BROKERS”. Thus the subscribers can make the identification request with the same
destination and only the brokers will listen. Each request includes a source identifier, thereby the brokers can send a private reply and let the sender know the brokers’ individual identities. Events include signals indicating that the CVT has modified and devices that connect or disconnect from the network. For these events the broker can use either the strings “SUBSCRIBERS”, “PUBLISHERS”, “BROKER”, or “ALL” to call only the nodes that are affected by the events. This represents different paths in Figure 3.2. For example, when the CVT is modified only the subscribers need to know about it. The “SUBSCRIBERS” string can be used and the publishers will not receive that event. The “ALL” string indicates relevant information for brokers, subscribers and publishers.

The interfaces from Figure 3.1 have similar functions that access the network. The \texttt{waitForEvent} functions will make the thread sleep until some event occurs. At that time, the main program is poked. Indicating that the event needs to be handled. The two upper rounded boxes of Figure 3.2, transmit data and handle subscriber are returned by the wait for event function to the main program. The transmit data event is implemented as an event in the pthread library. The main program needs to know what parameters to pack at transmission times, thus the handle subscriber event must be returned by the \texttt{waitForEvent} function as well. The broker library handles the remaining event of Figure 3.2 directly and the main program is not notified of the occurring events. The unknown event is also passed to the main program to let the user know something unexpected has happened. This is implemented as the default action and should not occur.

### 3.2.2 Data transmission

![Figure 3.3](image)

*Figure 3.3* The structure of the transmissions. The squares indicate actual content of the message sent through the network. The integer at the top equals the total amount of bytes sent in the buffer. The index in the head-part is the CVT index and the count is the number of samples of that index, since there can be multiple indexes sent in each transmission.

Figure 3.3 defines the structure of the transmissions used by the system. Each message begins with the amount of bytes that are sent in the buffer. This is used by the subscribers
when iterating the buffer to know when the message ends. For each new parameter in the
transmission, a header is included with the index of the CVT list together with the amount
of samples. Some parameters are updated more frequently than the transmission timer,
meaning it will contain several samples of the same index in each transmission. Thus the
count variable in the header equals the number of samples sent.

Figure 3.4 The workflow of the algorithm used by the brokers and publishers for sending
samples to the subscribers. The packets are stored in a buffer that is sent through the network.

To trigger the transmit data event, a timer was set to twenty milliseconds. This is also the
case for the existing implementation by Saab. The timer controls the function for
transmitting data. The algorithm, described in Figure 3.4, shows how the samples for new
transmissions are packed into a memory buffer that is to be sent to the subscribers. The
buffer starts with the size of the complete packet, enabling the subscribers to know how
much data to read. The publishers have different responsibilities and use a different
workflow than that of the brokers. However they use the same algorithm when sending new
samples. The box with the text “add all new samples with data and time” is a loop that
queues all samples one after another.
In Figure 3.5 the algorithm used by the subscribers when handling the incoming samples is described. Since the samples are assumed to be stored correctly using the defined structure it is simple to read the data from the buffer. The subscribers have subscribed to the brokers and publishers from which they need data, but it is not known in what transmission the wanted data is sent. Therefore the subscribers must check every transmission to look for the parameters of interest. After reading the size of the message from the top of the buffer both the structure and the size of the message is known. Using that information it is quite efficient to traverse the buffer in search for the parameters that are needed.

3.2.3 Network commands

The brokers have separate storage lists for the publishers and the subscribers to keep track of the parameters that each of them handles. The lists are also used to keep track of what parameters the publishers provide and what parameters are to be sent to all the subscribers. When the ZMQ network messages arrive the broker must check the command request of the message and handle it accordingly to the type of sending node. Requests from the different types of nodes have many overlapping commands, such as adding and removing parameters. The commands act over the different storage lists and issue different responses and events. Additionally the subscribers have more commands compared to the publishers since they have different responsibilities and needs.
Figure 3.6 displays the commands that are issued by the publishers. It includes the commands for adding additional parameters of subscription, as well as removing the parameters. When new parameters are added, the broker must set the indices of the new parameters in increasing order from the last index in the CVT. The broker replies the publisher with the given indexes, enabling the usage of the new parameters in the system transmissions. Additionally, the subscribers get a notification of the new parameters that are available. The broker library software handles the notification and the main program of the broker does not get any notice of this since it is not affected by the event.
Figure 3.7 displays the actions taken by the broker during events issued by subscribers. These actions to the events issued by the request parameter, add/remove parameters commands have many similarities compared to the publishers’ actions. When the subscribers add or remove parameters from its subscription list, the broker is used to communicate the information to the publisher. The broker must notify the affected publisher of the change of subscription. That event is handled by a private message ensuring it only concerns the issued publishers. At the request parameter command the broker replies with a list of the parameters available at the CVT, as well as the potential parameters at the connected publishers. The reply to the issuing subscriber contains a list of identities of those responsible for the parameters enabling subscription to the responsible publishers. The software for the subscriber library automatically subscribes to the identities of the message. The data will start flowing by utilizing the ZMQ publish–subscribe functionality. To ensure a flexible system where each node only knows what it needs to know, the list sent from the broker to the subscriber is kept short. This means that the list might not even contain the identity of the broker thus allowing the subscriber to only receive data from a single publisher.
The network performance command is used for testing the network latency during the final phase of the thesis and can be disabled by a define command in the setup. If enabled, the subscriber makes a time stamp attached in a message to the broker that identifies the command and resends the exact same message back to the subscriber. When received at the subscriber a new time stamp is taken and the two times can be compared. This gives the subscriber the round-trip time of the message, including the overhead for handling the message at the broker. This is explained more extensively in section 5.3 with graphs of the measured performance.

Both Figure 3.6 and Figure 3.7 contain a disconnect command that is issued when the respective node disconnects from the broker or is killed by the user. For the publisher this means that the CVT is modified and all subscribers must be updated with that information. When a subscriber disconnect it will only trigger an event to the publishers with subscriptions from said node. The reason for this is that not all publishers are affected by the event. The main program of the broker is also notified about the subscriber event, since it handles a list of all the parameters with subscriptions that need to be updated. At the issuing side of the node, all allocated memory is freed, the threads are killed and the program terminates.

The user of the main program of the broker issues the shutdown command in Figure 3.1. The shutdown concerns all connected nodes and a broadcast message with the identifier “ALL” is sent. The message ensures that all subscribers know that no new data will arrive from the broker. The publishers can then stop transmitting, since the data they transmit is only assumed to be relevant to that of the connected aircraft at the broker. If the aircraft isn’t there, the published data isn’t relevant any more. After all the devices have received the shutdown event, the broker frees up allocated memory, all threads terminate and the main program can now shut down. The subscribers can still be connected to other brokers and don’t need to shut down.

3.2.4 Workflow of publisher and subscriber clients
**Figure 3.9.** Workflow of the publishers. Notice the transmit data event from the broker diagram.

**Figure 3.9.** Workflow of the subscribers.
The corresponding diagram of Figure 3.2 for the subscriber is described in Figure 3.9 and describes the workflow of the subscriber. The subscriber is based on the same superclass as the broker, thus they have to implement the same structure. They do however trigger different events and the function for handling the incoming events differs. Additionally the subscriber is tightly dependent on the user input, which is not needed for the broker. The user input that is needed include connection to the broker and selection of the parameters.

Figure 3.9 describes the workflow of the publishers, which uses the same superclass as both broker and subscriber and thus utilizes the same functions for creating events, but with different set of commands. The publishers depend on user input to connect to the broker and to generate samples to transmit, but is more automated compared to the subscriber.

3.3 Interface to the network library

Described below are the utilized functionalities from the ZMQ library and how they complement each other into the complete solution.

3.3.1 Initial setup

Generally ZMQ uses a context that contains all the sockets, each thread should have one context to be initialized in the beginning of execution and to be terminated at the end (Akgul, 2013). This is solved using a common superclass that all the libraries initialize and use for the remainder of execution. The superclass handles all the sockets created and thus enabling to use the same context for each socket. Only the superclass uses the ZMQ sockets, which makes future changes of network library easier, if necessary.

The sockets of ZMQ need an IP address of the connection. A text file with a defaulted IP address was used to reduce the complexity of the system. The text file must be updated to a computer that runs an external program, which acts as a router that directs all transmissions through the network. This router must start before the rest of the system. This enables the subscribers’ to use the default IP addresses and connect to the brokers and the publishers via the router.

3.3.2 Publish subscribe connections

Since the publish-subscribe connection pattern can be fitted nicely into the design pattern discussed in section 1.2.1, it is used for the transmission of the samples. Each node generates a random, unique identity. By utilizing the identification system built into the publish-subscribe pattern of ZMQ the messages automatically get delivered to the right node.

Each node listens for messages that contain certain headers. The following headers are used in the network: ALL, BROKERS, SUBSCRIBERS, PUBLISHERS as well as the random unique identity of that specific node. Every node in the network listens to the headers ALL, the type of node header as well as the unique identifier. Messages that are
sent to a specific node are addressed with the identity of the receiver. The ZMQ library ensures that there are no other recipients of the message.

Each connected node creates multiple sockets for different types of messages. The API does not de-serialize and handle messages on all sockets, on some sockets it passes the messages to the main program. One example of the sockets that have such a pass through policy is the socket with the data stream.

### 3.3.3 Messaging

The API uses the router-dealer message system described in section 2.2.3 due to the added control of the bytes that are to be sent. ZMQ have functions for initializing the size and data of the message. The socket to be used for sending is passed as an argument to the function responsible for sending the message. The first step to sending messages through the network is to initialize it and copy data into it. After initialization the send function can be called and the message can be closed. At the receiving end, the message doesn’t have to be initialized since the size is unknown until the message has arrived. Thus the ZMQ receive function does that for you. There is a function that the API utilizes which polls the receiving sockets to see if messages have arrived. At such an event the receiver calls the ZMQ receive function and the message is stored into a ZMQ message. The API either decodes the message or passes it to the main program. To solve the decoding part, different sockets are used for different types of messages. Thereby the message type is known and decoding can be solved and the command that was included into the message can be read. Actions described in the workflows of section 3.1 can be performed in accordance to the command.

The drawback is that ZMQ only sends bytes and has no knowledge of the structure of the message. It is up to the API to decode the messages correctly and handle all the memory allocation. It is important to make sure that correct amount of data is sent, since it’s easy to send too much data. Improvements can be made if the amount of data can be reduced.

### 3.4 Solution architecture

To describe the structural design, the relevant classes as well as their relations are described in UML diagrams, which show their respective variables and functions. The layout of this section starts with the network messages, and then describes the system for handling said messages and finally the libraries that implement the system are described. The point of the solution is to describe how the requirements defined in section 1.2.2 have been implemented and how they can be utilized.
In Figure 3.10 the abstract class "Networking" contains the integers *command* and *extra* and the strings *source* and *destination*. Every message that passes through the network contains this data telling the system how to handle the sent data. Some messages are handled internally in the API and others pass through to the main program depending on the command. The source and destination is to address each program within the network following the system described in section 3.3.2. Each message type that is to be sent, needs to overload the *serialize*, *de-serialize* and the *clear* functions and optionally the *dump* function. The storage in the message is an allocated buffer whereby the content can be anything the user likes; meaning that the serialization leaves room for the optimization discussed in section 2.3. The serialized data is copied into the buffer of the message. The data in the message should contain both the integers *command* and *extra* from the super-class as well as any additional data from the sub-classes. In the implementation of “Bytes” and “Message” the storage of the message is casted as a char-array and all the relevant data is packed into that array.

The *clear* function makes sure that no unused data remains in the system. This function is called each time the message data is read to make sure that the messages are closed by the ZMQ library function. The dump function is a debug function but can also be used for measuring the system, since it can dump the data sizes as well as the time of arrival.
Figure 3.11. UML diagram of the core of the system consists of the Broker, Subscriber and Publisher classes that are sub-classes of the abstract class NetworkManager. The Arguments class consists of parameters and mutex locks that are used by the main program to send information to and from the created thread connected to the network.
Figure 3.11 displays the UML diagram for the wider system containing the classes that communicate over the network. There is an abstract class `NetworkManager` that implements the networking functionality for connecting, sending and receiving as well as a virtual function for handling the communications from different sockets. This makes a more flexible system where modification of network communication is easier. The abstract class includes two lists for each of the network message types, where the requests and replies can be queued and sent.

The class `Arguments` is the link between the thread running the library and the main program. This class contains two extra queues for the messages that need to be passed to the main program together with two mutex locks for the shared variables. Additionally the `Arguments` include the name of the broker as well as the identification code and pointers to the buffer that stores the samples, both for publication and for subscription. Other variables are the event conditions that are used by the pthreads library for waiting for the events to occur. The library notices the incoming messages and set the event of the `Arguments` and issue the main program to activate.

The event system has its own structure that is described in the figure. The main program will call the constructor of the `Arguments` class to initialize it. When calling the constructor of the `Broker`, `Subscriber` or `Publisher` classes, the `Argument` object is passed as a parameter to be stored in the class `NetworkManager`. When the `WaitForEvent` function from Figure 3.1 is called by the main program, the condition functionality of pthreads is used. This makes the main program sleep until the condition is set. The thread running ZMQ functionality will eventually receive a message and set the condition parameter using the `eventConditionSet` of the `eventCondition` class. This causes pthread to wake up the main program to read the status of the event and handle the event that occurred.

The `Broker` class contains the overwritten functions `handleEvent` and `handleByteStream`. It also contains functions that serve the subscribers and publishers for adding or removing parameters, `addParameters` and `removeParameters`. The helping class `ConnectionHandler` contains two lists where the parameters of interest are stored. One list containing parameters provided from the publishers and one list containing parameters of interest from the subscribers. Those lists also contain the identification strings, enabling the broker to address them. In the case of publishers, the names of the parameters are stored, giving the subscribers a possibility to get the extended parameter list. Extended meaning the regular CVT of the broker with the addition of the parameters from the publishers. In the future this could be adapted to handle the entire CVT list, thus enable the API system to have even more extensive control over the system and consequently be easier to implement in new software.

The libraries that are implemented by the subscriber create a new instance of the `Subscriber` class that also is a sub-class of the `NetworkManager`. It needs to handle all the commands from the main program, such as handling the parameters. Furthermore it needs to handle the selection of brokers, hence when new brokers connect, the main program needs to add it to the list of available brokers. The user can choose to connect and get the available CVT from different brokers. The library will get the brokers new identities and notify the main program.
The publisher library function is designed in a similar manner as the subscriber, but it calls the Publisher class. The main program of the publisher must control what broker to connect to and what parameters to transmit.

Figure 3.11 displays the functions and dependencies of the different classes of the system. Compared to Figure 3.1 this defines the functions and dependencies below the Broker-, Subscriber- and Publisher Implementation level. The declarations of the functions of the different implementations are very similar. All functions have a prefix that defines in what interface the function is described. The reason is to separate them from each other to make it possible for programs to include multiple libraries. All libraries use a common file where all commands of the API are defined. Those commands are needed when calling the broker-, sub- and pubSendCommand functions defined in the interface level of Figure 3.1. Arguments to these functions can, for example be lists of indexes or strings of the names of the parameters. The broker-, sub- and pubGetPtrStatus functions of the interface level, is needed when the main program must read or write data to the buffer that handles the samples. The buffer is treated as a shared resource and thus needs extra care to prevent reader-writer problems. This functionality is handled by the pthread lock mechanism defined in the Argument class. Other interesting functions include the init-, and waitForEvent- functions that have already been discussed in the thesis. These functions are also mitigated through to the NetworkManager.
Chapter 4

Integration with existing system

This chapter describes how the libraries have been integrated with the different aspects of the main program. The descriptions include how the main program utilizes the functionality as well as the problems encountered.

4.1 Implementation details

To fully utilize the unified API each subscriber, publisher and broker needs to include either of three libraries created depending on the desired functionality. This is shown in the top of Figure 3.1 where the main program interacts with the interface for each library. For each library the main program needs to be in a loop that calls the wait-for-event function of the library. That function has a timer that triggers an event every twenty milliseconds if no other events occur. The timer is controlled by a constant within the system and can be modified when compiling. When the timer is triggered, the publishers and brokers check if

![Figure 4.1](image-url)
they have new samples to send. The main program checks for events such as user input. For the API to function, the main program needs to follow the diagram displayed in Figure 4.1. The library needs access to the allocated memory used for sending information between the main program and the network API. The events the main program needs to handle include the CVT, parameters that are to be packed into the sample packets and then the messages that require sending new commands by the user input. First the main program needs to connect to the broker before any communication can commence since the library needs to know what identification code to address. The library will automatically find all new brokers and get the identities but the user of both the subscriber and the publisher needs to manually choose what broker to connect to.

Programs can include both subscriber and publisher if needed and enable both functionalities. For utilizing both libraries they need to call the wait-for-event function on both the publisher and the subscriber, it would require two arrays for storing the samples, one for the incoming samples and one for the outgoing. The other arrays that are required by the wait-for-event functions can be reused since the libraries write to them from the main thread running the program and not issuing any readers-writers problems of multi-threading.

4.1.1 Broker

The startup order of the system is important. For the system to work properly the broker needs to start and load the CVT before any other subscribers or publishers can connect to the broker. If this is not the case, some parameters could be overloaded during the loading process.

The responsibilities of the broker include applying new indexes to the parameters provided by the publishers. This is done from the broker library system and requires the library to have access to the number of parameters in the CVT. This is the reason why the CVT have to be loaded before the other nodes can connect to the broker. When a new publisher connects and adds parameters, the number of CVT parameters is increased within the library and connecting subscribers will automatically get the extended version of the CVT. When new publishers connect and start transmitting new parameters, all clients will be notified of an extension to the CVT.

The main program needs to allocate memory buffers for samples and incoming messages as described in Figure 4.1. The broker needs functions that create a message containing the CVT in reply to requests from the subscribers and packs the samples and sends them. This is done from the handleEvent and sendByteStream functions described in Figure 3.11.

4.1.2 Publisher and subscriber

The publisher and subscriber are of the same base class as the broker. The main programs that utilize them are assumed to operate in a similar manner compared to that of the broker. The libraries are simple since the data flows in one direction and they have simple responsibilities. The main program of the publisher has a similar function to that of the
broker that can create new transmissions and transmit them to the network. The main program of the subscriber has the reversed function that unpacks the samples and passes them to the main program. The libraries of both the publisher and the subscriber handle messages for adding and remember parameters.

4.2 Problems encountered

Interesting problems and observations were discovered during the implementation phase and they are described in the section below. The problems are described with a cause, solution and a discussion about the problem.

4.2.1 Message serializer

As discussed in section 3.1.3, the MessagePack message serializer is probably better to use due to the fact that it compresses messages much tighter and saves a lot of bytes. This made MessagePack the initial choice for message serializer and was successfully tested for packing and unpacking. MessagePack proved efficient in packing and unpacking. It was not complicated to implement the serializer into the solution. Problems were detected when the packed data was sent through ZMQ. MessagePack added a few bits to the packed data that was used during unpacking to tell the different types apart. Those bits were lost in the transmission and the subscribers could not unpack the messages. This seems to be some issue with the network library but insufficient information is available to make a more thorough investigation. It is unlikely to be a fault at MessagePack since it is designed to send serialized information to different destinations. The tests remained unsuccessful in deserialization and the more simple solution of using a large array for packing data was used. In the array the data just needs to be sorted in a specific manor and can then be transmitted to the destination.

The current solution did not implement any serializer and still proved efficient enough. As seen in section 5.3.1, the new API was also efficient. When the transmission frequency needs to be raised and the number of publishers increases the message serialization could be an important upgrade.

4.2.2 Network system

The telemetry system, now consisting of the broker and a subscriber is the major part of the existing solution by Saab. The main problem of creating the broker, besides integrating it to the telemetry server, was to get the network system to work properly. For this to work there had to be a decision taken how to access the IP addresses of all connected nodes within the network. The ZMQ did not need the IP address when connecting to different nodes for communication but it needed it for creating the sockets. The solution is to have a router program running in the background. The router had a fixed IP address that could be read from a file and each node would connect to the router to relay transmissions. This is not an excellent solution since it creates an overhead in both latency and complexity. When
another implementation is wanted it is only needed to update the socket creation function in the base class to update all implementations to get rid of the router program. The reason it was created was that it gave the solution more flexibility since no one needed to know the IP addresses of the broker and could simply find all brokers connected to the router. Another alternative could have been to have a separate identification system, using the UDP broadcast system to identify different nodes. That would not have been ideal either, since it would have broken the ZMQ methodology and added complexity to the system. The solution with the router is the one that is tested and delivered. The requirements on latency are still met, making the solution sufficient, but leaves room for improvement.

4.2.3 Telemetry system

The telemetry system software was created a long time ago and have been redesigned and upgraded by many programmers along the way. The resulting code is filled with nested if-statements, code that is copied to multiple places instead of placed in a function, etc. The code is messy and complex. The process of understanding it and cleaning it up in order to implement the API is time consuming and demanding. Many subsequent problems have risen from this root cause.

One problem is that the CVT implementation is tightly integrated in multiple places in the telemetry system. Multiple functions manipulate the CVT in different manners and that is done in places outside the reach of the API. This makes it difficult for the API to handle the CVT directly. There is a need to have the main program of the telemetry software handle transmissions that arrive from the API in its own way in order to insert into the CVT. It would have been cleaner and more efficient if the API were able to insert to the CVT directly. It would have reduced the complexity of the main program and saved some overhead. This also meant that it was more difficult to make changes to the telemetry system as well as the API. The solution of passing the transmission also meant that the API needed shared memory with the main program, which can be problematic when using multiple threads.

A problem with the telemetry subscriber was that it was not always ready to receive the transmissions. This meant that the API had to wait for the main program to signal it was ready before actually reading the transmission from the network library. Conducted tests showed that no packets were lost using this solution. Tests were also successful in using multiple publishers and subscribers, hence this solution was chosen for the final version. Another possible solution that was implemented and tested was to have a small message queue instead. However this proved inefficient due to memory being moved too much. ZMQ is, as the name suggest built to be used without a queue.

4.2.4 Matlab client

Another client to implement in the thesis work was the one running matlab. The previous solution was implemented as a java program that could receive data from the telemetry server.
Another command had to be added to the API, making the java program able to send a list of names of the parameters of interest. When the command arrived, the server searched for each parameter in the CVT and sent a reply containing the matching index of each parameter. Some parameter might not be available to the loaded CVT and then an error code was set as index. The java program needs that list to know what parameters to expect and what their indexes would be when receiving the parameters.

The parameters available to the broker were then used to set up a publisher for the matlab client. The data that arrived at the java program was retransmitted as new publications for others to subscribe to.

The problems with the matlab client were few, thanks to the architecture of the libraries that were created and all required functionality were already implemented except for the extra command for the telemetry server.

One problem that had to be solved was how to get java making the function calls necessary for utilizing the library files. Using the java native interface (JNI) allows java to transform the data structures into compatible ones used by the API (Liang, 1999). Java native interface is well documented and is not much work to implement. This made the transition from the previous client implementation into the new subscriber and publisher implementation effortless. Both the publisher and the subscriber part of the matlab client were working with the telemetry server within reasonable time.
Chapter 5

System evaluation

This chapter includes discussions about the different decisions that enabled verification of the design. Also described are the tests that were made in order to verify the performance of the system. The device under test is the system defined by the libraries. The requirements on performance defined in section 1.2.2 are also verified. The tests that were performed are defined in table 1.1.

5.1 Test methodology

During the tests, the telemetry software was used with different setups in terms of number of subscribers and brokers together with a dummy publisher. The dummy publisher used a random generator to transmit hardcoded parameters to the network. To make the measurements easier, the packets were of the same size each transmission. When the telemetry subscribers requested data from the publishers, the network data was increased by the size of the data, times the number of subscribers. The publisher implemented in the matlab client produced packets of various sizes and required a setup in order to work. By using the dummy publisher when testing the latency, logging was easier and more precise.

More extensive testing included several subscribers and publishers connected to the network. This aimed to test the limits on the system in term of latency was as well as the number of connections available to the broker.
Figure 5.1 Communication latency notation, $T_i$ represents the average time it takes for data to transmit from one box to the next. The variables $T_1$ and $T_3$ measure the time for data flow that uses the API. $T_2$ represents the time it takes for the broker to process the incoming data and $T_0$ represent the processing time of the publishers. The transmission time from the aircraft is not part of the API and is excluded from the graph.

There is no need for the publishers to send any data through the brokers, but there is a need for them to connect and communicated. The arrow labeled control signals in Figure 5.1 indicates this communication. The latency of the system is defined in Figure 5.1, either as $T_{\text{publisher}} = T_0 + T_1$ where the data flow comes from a publisher or $T_{\text{broker}} = T_2 + T_3$ where the data flow comes from the aircraft through the server. For all applications, it is necessary that the latency is as low as possible. Parameters sent from all sources should have an associated timestamp at the creation of the samples. The latency of both $T_{\text{publisher}}$ and $T_{\text{broker}}$ must be holding a validity interval of 200 milliseconds for the data to be relevant. This sets requirements on the broker and publisher to minimize $T_0$ and $T_2$ as much as possible. This also includes a requirement for an efficient use of the network to avoid overflowing it with messages, ensuring that $T_1$ and $T_3$ do not grow too large.

All computers in the network have different internal clocks, they are not easily synchronized, and it is hard to measure latency and throughput. It is not sufficient for computer A to send a message containing the current time to computer B and simply check the difference in time of computer B. The clocks could differ a lot and the clock of computer A could be ahead of time making the latency negative. If the clocks were synchronized, computer B could get the latency by checking its own time and subtract the time in the message. If computer B’s time if behind computer A, it would seem like the message would have been sent from the future. That is not the case and this problem makes it complicated to measure latency. For the most accurate measure of latency, the time that is relevant is the time the messages was travelling in the network without any additional time for message processing. This problem must be addressed during the performance analysis. To calculate the latency, the subscriber could compare the timestamp to time at the arrival of the data. This assumes synchronization between all nodes in the network, making the comparison of timestamp incorrect. Another solution is to make the subscribers send a specific message with the timestamp to the broker and make it send the same message back again. When it arrives back at the subscriber it can check the current time and compare it to the initial time in the message.
When testing the system, a small local network consisting of one broker and one subscriber were used. This enabled control over the amount of data that were sent and made it possible to test different parts of the system individually. By connecting several instances of the dummy publisher, the amount of traffic in the network could be increased. The subscriber was set to automatically subscribe to the new traffic and start measure the latency with the added traffic. This required a slight modification to the normal behavior of the subscriber and thus the tests were performed in the local network.

The second part of the testing was made in the large control room at Saab where multiple clients and brokers were used. For this test the telemetry broker was set to transmit saved data from the aircraft. Twenty telemetry subscribers were connected to the system and subscribed to all available data of the broker. Since the subscribers needed to run as a full version without any modification, the measurement tool was disabled. However, the built in system monitor functionality of the operating system on the broker showed minimal activity on both the processor and the network.

### 5.2 Scalability requirement

The unification requirement on the system is considered met thanks to loosely coupled design with the publish-subscribe pattern. The classes are implemented in such manner that they are open for modification and they share base classes where applicable. The classes have few dependencies and to upgrade certain parts of the system would in most cases be to update the base class. This makes it easy to adapt the API and therefore comply with the unification requirement.

Future clients that need to access the network would simply have to initialize the API, create the memory buffers and have a loop that called the `waitForEvent` function in the API. The loop function will have to check the return status of the `waitForEvent` function and handle the event appropriately.

In limitation section 1.3.3 the requirement for unification was deemed met if three clients could be adapted for using the API. The thesis work has successfully implemented the API into the telemetry client as a subscriber and the telemetry server as a broker. The client running matlab has successfully implemented both the subscriber part and the publisher parts. This makes the requirement for unification well met since one broker, two subscribers and one publisher has been implemented and tested in the system.

### 5.3 Latency requirement

The most relevant design choices regarding latency are the network library, ZMQ. It is fast and reliable, making it a good choice. The other relevant decision was that the publisher could send the transmissions directly to the subscribers without letting the broker handle the data. This made the system a bit more complex since it added responsibility to the publishers to keep track of the subscribers, but the latency is favored by that choice.
The limitations on latency are considered satisfied if the latency is below 200 milliseconds. The latency on the system can be defined as two different measures. The first defines the time it takes for the publisher to process and create a transmission, \( T_{\text{publisher}} \). The second definition is the time it takes for the broker to process and transmit data from the aircraft, \( T_{\text{broker}} \).

### 5.3.1 Verification and results

The functional test in the control room at Saab was successful. The system could run with one telemetry broker that was transmitting all available parameters. The parameter values were received among multiple subscribers. The system had over 20 subscribers connected, which was the limitation set in section 1.3.2. The only measurement that was done in this test was to monitor the system on the broker computer. The reason for this was that the environment in the control room required the full version of the system without the modification of the measurement functionality, in order to avoid the probe effect. The monitoring on the broker got minimal activity on both the network and the processor. This means that the API is efficient in terms of performance of the whole system.

![Packet creation time](image)

**Figure 5.2** The time it takes for the broker to create packets of different sizes to transmit. The time is measured in milliseconds.

In Figure 5.2 the time it takes for both broker and publisher to create a transmission packet is displayed, this is the big part of both \( T_0 \) and \( T_2 \). The frequency of the transmission
keeps the messages small, usually around 2 kB, so testing bigger packets than 85 kB isn’t necessary. The most accurate measurement tool of the runtime environment of the broker can only measure milliseconds. This means that the chart is only a rough estimation of the transmission time. What can be concluded from the chart is that the messaging time can be neglected as the times are rounded up by the operating system, meaning 1.1 [ms] is rounded to 2 [ms]. Furthermore there is no difference in the output of the charts for big message sizes. This is the first step towards confirming the latency requirement.

![Graph](image)

**Figure 5.3** The time, in milliseconds, it takes for the subscriber to issue and receive a request for parameters from the broker.

In Figure 5.3 the dummy subscriber is set to repeatedly issue a request for parameter from the broker. That request is sent to the main program of the broker, meaning it will take the maximum amount of time to process it before the reply can be sent back. In the graph it peaks at 33 milliseconds, which can be explained the broker got creating a new transmission at the time it the request. The request was delayed since the broker had to finalize the transmission. The remaining requests take around 23 milliseconds, which can be considered good. In that time, both the broker and the subscriber have sent, received and handled the message.

Figure 5.4 and Figure 5.5 shows the usage of the measurement function. The subscriber was modified to start a new measurement each time it subscribed to a broker or a publisher. It sent the measurement command to the broker 20 times and saved the results to a file. As a reply was received, a new request was immediately sent. When a new publisher was discovered the subscriber would subscribe to all new available parameters and start a new measurement. All new subscribers would start the measurement by themselves since the subscribers do not know about each other. Using this functionality, two test cases were developed.
The first test consists of using one broker, one publisher and manually starting twenty subscribers that would cause an increase in network traffic that affects the latency. Each new subscriber saves the messaging times, affected by increasing traffic flow in the network to a file. This test was conducted four times. For each number of subscribers, the average of the saved message times was calculated. By taking the average of each average messaging time, the graph in Figure 5.4 was created. While performing the measurements, the publisher is transmitting data that the subscribers need to handle, adding to the roughness of the measurement tool. By taking the average out of several steps, a smooth graph was created that is almost linear. This is even better compared to the expected results of table 1.1. This might be due to some optimization in ZMQ, but is regardless good for the system. For one publisher there is very low latency and there is an increase in latency for increasing number of publishers. It was expected that the latency increased by a third when adding another publisher. Due to the inaccuracy of the timestamp system it is difficult to get the exact increase. Based on the small difference of the graph it is fair to say that the system is better than the estimations done in the table. The average line is well aligned with the minimum and maximum values.

**Figure 5.4** The average latency for sending and receiving 20 ping messages in a network with one broker, one publisher and differentiating number of subscribers. The graph shows the average out of four tests. The minimum and maximum lines show the extreme values of all 20 times four messages.
Figure 5.5 The average latency for sending and receiving 20 ping messages in a network with one broker, one subscriber and differentiating number of publishers. The graph shows the average out of four tests. The minimum and maximum lines show the extreme values of all 20 times four messages.

The second test consists of one broker, one subscriber and manually starting twenty publishers and measures $T_{\text{publisher}}$. The subscriber will start new measurements for each new publisher. This test was conducted four times and for each number of publishers, the average message time was calculated. The graph of Figure 5.5 shows the average of the average message times from the second test. There is an anomaly around 11 publishers, this is due to a fluctuation in one of the four tests. This is shown in the maximum value line as well. This might be due to the roughness of the measuring system. By taking the average from all tests, the fluctuation can be disregarded since the other tests show good results. Using many publishers, each generating the same amount of traffic will increase the latency more compared to the first test. This is due to the publish-subscribe system of ZMQ that handles the distributions to multiple subscribers in an efficient manner. The average latency of using 20 publishers was 10.4 milliseconds per ping and can be compared to the 6.9 milliseconds of the 20 subscribers test, this result is reasonable and expected. Over all, the average line is well aligned with the maximum and minimum value lines.

5.3.2 Requirement verification

In addition to the tests described in section 5.3.1, all tests described in table 1.1 have been successfully verified. A test with two brokers, both every parameter available, to 30 subscribers gave no loss of data or any latency issues. No data was however recorded of these tests.

The required requirements 1-5 in section 1.2.2 have successfully been met:

- Requirement 1 are verified by the client running matlab
• The functionality from requirement 2 has been used to create the graphs of Figure 5.4 and Figure 5.5.
• During the implementation of requirement 3, it was concluded that it was more efficient to send data from the publishers to the subscribers directly. Using the publish-subscribe functionality form ZMQ improved the system greatly. If the publishers were to communicate to the broker for re-transmission, as suggested by the requirement, the overhead had been greater compared to the actual implementation.
• Requirement 4 is implemented with the *Found broker* event of Figure 3.9.
• Requirements 5-6 were not deemed mandatory and have been left for the future.

### 5.4 Test conclusion

The tests performed in smaller scale shows that the system is well within the required latency. Together with all the results from the graphs as well as the full-scale test in the control room where the system was confirmed to be working, the latency requirements are considered met.
Chapter 6

Conclusions

This chapter summarizes the work of the thesis and suggests some future improvement that can be done to further add to the implementation.

6.1 Summary of the work

The result of this thesis work is an API that can be implemented by different clients that need to access the network. A measuring system, DAS, on the Swedish aircraft JAS 39 Gripen continuously transmits sensor data during test flights. The sensor data is transmitted using radio and is received by a server at Saab. The server is running the broker software developed by the thesis work. It utilizes the API to distribute all parameters that are sent to the subscribers that need to monitor the changes transmitted by the fighter.

The existing solution by Saab utilized many protocols that created a big and complex system that was difficult to maintain. The advantage of the API is the unification of protocols. The broker manages each connected node in a unified manner. Nodes can connect as publishers and contribute with additional parameters that the subscribers can receive.

The resulting API has been implemented into two nodes where one works as both subscriber and publisher. The API is considered usable by all future nodes, with some slight modification. The API is proven to be fast enough for the requirements set by Saab.

The interesting technologies that were utilized include ZMQ as networking library and pthreads as threading system. Architecturally the publish-subscribe design pattern inside ZMQ was utilized. The results of the created API show that these choices were well made and work well together. Using pthreads together with ZMQ as network layer gave flexibility and structure to the system. Using the publish-subscribe design pattern gave logic to the classes and the dependencies within the system.

6.2 Future work

This thesis work laid ground for the start of the API that is to be used for a new version of the distribution system at Saab. The API has been implemented in two of the clients used in the previous solution and tested in full scale in the control room at Saab. The current status
of the system leaves room for further implementations and optimizations. Some of the most important features that are to be implemented are listed below.

- Not all requirements from section 1.2.2 are implemented, requirements 6-7 are left for the future. Those requirements include the ability to request data from the hardcoded stream of the existing system. The other requirement is about data optimization making it possible to send data as their true data types.

- Make the broker send the CVT as a list of parameters instead of a link to a file, enabling connections to multiple brokers in a better way. While it is possible to connect to multiple brokers in the API it is required that they use the same CVT since the subscribers cannot for now, handle multiple CVTs. By turning the CVT into a list handled by the brokers it will be possible for the subscribers to distinguish between different CVTs of different brokers. This also requires modification of the main program of the telemetry broker and subscriber.

- Make every node connect to each other more directly and remove the need for the router program that now handle all the traffic. This requires some extra communication in the beginning or static IP addresses or some prior knowledge that was not given to the API.

- Better message serializer, minimizing the amount of data that has to be transmitted through the network.

- By using the threads of ZMQ it could be possible to get a cleaner solution compared to the pthreads. It would be interesting to test how the ZMQ threads would affect both the design and the performance.
References


Sebastopol.


Upphovsrätt
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.
Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.
Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart.
För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright
The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.
The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.
According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.
For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Jonathan Karlsson 2015