Reliable Object Management

Johnny Berentsen

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Reliable Object Management

Johnny Berentsen

Reliable Object Management

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

Printed: Reprosentralen, University of Oslo

Abstract

In this paper different aspects of reliable object management within a distributed system is discussed. The overall goal is to have data objects in a consistent state within the system with minimal network overhead and with as low latency as possible. Additionally the solutions should be easy to use for developers and minimize system administration needed to set up.
The problem description can be divided into three levels of abstraction: reliable multicast, object management and object contents. Each level may be solved independently or together by combining different technologies.
There are various problems and solutions depending on the use case, so an already deployed and widely used Air Traffic Control (ATC) system was used as a basis of the analysis.
By reading RFC's and other publicly available documents and gathering information on technical forums on the Internet a broader view on this topic was found. Wherever possible, real implementations were used as examples and as reference.
A lot of different solutions exists; from socket libraries that help build your own management implementation, through broker type solutions in the form of message queues to large enterprise messaging systems meant for large heterogeneous and physically distributed systems.
For the problems that is to be solved for the ATC software analyzed here, a suitable solution was found with OpenPGM, ZeroMQ and Google Protocol Buffers.

Contents

List of Figures

List of Tables

Listings

Part I
Introduction

1 Introduction

In distributed systems there will eventually arise a need to keep bits of information synchronized across all or a subset of all processes running in the system. This can be done in a number of different ways depending on the architecture and characteristics of the applications in the distributed system.

This paper shall discuss reliable object management, primarily based on reliable multicast - RMC/OM for short. In this context RMC/OM consists of three parts:

- Reliable multicast
- Object distribution
- Object contents

Objects refers to a collection of data that are serialized into objects and distributed to any number of clients reliably and in a timely manner. Reliability implies that messages shall not be lost and all clients shall receive and operate on the same data. Consistency implies that object contents shall not be inadvertently changed before reaching destination. Timely updates implies that changed objects shall be distributed as soon as the change is done and make sure clients never work on stale data.

To be able to concentrate on a single set of challenges and solutions a tower air traffic control system, Tower ATC, is studied. This is a system that is deployed on many airports around the world, including many of the biggest. The Tower software, Tower SW is currently using a proprietary reliable object management system internally called Object Manager, or Tower OM for short. The question is, can it be easily replaced with a more flexible and modern system, and what options are there?

1.1 Motivation

In a distributed system with high demands on availability, keeping data consistent and synchronized on all nodes is essential. Finding a reasonably good replacement for the Tower OM should not be addressed lightly; it will be a basis for a vital part of the communication and making the correct choice will affect development and system capability for many years.
On the way to finding this solution it is important to understand what the fields of RMC and OM looks like. How they have developed over time, what solutions exists, their strengths and weaknesses and how they fit in the case of Tower OM.

1.2 Problem Statement

The use-case for this study is the RMC/OM solution found in the Tower ATC software. This is a fully functional and quite robust system that is in active use

daily. However, as the OM implementation was done at the end of the nineties it is showing its age and have some issues that could be improved upon. This thesis will try to address that.

1.3 Goal

The results of this thesis are threefold: analysis of the different RMC/OM solutions, and as a consequence, find possible replacements and finally implement a proof of concept with the most promising solution found. This solution will be integrated with an Tower ATC system to send and receive actual data.

1.4 Approach

A first initial step was to analyze the existing RMC/OM system more in depth to get a clear understanding of the actual problem domain.
Next, analysis of available RMC/OM solutions was done by reading and gathering information through the Internet. There is an abundance of written material on this topic, ranging from small Q/A sections on the Internet to RFC's and real implementations. The available solutions also range from small libraries that do only one thing and need to be chained together with other small libraries, to fully fledged enterprise server systems. By looking at all this material a good understanding of the topic was found.

From the analysis, possible replacements for the Tower OM was found and discussed further.

Finally, an actual proof of concept implementation written in C++ was created based on one the found replacements. It will contain a reasonably realistic implementation of the core functionality necessary.

1.5 Work Done

The available documentation for the Tower OM solution is scarce, so most of the information gathered came from interviewing developers and long time users of the software. Some inspection of the source code was also done.
The Google search engine played a large role in gathering information. The RMC/OM domain stretches into a large field of different use/cases with different requirements and solutions, leading to a lot of relevant and not so relevant information. There were many RFC's, company white papers, discussion forums, source code for various implementations and other related material available. The ones found to be most relevant were analyzed further, and some were even tested with actual implementations. These test implementations were done using either Python or C++.
A proof of concept implementation was created using OpenPGM, ZeroMQ and Protocol Buffers. This implementation included a server and any number of clients. The server and clients understood two object types, and several aspects of RMC/OM were tested using these. Finally, to see if the software could easily replace a part of the current implementation, a specific object type was selected as a use case, and the proof of concept solution was integrated with the current Tower software for this object type.

1.6 Evaluation

The available solutions were evaluated on their fit for purpose as a replacement for Tower OM. Fit for purpose is a rather vague evaluation criteria but the goal is not to find the best possible based on capacity, network throughput or other hard metrics, but rather something that could easily be integrated in the Tower ATC software and solve the Tower OM issues. One of these solutions were chosen to be used for the proof of concept.

The proof of concept was then implemented to see if it would reduce or eliminate the shortcomings of the existing solution as well as support the same range of functionality.

By integrating the replacement into the existing system it can be shown that it is a valid replacement and should be an easier system to maintain than a complex proprietary solution with no original developer available.

1.7 Results

The following issues were found to be candidates for improvements:

Layer	Problem	Result
RMC	Network problems	Lots of resends may result in system halt.
	No one knows the implementation details	Difficult to make changes, do not fully understand the effect
OM	Unrelated data types are coupled	The flight plan sub system must handle unrelated data which adds complexity both in code and server configuration.
Object contents	Binary format	Not backwards compatible
	Limited number of types	Available data types are limited and not very flexible in use

Table 1: Problems With Current Solution

Two solutions with slightly different approaches emerged as very good replacements for the Tower OM.

The first one is DDS - Data distribution Service. It is an open standard put forward by OMG - Object Management Group, and enjoys several implementations both open source and proprietary. DDS is a full stack RMC/OM implementation which covers everything from reliable data transfer to supporting backwards compatible data objects. It has become very popular the last few years, especially within air traffic control systems.

The second is a layered approach based on OpenPGM for reliable multicast, ZeroMQ for object distribution and Protocol Buffers for object contents. It is decoupled in a way that it is possible to replace any of the layers with a different option if needs be.

For the proof of concept, the second solution was chosen. This was mostly due to the fact that only the proprietary implementations of DDS supported

the full range of functionality in the DDS specifications. In the proof of concept there were several tests done to see if the replacement could provide the same kind of basic functionality and reliability as the current implementation. The complexity of the original implementation did not allow for one to one evaluation, but the base statistics showed that it is feasible to assume it will also provide equal or better performance. It is nevertheless clear that it would greatly improve the situation of object definition, backwards compatibility and the possibility of understanding the code base and addressing problems with help from external sources.

1.8 Conclusion

From this we conclude that it is indeed possible to replace a RMC/OM stack that has done its time with an updated and more modern solution. There are numerous different options, and selecting the correct one is highly dependent on the type of application to make use of it.
For the use/case discussed in this paper, the selected software stack, OpenPGM, ZeroMQ and Google Protocol Buffers, suits the purpose very well and as the proof of concept shows, can replace the existing solution piece by piece rather than all at once.

1.9 Contributions

During the course of this thesis, a broad analysis of the different RMC/OM solutions was done. Even though the particular use case used for this thesis has a rather narrow focus, the overview still gives a good starting point for further analysis even for other areas of RMC/OM. This also includes areas of data distribution in general that is not necessarily dependent on reliable multicast.

Likewise, the proof of concept implementation is very much targeted towards the use-case example, but shows a good use of existing technology to solve a common task in modern distributed systems.

1.10 Limitations

As the domain of RMC/OM is so broad, certain limitations had to be imposed. Naturally, the fact that a solution should fit into an existing software easily reduced the scope. It was not clear from the beginning whether a completely different technology would be more appropriate than something resembling what was already in place. Because of this, other types of solutions had to be analyzed for the overview.

Additionally, there are a lot of details omitted that could help paint a clearer picture on this topic. The overview should thus be considered a starting point for further study rather than being complete.

The search results from the Google search engine is largely dependent on the search words and criteria used. Important or interesting solutions may have been missed by oversight, wrong keywords or suppression from the search engine.

Ideally, a full blown replacement for all existing functionality should have been implemented for real capacity and network related tests to be performed. In the end, there were two solutions with different approaches that stood out as good replacement candidates. Making implementations based on both would

have been a big improvement to the study, but due to time constraints, only one was actually implemented. For the same reason, only rudimentary network and capacity tests were performed.

The selected solution for the proof of concept opted for Protocol Buffers, protobuf, as the object contents layer. It would have been very easy to replace that with other types of structured data formats, and could have given more insight into pros and cons of the different formats. However, the study showed that protobuf did include the functionality sought after, and replacing it with something else would probably not change the end result much.

One important functionality of the current system was not implemented in the proof of concept solution; recording and playback. It is a vital part of ATC software and would have been included had time allowed it.

As the final solution would be running in a small local LAN in a Linux only environment, the focus lied on finding a replacement that fit within those criteria.

As always, regardless of how hard one tries, all decisions have a subjective component that may skew the end result one way or the other. This problem could have been reduced by having someone take the role as a devils advocate pointing out flaws with selected technologies, and pushing for other solutions.

1.11 Outline

There are six parts in this paper.
Part one gives an overview of the study and briefly sums up the results and conclusions.
Part two analyze the background and main problem areas of the current RMC/OM implementation.
Part three looks at available literature and discusses different solutions for each area of RMC/OM. It also looks at some source code examples for a better understanding of its use in the context of the Tower ATC software.
Part four describes actual implementations using the solutions found to be most suitable as a replacement for Tower OM.
Part five sums up the findings
Part six is an appendix holding all source code examples and other data referenced throughout the paper.

Part II
Background

2 Use Case Analysis

The Tower ATC software is a system used on very busy airports around the world. This requires that data are reliably distributed among all parts of the system. The system will be used as the basis for this discussion.
Airports come in many sizes so the amount of data to distribute can not be determined exactly up front. There will also be times where the system will meet bursts of data, so it needs to be able to handle an unknown amount of data at different rates at all times. A multicast solution is a good way to keep the total amount of data as low as possible. There is a reliable multicast solution, RMC, in place already, but there are issues that need to be addressed. On top of the RMC there is an object management solution that have separate issues. Finally, the objects have a proprietary format that lead to undesirable side effects.

2.1 System overview

Figure 1 on the following page gives an overview of a simple ATC system. The heart of the system is the two SDS'es (Surveillance Data Server). They will receive data from external sensors such as radars and flight plan generators and provide the system with current data. They operate in master/slave mode and only one will be active at any time.
The data will be received by Controller Working Positions (CWP) and/or Electronic Flight Strip Boards (EFSB) that are used by the air traffic controllers to make qualified decisions about flight movements.
In addition there is a technical configuration and monitoring system (TECAMS) that is used to configure the system and monitor the overall status. Finally, a recorder (RPS) will record all data in the system, and the playback machine will allow playback of any situation exactly how the operator witnessed it.

An integral part of Tower SW is handling flight plans. A flight plan is a document created prior to departure and gives information about arrival points, arrival times, type of aircraft and a number of other things related to the flight. Once entered into the system it is used by the controllers to plan the flight movement, from leaving the gate to entering the gate at the destination airport. Throughout the flight the data of the flight plan is updated several times, and will move between several jurisdictions. Internally the flight plan will contain data from an external system defining the base flight plan data with some additional system data added. The system flight plan will normally be updated from one place at a time, but may get updates from the originator due to delays etc, or by controllers working on different parts of the flight. In essence, reliable object management is essential...

2.2 Current Solution

The data flow in the Tower SW system can be divided in three:

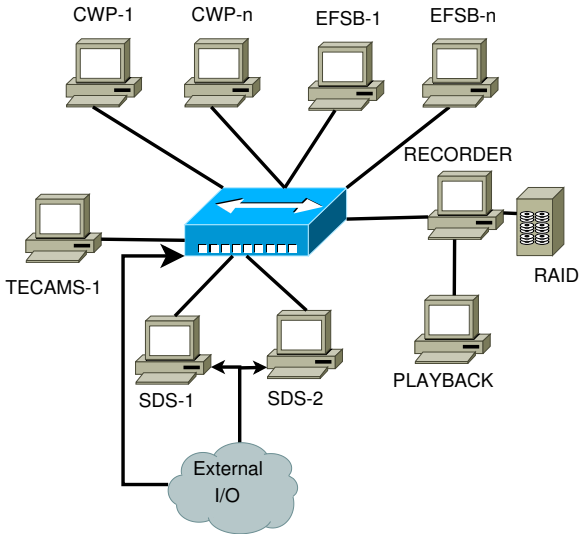

Figure 1: Tower System Overview

- RMC - An in house reliable multicast implementation
- OM/LWOM - (Light Weight) Object Manager. Objects make the basis of how data is delivered throughout the system and a separate layer manages these
- Object contents - There are a number of different objects that hold different types of information originating from different places and of interest to different entities in the system

2.2.1 RMC - Reliable Multicast

Implementation of the reliable multicast layer was started around 1997 and it took a few years to make it stable. The reliable part of RMC is crucial as the whole system will fail if there are problems in this layer, such as clients working on obsolete or stale objects.
There are certain limitations and requirements put on the RMC implementation:

- One to many only. Ie. only one sender/server allowed. All other participants are receivers. This also implies that only one can change the contents of the objects
- All receivers shall have the full database at all times
- Changing master requires a full database resend
- It shall support a standby server, however initiated and controlled by the application
- Efficiency shall be prioritized
- Alive messages to have all clients know the presence of the server

- API to be simple to use, and to remove the need for the different applications to know the underlying technology
- The communication between the RMC and the applications shall be implemented using callbacks
- Different types of objects shall be kept in separate domains. Applications may subscribe to only those domains that are of interest. All other objects shall be discarded.

The second point have some interesting effects that need particular attention. The first time a client starts it needs to fetch the full database from the server. If the whole system starts up at almost the same time, as often is the case, a lot of communication will be done, and it is a fair chance that the network will be saturated and result in missing packets and in the end overload of the system. With the first implementation of the RMC solution all data within the system used the same RMC/OM system and this increased the risk of packet loss and network saturation at startup. This problem was greatly reduced when it was decided to move the bulk of the data out of the fully fledged RMC/OM system, and into something lighter - described later as the LWOM.
In addition, some improvements to RMC was done to make it more robust to startup problems. One of these was to add a random timeout before a receiver asks for a resend. This will make sure resends will be spread out over time. As we will see later this is a normal procedure within RMC.
There have been reported problems when the network is not behaving the way it is supposed to. A rogue switch in the network may for instance lead to a lot of dropped/discarded packets and eventually the number of resend requests may bring the system to a halt or result in undefined behavior.
Another issue that occasionally is brought up is when the two redundant servers are separated and will start to run as two separate master servers. When the network connection is reestablished the status will be undefined.

Currently the implementation works well, with the following possible improvements:

- Further improve tolerance of network appliances that are not correctly configured or are misbehaving. This also includes faults in cabling, network interface cards etc.
- Map/document the current implementation. There are a number of states both in the server and in the clients, and no one has the full overview. The original developers have long left.
- Improve handling of separated networks, i.e. when two servers have been separated and later reconnected.

2.2.2 OM/LWOM

Object Management constitutes handling of the objects in the different domains between the participants. This includes keeping a full database of all objects in the domain and reliably update all recipients of any changes.
To remedy the problem of many resends at startup, the LWOM - Light Weight Object Manager - was introduced. LWOM still relies on RMC for the data

transfer, but instead of having the clients update their database with new information and ask for resends of missing/faulty packets, the full database is sent periodically.
As it stands now there are two parts using RMC and the full OM: fdps - flight data processing system and certain parts of the playback system. Given the way OM and LWOM are organized, and how the data flow is constructed the fdps is used for a lot more object types than just flight plan related.
Whenever a receiver shall change a value in the object database, it needs to send a message to the server with a request to change the value. The term used for this task is object transaction.
If the server acknowledges the new value it will update the database and transfer the updated object(s) to all the receivers using normal RMC/OM.
The biggest candidate for improvement in this area is this:

- Decouple the different data types in the OM domain from the fdps and make it easier to handle these independently.

2.2.3 Object Contents

An object is a specific class implementation of OM<type>. The base types are OMIntX (X=8, 16, 32), OMBool, OMString and a few others. Other OM type classes can be built up of any number of these base types. The most notable one is the OMFlightPlan class which include more than a hundred OM type attributes. All these attributes are searchable throughout the RMC/OM domain they belong to.
The objects transmitted over the network is a header followed by a binary representation of the object described above. This makes the objects tightly linked to the source code and as there are currently little validation of the incoming messages the processes are prone to crash when the messages originate from different source bases.
The Tower SW recorder will record the raw RMC messages together with some bookkeeping data. Playback only replays the same raw messages. This clearly leads to problems if the receiving applications has changed their data format, which very often happens. These are the most prominent issues with regards to object contents:

- No backwards compatibility
- Available data types are limited and not very flexible in use

2.3 Summary

Table following table shows a summarized list of problems found in the current system.

Layer	Problem	Result
RMC	Network problems No one knows the implementation details	Lots of resends may result in system halt. Difficult to make changes, do not fully understand the effect
OM	Unrelated data types are coupled	The flight plan sub system must handle unrelated data which adds complexity both in code and server configuration.
Object contents	Binary format Limited number of types	Not backwards compatible Available data types are limited and not very flexible in use

Table 2: Problems With Current Solution

The biggest problem is object contents and how to handle evolving datasets. It would be possible to replace just this layer with something more appropriate, but as it is tighly linked with the other layers, changes will need to be done throughout.
There are additional issues regarding playback that will not be discussed in detail in this paper.

Part III
Overview of Alternatives

The following sections will look at alternatives for RMC, Object Contents and Object Management. A general view of the topics will be given, and specific solutions will be analyzed in respect to the issues found with Tower OM.

3 RMC

Reliable Multicast differs from ordinary multicast in some specific ways. It will still have the advantages like one to many distribution with a single transmission and a higher data transmission rate than its typical counterpart, TCP/IP, but will have some extra overhead. There are various use cases and implementations of RMC and the following sections will give some details about those.

3.1 Background Information

The previous discussion listed three areas that need to be considered: RMC, Object Management and Object Contents.
According to different specifications listed in RFC's and documents from the standard organization IETF, Internet Engineering Task Force, a lot of work was done on RMC in the nineties and early 2000. Several specifications were released, where the most notable ones are RFC 3208 [22] which describes PGM - Pragmatic General Multicast and RFC 5401 [3] and RFC 5740 [4] which describe NORM - NACK-Oriented Reliable Multicast. In addition SRM - Scalable Reliable Multicast [7] has gained some attention.There are a number of other reliable multicast protocols as well with a varying degree of available implementations and proper documentation. IETF has a workgroup [13] specifically targeting reliable multicast. As with many of the other standards, the work is only slowly progressing.
Object management is the part that depends the most on the type of application(s) and environment. There are a lot of different solutions ranging from simple libraries that provides some functionality to large enterprise systems made for stock exchanges and the like. There are both open source and closed solutions in this realm. A popular open one is 0MQ or ZeroMQ [zeromq.com] as it is often called, which as a bonus has PGM as a possible transport protocol. AMQP [amqp.org] is another messaging standard with several implementations. AMQP is supported by some of the largest IT companies in the world, such as Red Hat, Cisco, VMWare and Microsoft. For systems with very large object management requirements, IBM Websphere might be a possible solution.
The most important part is in the end the object contents. The Tower ATC software uses a proprietary binary representation of the data which inherently creates some problems. Ideally it should be easy to define the objects, the object schema, and easy to create the corresponding source code with support for evolving schemas. XML has 'always' been there, but eventually other standard emerged, among those JSON and Google's Protocol Buffers.
Some protocols try to focus on large file transfers to multiple receivers. To name a few:

MFTP - Used for large files where the file is sent in several passes; a big chunk or possibly the whole file is sent by multicast in one go. Then the missing packets from all receivers are collected and sent in the next pass etc. till the whole file has been successfully transferred.

UFTP - A protocol designed to transfer files encrypted by handshaking with receivers before the transfer and exchange keys if applicable.
Likewise there are other protocols well suited for large files but using other techniques than multicast. One of the more popular examples is the bittorrent technology.

3.2 RMC Discussion

The key concept of RMC is to make the inherently unreliable multicast a reliable transportation mechanism. The challenges that is met will be equal no matter what kind of protocol one uses, but there are different ways to overcome these.
The following key concepts are used by the different RMC solutions described in the next section.

RMC Message flow The sender transmits the uniquely identified packets on the selected multicast address and port.
Depending on protocol the sender or receivers identify missing packets and request a resend of these. The packets will be buffered by the sender or some kind of proxy for a while in case of a resend.
Missing packets are resent until all receivers are up to date, or a timeout determine that the receiver is out of reach.

Identification Most, if not all, protocols use some way of unique identification on the packets. This could be as simple as a sequence number that is increased for each sent packet. These numbers are used to identify missing packets and to allow for resend request on those.

Resends For resends the following mechanism is typical:
The sender will hold a backlog of all messages sent, at least up to a predefined maximum. If a receiver sends a request for a missing message and the message is not present in the backlog, the whole database may be sent or in some way handled at the application level.

Handling missed packets The reliable part of RMC is making sure that missed packets eventually arrive at all receivers. There are two main routes:

ACK - Acknowledgement messages sent from all receivers to the sender. Also called positive ACK's. Sender is responsible for resends.
There are different ways of accomplishing this. The simplest is to send an acknowledgment for every received packet. This will lead to a lot of excessive traffic so to improve on this some protocols only send updates in predetermined intervals and then only the sequence number of the last received valid packet. Valid meaning all sequenced packets before it has been received.
The disadvantages of this model is:

- Sender need to keep a list of all receivers and also their state.
- ACK implosion - A term used to explain the effect of many receivers submitting ACK's for received messages. Many receivers lead to many ACK's lead to a lot of traffic.

NACK/NAK - Negative acknowledgment - Receivers send messages with packets not received. Receivers are responsible for resends.
There are several ways of doing this as well, but the main concept is that the receivers keep track of the sequence numbers of the arrived packets. In the occasion of gaps in the packets, a retransmission message is sent.
When there is no data to be sent, a low rate multicast message with the last sequence number is sent.
If many receivers have missed packets an implosion situation may occur also for NACK based protocols - request implosion in this case. There are several ways to improve on this:

- If a request for a missing packet has already been sent by any of the other receivers, a new will not be sent. All receives need to listen for missing packets.
- Adding a randomized timeout before a resend request is done will prevent a lot of requests at the same time
- Adding a timeout depending on the distance (number of network hops) to the sender will have the furthest away send first and thus all receivers on the path to the sender will be made aware of this and not send requests themselves
- If any local receivers have the requested packet they may send the packet directly
- DR - Designated Receivers - used as proxies for the sender. Messages will be received by the DR's and logged here. Any local or upstream receivers will use the nearest DR as their sender.
- Request is sent unicast directly to sender - will not send unnecessary traffic to the other receivers but may put some extra burden on the sender.

Packet loss Receivers may request a resend of packets due to two situations: Packet not received or packet not received correctly.
If there are packet loss the ideal situation is that all receviers that missed a packet missed the same one - correlated misses. In an ideal implementation only one resend request would be needed, and only one repair message.
Normally though, there will be different packets lost to different receivers - uncorreleated misses - so a possible efficient implementation would have the sender collect all missing packets and combine them into one before transmission.

FEC To help minimize the number of resend requests due to incomplete/misformed packets a methodology called FEC - Forward Error Correction - can be applied. FEC is a term used on a variety of different methods and IETF has authored a number of RFC's on this topic. See Appendix A for a detailed list.

General concerns with reliable multicast as opposed to ordinary multicast

- Increased latency over unreliable UDP, from marginal, measured in microseconds to significant for recovered packets.
- Back-link from subscriber to publisher.
- High packet loss rates can overwhelm senders re-transmitting repair data.

The following sections describe some of the RMC implementations/standards available.

3.2.1 SRM - Scalable Reliable Multicast

Scalable Reliable Multicast is not a specific protocol definition but a collection or a discussion of possible solutions. Two papers, [7] and [21] explains some of the key concepts. The goal is slightly broader than some of the other protocols, in that it is oriented towards ALF - Application Level Framing. This indicates that SRM understands that different applications have different requirements and creates building blocks that can be used and altered by the application.
Some of the methods used by SRM for reliable multicast are of a general nature and described here:
When a packet is lost, a negative acknowledgment (NACK) is sent to all members using the same method of transportation as the original data. As all members see these NACK's, everyone can participate in the repair of the traffic. This is important because it lessens the burden on the original sender and makes the repair process faster if the distance between the NACK sender and the original sender is large.
Each member of the session will cache the latest data-packets received (and sent) and if they see a NACK for a packet they have in their cache, they retransmit that packet to the whole group as a repair.
To minimize the number of NACK's and repairs, these two operations are preceded by exponential back-off. This means that instead of sending the packet directly, the client waits for a random time (based on the distance to the related other party) and if another client sends a corresponding packet it withdraws its own potential transmission or increases its own timeout. So timers are an important mechanism within SRM both for NACK and repair packet reduction.
A mechanism which is not so common is the possibility of automatically creating local groups which will interact to help repair missing packets. A local receiver will act as a local sender as well and create a small multicast group and send an error fingerprint out to all members. The members that have a certain number of the missing packets listed in the error fingerprint can join this local group and repair other receivers. This might help fix the problem of a "crying baby" - a receiver that is losing too many packets too often, but it will increase the bandwidth usage and add to the general network overhead.
There are only a couple of implementations mentioned, a white board type application that is cited many times, and a SRM library. However, these were created in the mid-nineties and doesn't seem to be available anymore.

3.2.2 PGM - Pragmatic General Multicast

PGM, also called Pretty Good Multicast, is a reliable multicast protocol created by the RMRG, Reliable Multicast Research Group, within the IETF, Internet Engineering Task Force, and has resulted in RFC 3208 [22].
From the definition in the RFC: "Pragmatic General Multicast (PGM) is a reliable multicast transport protocol for multicast applications that require reliable, ordered, duplicate-free multicast data delivery from multiple sources to multiple receivers. PGM guarantees that a receiver in a multicast group either receives all data packets from transmissions and retransmissions, or can detect unrecoverable data packet loss. PGM is intended as a solution for multicast applications with basic reliability requirements."
PGM was designed with simplicity in mind and follow the repair pattern of

NACK's. An implementation only requires a few data types with the following use pattern: message data, ODATA, from the sender to all receivers. This message is tagged with a Transport Session Identifier, TSI. This is an unique identifier that is used by the receivers to identify any missing packets. For any missing packet a NACK with only the TSI is sent to the sender or to a designated local repairer - DLR, and a repair packet, RDATA, is sent in return.
To reduce the amount of NACK's sent, support for PGM has been implemented in routers. This is called PGM Router Assist and can be found in equipment by Cisco Systems, Juniper Networks, Nortel Networks and possibly others. On a small LAN this may not be an important function, but will be when a wide area network is used. A NACK is sent unicast to nearest PGM capable router which in turn will keep sending the NACK until a repair data packet has been received. To inform the router of correct path of NACK packets, a SPM - Source Path Message - is sent at regular intervals.
An important detail about NACK handling is that the sender will only keep packets available for retransmit within a set transmit window. If a NACK has not been received within this time window, repairs will not be available. The receiver will be notified and will have to deal with this at the application level. Forward Error Correction in the form of packet-level Reed-Solomon Erasure techniques [24] is also part of PGM.
The most notable implementation of the PGM protocol is in the form of the open source project OpenPGM [14]. It is in use by several other open source projects. For unknown reasons, continuing development has more or less stalled. A few commits the last few years, but not much else.
As stated in the OpenPGM Google Code Archive [9] there are also proprietary implementations by TIBCO SmartPGM (originally by WhiteBarn, bought by Talarian), Microsoft Windows XP and later, IBM WebSphere MQ, and RT Logic's RTPGM.
An effort was done by Christoph Lameter in 2010 to get PGM support in the official Linux kernel, but that work seems to have stalled. Kernel support would have improved the throughput input further.

3.2.3 NORM - NACK-Oriented Reliable Multicast

As the name implies this is another NACK based RMC solution. RFC5401 [3] and RFC5740 [4] defines the implementation details and obsoletes RFC3940 [2] and RFC3941 [1].
In short, NORM defines three types of data: NORM_OBJECT_DATA, NORM_OBJECT_FILE and NORM_OBJECT_STREAM . The two former is used for bulk data and is divided in two only to inform the receiving application of what to expect and to acquire memory space or disk space depending on the type. What differentiate it further from PGM is two other message types: NORM_INFO and NORM_CMD. These can be used by sender and receivers to control the message flow in various ways.
The extra complexity, and thereby also flexibility, might be the reason that the only readily available implementation is a C++ library being maintained by the Naval Research Laboratory (NRL) [15]. Last update was July 2015, so it is more or less in the same state as OpenPGM. The NORM implementation does not seem to have gotten the same degree of use though.

3.2.4 SRMP - Selectively Reliable Multicast Protocol

This protocol is meant for applications where a large portion of the data is of a type that does not need reliablity and a smaller part needing reliablity.
It uses a NACK based scheme where each unreliable packet has the sequence number of the last reliable packet attached. This way the receiver will know if it missed a packet and can request a retransmission. SRMP also makes use of TCP-Friendly Multicast Congestion Control as described in RFC4654 [23] and a number of other components described in other RFC's. The full protocol description can be found in RFC4410 [18]

3.2.5 RMC Summary

With regular multicast it's basically fire and forget. Messages will either get to the recepients or not. No way to retrieve lost messages without some extra measures. This is what reliable multicast tries to solve. All solutions require some kind of back traffic to keep in sync with the sender. It will be of either type ACK or NACK, where the first will send an acknowledge message for each packet, or a variant thereof, and the second will send a negative acknowledge packet for packets *not* received. In either case it is required that each packet is tagged with a unique id, and that some bookkeeping will be needed on either sender or receiver side. There are also ways to make an intermediate receiver also act as a proxy sender and also ways to include network units in the packet repair procedure. Most of the techniques used in RMC are introduced to reduce the amount of back traffic.
There are two open and standardized RMC protocol implementations. OpenPGM [14] of the Pragmatic General Multicast, PGM, specifications in RFC3208 [22] and NORM [15] of the NACK-Oriented Reliable Multicast, NORM, specifications of RFC5401 [3] and RFC5740 [4]. PGM and also OpenPGM is cited more places and seems to be in more general use. In addition, there are built-in support for PGM in many switches such as those from Cisco.

4 Object Contents

Whatever the underlying technology the important part is the actual data. It comes in all shapes and forms and will evolve and change in unpredictable ways during the lifetime of the system. Depending on the application at hand, the type of data may vary quite a lot and require different solutions regarding object handling. An ATC flight plan messaging system will mostly handle numerous fields of textual data and this section will concentrate on solutions to this.
The terms used when preparing an object to be transmitted over the network and received by clients serialization and deserialization. Wikipedia has this definition: "the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and resurrected later in the same or another computer environment". The important part here is that the same data sent from one side shall be received as the exact same copy on the other end. Typically this has often been done using proprietary binary protocols only applicable to the applications from the same company, or even only within the same department. With the emerge of the Internet the need for easier exchange of data objects got more and more important. It more or less started with XML but more and more interesting contenders have gotten widespread adoption.
An object definition is called a schema and is defined by an IDL, interface description language. The IDL consists of types that tell what the elements in the objects contain. That can be numbers, strings, arrays and more specialized types like a date. The number of different types vary a lot between IDL's and the complexity between the corresponding schema's do too. This should be taken into consideration when choosing one. How the IDL's cope with applications that evolve and how the accompanying object change is also a factor to consider.
In the next sections we will have a look at these and how they cope with different aspects of serialization/deserialization. A simple python implementation that operates on a flight plan object will be made for each IDL. This will help to more easily compare and visualize the different technologies. The following attributes will be handled:

- SSR Code - Type: Octal numeric value
- Callsign - Type: 7 character string
- WTC - Type: enum
- Route - Type: Array of strings

The main aspects that will be discussed is backwards compatibility, type support, language support and ease of use.

4.1 Proprietary Binary Protocol

For applications that have lived for a while, it is not uncommon to see completely binary protocols closely linked to the source code. The serialization can be done as easily as just copy the entire data structure into a buffer and transfer to the receiver which again copies the data into the same data structure. This will be very simple and efficient processing wise; there is no need for any external

processing of the data before it is sent or after it is received. It can also be very efficient if the data structure is made efficient, i.e. no large mostly unused arrays or lots of attributes with only a subset in actual use. This part can be eliminated by moving the data from the verbose data structure into a smaller structure designated for network transmission. However, this adds processing overhead but will most likely still be efficient, both size and computation wise. It is quite a different story when it comes to validation and schema evolution. If the data structure on the sender and the receiver end is different one can end up with wrong data on the receiver perhaps without even noticing. It might also lead to crashes when the receiver starts using unexpected data. A binary protocol more or less mandates same version on sender and receiver, or some complex processing to reduce the problems.
As there are no common way of handling data in a simple binary protocol it is not very interesting to create a sample application for this.

4.2 XML

Usable working drafts of XML started to emerge at the end of the nineties. It is goverened by World Wide Web Consortium, W3C, and has had many updates and side tracks since then. It is quite ubiquitous for inter process messaging, especially on the Internet, and will be a safe choice for the foreseeable future.
However, it is often criticized as being very complex and verbose and with large message size overhead, so it is generally not considered to be the first choice as a messaging protocol. Even for the very simple example used in this section the schema file became quite complex. See listing 16 in Appendix B for reference. The XML data file shown here is not as complex though:

Listing 1: XML Data

```
<?xml version="1.0"?>
<FPL>
 <SSRCode>5523</SSRCode>
 <callsign>JLJ3105</callsign>
 <wtc>J</wtc>
 <route>
  <leg>LEG1</leg>
  <leg>LEG2</leg>
  <leg>LEG3</leg>
 </route>
</FPL>
```

However, it is tiresome to read and more complex data sets become almost impossible to read and understand for human beings. Its complexity is one of XML's biggest drawbacks.
Because of its wide adoption there are tools and libraries for every possible programming language out there and an abundance of documentation. XML supports a lot of data types, and it also possible to create complex ones out of simpler ones. The use of a XML schema makes it possible to validate any data and act when wrong data is encountered. The small example used here shows that a rather complex schema file can remove a lot of expressive and detailed code. Since the boundaries of data is expressed in the schema the programmer can concentrate on dealing with boundary crossing more than checking the boundaries. For instance, adding or removing a character in the callsign will

directly lead to an exception due to its restriction of string length of 7. Reading and validating the above xml file can be done in very few steps as can be seen in listing 18 in Appendix B.
If the schema changes, it is possible to stay backwards compatible by having new data members optional, but staying backwards compatible using XML is hard. An old, but still interesting writeup on the subject can be found in [8]

The steps of creating a python xml reader/writer is quite simple using the PyXB xml libraries.
1. Create the XML schema file - FPL.xsd
2. Create the corresponding class files using the PyXB generation tool. pyxbgen -u FPL.xsd -m FPL
3. Create the XML data file - FPL.xml
4. Create the python source code - fdpsServer.py. Most importantly:

```
import FPL
xml = open('FPL.xml').read()
print('Flightplan details:')
print('\tSSR Code: %d\n\tCallsign: %s\n\tWTC:
 %s\n\tRoute:' % (fpl.SSRCode, fpl.callsign, fpl.wtc))
```

4.3 JSON

JSON is an acronym for JavaScript Object Notation. Even though the name suggests that this is something related to Javascript, it is in fact one of the most language agnostic protocols out there. It has been available since the beginning of 2000 and is possibly the technology used by most online providers for their web services. Fetching data using a public API from companies such as Flickr and Facebook will result in JSON strings with the requested data.
JSON is one of the few data serialization formats that is IETF standardized with RFC4627 [6]. The standard only defines a few basic data types, but it is quite extendable and unsupported data types such as date and regular expression is available in most implementations. Note that those are not real types, but rather a specially formatted string that represents the type of interest. For instance, a typical toJSON string representation of a date is:
2018-05-31T18:25:43.511Z
Both the JSON schema and data file is a lot simpler than their XML counterparts, as can be seen with the following example schema and data file respectively:

Listing 2: JSON Schema

```
{
    "title": "FPL Schema",
    "type": "object",
    "properties": {
        "SSRCode":  {
                        "type": "integer"
        },
        "callsign": {
                        "type": "string",
                        "minLength": 7,
                        "maxLength": 7
        },
```

```
        "wtc":      {
                        "enum": [ "L", "M", "H", "J" ]
        },
        "route":    {
                        "type": "array",
                        "items": {
                                  "type:": "string"
                        }
        }
    }
}
```

Listing 3: JSON Data

```
{
    "SSRCode": 5523,
    "callsign": "JLJ3105",
    "wtc": "J",
    "route": ["LEG1", "LEG2", "LEG3"]
}
```

Both can easily be read by humans and can be verified by command line, in code or with the help of two online services - http://jsonschemalint.com/ and http://jsonlint.com/. Most programming languages come with several tools to make working with JSON as simple as possible. The test implementation (see listing 21) shows that reading a schema and a JSON data file and validating it can be done in very few lines of code:

```
import json
from jsonschema import validate
with open('FPL.schema.json', 'r') as
json_schema:
   schema = json.loads(json_schema.read())
json_schema.close()
with open('FPL.json', 'r') as json_data:
   fpl = json.load(json_data)
json_data.close()
```

The example translates both the schema and the data file into simple Python dictionaries which is one of the most common ways to handle data in Python - see 21 for the full listing. There are also ways of serializing full objects automatically. Inherently JSON doesn't support schema evolution very well, so that is a task that will mostly be left to the application developer.

4.4 Apache Thrift

Apache Thrift is another project in the large portfolio of programs and technologies from the Apache Foundation. It was originally developed by Facebook but was open sourced in 2007 and added to Apache in 2008. Although it can be used as simple a serialization/deserialization framework, it is a lot more than that, adding automatic client/server support, RPC and a fairly rich IDL. Bindings for most languages and all necessary tools are included in the Apache Thrift install package.
Typical usage is to write a .thrift file with the object definition - see listing 22 below. It uses a very C-like language, with structs, a number of base types, containers (list, set, map), exceptions and services - akin to an interface or pure

virtual abstract class. The .thrift file is then used to generate stub classes and methods that can be used directly for client/server communication. Included in the python framework is a transport library that by default uses a simple binary protocol. If one desire a text based protocol, it is possible to use a JSON protocol instead. This can be very useful when debugging as one can read the values directly using Wireshark or other net sniffing tools. There are two python JSON implementations where the first is very simple and exchange the variable names with id numbers and thus gives little meaning without the schema. The other implementation, ironically called SimpleJSON, includes the variable names too and can be very valuable during debugging.
Client/server support can be used directly with the included methods as defined in the API as well as RPC functionality. All this is built into the framework so it is a simple way to get client/server and RPC functionality automatically. For a lot of projects this setup should cover most, if not all, of the functionality needed.

Listing 4: Thrift Schema

```
enum WTC {
      L,
      M,
      H,
      J
}

struct FPL {
      1: i32 SSRCode;
      2: optional string callsign;
      3: optional WTC wtc;
      4: list<string> route;
}
```

As can be seen from the thrift schema file, the struct fields are uniquely numbered. This, in addition to tagging each field as required or optional, provides the basis for handling schema evolution. As long as the following rules are obeyed, different versions of a thrift schema can exist within the same network:

- Do not change the tag numbers of any existing fields.
- Do not add or delete any required fields.
- It is ok to delete optional or repeated fields.
- It is ok to add new optional or repeated fields but use fresh tag numbers

Thrift has many of the features sought after; fairly rich IDL, backwards compatibility, good language support. However its biggest weakness is the lack of proper documentation. Even to get the information to write the simple example in this description was a struggle. Most examples concentrate on client/server setup, but anything further quickly leads to the scarcely documented API on the thrift website. Listing 24 shows a simple approach to creating an object, serialize to json and save to disk.

4.5 Google Protocol Buffers

Google Protocol Buffers, or protobuf for short, was initially developed by Google for internal use in 2001. It was open sourced as version 2 in 2008 and has a

large user base, including all internal systems in Google. Only C++, Java and Python are officially supported but there are third party bindings for a number of other languages.
One of the main reasons for creating protobuf was to allow for backwards compatibility/schema evolution. This is done in the exact same way as Apache Thrift, by using field identifiers in the definition. The same guidelines for proper use will be the same as for Thrift.
Protobuf aims to be quick and create small objects when transferred. Knowing that Google handles a tremendous amount of data in its data centers, one can assume that great effort has been put in getting the size and speed as optimal as possible. Some benchmarks [10] and [20] also backs this up.
Before transmission the data is converted to a very efficient binary protocol that is still backwards and forwards compatible. The binary data created for the test example is only 32 bytes. Although very small it is near to impossible to debug without direct access to the schema as well. An ASCII serialization implementation can be used when debugging, but that is not backwards compatible. In addition a Wireshark plugin is available that with the help of the .proto file (the protobuf schema file) can decode any messages sent.
As for Thrift, a number of base types are supported. Protobuf is purely an interchange format and does not support or create RPC and client/server code automatically. However, another project, gRPC [11] is created for this purpose. From Wikipedia: "gRPC (Google Remote Procedure Call) is an open source remote procedure call (RPC) system initially developed at Google. It uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts. It generates cross-platform client and server bindings for many languages"
Another feature provided by the message classes is reflection. You can iterate over the fields of a message and manipulate their values without writing your code against any specific message type. One very useful way to use reflection is for converting protocol messages to and from other encodings, such as XML or JSON. A more advanced use of reflection might be to find differences between two messages of the same type, or to develop a sort of "regular expressions for protocol messages" in which you can write expressions that match certain message contents.
Usage pattern is the same as with thrift; create protobuf schema file, run protobuf compiler to create the corresponding classes. Include the automatically created header file, create an instance of the class and populate with values. The following listings shows the example schema file and the three lines of code necessary to populate one specific field.

Listing 5: Protocol Buffers Schema

```
syntax = "proto3";

package FPL;

message FPL {
  enum WTC {
    NOTSET = 0;
    L = 1;
    M = 2;
    H = 3;
```

```
    J = 4;
  }

  uint32 SSRCode = 1;
  string callsign = 2;
  WTC wtc = 3;
  repeated string route = 4;
}
```

```
import FPL_pb2
newfpl = FPL_pb2.FPL()
newfpl.SSRCode = 2323
```

As can be seen, the schema looks a lot like the Thrift schema. Both are easy to read and create, and vastly easier than XML or even JSON. What really differentiate Protobuf and Thrift is the vastly superior documentation of Protobuf. While getting information on Thrift usage was a struggle, examples and discussions on Protobuf were readily available.

4.6 Apache Avro

Apache Avro is yet another Apache technology covering serialization/deserialization. It uses JSON for its schema declaration and one could think that schema evolution is a bit lacking. However, Avro solves this in an elegant way by actually including the schema with the encoded data. The data can thus be more compact as there are practically no overhead with the actual data. Whether the size end up larger or smaller than other solutions would depend a lot on the actual data. Lately a separate Avro specific IDL has been added. It resembles the way protocol buffers defines the schema, but adds a number of extras, such as remote procedure calls and java type annotations. The c++ implementation does not support the full range of the Avro functionality, namely RPC support, as of v1.8.2 [5]. Even though RPC is not needed for the Tower SW, being a second class citizen is not preferable.
Dynamic languages may also enjoy the fact that the schema is attached to the data, and one can use the data with no code generated upfront. This facilitates construction of generic data-processing systems and languages.
An Avro schema, as the following listing shows, is more complicated than its Thrift/Protobuf siblings, even though it is based on JSON.

Listing 6: AVRO Schema

```
{
 "type": "record",
 "name": "FPL",
 "fields": [
     {"name": "SSRCode", "type": "int"},
     {"name": "callsign",
      "type": {"type": "string", "minLength":7, "maxLength":7}
     },
     {"name": "wtc",
      "type": {"type": "enum", "name": "WakeTurbulenceCategories",
               "symbols": ["L", "M", "H", "J"]}
     },
     {"name": "route", "type": {"type": "array", "items": "string"}
         }
 ]
```

```
}
```

Avro struggles with the same major problem as Thrift; lack of documentation.

4.7 DDS IDL

Data Distribution Service, DDS, is not only for serialization, but is a full messaging library or middleware if you like. It is commercially supported and advertised by very dedicated vendors, such as PrismTech with Vortex OpenSplice and RTI, Real Time Innovations with Connext DDS. Because of this it has gotten great attention within certain industries, such as air traffic control, financial and other big data providers. Both provide open source versions in addition to their commercial offerings. However, getting to the open source versions are more cumbersome than for instance getting the open source at heart OpenDDS. Feature wise OpenDDS is a bit behind the other solutions though.
DDS was started by RTI and Thales in 2001, and standardized through the Object Management Group, OMG, in 2003. Current version 1.4 was released in 2015 [16]. In this section we will concentrate on the IDL and how that differentiate from the other offerings. First of all, DDS is only fully available for C++ and Java. A Python version is possible through a bridge only, so a full example will not be created here.
Everything in DDS revolves around topics. Each topic is a collection of fields building a particular object. Anyone interested in this topic only need to subscribe to it, and DDS will automatically make that happen. The IDL supports all common base types including union, enum, arrays and nested structures. A possible schema for the FPL example can be seen here:

Listing 7: DDS IDL/Schema

```
enum WTC { L, M, H, J };

struct FPL {
    long SSRCode;
    string callsign;
    WTC wtc;
    sequence<string> route;
}
```

This shows a very C-like structure and is both easy to use and create.
For backwards compatibility, OMG has added XTypes to the DDS specifications. It allows new types to be added to the end of existing schema's. RTI has a good write-up of this topic in [19]. OpenDDS does not support XTypes so for Tower OM, only the commercial offerings are suitable.

4.8 Summary

For pure serialization/deserialization the best options currently seem to be Google Protocol Buffers and Apache Thrift, especially when schema evolution is a priority. Both are developed by big organizations but the former has better documentation. Both support schema evolution by using field identifiers, and by following some general rules about what and how to change the schema, it is possible to be backwards compatible. Both aim to create small data messages and have efficient serialization/deserialization. As both are used by big data

movers - Google and Facebook - one can assume that great effort has been put in making the implementations efficient.
DDS is also very suitable alternative. Although it is a complete framework and not only for serialization/deserialization it solves the issues of Tower OM and adds a very rich IDL and a standard organization, OMG, that is constantly working on improving the specifications. Unfortunately. the most prominent open source implementation, OpenDDS, does not support all the latest functionality and is therefore not a choice for Tower OM. Basing the inner and most important communication in the Tower SW on proprietary implementations may not be a good option. If there are bugs in the implementation, it will be very hard to debug and there is always a question of where the software will be heading in ten-twenty years time.
The other alternatives do not support schema evolution in a particularly good way, are lacking proper documentation or implementations, has varying degree of use and backing or have complicated schema requirements. There are other similar technologies such as Kryo, ASN.1, MsgPack or hessian that were not discussed due to the same reasons. These limitations may quickly change so they should be given a chance in any new technology evaluation.

5 Object Management

Object management has got its own acronym - MOM - Message-oriented middleware. From Wikipedia: "Message-oriented middleware (MOM) is software or hardware infrastructure supporting sending and receiving messages between distributed systems."
Depending on the definition one uses, a MOM can come in a variety of different forms, from a simple software library that can be used within the software developed, to full enterprise scale software ecosystems that require full time software engineers to operate.
From the simple solutions to the complex ones there are solutions that gradually add more and more functionality and also complexity. Terms often used are Integration Framework, ESB (Enterprise Service Bus), Integration Suite, Message Queue and the beforementioned MOM which is used both as an individual solution and as a general term for all of the other terms. Integration Framework gives the software developer a software stack that will help getting a message from A to B more or less regardless of protocol and implementation details. One can also think of it as a layer on top of the existing network stack that removes some of the complexity in a distributed system where components may even speak different languages or use different data formats. It is also often provided as part of a framework that concentrate mostly on data formats but add integration functionality as well. When moving further towards ESB and Integration Suites the included functionality increases and also adds tools and monitoring systems to further separate the messaging handling part from the core system implementation. The two latter will use what is known as a broker for message distribution. I.e. the message is sent to a central unity that is responsible for routing the message to the correct recipient(s).
Within this mix, there is also something called a Message Queue. It is often used interchangeably with MOM, and also has a wide definition. As the name suggest, messages may be queued up and sent to recipients whenever ready. This can be interpreted as a possibility for delays, but many of the solutions can be set up with a very high throughput.
Selecting a MOM solution to use shall be given a considerable amount of thought. The solution will be a core part of the system and have great implications if selected on wrong qualifications.
The following sections will give information on some of the offerings. Even though the main focus has been on solutions using RMC, other possibilities are explored as well both to get a clearer picture and too see if other options could be used as well.

5.1 MQ - Message Queues

Advanced Message Queing Protocol, AMQP, is actually a standard, and one of the oldest ones of such. It was initiated in 2003 by JPMorgan Chase and later involved a number of other large enterprises. In 2011 it was reorganized as an OASIS member. Initially a few specifications were made available, versioned 0-8, 0-9, 0-10 and 0-9-1. 0-9-1 was what most implementations started using, and several are still only supporting that version, even though the 1-0 version was released 2011/2012. However, the specification difference is so large that they can be considered different protocols.

The goal was to create a standard for inter business message passing that guarantee interoperability between implementations. As long as they 'spoke' AMQP they would be able to communicate. The current specification(s) rely on a reliable transport mechanism such as TCP/IP, and even though support for reliable multicast using PGM is mentioned, nothing has yet come out of it.
Figure 2 below show a typical setup. A publisher transmits packets to a broker or an exchange that will transmit to whatever recipients configured to want them. There are several implementations of AMQP and the ones often brought

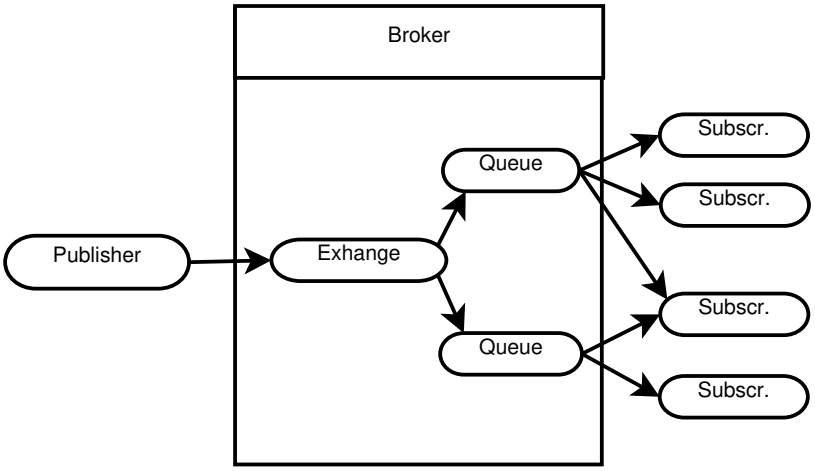

Figure 2: Message Queue Connection/Message Flow

up are OpenAMQ, SwiftMQ, Apache QPID, Apache ActiveMQ and RabbitMQ. Interoperability wise one can choose to have clients using RabbitMQ and the broker set up with Apache QPID. Some of these also support other connection protocols/API's such as JMS - Java Messaging Service. JMS is an integral part of the Enterprise Java Platform, and is very commonly used in Java environments. AMQP is by some considered an evolution of JMS, basing many of the semantics on it.
IBM Websphere MQ has by far the largest market share for message queue systems. It uses a proprietary wire protocol, and will thus not be directly compatible with other message queue protocols, such as AMQP. However, in 2015 [12] support for AMQP clients were added in the form of IBM MQ Light. The fact that it is so widely used means that it will generally have more features, larger third party support and a larger distribution of technical support personnel than other solutions.
A solution based on a Message Queue technology may be able to solve the issues found in the Tower OM implementation. However, as it adds a broker to the equation, the complexity increases and given a system that has spent a great deal on reducing complexity this might not be desirable. If the whole system was more heterogeneous or there was reason to believe it had to scale a great deal more using something based on AMQP might have been the correct choice.

5.2 ZeroMQ

ZeroMQ [26] is very often referred to in the same context as Apache ActiveMQ, RabbitMQ and other MQ solutions. Even though they try to solve the same problem, getting messages from publisher to subscribers, they solve them in fundamentally different ways. One can be misguided by the MQ in the name, but the Zero in front indicates *no* messages queue and not a message queue named Zero. In essence ZeroMQ works on a much lower level than the message queue counterparts. It tries to replace/or live on top of the low level socket API we normally work with when programming applications. Instead of opening for instance a TCP/IP socket a ZeroMQ socket is opened and used very much like the ordinary sockets. However, under the surface a lot of 'magic' is going on. At the base, it's just a library that will be included in the software, and thus will not need any external software or setup to be used; include the header files, apply the ZeroMQ API, link the application and run it. ZeroMQ is obviously well suited for communication based on TCP/IP, but can also work interprocess and inter thread as well as multicast. Reliable multicast support is done with either OpenPGM or the NORM implementation described in section 3.2.2 and 3.2.3, which makes ZeroMQ a very flexible and modular socket type implementation. It touts itself to be simple to use, and does in many ways live up to it. This is what it takes to create a ZeroMQ socket using PGM and IPC at the same time and send a line of text.

```
zmq::context_t* context  =new zmq::context_t(1);
zmq::socket_t* fplsocket =new zmq::socket_t(*context, ZMQ_PUB);
fplsocket->bind("epgm://eth0;239.10.10.10:3105");
fplsocket->bind("ipc:///tmp/simple_pubsub");

std::string msg = "HELLO WORLD";
zmq::message_t zmsg(msg.length());
memcpy(zmsg.data(), msg.data(), msg.length());
fplsocket->send(zmsg);
```

Under the hood it will take care of I/O asynchronously, automatically reconnect, queue messages, route messages and many other tasks. This using only a simple API. Figure 3 shows that the publisher and subscribers connect and operate in the same manner as a normal socket. Note however that ZeroMQ also support other communication patterns, such as request/reply, pipeline and routing.

It is also agnostic when it comes to wire protocol, data contents, for the

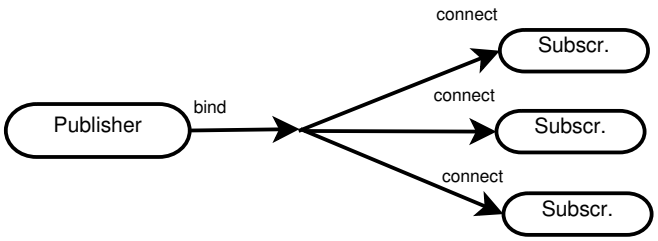

Figure 3: ZeroMQ Pub/Sub Connection/Message Flow

messages and can thus support any type of object content protocol, be it a proprietary solution or something based on any of the aforementioned options.

From this we can gather that ZeroMQ is a layer between the transport layer and the object layer fitting very well with a Unix philosophy of doing one and only one thing, but do it well. In fact, its implementation using TCP/IP has a higher throughput than pure TCP/IP, at least for small (<100bytes) messages, but a slightly higher latency [27]. There are also bindings for 40+ languages, but note that not all have been updated to support the latest 4.x version.
As an integral part of solving the issues of the Tower OM, ZeroMQ is a good candidate. This is due to its simplicity and focus on speed and functionality as well as its direct support for both PGM and NORM. One perceived shortcoming of ZeroMQ is that its internal structure does not allow for easy integration of new transport protocols, but for this setup it does not really matter. There are other issues too, such as POSIX compatibility and sockets not being thread safe. Another socket library called nanomsg tries to solve all these issues and then some, but it has not gotten such a large user base, and it does not support reliable multicast. It also seems it is now being rewritten in the name of nng - nanosg-next-gen.

5.3 DDS - Data Distribution Service

As described in section 4.7, DDS is a protocol specification from Object Management Group, OMG. One of the key selling points of DDS is the simplicity to both publish and subscribe to data feeds. A single data feed from a publisher is called a topic and is distributed to any subscriber subscribing to this topic. There are no need to handle network entities like addresses and ports, as this is handled transparently by DDS. Figure 4 shows a diagram of a simple

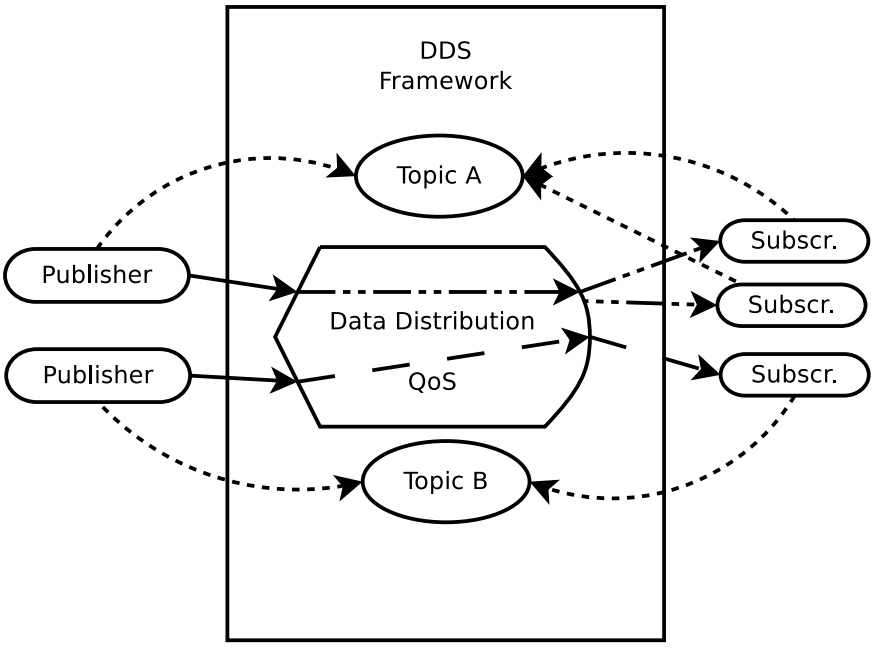

Figure 4: DDS Connection/Message Flow

setup with two publishers distribution Topic A and B to some subscribers. Two subscribing to Topic A, and one to Topic B. The DDS framework ensures that the subscribers will receive the designated messages for the different topics they have subscribed to. The following simplified code listings shows the necessary method calls needed to create a publisher and subscriber for a specific topic:

Listing 8: DDS Publisher

```
int main(int argc, char *argv[])
{
    participant = DDSDomainParticipantFactory::get_instance()->
                  create_participant(0,QOS,NULL,StatusMask);
    topic = participant->create_topic("Hello, World",TypeName,QOS,
        NULL,StatusMask);
    data_writer = participant->create_datawriter(topic,QOS,NULL,
        StatusMask);
    string_writer = DDSStringDataWriter::narrow(data_writer);
    string_writer->write("Hello, World",DDS_HANDLE_NIL);
}
```

Listing 9: DDS Subscriber

```
int main(int argc, char *argv[])
{
    participant = DDSDomainParticipantFactory::get_instance()->
                  create_participant(0,QOS,NULL,StatusMask);
    topic = participant->create_topic("Hello, World",TypeName,QOS,
        NULL,StatusMask);
    data_reader = participant->create_datareader(topic,QOS,&
        listener,...);
}
// called automatically when data is available
void HelloListener::on_data_available(DDSDataReader *reader)
{
    string_reader = DDSStringDataReader::narrow(reader);
    while(1)
    {
        retcode = string_reader->take_next_sample(ptr_sample,info);
        cout << sample << endl;
    }
}
```

Note the small amount of source code necessary to actually get a publisher and subscriber ready. With this you can start up as many subscribers as you like. Also note that there are parameters for Quality of Service, QoS, as well. These can be used to control reliability, performance and a number of other things. These variables can also be controlled via a XML file as well, to remove the need to recompile if any QoS needs are changed. There are also other parameters one needs to get acquainted to. Event though it is simple to get started, the flexibility of DDS requires the developer to study the API guides thoroughly.
There are several vendors providing DDS implementations, and they are all focusing on interoperability. In fact, there are regularly interoperability tests done to make sure the different implementations can work together. DDS is also focused on being light weight with low overhead and high performance, and communication is based on multicast. As a solution it looks to sit neatly in between a message queue system and ZeroMQ.

5.4 SOA and ESB

Finally, we shall briefly look at some of the buzzwords that have become popular the last few years: SOA and ESB. SOA is short for Service Oriented Architecture and more or less relies on ESB, Enterprise Service Bus, and an Integration Suite.
Now, what does this really mean? In short SOA describes a way to modularize a complex distributed system into separate services that integrate into the greater whole via messaging. This is where the ESB comes into play. Once a system evolves beyond a certain threshold the job of keeping the different services synchronized when they evolve asynchronously starts to be become overwhelming. An ESB removes some of this complexity by being a central point of message handling and routing in the form of a software suite to be run in a server environment. It can take most common protocols, easily convert to another and route to any number of other services as well as 'massaging' or translating the message contents according to certain rules. Add to this security layers, authentication, QoS, message filtering and a host of other possibilities and one can see that this can be a powerful tool in a large enterprise system. A Message Queue/broker solutions can also provide many of these functions so there is a large and confusing overlap between the two.
There are many contenders in this field, with commercial offerings by, among others, IBM WebSphere, Oracle and Microsoft. Notable open source offerings are Mule ESB, WSO2, JBoss and a number of others. See [17] and [25] for a rundown of different solutions.
Note that message queue/broker solutions clearly enters the realm of an ESB, and there is big confusion of what, if any, differences there are.
ESB is often considered a different technology than the simpler MOM systems, but it is within the definition by operating as a central point for message routing. However, the flexibility and sheer amount of possibilities also bound for a very complex environment. Even though most products come with graphical workbenches that helps configuring and monitoring the different parts, it will need a substantial effort to get configured and operating correctly. In fact, the most common argument against using an ESB or an Integration Suite is the complexity.
As these solutions are aiming at large, heterogeneous systems they do not fit very well with a smaller scale Tower SW as are discussed in this paper.

5.5 Summary

When it comes to selecting an object management system one really has to analyze the needs of the applications that shall be implemented and/or integrated. A simple in-house system where you control all parts will require something completely different from a financial system involving several thousand clients and messages routed across the globe. There are typically three types of solutions:

- Socket libraries such as ZeroMQ which is used as a building block for rolling your own communication system
- Message queue systems, with a broker responsible for distributing messages to interested subscribers. The broker is mostly a service running

on a dedicated server, but may, as in DDS, be included in the framework itself.

- Enterprise Service Bus meant for really large heterogeneous and complex environments.

When choosing a socket library such as ZeroMQ, you need tight control of all aspects of the communication and will need to create some amount of boilerplate code to integrate everything together. In return you will get fast and efficient delivery with as little administrative and network overhead as possible and full flexibility.
For a more integrated solution, you find Data Distribution Service, DDS. It includes functionality for automatic network discovery, Quality of Service, reliable message distribution and an IDL for object definition. DDS is backed by several vendors which will provide support in different ways, all from a typical Open Source community to fully commercial variants.
Message Queue, MQ, is useful when your application is very message centric and have a very diverse portfolio of subscribers in terms of language, type of application, network distance etc. Distributing messages in such an environment is done through a broker that will route messages to correct receivers and also add persistent storage to allow for restart of services etc. The biggest commercial vendor in this area is IBM with their IBM Websphere MQ and associated technologies. Another solution that should be considered is any of the AMQP implementations, especially if vendor lock-in is a concern. AMQP is an open standard that has several implementations such as ActiveMQ and RabbitMQ.
Enterprise Service Bus and Integration Suites/Buses are a further development of the MQ domain, where even more control can be made on how messages are treated and adds security layers, authentication and more QoS possibilities. These solutions are for large heterogeneous enterprise systems that integrate a large number of different systems.
To solve the problems described for the Tower OM, a simpler solution such as ZeroMQ or DDS is the preferred option. The Tower software is based on simplicity both in configuration, administration and code wise and using ZeroMQ and DDS will mostly keep it this way with the bonus of improving the functionality. DDS adds somewhat more administration and engineering complexity than ZeroMQ, but on the other hand adds some useful functionality for 'free' such as quality of service and automatic network discovery.

6 Summary

As have been seen, there is a large amount of material and discussion on the topic of reliable object management.
For reliable multicast a lot of work and discussion was done during the late nineties, especially within IETF. However the pace seemed to slow down around the year 2000 and has only lately been gaining steam again. Table 3 gives a brief summary of some of the variants:

Manufacturer/Name	**Comment**
SRM	NACK based Used as reference for other protocols and have two old implementations that seem to be unavailable now.
PGM	NACK based Supported by some network units Only standard widely used.
NORM	NACK based Apparently there is only one implementation available, with no widespread adoption
SRMP	NACK based Uses two data streams - one for reliable data, and one for data that will allow lost packets. No available implementations

Table 3: Summary: Reliable Multicast

The specifications for PGM is described in RFC3208 [22] and is the most widely implemented variant. OpenPGM is the one which is most widespread, but other implementations also exists, for instance as an integral part of the Windows operating system since Windows XP. NORM has an open source implementation by the Naval Research Laboratory (NRL) as well.
When it comes to object contents there are even more solutions. Some are solely for serialization/deserialization, and some add support for remote procedure calls to very easily build client/server systems. Table 4 gives a brief summary.

Name	Comment
Proprietary	All languages Simple/efficient, not easy to integrate with other systems
XML	All languages Ubiquitous and abundance of tools but tend to get very complex. Poor backwards compatibility.
JSON	Supports a lot of languages Used by a wide range of applications. Message definition easier than XML but can be convoluted Does not support schema evolution
Apache Thrift	Supports a lot of languages Supports schema evolution Clean message definition Lack proper documentation
Google Protocol Buffers	Officially only supports C++, Java and Python Third party support for a lot of other languages Supports schema evolution Clean message definition Provides good documentation
Apache Avro	Supports C, C++, Java, Python, Ruby, and PHP although focus is on Java Message definition in JSON but included in data messages and thus supports schema evolution Lack proper documentation
DDS IDL	Supports C++ and Java. A few others depending on implementation. Supports schema evolution throught DDS XTypes. Only supported by the commercial offerings.

Table 4: Summary: Object Contents

Google Protocol Buffers includes all functionality necessary for Tower OM, is simple and efficient in use. DDS IDL is as well, but given the fact that only the commercial, proprietary implementations supports everything needed, using it should be considered carefully.
Object management has gained more available solutions. The functionality and complexity varies a great deal. Table 5 gives a brief excerpt from the findings in section 5.

Name	Comment
ZeroMQ	Socket library focused on efficiency and simplicity.
AMQP	Message Queue standard with a number of different implementations. Reliable multicast not supported yet. Depend on a broker.
DDS	A framework for message distribution. Great commercial support. Has gained a great momentum lately. Some configuration needed, but simple and efficient to use.
SOA/ESB	Large enterprise solutions. Adds a lot of complexity, but also functionality.

Table 5: Summary: Object Management

ZeroMQ has received widespread adoption and as it also support/use both OpenPGM and NORM seem to make it a good candidate for a general purpose reliable object management solution.

All in all, the two possible solutions that stand out are OpenPGM + ZeroMQ + Google Protocol Buffers and DDS. Because of the proprietary nature of the DDS implementations, the former solution was selected as the Tower OM replacement and tested further in the Proof of Concept section - 7.

Part IV
Proof Of Concept

7 Introduction

This part shows two actual implementations using OpenPGM, ZeroMQ and Protocol Buffers. The goal is to explore the different aspects of the selected technologies and see if they are useful in the context described in this paper.

The first one will be a flight plan distribution system (FPDS) with a server and any number of clients that resembles what is an essential part of any ATC software.

The second will replace one specific object type and corresponding distribution in the Tower SW with one built using OpenPGM, ZeroMQ and Protocol Buffers.

8 Flight Plan Distribution

8.1 Introduction

The main user of RMC/OM in the Tower ATC software stack is the flight plan server and its various clients. Building a generic server and client based around the data flow of flight plans gives a good idea of how this technology can solve the main message pattern for this software.

8.2 Goals

Three factors are to be investigated with this simple flight plan publisher/subscriber implementation:

- Throughput measurements
- Backwards compatibility
- Subjective feel for how easy and useful the technology is

The results are listed in section 8.5

8.3 Design Overview

The fundamental functionality for this setup is a server that holds any number of flight plans and distribute those to all connected clients. To allow this to happen, the following key design choices were made:

- No redundancy; only one server.
- Updates from clients, be it additions, deletes or value updates, shall go as a transaction to server which updates the central database and then sends the same update to all clients.
- Keep alive message going both ways to make sure everything is in sync and to keep statistics

8.4 Implementation Details

8.4.1 Basic publisher/subscriber

At first, a very simple publisher/subscriber implementation was done with the data flow as depicted in figure 5. The publisher creates an array of strings with a size of 4 kB each. Every ten seconds all strings are sent one by one as quickly as possible. A simple publisher is listed in listing 10 and shows that only a

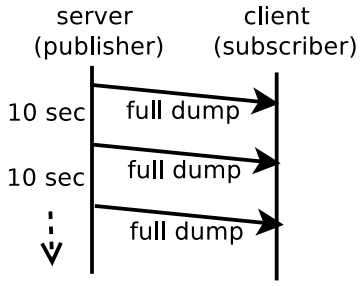

Figure 5: Simple Pub/Sub Setup

few lines are needed to setup the communication with ZeroMQ using PGM (see lines 18-24). This also includes setting socket options to raise the very lenient default values for transfer rate and high water mark. Transfer rate is set to 40 Mb/s to be well within limits for a 100 Mb/s network and high water mark, HWM, is set to unlimited. HWM sets the maximum number of messages that will be buffered before being sent. It is assumed that for this setup other factors are more limiting. A ZeroMQ specific can be seen on line 18:

```
zmq :: context_t * context = new zmq :: context_t (1);
```

The context is an internal structure of ZeroMQ that acts as a container for all sockets in a process. This means that the context can be reused throughout the program. As an extension to this, ZeroMQ is structured such that it is possible to share multiple connection types over the same socket. In our case, there's a fair chance that clients reside on the same host, and Inter Process Communication/IPC would be preferable. To achieve this, the following line could be added:

```
fplsocket->bind("ipc:///tmp/simple_pubsub)"
```

Messages sent using `fplsocket->send(...)` will be sent either using IPC or multicast/PGM, depending on connected clients and what ZeroMQ finds most suitable. The following three lines shows the necessary steps to send the strings across the network:

```
zmq::message_t fplmsg(str.length());
memcpy(fplmsg.data(), str.data(), str.length());
fplsocket->send(fplmsg);
```

These steps are repeated for each string in the vector. Additionally, one START and one END message is sent prior and after the loop is started. Time spent sending all messages is measured and presented for every run.

Listing 10: Simple Publisher

```
#include <iostream>
#include <sstream>
#include <chrono>
#include <thread>
#include <unistd.h>
#include <zmq.hpp>

using namespace std;

int main(int argc, char *argv[])
{
    stringstream connstr;
    connstr << "epgm://";
    if( argc > 0 ) connstr << argv[1] << ";239.10.10.10:50000";
    else       { cout << "Missing argument" << endl; exit(1); }

    cout << "Initializing network: " << connstr.str() << endl;
    zmq::context_t* context  =new zmq::context_t(1);
    zmq::socket_t* fplsocket =new zmq::socket_t(*context, ZMQ_PUB);
    const int rate = 40000;  // Set TX- and RX- rate - b/s
    fplsocket->setsockopt(ZMQ_RATE, &rate, sizeof(rate));
    const int hwm = 0;     // Set highwater mark to 0 - no limit
    fplsocket->setsockopt(ZMQ_SNDHWM, &hwm, sizeof(hwm));
    fplsocket->bind(connstr.str());

    vector<string> objects;
    int len=4096;
    for( auto id=0 ; id < 10000 ; id++ ) {
        string str(len, '#');
        objects.push_back(str);
    }

    int runs=0;
    while( 1 ) {
        sleep(10);
        auto start = chrono::system_clock::now();

        string tag = "START";
        zmq::message_t startmsg(tag.length());
        memcpy(startmsg.data(), tag.data(), tag.length());
        fplsocket->send(startmsg);

        vector<string>::iterator it;
        for( it = objects.begin() ; it != objects.end() ; it++ )
        {
            std::string str;
            str = *it;

            zmq::message_t fplmsg(str.length());
            memcpy(fplmsg.data(), str.data(), str.length());
            fplsocket->send(fplmsg);
        }

        tag = "END";
        zmq::message_t endmsg(tag.length());
        memcpy(endmsg.data(), tag.data(), tag.length());
        fplsocket->send(endmsg);

        auto end = chrono::system_clock::now();
        chrono::duration<double, milli> send_ms = end - start;
        runs++;
        cout << runs << " Finalized full sync of "
```

```
63                 << objects.size() << " flight plans in "
64                 << send_ms.count() << " ms" << endl;
65     }
66 }
```

Listing 11 shows an equally simple subscriber with the same characteristics as the server. Each received message is added to a vector which is cleared before each new run. ZeroMQ assures that all packets are whole, so no need to check buffer size and stitch messages together. Time used receiving the published messages is measured and presented for every run.

Listing 11: Simple Subscriber

```
#include <iostream>
#include <sstream>
#include <chrono>
#include <unistd.h>
#include <zmq.hpp>

using namespace std;

int main(int argc, char *argv[])
{
    stringstream connstr;
    connstr << "epgm://";
    if( argc > 0 ) connstr << argv[1] << ";239.10.10.10:50000";
    else       { cout << "Missing argument" << endl; exit(1); }

    cout << "Initializing network: " << connstr.str() << endl;
    zmq::context_t *context  =new zmq::context_t(1);
    zmq::socket_t *fplsocket =new zmq::socket_t(*context, ZMQ_SUB);
    fplsocket->setsockopt( ZMQ_SUBSCRIBE, "", 0 );
    const int rate = 40000;  // Set TX- and RX- rate - b/s
    fplsocket->setsockopt(ZMQ_RATE, &rate, sizeof(rate));
    const int hwm = 0;
    fplsocket->setsockopt(ZMQ_RCVHWM, &hwm, sizeof(hwm));
    fplsocket->connect(connstr.str());

    int i=0;
    int runs=0;
    int rc;
    vector<string> objects;
    zmq::message_t rs;
    auto start = chrono::system_clock::now();
    while( 1 ) {
        if( (rc = fplsocket->recv(&rs)) == true) {
            string msgstr(static_cast<char*>(rs.data()),rs.size());
            if( msgstr.length() < 10 ) {
                if( msgstr == "START" ) {
                    i = 0;
                    start = chrono::system_clock::now();
                    objects.clear();
                } else if( msgstr == "END" ) {
                    auto end = chrono::system_clock::now();
                    chrono::duration<double, milli> send_ms = end -
                        start;
                    runs++;
                    cout << runs << " Received "
                        << i << " messages in "
                        << send_ms.count() << " ms. Size of vector
                            : "
                        << objects.size() << endl;
```

```
                }
            } else {
                i++;
                objects.push_back(msgstr);
            }
        } else
            cout << "rc = " << rc << endl;
    }
}
```

For reference, details of how to compile, link and run the example is shown in listing 12

Listing 12: Compile, Link and Run

```
# publisher
g++ -c -m64 -pipe -g -std=gnu++1y -Wall -W -D_REENTRANT -fPIC -
    DZMQ_HAVE_OPENPGM -I. -isystem /usr/include/pgm-4.2 -o
    publisher.o publisher.cc
g++ -m64 -o publisher publisher.o -lzmq -lpgm -lprotobuf -L/usr/
    local/lib
# subscriber
g++ -c -m64 -pipe -g -std=gnu++1y -Wall -W -D_REENTRANT -fPIC -
    DZMQ_HAVE_OPENPGM -I. -isystem /usr/include/pgm-4.2 -o
    subscriber.o subscriber.cc
g++ -m64 -o subscriber subscriber.o -lzmq -lpgm -lprotobuf -L/usr/
    local/lib
# Host 1
./publisher <name of network interface>
# Host 2
./subscriber <name of network interface>
```

To finalize the simple pub/sub implementation, a protocol buffers data object (proto object) with a few flight plan related fields is added. The definition of the proto object, named FPL, is listed in listing 13.

Listing 13: FPL Object Definition

```
syntax = "proto3";

message FPL {
  enum WTC {
    NOTSET = 0;
    L = 1;
    M = 2;
    H = 3;
    J = 4;
  }

  uint32 SSRCode = 1;
  string callsign = 2;
  WTC wtc = 3;
  string blob = 4;
}
```

The necessary code changes to the publisher is rather limited. Apart from including the header file for the FPL object definition and populate the vector with FPL objects rather than strings, only one line had to be changed; when looping through the the assignment of the string to be sent, a protocol buffers

assignment was used instead:
`str = *it;`
replaced with
`it->SerializeToString(&str);`

On the client side, even less had to be done; include header file, change vector definition, and instead of pushing the string directly to the vector, the received string is deserialized to a FPL proto object first, like this:

```
FPL fpl;
fpl.ParseFromString(msgstr);
objects.push_back(fpl);
```

A typical output from running one publisher and one subscriber:

```
$ ./publisher enxa44cc8f7dbd3
Initializing network: epgm://enxa44cc8f7dbd3;239.10.10.10:50000
1 Finalized full sync of 5000 flight plans in 25.7126 ms
2 Finalized full sync of 5000 flight plans in 20.5641 ms
3 Finalized full sync of 5000 flight plans in 17.3434 ms
4 Finalized full sync of 5000 flight plans in 21.6588 ms

$ ./subscriber em1
Initializing network: epgm://em1;239.10.10.10:50000
1 Received 5000 messages in 4775.59 ms. Size of vector: 5000
2 Received 5000 messages in 4791.34 ms. Size of vector: 5000
3 Received 5000 messages in 4790.02 ms. Size of vector: 5000
4 Received 5000 messages in 4788.01 ms. Size of vector: 5000
```

Section 8.5 shows results from running with different parameters with regards to number of objects, size of each object and transfer rate.
To see how protocol buffers handles evolving object definitions, certain tests were done using one publisher and two clients. All changes were done without breaking the backwards compatibility rules mentioned earlier. The following table lists different versions of the object definition, and the difference from the original version.

Version	**Difference from original**
v1.0	No change
v1.1	Added field: uint32 val1 = 5;
v1.2	Removed field: uint32 val1 = 5; Added field: string str1 = 6;

Table 6: Object Definition - Evolution

The following table shows the setup of one publisher and two subscribers and which definition version they use. During the test, all fields the subscriber knows about is printed to verify correct values.

Publisher	Subscriber 1	Subscriber 2
v1.0	v1.0	v1.1
v1.0	v1.2	v1.1
v1.1	v1.0	v1.1
v1.1	v1.2	v1.1
v1.2	v1.0	v1.1
v1.2	v1.2	v1.1

Table 7: Object Definition - Evolution Test Setup

8.4.2 Flightplan server/client

The simple flightplan distribution system builds on the pub/sub example in the previous section. The basic class structure is as shown in figure 6. The server and client shares most of the member variables and methods so a Base class holds these. Only the most important members are shown. Implementations for Base, DataServer and DataServer classes, in addition to the keepalive and flight plan data protobuf definitions can be found in Appendix C.

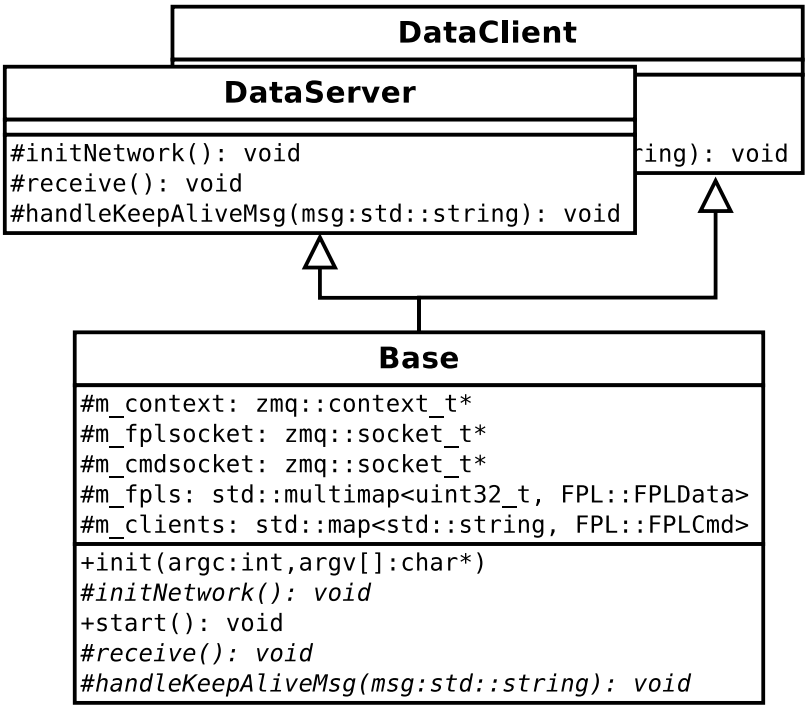

Figure 6: FPL Server/Client Data Flow

Figure 7 shows the general call stack for both server and client. The left side shows the Base class methods which is used to call the subclass implementations. The initialization phase initializes the ZeroMQ/PGM network and also some logging and configuration functionality.

The start method, listing 14, is an endless loop that does two things - send

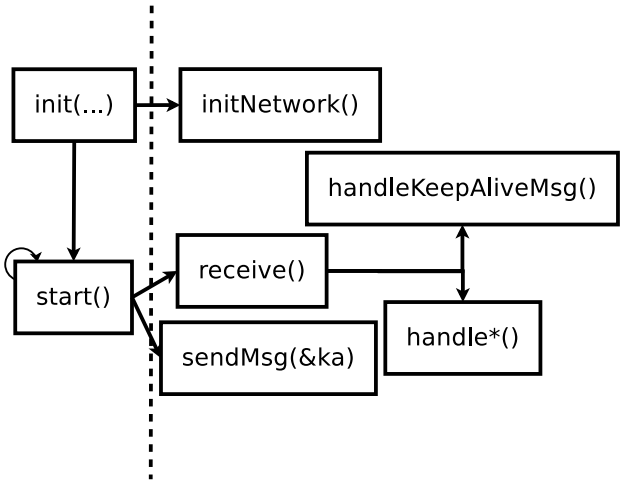

Figure 7: FPL Server/Client Call Stack

the keep alive message every second, and run the receive() method, listing 15. This will in turn run a handle*() method depending on the type of message that is received, in this case either a keep alive message or some kind of flight plan update. To keep things very simple, both receive and send run in the same thread, so the recv() call will block for 100 ms before continuing sending keep alive message and other janitor tasks.

Listing 14: start()

```
void Base::start()
{
    auto start = chrono::system_clock::now().time_since_epoch();
    while (1)
    {
        receive();

        // Every second: Send keepalive and print number of flight
            plans in map
        auto now = chrono::system_clock::now().time_since_epoch();
        auto elapsed = chrono::duration_cast<chrono::milliseconds>(
            now - start);
        if( elapsed.count() < 1000 )
            continue;

        start = chrono::system_clock::now().time_since_epoch();

        if( m_commstate != FPL::FPLCmd::Synchronizing )
        {
            zmq::message_t msg = createKeepAliveMsg();
            sendMsg(&msg);
        }
    }
}
```

Listing 15: receive()

```
void DataServer::receive()
{
    int rc;
    zmq::message_t resultset;

    try {
        if( (rc = m_cmdsocket->recv(&resultset)) == true) {
            string msg_str(static_cast<char*>(resultset.data()),
                resultset.size());
            string::size_type pos = msg_str.find('\0');
            string type = "DATA";
            if (pos != string::npos)
            {
                type = msg_str.substr(0, pos);
                msg_str = msg_str.substr(pos+1, msg_str.length());
            }
            if( type == "KA" )
                handleKeepAliveMsg(msg_str);
            else
            {
                if( m_config->value("datatype") == "fpl" )
                    addFpl(msg_str);
                else if( m_config->value("datatype") == "
                    selected_fpl" )
                    handleSelectedFlightplan(msg_str);
                else
                    BOOST_LOG_SEV(m_log, kw::fatal) << "No handler
                        for datatype "
                                                    << m_config->
                                                        value("
                                                        datatype")
                                                    << resultset.
                                                        size();
            }
        }
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            endl;
    }
}
```

As the message sent over the network is a simple std::string, a short null-terminated keyword has been prepended the protobuf serialized message to allow receiving side to correctly handle the message. This way it is easy to send keep alive and flight plan messages, or any other kind of message, down the same socket.

To replicate a true FPDS, a flight plan simulator was built into this server and client concept implementation. There are three typical actions in such a system: add a flight plan, update a flight plan and delete a flight plan. So the simulator will follow this list of events to replicate the typical actions that will take place:

- Server and client (participants) start. Order is of no importance.
- Server creates a large number of flight plans with different SSR codes and call signs
- The participants start sending keep alive messages as soon as the network is properly set up.

- Once both client and server has received keep alive messages, the connection is established
- The server initiates a transfer of all flight plans, with a start and end message prior and after
- The keep alive message will have field stating number of flight plans and a crc of all flight plans. Handling differences is up to an agreed algorithm, but this implementation only prints the difference.
- Then the following events are induced, some on the server side and some on the client side
  - Server deletes one flight plan
  - Server adds one flight plan
  - Server updates one flight plan,
  - Server deletes all flight plan's
  - Client add many flight plan's
  - Client deletes one flight plan
  - Client adds one flight plan
  - Client updates one flight plan,
  - Client deletes all flight plan's

The flight plans are held in a standard multi map so actually implementing this is straight forward. The base concepts of serializing/deserializing and sending/receiving protobuf messages follows the same pattern regardless.

8.5 Results

Test setup:
1 Dell XPS 15 laptop, 32 GiB RAM, Gb NIC, 8 core i7-7700HQ CPU @ 2.80GHz
1 Dell Latitude E6520 laptop, 8 GiB RAM, Gb NIC, 4 core i7-2620M CPU @ 2.70GHz Connected via a simple desktop gigabit switch.

The following table shows the throughput measurements when running with different parameters. Obviously, this is highly dependent on the test setup and complexity of the running processes, but will nevertheless give a good indication on what to expect.

Rate	Packet Size	Num Msg's	Send	Receive
40 Mb/s	1041 b	5000	<10 ms	1250 ms
40 Mb/s	4113 b	5000	8-15 ms	4800 ms
40 Mb/s	4113 b	10000	15-40 ms	9600 ms
60 Mb/s	1041 b	15000	10-20 ms	2400 ms
60 Mb/s	1041 b	60000	20-50 ms	9600 ms
60 Mb/s	4113 b	10000	8-30 ms	6300 ms
60 Mb/s	4113 b	15000	15-40 ms	9600 ms
1 Gb/s	1041 b	15000	5-25 ms	1400 ms
1 Gb/s	1041 b	60000	25-50 ms	5600 ms
1 Gb/s	4113 b	25000	ms	ms

Table 8: Summary: Pub/Sub Message Throughput Multicast

Rate	Packet Size	Num Msg's	Send	Receive
40 Mb/s	4113 b	10000	15-40 ms	1150 ms
60 Mb/s	1041 b	200000	200 ms	4300 ms
1 Gb/s	4113 b	20000	30-60 ms	50-90 ms

Table 9: Summary: Pub/Sub Message Throughput IPC

As predicted, transport over IPC is much faster, so finding the best parameters should be done based on the ordinary network setup.
Even though CPU usage was not measured explicitly, it was observed during the tests. During the most intense sending sessions, the publisher process peaked around 50%. The subscriber around 15-20%.
Testing for backwards compatibility was simply done by running publishers/subscribers compiled with different object definitions and see how gracefully they handled missing/unknown data. To see if the data was deserialized correctly, the built-in DebugString() method in protobuf was used to print all fields with corresponding value. A typical printout of version 1.2 of the object definition looks like this:

```
SSRCode: 1761
callsign: "NA1761"
wtc: L
blob: "##########"
str1: "PUBSUB"
```

The results from running all different versions are listed here:

Pub	Sub 1	Sub 2	Result
v1.0	v1.0	v1.1	Both subscribers printed correct information. New val1 not printed by any.
v1.0	v1.2	v1.1	Both subscribers printed correct information. New val1 or str1 not printed by any.
v1.1	v1.0	v1.1	Subscriber 2 printed correct information, including val1. Subscriber 1 also printed the val1 data, but as the definition did not include field 5, it only printed the field tag value and field value: 5: 99
v1.1	v1.2	v1.1	Subscriber 2 printed correct information, including val1. Subscriber 1 also printed the val1 data, but as the definition did not include field 5, it only printed the field tag value and field value: 5: 99
v1.2	v1.0	v1.1	Subscriber 2 printed only values from the initial definition. ubscriber 1 also printed the val1 data, but as the definition did not include field 5, it only printed the field tag value and field value: 6: "PUBSSUB"
v1.2	v1.2	v1.1	Subscriber 1 printed correct information, including str1. Subscriber 2 printed only values from the initial definition.

Table 10: Object Definition - Evolution Test Results

As advertised, deserializing worked flawlessly regardless of version of object definition and whether publisher or subscriber used the new version. The only thing worth noting is that when running publisher with version 1.2 of the definition - one field removed another field added - the v1.1 subscriber did not print the new field with field tag only, i.e. 6: “PUBSUB”. For other combinations it did print unknown fields with field tag value only, . Even though this seems to be semi random, it is according to documentation:

```
Proto3 implementations can parse messages with unknown fields successfully,
however, implementations may or may not support preserving those unknown
fields.
```

The final discussion point was the subjective feeling of how it is to work with this technology. It is hard to beat the simplicity of both setup and use of both ZeroMQ and Protocol Buffers. With very few lines of code everything is ready and one can start to build the infrastructure on top of the base building blocks. ZeroMQ gives the feeling of keeping out of the way. You only need to pay attention when initially setting it up. Protocol Buffers is the part that needs to be dealt with on a daily basis. It has a clear and concise object definition language, numerous different field types, creates necessary methods for working with the objects automatically and supports backwards compatibility. For the kind of software discussed in this paper, this is a good software stack to build a distributed software on top.

8.6 Summary

Even though the server and client only implements a subset of all functionality needed for a flight plan distribution system, the base functions are there and all the issues with the current systems seems to be met.

9 Real World Application

9.1 Introduction

The Tower ATC software is a large system with lots of data objects moving between the different servers and clients. The object types vary a lot in size and complexity, from the flight plan objects with hundreds of variables to very simple ones, such as the Selected Flight Plan object type. Its sole purpose is to synchronize two or more paired workstations with which flight plan the operator is working on. This is a simple and well understood object type, and will be used as the basis for the OpenPGM/ZeroMQ/Protocol Buffers integration.

9.2 Goals

The purpose of this implementation is to see what it takes to integrate the chosen replacement technology into the Tower ATC software.

9.3 Design Overview

The object definition, named DOSelectedFlightPlan, needed is very simple:

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";

message DOSelectedFlightPlan {
  map<string, string> selfpl = 1;

  google.protobuf.Timestamp timestamp = 100;
  bool processed = 101;
}
```

The *selfpl* field is the important one here; it holds a map of which working position has selected which flight plan. The *timestamp* field is used to prevent old messages from being processed, and the *processed* field is a flag used to signal the underlying software that updates are available. An interesting feature of protobuf is the different data types it supports. In this object definition the field value is a map which provides the same methods as the std::map data type. This makes it trivial to operate on the value, and allows programmers to quickly start using protobuf features.

Figure 8 shows the data flow when an operator selects a flight plan until it is reported to all clients and the HMI is correctly updated. The figure will be further discussed in the next section.

9.4 Implementation Details

The current code base is highly complicated with several layers of inheritance, signal mechanisms between different parts of the code and generally a lot of code. Finding the best places to inject the new code was thus rather challenging. Nevertheless, in the end only a few lines of code had to be changed in the existing code to accommodate the new system. The *SFManager* instance shown in figure 8 implements everything related to PGM/ZeroMQ/Protocol Buffers and the existing code only needed to have access to three methods to

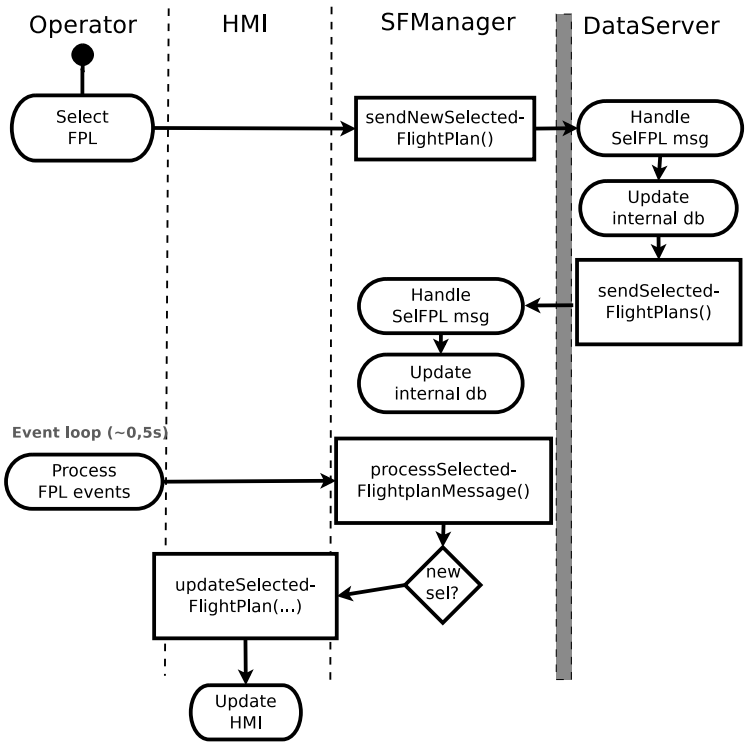

Figure 8: Selected Flight Plan Event Diagram

get the job done. Figure 8 only shows two of them as the last one is used to signal missing connection to data server and is not related to the core functionality.
The full implementation of SFManager is listed in Appendix D, but the important parts are the initNetwork() method, and the send() and receive() threads. Network is set up as described in the previous section, and serializing/deserializing is also done the same way so the rest of the code is just boilerplate to handle the messages and interconnect with the rest of the software.
It turned out that reusing the dataServer created for FPDS was possible without much integration effort needed. Adding support for the new Selected FPL object type was a matter of adding a handler for the specific message and a send method to return the values to all clients. Here's a listing of the two methods:

```
void DataServer::handleSelectedFlightplan(std::string msg_str)
{
    m_selFplPB.ParseFromString(msg_str);
    sendSelectedFlightPlans();
}

void DataServer::sendSelectedFlightPlans()
{
    std::string protomsg;
    m_selFplPB.SerializeToString(&protomsg);
    zmq::message_t fplmsg(protomsg.length());
```

```
    memcpy(fplmsg.data(), protomsg.data(), protomsg.length());
    sendMsg(&fplmsg);
}
```

As figure 8 states, the chain of events is started when the operator selects a flight plan - visually shown in figure 9.

SFManager updates its internal DOSelectedFlightPlan object and transfers

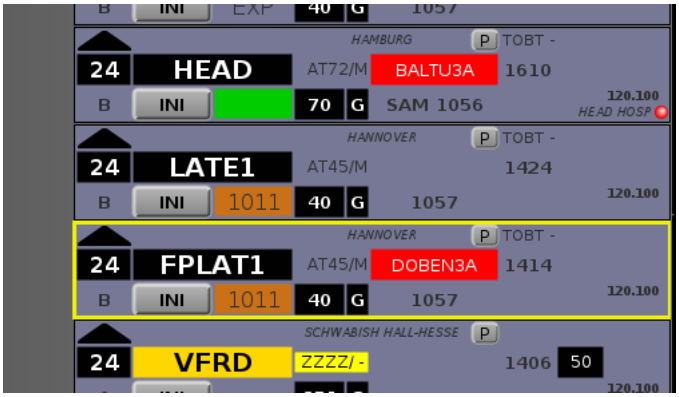

Figure 9: Selected Flight Plan

the whole object to the DataServer. It will, as shown in the listing above, save the object contents to a member variable of DOSelectedFlightPlan type and send it to all clients. The SFManager will, on all working positions, receive the updated object, update the internal data object and set the processed flag false. Whenever a new client connects, the same message will also be sent.
An event loop is running a couple of times a second and will process all events, including the selected flight plan. If the processed flag is false, it will fetch the selected flight plan value for its own workstation id. If it is different from its current value, a signal is sent to the HMI which will update the visual characteristics of a selected flight plan.

9.5 Results

It was, without much effort, possible to integrate it into the current software.

9.6 Summary

Apart from the challenges related to understanding where in the original code the integration should take place, the actual implementation was very straight forward. Building on the work done with the FPDS in the previous section, it was merely a matter of finding the correct protobuf object types to use, and reuse most of the logics from the FPDS. The fact that reusing the DataServer from the FPDS implementation was possible with only minor modifications helped tremendously.

10 Summary

Both the simple flight plan distribution system and the replacement done in the actual ATC software shows that the chosen technology is a very good match for this kind of usage. It should be possible to gradually replace the current object types over time and minimize any risks that may not currently be known.

Part V
Conclusion

11 Conclusion

The problem analysis found that the Tower OM implementation had, in short, these issues:

Layer	Problem	Result
RMC	Network problems	Lots of resends may result in system halt.
	No one knows the implementation details	Difficult to make changes, do not fully understand the effect
OM	Unrelated data types are coupled	The flight plan sub system must handle unrelated data which adds complexity both in code and server configuration.
Object contents	Binary format	Not backwards compatible
	Limited number of types	Available data types are limited and not very flexible in use

Table 11: Problems With Current Solution

The problem solution analysis found a wide specter of possibilities, ranging from libraries sought out to solve one part and thus needed to be chained with others to make a full solution to large broker systems suitable for stock exchanges and others with corresponding requirements. Given the nature of the Tower ATC software it was clear that the possible solutions lay in the less extensive end of the spectrum.

There were two solutions that stood out as good replacements for Tower OM. The first being a combination of OpenPGM for reliable multicast, ZeroMQ for distribution and Google Protocol Buffers for object contents. The second is DDS which is a complete solution for reliable object management. As these have slightly different approaches, the following two sections will discuss them separately with regards to the issues found.

11.1 OpenPGM/ZeroMQ/Protocol Buffers

OpenPGM is an open source implementation of Pragmatic General Multicast, based on RFC 3208. A possible alternative to this is NORM, an open source implementation of NACK-Oriented Reliable Multicast, based on RFC 5740. ZeroMQ is an open source socket library that has bindings to both OpenPGM and NORM. It takes care of all the low level socket handling and also introduces different patterns that can be used depending on the use case. It easily allows mixing TCP, multicast and other distribution mechanisms and supports numerous languages. As one of the issues with Tower OM was handling rogue network or misbehaving clients, it is expected that using software that is based on a thoroughly researched and designed solution will help in that matter.

There is an abundance of ways to structure and serialize/deserialize object data. As one of the issues set out to be solved was problems with backwards compatibility, Google's Protocol Buffers was a great contender. It was designed specifically to provide backwards compatibility as well as being small and compact. By following a few simple guidelines it is possible to retain interoperability between components using different versions of the object definition.

11.2 DDS

DDS, Data Distribution Service, is an open standard with several implementations, both open source and proprietary. The most notable open source implementation is OpenDDS, whereas there are several proprietary alternatives. The standard has gained a lot of grounds the last few years, and is becoming increasingly popular within air traffic control. It is slightly different from the other solution as it is based on publishing and subscribing to topics. The underlying network details are abstracted from the developers point of view.

Communication can be done using different transports, but multicast is one of the major components. The standard describes some details for how to handle reliability but the different implementations may handle this slightly different.

Backwards compatibility is solved by using the DDS standard defined as 'Extensible and dynamic topic types for DDS' - DDS-XTypes. Unfortunately, OpenDDS does not currently support this feature, so only the proprietary implementations of DDS that support XTypes would be useful for Tower OM.

11.3 Final words

In essence, a reliable object management solution based on OpenPGM, ZeroMQ and Google Protocol Buffers is very potent and gives scalability, simplicity and performance into the hands of the developers.

Even though it is a good replacement for Tower OM, any software in need for a good RMC/OM solution has to study its own use cases and requirements. The field is very broad, and the available solutions are very diverse.

12 Future Work

The most important next step would be to create a similar implementation based on DDS and compare it with the prototype. Both should be extended with more tests regarding network throughput, rogue network and clients and stress tests of all kinds.

Second would be to create recording and playback systems for both solutions. It is a vital part of the Tower ATC software, and failing to provide the necessary functionality would basically make it useless.

Part VI
Appendix

A Appendix A - RFC's

Bulk data transfer/File transfer

RFC2887
The Reliable Multicast Design Space for Bulk Data Transfer

RFC3048
Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer

RFC3926
FLUTE - File Delivery over Unidirectional Transport

PGM

RFC3208
PGM Reliable Transport Protocol Specification

SRMP

RFC4410
4410 - Selectively Reliable Multicast Protocol (SRMP)

General

RFC1458
Requirements for Multicast Protocols

RFC3170
IP Multicast Applications: Challenges and Solutions

RFC3269
Author Guidelines for Reliable Multicast Transport Building Blocks and Protocol Instantiation documents

RFC3450
Asynchronous Layered Coding (ALC) Protocol Instantiation

RFC3451
Layered Coding Transport (LCT) Building Block

RFC3738
Wave and Equation Based Rate Control (WEBRC) Building Block

RFC4654
TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification

RFC5651
Layered Coding Transport (LCT) Building Block

RFC5775
Asynchronous Layered Coding (ALC) Protocol Instantiation

NORM

RFC6584
Simple Authentication Schemes for the Asynchronous Layered Coding (ALC) and NACK-Oriented Reliable Multicast (NORM) Protocols

RFC5401
Multicast Negative-Acknowledgment (NACK) Building Blocks obsoletes

RFC3941
Negative-Acknowledgment (NACK)-Oriented Reliable Multicast (NORM) Building Blocks

RFC5740
NACK-Oriented Reliable Multicast (NORM) Transport Protocol

RFC3940
Negative-acknowledgment (NACK)-Oriented Reliable Multicast (NORM) Protocol

Forward Error correction

RFC3452
Forward Error Correction (FEC) Building Block

RFC3453
The Use of Forward Error Correction (FEC) in Reliable Multicast

RFC3695
Compact Forward Error Correction (FEC) Schemes

RFC5052
Forward Error Correction (FEC) Building Block

RFC5053
Raptor Forward Error Correction Scheme for Object Delivery

RFC5170
Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes

RFC5445
Basic Forward Error Correction (FEC) Schemes

RFC5510
Reed-Solomon Forward Error Correction (FEC) Schemes

RFC6330
RaptorQ Forward Error Correction Scheme for Object Delivery

B Appendix B - Object Contents - Programming Examples

B.1 XML

B.1.1 Schema

Listing 16: XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="FPL" type="FPLType"/>

  <xsd:complexType name="FPLType">
    <xsd:sequence>
      <xsd:element name="SSRCode"  type="xsd:decimal"/>
      <xsd:element name="callsign">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value="7"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="wtc"       type="WTCenum"/>
      <xsd:element name="route"     type="RouteLegs"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="WTCenum">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="L" />
      <xsd:enumeration value="M" />
      <xsd:enumeration value="H" />
      <xsd:enumeration value="J" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="RouteLegs" minOccurs="0" maxOccurs="1">
    <xsd:sequence>
      <xsd:element name="leg" minOccurs="1" maxOccurs="unbounded">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxlength value="100"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

B.1.2 Data File

Listing 17: XML Data

```
<?xml version="1.0"?>
<FPL>
 <SSRCode>5523</SSRCode>
 <callsign>JLJ3105</callsign>
```

```
 <wtc>J</wtc>
 <route>
  <leg>LEG1</leg>
  <leg>LEG2</leg>
  <leg>LEG3</leg>
 </route>
</FPL>
```

B.1.3 Source Code

Listing 18: XML Source Code Example

```
#!/usr/bin/env python

from __future__ import print_function
import FPL
import pyxb

# Read existing FPL
xml = open('FPL.xml').read()
try:
    fpl = FPL.CreateFromDocument(xml)
except pyxb.UnrecognizedContentError as e:
    print('*** ERROR validating response:')
    print('Unrecognized element "%s" at ' % (e))
except Exception as e:
    print('*** Parse error')
    print(e)
    exit()

if fpl is None:
    pyxb.RequireValidWhenParsing(False)
    fpl = po1.CreateFromDocument(rxml)

print('Flightplan details:')
print('\tSSR Code: %d\n\tCallsign: %s\n\tWTC: %s' %
(fpl.SSRCode, fpl.callsign, fpl.wtc))
if fpl.route is not None:
    print('\tRoute: ')
    for leg in fpl.route.leg:
        print('\t\tLeg: %s' % leg)
else:
    print('\tRoute: <Not defined>')

# Create new FPL
newfpl = FPL.FPL()
newfpl.SSRCode=2323
newfpl.callsign="KAB2304"
newfpl.wtc="L"

route=FPL.RouteLegs()
route.leg=[ "LEG3", "LEG2", "LEG1" ]

newfpl.route = route

print('Flightplan details:')
print('\tSSR Code: %d\n\tCallsign: %s\n\tWTC: %s' %
(newfpl.SSRCode, newfpl.callsign, newfpl.wtc))
if newfpl.route is not None:
    print('\tRoute: ')
```

```
    for leg in newfpl.route.leg:
        print('\t\tLeg: %s' % leg)
else:
    print('\tRoute: <Not defined>')
f=open('newfpl.xml', 'w')
f.write(newfpl.toxml("utf-8"))
```

B.2 JSON

B.2.1 Schema

Listing 19: JSON Schema

```
{
    "title": "FPL Schema",
    "type": "object",
    "properties": {
        "SSRCode":  {
                        "type": "integer"
        },
        "callsign": {
                        "type": "string",
                        "minLength": 7,
                        "maxLength": 7
        },
        "wtc":      {
                        "enum": [ "L", "M", "H", "J" ]
        },
        "route":    {
                        "type": "array",
                        "items": {
                                  "type:": "string"
                        }
        }
    }
}
```

B.2.2 Data File

Listing 20: JSON Data

```
{
    "SSRCode": 5523,
    "callsign": "JLJ3105",
    "wtc": "J",
    "route": ["LEG1", "LEG2", "LEG3"]
}
```

B.2.3 Source Code

Listing 21: JSON Source Code Example

```
#!/usr/bin/env python
import json
import sys
from jsonschema import validate
```

```
with open('FPL.schema.json', 'r') as json_schema:
    schema = json.loads(json_schema.read())
json_schema.close()

with open('FPL.json', 'r') as json_data:
    fpl = json.load(json_data)
json_data.close()

# Print results
print('Flightplan details:')
print('\tSSR Code: %d\n\tCallsign: %s\n\tWTC: %s' % (fpl["SSRCode"
    ], fpl["callsign"], fpl["wtc"]))
if "route" in fpl:
    for leg in fpl["route"]:
        print('\t\tLeg: %s' % leg)
else:
    print('\tRoute: <Not defined>')
# Simple validation
try:
    validate(fpl, schema)
except:
    print("Some error with JSON data")

# Create new fpl
newfpl = {}
newfpl['SSRCode']=2323
newfpl['callsign']="KAB2304"
newfpl['wtc']="M"
newfpl['route']=["LEG3", "LEG2", "LEG1"]
print('Flightplan details:')
print('\tSSR Code: %d\n\tCallsign: %s\n\tWTC: %s' % (newfpl["
    SSRCode"], newfpl["callsign"], newfpl["wtc"]))
if "route" in newfpl:
    for leg in newfpl["route"]:
        print('\t\tLeg: %s' % leg)
else:
    print('\tRoute: <Not defined>')
try:
    validate(newfpl, schema)
except:
    print("Some error with JSON data")
with open('newfpl.json', 'w') as json_fpl:
    json_fpl.write(json.dumps(newfpl, indent=4))
json_fpl.close()
```

B.3 Apache Thrift

B.3.1 Schema

Listing 22: Thrift Schema

```
enum WTC {
     L,
     M,
     H,
     J
}

struct FPL {
```

```
    1: i32 SSRCode;
    2: optional string callsign;
    3: optional WTC wtc;
    4: list<string> route;
}
```

B.3.2 Data File

Listing 23: Thrift Data File (json)

B.3.3 Source Code

Listing 24: Thrift Source Code Example

```
#!/usr/bin/env python

import sys
sys.path.append('gen-py')

import FPL
from FPL.ttypes import *
from thrift.protocol import TJSONProtocol

# Create and write FPL to file and read it back
fpl = FPL()
fpl.SSRCode = 5523
fpl.callsign = "JLJ3105"
fpl.wtc = WTC.J
fpl.route = []
fpl.route.append("LEG1")
fpl.route.append("LEG2")
fpl.route.append("LEG3")

#f = open("FPL.thrift.data.binary", "wb")
f = open("FPL.thrift.data.json", "wb")
transportOut = TTransport.TFileObjectTransport(f)
#protocolOut = TBinaryProtocol.TBinaryProtocol(transportOut)
protocolOut = TJSONProtocol.TJSONProtocol(transportOut)
fpl.write(protocolOut)
f.close()

#f2 = open("FPL.thrift.data.binary", "r")
f2 = open("FPL.thrift.data.json", "r")
transportIn = TTransport.TFileObjectTransport(f2)
#protocolIn = TBinaryProtocol.TBinaryProtocol(transportIn)
protocolIn = TJSONProtocol.TJSONProtocol(transportIn)
afpl = FPL()
afpl.read(protocolIn)
print afpl

newfpl = FPL()
newfpl.SSRCode = 2323
newfpl.callsign = "KAB2304"
newfpl.wtc = WTC.M
newfpl.route = ["LEG3", "LEG2", "LEG1"]
print newfpl
```

B.4 AVRO

B.4.1 Schema

Listing 25: AVRO Schema

```
{
 "type": "record",
 "name": "FPL",
 "fields": [
     {"name": "SSRCode", "type": "int"},
     {"name": "callsign",
      "type": {"type": "string", "minLength":7, "maxLength":7}
     },
     {"name": "wtc",
      "type": {"type": "enum", "name": "WakeTurbulenceCategories",
               "symbols": ["L", "M", "H", "J"]}
     },
     {"name": "route", "type": {"type": "array", "items": "string"}
         }
 ]
}
```

B.4.2 Data File

N/A - some binary data

B.4.3 Source Code

Listing 26: AVRO Source Code Example

```
#!/usr/bin/env python
from avro import schema, datafile, io, protocol
import json

fplschema = schema.parse(open("FPL.avsc").read())

# Create and write FPL to disc
data = {}
data['SSRCode']    = 5523
data['callsign']     = 'JLJ3105'
data['wtc'] = 'J'
data['route']   = ["LEG1", "LEG2", "LEG3"]
rec_writer = io.DatumWriter(fplschema)
df_writer = datafile.DataFileWriter(
                    open("FPL.avro", 'wb'),
                    rec_writer,
                    writers_schema = fplschema,
                    codec = 'null')
df_writer.append(data)
df_writer.close()

# Read FPL
rec_reader = io.DatumReader()
df_reader = datafile.DataFileReader(
    open("FPL.avro"),
    rec_reader
    )
```

```
# Read all records stored inside
for fpl in df_reader:
    print('Flightplan details:')
    print('\tSSR Code: %d\n\tCallsign: %s\n\tWTC: %s' % (fpl["
        SSRCode"], fpl["callsign"], fpl["wtc"]))
    if "route" in fpl:
        print('\tRoute:')
        for leg in fpl["route"]:
            print('\t\tLeg: %s' % leg)
    else:
        print('\tRoute: <Not defined>')
```

B.5 Protocol Buffers

B.5.1 Schema

Listing 27: Protocol Buffers Schema

```
syntax = "proto3";

package FPL;

message FPL {
  enum WTC {
    NOTSET = 0;
    L = 1;
    M = 2;
    H = 3;
    J = 4;
  }

  uint32 SSRCode = 1;
  string callsign = 2;
  WTC wtc = 3;
  repeated string route = 4;
}
```

B.5.2 Data File

N/A - binary only

B.5.3 Source Code

Listing 28: Protocol Buffers Source Code Example

```
#!/usr/bin/env python
import FPL_pb2

# Read fpl
fpl = FPL_pb2.FPL()
f = open("FPL.proto.data", "rb")
fpl.ParseFromString(f.read())
print fpl

# Create new fpl
newfpl = FPL_pb2.FPL()
newfpl.SSRCode = 2323
```

```
newfpl.callsign = "KAB2304"
newfpl.wtc = FPL_pb2.FPL.M
newfpl.route.append("LEG3")
newfpl.route.append("LEG2")
newfpl.route.append("LEG1")
print newfpl
f = open("newfpl.proto.data", "wb")
f.write(newfpl.SerializeToString())
```

B.6 DDS IDL

B.6.1 Schema

Listing 29: DDS IDL/Schema

```
enum WTC { L, M, H, J };

struct FPL {
    long SSRCode;
    string callsign;
    WTC wtc;
    sequence<string> route;
}
```

B.6.2 Data File

N/A - No example created

B.6.3 Source Code

N/A - No python implementation

C Appendix C - Simple Flight Plan Server/Client

C.1 Base Class

Listing 30: Base Class Implementation

```
#define BOOST_LOG_DYN_LINK 1

#include <iostream>
#include <unistd.h>
#include <regex>
#include <random>
#include <algorithm>

#include <google/protobuf/text_format.h>

#include <boost/filesystem.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/crc.hpp>

#include "Base.h"

std::string to_oct(int to_convert);
std::string string_to_hex(const std::string& input);

Base::Base()
{
}

Base::~Base()
{
    delete m_context;
    delete m_cmdsocket;
    delete m_fplsocket;

    delete m_config;
}

// Do all initializing stuff here to make sure
// overriden methods are called, and not the empty
// Base method.
void Base::init(int argc, char *argv[])
{
    GOOGLE_PROTOBUF_VERIFY_VERSION;
    m_config = new Config(argc, argv);

    m_calc_checksum = true;
    m_simstage = FPL::FPLCmd::TransferManyFplsFromServer;

    m_uuid = boost::uuids::random_generator()();
    unsigned char uuid_clear[16];
    memcpy(&m_serveruuid, uuid_clear, 16);

    initLog();
    BOOST_LOG_SEV(m_log, kw::info) << "My UUID: " << m_uuid;
    BOOST_LOG_SEV(m_log, kw::info) << *m_config;

    setState(FPL::FPLCmd::Initial);

    initNetwork();
```

```
}

void Base::setState(FPL::FPLCmd::CommState newstate)
{
    m_commstate = newstate;

    const google::protobuf::EnumDescriptor *descriptor = FPL::
        FPLCmd::CommState_descriptor();
    BOOST_LOG_SEV(m_log, kw::info) << "Setting new state to "
                                   << descriptor->FindValueByNumber
                                       (newstate)->name();
}

void Base::initLog()
{
    boost::log::register_simple_formatter_factory< boost::log::
        trivial::severity_level, char>("Severity");
    boost::log::add_file_log
    (
     boost::log::keywords::auto_flush = true,
     boost::log::keywords::file_name = m_config->value("processname
         ")+".log",
     boost::log::keywords::rotation_size = 10 * 1024 * 1024,
     boost::log::keywords::format = "[%TimeStamp%][%Severity%]: %
         Message%"
    );

    boost::log::add_console_log(
     std::cout,
     boost::log::keywords::filter = kw::severity >= kw::info,
     boost::log::keywords::format = "[%Severity%] %Message%"
    );

    boost::log::core::get()->set_filter
    (
        boost::log::trivial::severity >= kw::trace
    );

    boost::log::add_common_attributes();

    // BOOST_LOG_SEV(m_log, kw::trace) << "A trace severity message
        ";
    // BOOST_LOG_SEV(m_log, kw::debug) << "A debug severity message
        ";
    // BOOST_LOG_SEV(m_log, kw::info) << "An informational severity
         message";
    // BOOST_LOG_SEV(m_log, kw::warning) << "A warning severity
        message";
    // BOOST_LOG_SEV(m_log, kw::error) << "An error severity
        message";
    // BOOST_LOG_SEV(m_log, kw::fatal) << "A fatal severity message
        ";
}

void Base::printMessageContents(FPL::FPLData fpl)
{
    const google::protobuf::Descriptor *desc       = fpl.
        GetDescriptor();
    // const google::protobuf::Reflection *refl       = fpl.
        GetReflection();
    int fieldCount= desc->field_count();
```

```
    fprintf(stderr, "The fullname of the message is %s \n", desc->
        full_name().c_str());
    for(int i=0;i<fieldCount;i++)
    {
        const google::protobuf::FieldDescriptor *field = desc->
            field(i);
        fprintf(stderr, "The name of the %i th element is %s and
            the type is  %s \n",i,field->name().c_str(),field->
            type_name());
      }
}

// Ikke i bruk
uint32_t Base::calccrc(FPL::FPLData fpl)
{
    const ::google::protobuf::Descriptor *d    = fpl.GetDescriptor
        ();
    const ::google::protobuf::Reflection *refl = fpl.GetReflection
        ();
    std::cout << d->name() << std::endl;
    for (int j = 0; j < d->field_count(); j++) {
        const ::google::protobuf::FieldDescriptor *f = d->field(j);
        if( f->name() == "crc" )
            continue;

        std::cout << "Field: " << f->name() << "  " << f->label()
            << "  " << f->type() << std::endl;
        switch(f->type())
        {
        case 13: std::cout << refl->GetString(fpl, f) << std::endl;
             break;
        default: std::cout << "Unknown" << std::endl;

        }
        // if( !f->is_repeated() )
        // {
        //      std::string g = refl->GetString(fpl, f);
        //      std::cout << g << std::endl;
        // }
    }

    std::string  data1, data2;
    boost::crc_32_type  crc;

    data1 = "This is a test string";
    data2 = "test 2";
    crc.process_bytes(data1.c_str(), data1.length());
    crc.process_bytes(data2.c_str(), data2.length());

    return crc.checksum();
}

void Base::printFpls()
{
    for( m_fpliter it = m_fpls.begin() ; it != m_fpls.end() ; it++
        )
    {
        BOOST_LOG_SEV(m_log, kw::trace) << "Flightplan" << std::
            endl << it->second.DebugString();
    }
}
```

```
void Base::createTestFpls()
{
    std::cout << "Creating flight plans" << std::endl;

    std::random_device rd;
    std::mt19937 rnd(rd());
    int len=4096;

    // int numfpls = atoi(m_config->value("general.num_test_fpls").
        c_str());
    int numfpls = atoi(m_config->value("general.num_test_fpls").
        c_str());
    // std::cout << "numtfpl" << numtfpl << std::endl;
    // int numfpls = 20;

    auto start = std::chrono::system_clock::now();
    for( auto id=1 ; id <= numfpls ; id++ )
    {
        FPL::FPLData fpl;

        std::string octid=to_oct(id+1000);

        fpl.set_ssrcode(atoi(octid.c_str()));
        fpl.set_callsign("NA"+octid);
        fpl.set_wtc(FPL::FPLData_WTC_L);

        // Random binary blob
        std::vector<bool> bits(len);
        for (int i = 0; i < len/2; i++)
            bits.push_back(1);
        std::shuffle(bits.begin(), bits.end(), rnd);
        std::string bitsstr(bits.begin(), bits.end());
        fpl.set_blob(bitsstr);

        addFpl(fpl);
    }
    auto end = std::chrono::system_clock::now();
    auto elapsedsec = std::chrono::duration_cast<std::chrono::
        seconds>(end - start);
    auto elapsedms = std::chrono::duration_cast<std::chrono::
        milliseconds>(end - start);

    BOOST_LOG_SEV(m_log, kw::trace) << "Creating " << numfpls << "
        flightplans took " << elapsedsec.count() << " s " <<
        elapsedms.count() << " ms" << std::endl;
    BOOST_LOG_SEV(m_log, kw::trace) << "After initial inserts";
    // printFpls();

    FPL::FPLData fpl;
    fpl.set_ssrcode(1754);
    fpl.set_callsign("NA1754");
    fpl.set_wtc(FPL::FPLData_WTC_J);
    BOOST_LOG_SEV(m_log, kw::trace) << "After update";
    updateFpl(fpl);

    // printFpls();

    BOOST_LOG_SEV(m_log, kw::trace) << "After updating exactly the
        same";
    updateFpl(fpl);

    BOOST_LOG_SEV(m_log, kw::trace) << "Testing incorrect ssrcode";
```

```
    fpl.set_ssrcode(1238);
    updateFpl(fpl);
}

void Base::sendFpl(FPL::FPLData fpl)
{
    std::string protomsg;
    fpl.SerializeToString(&protomsg);
    int msgsize = protomsg.length();

    zmq::message_t fplmsg(msgsize);
    memcpy(fplmsg.data(), protomsg.data(), msgsize);
    BOOST_LOG_SEV(m_log, kw::info) << "Sending flight plan with
        size: "
                                   << fplmsg.size()
                                   << " SSR Code: " << fpl.ssrcode
                                       ()
                                   << " Callsign: " << fpl.callsign
                                       ();
    sendMsg(&fplmsg);
}

void Base::sendMsg(std::string msg)
{
    zmq::message_t zmsg(msg.length());
    memcpy(zmsg.data(), msg.data(), msg.length());
    sendMsg(&zmsg);
}

bool Base::fplExists(FPL::FPLData fpl)
{
  std::pair<m_fpliter, m_fpliter> fpls = m_fpls.equal_range(fpl.
      ssrcode());
    for( m_fpliter it = fpls.first ; it != fpls.second ; it++ )
    {
        if( it->second.ssrcode() == fpl.ssrcode() and
            it->second.callsign() == fpl.callsign() )
            return true;
    }
    return false;
}

bool Base::isValid(FPL::FPLData fpl)
{
    // Very simple SSR code regex check
    std::regex ssr_regex("^[1-7][0-7][0-7][0-7]$");
    if( !std::regex_match(std::to_string(fpl.ssrcode()), ssr_regex)
        )
        return false;

    // Check valid callsign
    //todo
    return true;
}

void Base::rmFpl(std::string msg_str)
{
    uint32_t ssrcode = 9999;
    std::string callsign;

    std::string::size_type pos = msg_str.find('\0');
    if (pos != std::string::npos)
```

```
    {
        ssrcode = atoi(msg_str.substr(0, pos).c_str());
        callsign = msg_str.substr(pos+1, msg_str.length());
    }

    if( ssrcode == 9999 )
    {
        BOOST_LOG_SEV(m_log, kw::trace) << "Unable to decode delete
             string - " << msg_str;
        return;
    }

    deleteFpl(ssrcode, callsign, false);
}

bool Base::deleteFpl(uint32_t ssrcode, std::string callsign, bool
    notify, bool eraseFpl)
{
  std::pair<m_fpliter, m_fpliter> fpls = m_fpls.equal_range(ssrcode
      );
    for( m_fpliter it = fpls.first ; it != fpls.second ; it++ )
    {
        if( it->second.ssrcode() == ssrcode and
            it->second.callsign() == callsign )
        {
            if( eraseFpl )
            {
                BOOST_LOG_SEV(m_log, kw::info) << "Removed flight
                    plan " << ssrcode << ", " << callsign;
                m_fpls.erase(it);
            }

            if( not notify )
                return true;

            std::string msg = "DELETE";
            msg += '\0';
            msg += std::to_string(ssrcode);
            msg += '\0';
            msg += callsign;
            sendMsg(msg);
            BOOST_LOG_SEV(m_log, kw::info) << "Notify delete of
                flight plan " << ssrcode << ", " << callsign;

            return true;
        }
    }

    BOOST_LOG_SEV(m_log, kw::info) << "Unable to find and delete
        fpl: " << ssrcode << ", " << callsign;
    return false;
}

void Base::updateFpl(std::string msg_str)
{
    FPL::FPLData newfpl;
    newfpl.ParseFromString(msg_str);
    updateFpl(newfpl);
}

void Base::updateFpl(FPL::FPLData fpl)
{
```

```
    BOOST_LOG_SEV(m_log, kw::info) << "Updating flight plan (" <<
        fpl.ssrcode() << ", " << fpl.callsign() << ")";

    if( not isValid(fpl) )
    {
        BOOST_LOG_SEV(m_log, kw::error) << "Trying to update flight
             plan with invalid SSR code";
        BOOST_LOG_SEV(m_log, kw::trace) << "Flightplan: " << std::
            endl << fpl.DebugString();

        return;
    }

    for( m_fpliter it = m_fpls.begin() ; it != m_fpls.end() ; it++
        )
    {
        if( it->second.ssrcode() != fpl.ssrcode() )
            continue;

        if( it->second.callsign() != fpl.callsign() )
            continue;

        m_fpls.erase(it);
        break;
    }
    m_fpls.insert(std::make_pair(fpl.ssrcode(),fpl));
    m_calc_checksum = true;


    // Update instance....
    // std::map<std::string, FPL::FPLCmd>::iterator it = m_clients.
        find(ka.uuid().value());
    // if (it != m_clients.end())
    //     it->second.set_synchronizing(true);

}

void Base::addFpl(std::string msg_str)
{
    FPL::FPLData newfpl;
    newfpl.ParseFromString(msg_str);
    addFpl(newfpl);
}

void Base::addFpl(FPL::FPLData fpl)
{
    // BOOST_LOG_SEV(m_log, kw::trace) << "Adding flight plan (" <<
         fpl.ssrcode() << ", " << fpl.callsign() << ")";
    if( not isValid(fpl) )
    {
        BOOST_LOG_SEV(m_log, kw::error) << "Trying to add flight
            plan with invalid SSR code";
        BOOST_LOG_SEV(m_log, kw::trace) << "Flightplan: " << std::
            endl << fpl.DebugString();

        return;
    }

    // Check if already exists
    if( fplExists(fpl) )
    {
```

```
        BOOST_LOG_SEV(m_log, kw::error) << "Trying to add flight
            plan that already exists";
        BOOST_LOG_SEV(m_log, kw::trace) << "Flightplan: " << std::
            endl << fpl.DebugString();

        return;
    }

    m_fpls.insert( std::make_pair(fpl.ssrcode(),fpl) );

    m_calc_checksum = true;
}

zmq::message_t Base::createKeepAliveMsg()
{
    std::string msg = "KA";
    msg += '\0';

    FPL::FPLCmd      kamsg;

    // Static keepalive data
    // UUID
    UUID uuid;
    uuid.set_value(boost::lexical_cast<std::string>(m_uuid));
    kamsg.set_allocated_uuid(&uuid);
    // hostname
    kamsg.set_hostname(m_config->value("hostname"));
    // processname
    kamsg.set_processname(m_config->value("processname"));
    // commstate
    kamsg.set_commstate(m_commstate);
    // simstage
    kamsg.set_simstage(m_simstage);
    // synchronizing?
    kamsg.set_synchronizing(false);

    // num fpls
    kamsg.set_num_fpls(m_fpls.size());

    // fpls_crc
    if( m_calc_checksum )
    {
        boost::crc_32_type  crc;
        std::string crcmsg;
        for( m_fpliter it = m_fpls.begin() ; it != m_fpls.end() ;
            it++ )
        {
            it->second.SerializeToString(&crcmsg);
            crc.process_bytes(crcmsg.c_str(), crcmsg.length());
        }
        kamsg.set_fpls_crc(crc.checksum());
        m_fpl_checksum = crc.checksum();

        m_calc_checksum = false;
    }
    else
        kamsg.set_fpls_crc(m_fpl_checksum);


    // Timestamp
    google::protobuf::Timestamp timestamp;
    struct timeval tv;
```

```
    gettimeofday(&tv, NULL);
    timestamp.set_seconds(tv.tv_sec);
    timestamp.set_nanos(tv.tv_usec * 1000);
    kamsg.set_allocated_timestamp(&timestamp);

    // Serialize
    std::string kastr;
    kamsg.SerializeToString(&kastr);
    msg += kastr;

    // Free objects
    kamsg.release_uuid();
    kamsg.release_timestamp();

    zmq::message_t fplcmd(msg.length());
    memcpy(fplcmd.data(), msg.data(), msg.length());
    // BOOST_LOG_SEV(m_log, kw::debug) << "Sending keepalive
        message with size " << fplcmd.size();
    return fplcmd;
}

void Base::start()
{
    BOOST_LOG_SEV(m_log, kw::info) << "Ready to receive data" <<
        std::endl;

    auto start = std::chrono::system_clock::now().time_since_epoch
        ();
    while (1)
    {
        receive();

        // Every second: Send keepalive and print number of flight
            plans in map
        auto now = std::chrono::system_clock::now().
            time_since_epoch();
        auto elapsed = std::chrono::duration_cast<std::chrono::
            milliseconds>(now - start);
        if( elapsed.count() < 1000 )
            continue;

        start = std::chrono::system_clock::now().time_since_epoch()
            ;

        if( m_commstate != FPL::FPLCmd::Synchronizing )
        {
            zmq::message_t msg = createKeepAliveMsg();
            sendMsg(&msg);
        }


        const google::protobuf::EnumDescriptor *descriptor = FPL::
            FPLCmd::SimStage_descriptor();
        BOOST_LOG_SEV(m_log, kw::info) << "Fpls: " << m_fpls.size()
                                       << " Checksum: " <<
                                           m_fpl_checksum
                                       << " Clients: " << m_clients
                                           .size()
                                       << " Sim stage: " <<
                                           descriptor->
                                           FindValueByNumber(
                                           m_simstage)->name();
```

```
        for( auto it = m_clients.begin() ; it != m_clients.end() ;
            it++ )
        {
            auto sec = std::chrono::duration_cast<std::chrono::
                seconds>(now);
            if( (sec.count() - it->second.timestamp().seconds()) >
                10.0 )
            {
                std::cout << "Client timed out " << it->second.
                    hostname() << ", "
                            << it->second.processname() << std::endl;
                m_clients.erase(it);
            }
        }
    }
}
```

C.2 DataServer Class

Listing 31: DataServer Class Implementation

```
#define BOOST_LOG_DYN_LINK 1

#include <iostream>
#include <unistd.h>
#include <thread>

#include <google/protobuf/text_format.h>
#include <google/protobuf/duration.pb.h>
#include <google/protobuf/util/time_util.h>

#include <boost/filesystem.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/crc.hpp>

#include "DataServer.h"

std::string string_to_hex(const std::string& input);

DataServer::DataServer() : Base()
{
}

void DataServer::initDS()
{
    if( m_config->value("datatype") == "fpl" )
        createTestFpls();

    setState(FPL::FPLCmd::Ready);
}

void DataServer::initNetwork()
{
    m_context = new zmq::context_t(1);

    /////
    // Subscriber - Keepalive message from clients
```

```
    /////
    m_cmdsocket  = new zmq::socket_t(*m_context, ZMQ_SUB);
    m_cmdsocket->setsockopt( ZMQ_RCVTIMEO, 100 );

    // Set socket option
    try {
        BOOST_LOG_SEV(m_log, kw::info) << "Subscriber: Setting
            socket option - subscribe to all messages";
        m_cmdsocket->setsockopt( ZMQ_SUBSCRIBE, "", 0 );
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    // Connect to pgm socket
    try {
        std::stringstream connstr;
        connstr << "epgm://"
            << m_config->value("network.interface")  << ";"
            << m_config->value("network.fpladdress") << ":"
            << m_config->value("network.dsservice");
        BOOST_LOG_SEV(m_log, kw::info) << "Subscriber: Connect MC
            using " << connstr.str();
        m_cmdsocket->bind(connstr.str());
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    // Connect to IPC
    try {
        using namespace boost::filesystem;
        if ( !exists( "/tmp/InNOVA" ) )
            create_directory("/tmp/InNOVA");
        std::string connstr = "ipc:///tmp/InNOVA/3106";
        BOOST_LOG_SEV(m_log, kw::info) << "Subscriber: Connect ipc
            using " << connstr;
        m_cmdsocket->bind(connstr);
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    /////
    // Publisher - FPL data to clients
    /////
    m_fplsocket  = new zmq::socket_t(*m_context, ZMQ_PUB);

    // Set transmission rate. Default is 100kb/s
    try {
        BOOST_LOG_SEV(m_log, kw::info) << "Publisher: Setting
            socket option - set rate to 1Gb";
        const int rate = 1000000;                                  // 1
            Gb TX- and RX- rate
        m_fplsocket->setsockopt(ZMQ_RATE, &rate, sizeof(rate));
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
```

```
    }

    // Connect to pgm socket
    try {
        std::stringstream connstr;
        connstr << "epgm://"
            << m_config->value("network.interface")  << ";"
            << m_config->value("network.fpladdress") << ":"
            << m_config->value("network.fplservice");
        BOOST_LOG_SEV(m_log, kw::info) << "Publisher: Connect MC
            using " << connstr.str();
        m_fplsocket->bind(connstr.str());
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    // Connect to IPC
    try {
        using namespace boost::filesystem;
        if ( !exists( "/tmp/InNOVA" ) )
            create_directory("/tmp/InNOVA");
        std::string connstr = "ipc:///tmp/InNOVA/3105";
        BOOST_LOG_SEV(m_log, kw::info) << "Publisher: Connect ipc
            using " << connstr;
        m_fplsocket->bind(connstr);
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

}

void DataServer::sendMsg(zmq::message_t *msg)
{
    m_fplsocket->send(*msg);
}

void DataServer::handleSimulation()
{
    if( m_config->value("datatype") != "fpl")
        return;

    if( m_simstage == FPL::FPLCmd::TransferManyFplsFromServer )
    {
        m_simstage = FPL::FPLCmd::DeleteOneFplOnServer;
    }
    else if ( m_simstage == FPL::FPLCmd::DeleteOneFplOnServer )
    {
        deleteFpl(1752, "NA1754", true); // Non existing
        deleteFpl(1754, "NA1754", true);
        m_simstage = FPL::FPLCmd::AddOneFplOnServer;
    }
    else if ( m_simstage == FPL::FPLCmd::AddOneFplOnServer )
    {
        FPL::FPLData fpl;
        fpl.set_ssrcode(1754);
        fpl.set_callsign("NA1754");
        fpl.set_wtc(FPL::FPLData_WTC_J);
        addFpl(fpl);
```

```
        m_simstage = FPL::FPLCmd::UpdateOneFplOnServer;
    }
    else if ( m_simstage == FPL::FPLCmd::UpdateOneFplOnServer )
    {
        FPL::FPLData fpl;
        fpl.set_ssrcode(1754);
        fpl.set_callsign("NA1754");
        fpl.set_wtc(FPL::FPLData_WTC_J);
        addFpl(fpl);
        m_simstage = FPL::FPLCmd::DeleteAllFplsOnServer;
    }
    else
        std::cout << "Unknown sim stage" << std::endl;
}

void DataServer::sendSelectedFlightPlans()
{
    std::string protomsg;
    m_selFplPB.SerializeToString(&protomsg);
    zmq::message_t fplmsg(protomsg.length());
    memcpy(fplmsg.data(), protomsg.data(), protomsg.length());
    sendMsg(&fplmsg);
}

void DataServer::handleKeepAliveMsg(std::string kastr)
{
    FPL::FPLCmd ka;
    ka.ParseFromString(kastr);
    bool send_fpls = false;
    if( ka.commstate() == FPL::FPLCmd::Initial )
    {
        ka.set_synchronizing(true);
        send_fpls = true;
        BOOST_LOG_SEV(m_log, kw::info) << "Received initial
            keepalive message from "
                                       << ka.hostname()
                                       << ", "
                                       << ka.processname()
                                       << " ("
                                       << ka.uuid().value()
                                       << ")";
    }
    else
    {
        BOOST_LOG_SEV(m_log, kw::trace) << "Received keepalive
            message from "
                                        << ka.hostname()
                                        << ", "
                                        << ka.processname()
                                        << " ("
                                        << ka.uuid().value()
                                        << ")";
    }

    // Check timestamp - only accept 0.5 sec latency
    struct timeval tv;
    gettimeofday(&tv, NULL);
    google::protobuf::Timestamp timestamp;
    timestamp.set_seconds(tv.tv_sec);
    timestamp.set_nanos(tv.tv_usec * 1000);
    double now = tv.tv_sec + tv.tv_usec/1000000.0;
```

```
    double then = ka.timestamp().seconds() + ka.timestamp().nanos()
        /1000000000.0;
    if( now - then > 0.5 )
    {
        BOOST_LOG_SEV(m_log, kw::error) << "Update from " << ka.
            hostname() << ", " << ka.processname() << " was too
            late! (" << now - then << " secs) Ignoring message.";
        return;
    }

    // Check checksums
    bool stateOk = true;
    if( m_fpl_checksum != ka.fpls_crc() and m_fpl_checksum != 0 and
         ka.fpls_crc() != 0)
    {
        BOOST_LOG_SEV(m_log, kw::error) << "Checksums differ! " <<
            m_fpl_checksum << " vs " << ka.fpls_crc() << std::endl;
        stateOk = false;
    }

    // Add/update client
    if( m_clients.find(ka.uuid().value()) == m_clients.end() )
    {
        BOOST_LOG_SEV(m_log, kw::info) << "Adding client " << ka.
            hostname() << ", " << ka.processname() << " (" << ka.
            uuid().value() << ")";

        if( m_config->value("datatype") == "selected_fpl" )
            sendSelectedFlightPlans();
    }
    m_clients[ka.uuid().value()] = ka;

    // Check simulation stage
    if( stateOk )
        handleSimulation();

    // Stage: Send all flight plans
    if( not send_fpls )
        return;

    BOOST_LOG_SEV(m_log, kw::trace) << "Starting full sync of
        flight plans";
    auto start = std::chrono::system_clock::now();
    std::string msg = "START SYNC";
    msg += '\0';
    zmq::message_t fplcmd(msg.length());
    memcpy(fplcmd.data(), msg.data(), msg.length());
    sendMsg(&fplcmd);

    for( m_fpliter it = m_fpls.begin() ; it != m_fpls.end() ; it++
        )
    {
        sendFpl(it->second);
        std::this_thread::sleep_for(std::chrono::microseconds(200))
            ;
    }

    sleep(1);
    msg = "END SYNC";
    msg += '\0';
    zmq::message_t fplcmd_end(msg.length());
    memcpy(fplcmd_end.data(), msg.data(), msg.length());
```

```
    sendMsg(&fplcmd_end);

    auto end = std::chrono::system_clock::now();
    std::chrono::duration<double, std::milli> send_ms = end - start
        ;
    BOOST_LOG_SEV(m_log, kw::trace) << "Finalized full sync of
        flight plans in " << send_ms.count() << " ms";
}

void DataServer::handleSelectedFlightplan(std::string msg_str)
{
    m_selFplPB.ParseFromString(msg_str);
    std::stringstream str;
    str << "Selected flight plan> Timestamp: "
        << google::protobuf::util::TimeUtil::ToString(m_selFplPB.
            timestamp())
        << " Map<WpIf, fplKey>: ";
    for( auto it = m_selFplPB.selfpl().begin() ; it != m_selFplPB.
        selfpl().end() ; it++ )
        str << "<" << it->first << ", " << it->second << "> ";
    str << std::endl;
    BOOST_LOG_SEV(m_log, kw::info) << str.str();

    sendSelectedFlightPlans();
}

void DataServer::receive()
{
    int rc;
    zmq::message_t resultset;

    try {
        if( (rc = m_cmdsocket->recv(&resultset)) != true)
            return;
    } catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    std::string msg_str(static_cast<char*>(resultset.data()),
        resultset.size());
    std::string::size_type pos = msg_str.find('\0');
    std::string type = "DATA";
    if (pos != std::string::npos)
    {
        type = msg_str.substr(0, pos);
        msg_str = msg_str.substr(pos+1, msg_str.length());
    }
    if( type == "KA" )
        handleKeepAliveMsg(msg_str);
    else
    {
        if( m_config->value("datatype") == "fpl" )
            addFpl(msg_str);
        else if( m_config->value("datatype") == "selected_fpl" )
            handleSelectedFlightplan(msg_str);
        else
            BOOST_LOG_SEV(m_log, kw::fatal) << "No handler for
                datatype "
                                            << m_config->value("
                                                datatype")
                                            << resultset.size();
```

```
    }
}
```

C.3 DataClient Class

Listing 32: DataClient Class Implementation

```
#define BOOST_LOG_DYN_LINK 1

#include <iostream>
#include <unistd.h>

#include <google/protobuf/text_format.h>

#include <boost/filesystem.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/crc.hpp>

#include "DataClient.h"

DataClient::DataClient() : Base()
{
}

void DataClient::initNetwork()
{
    m_context = new zmq::context_t(1);


    /////
    // Publisher - Keepalive message to server
    /////
    std::stringstream connstr;
    connstr << "tcp://"
            << m_config->value("network.dsaddress")
            << ":"
            << m_config->value("network.dsservice");
    BOOST_LOG_SEV(m_log, kw::info) << "Push: Connect tcp using " <<
         connstr.str();

    m_cmdsocket = new zmq::socket_t(*m_context, ZMQ_PUB);
    const int rate = 1000000;                                  // 1Gb
        TX- and RX- rate
    m_cmdsocket->setsockopt(ZMQ_RATE, &rate, sizeof(rate));

    // Connect to pgm socket
    try {
        std::stringstream connstr;
        connstr << "epgm://"
            << m_config->value("network.interface")  << ";"
            << m_config->value("network.fpladdress") << ":"
            << m_config->value("network.dsservice");
        BOOST_LOG_SEV(m_log, kw::info) << "Publisher: Connect MC
            using " << connstr.str();
        m_cmdsocket->connect(connstr.str());
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
```

```
    }

    // Connect to IPC
    try {
        using namespace boost::filesystem;
        if ( !exists( "/tmp/InNOVA" ) )
            create_directory("/tmp/InNOVA");
        std::string connstr = "ipc:///tmp/InNOVA/3106";
        BOOST_LOG_SEV(m_log, kw::info) << "Publisher: Connect ipc
            using " << connstr;
        m_cmdsocket->connect(connstr);
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    m_cmdsocket->connect(connstr.str());

    /////
    // Subscriber - FPL data from server
    /////
    m_fplsocket  = new zmq::socket_t(*m_context, ZMQ_SUB);
    m_fplsocket->setsockopt( ZMQ_RCVTIMEO, 100 );

    // Set socket option
    try {
        BOOST_LOG_SEV(m_log, kw::info) << "Subscriber: Setting
            socket option - subscribe to all messages";
        m_fplsocket->setsockopt( ZMQ_SUBSCRIBE, "", 0 );
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    // Connect to pgm socket
    try {
        std::stringstream connstr;
        connstr << "epgm://"
            << m_config->value("network.interface")  << ";"
            << m_config->value("network.fpladdress") << ":"
            << m_config->value("network.fplservice");
        BOOST_LOG_SEV(m_log, kw::info) << "Subscriber: Connect MC
            using " << connstr.str();
        m_fplsocket->connect(connstr.str());
    }
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    // Connect to IPC
    try {
        using namespace boost::filesystem;
        if ( !exists( "/tmp/InNOVA" ) )
            create_directory("/tmp/InNOVA");
        std::string connstr = "ipc:///tmp/InNOVA/3105";
        BOOST_LOG_SEV(m_log, kw::info) << "Subscriber: Connect ipc
            using " << connstr;
        m_fplsocket->connect(connstr);
    }
```

```
    catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

}

void DataClient::handleSimulation()
{
}

void DataClient::sendMsg(zmq::message_t* msg)
{
    m_cmdsocket->send(*msg);
}

void DataClient::handleKeepAliveMsg(std::string kastr)
{
    FPL::FPLCmd ka;
    ka.ParseFromString(kastr);
    // BOOST_LOG_SEV(m_log, kw::trace) << "Received keepalive
        message from " << ka.hostname()  << ", " << ka.processname
        ();

    // Check timestamp - only accept 0.5 sec latency
    struct timeval tv;
    gettimeofday(&tv, NULL);
    double now = tv.tv_sec + tv.tv_usec/1000000.0;
    double then = ka.timestamp().seconds() + ka.timestamp().nanos()
        /1000000000.0;
    if( now - then > 0.5 )
    {
        BOOST_LOG_SEV(m_log, kw::error) << "Update from " << ka.
            hostname() << ", " << ka.processname() << " was too
            late! (" << now - then << " secs) Ignoring message.";
        return;
    }

    // Check checksums
    if( m_fpl_checksum != ka.fpls_crc() and m_fpl_checksum != 0 and
         ka.fpls_crc() != 0)
        BOOST_LOG_SEV(m_log, kw::error) << "Checksums differ! " <<
            m_fpl_checksum << " vs " << ka.fpls_crc() << std::endl;
}

void DataClient::receive()
{
    int rc;
    zmq::message_t resultset;

    try {
        if( (rc = m_fplsocket->recv(&resultset)) != true)
            return;
    } catch(zmq::error_t& e) {
        BOOST_LOG_SEV(m_log, kw::error) << zmq_strerror(errno) <<
            std::endl;
    }

    std::string msg_str(static_cast<char*>(resultset.data()),
        resultset.size());
    std::string original_str = msg_str; // Save it in case this is
        a complete FPL message
```

```
    std::string::size_type pos = msg_str.find('\0');
    std::string type = "DATA";
    if (pos != std::string::npos)
    {
        type = msg_str.substr(0, pos);
        msg_str = msg_str.substr(pos+1, msg_str.length());
    }
    if( type == "KA" )
        handleKeepAliveMsg(msg_str);
    else if( type == "START SYNC" )
    {
        m_fpls.clear();
        m_calc_checksum = true;
        setState(FPL::FPLCmd::Synchronizing);
    }
    else if( type == "END SYNC" )
        setState(FPL::FPLCmd::Ready);
    else if( type == "DELETE" )
        rmFpl(msg_str);
    else
        addFpl(original_str);
}
```

C.4 FPLCmd - Keepalive Message

Listing 33: Keepalive Proto Definition

```
syntax = "proto3";

import "FPLDefines.proto";
import "google/protobuf/timestamp.proto";

package FPL;

message FPLCmd {
  enum CommState {
    Initial = 0;
    Synchronizing = 1;
    Ready = 2;
  }

  enum SimStage {
    TransferManyFplsFromServer = 0;

    DeleteOneFplOnServer = 1;
    AddOneFplOnServer = 2;
    UpdateOneFplOnServer = 3;
    DeleteAllFplsOnServer = 4;

    TransferManyFplsFromClient = 5;

    DeleteOneFplOnClient = 6;
    AddOneFplOnClient = 7;
    UpdateOneFplOnClient = 8;
    DeleteAllFplsOnClient = 9;

    Finished = 10;
  }

  UUID uuid = 1;
```

```
  string hostname = 2;
  string processname = 3;
  google.protobuf.Timestamp timestamp = 4;
  uint32 num_fpls = 5;
  uint32 fpls_crc = 6;

  CommState commstate = 100;
  SimStage simstage = 101;
  bool synchronizing = 102;
}
```

C.5 FPLData - Flight Plan Data

Listing 34: FPL Data Proto Definition

```
syntax = "proto3";

package FPL;

message FPLData {
  enum WTC {
    NOTSET = 0;
    L = 1;
    M = 2;
    H = 3;
    J = 4;
  }

  uint32 SSRCode = 1;
  string callsign = 2;
  WTC wtc = 3;
  string blob = 4;

  repeated string route = 5;
}
```

D Appendix D - SFManager

D.1 SFManager - Header File

Listing 35: SFManager Header File

```
#ifndef SFMANAGER_H_INC
#define SFMANAGER_H_INC

#include <chrono>

#include <zmq.hpp>

#include <CLSignal.h>
#include <CLUuid.h>
#include <PThread.h>

// protobuf definitions
#include <DOSelectedFlightPlan.pb.h>
#include <FPLKeepAlive.pb.h>

class PThread;
class FlightDataCollection;

class SFManager
{
private:
    zmq::context_t*        m_sendcontext;
    zmq::socket_t*         m_sendsocket;

    std::string            m_mywpid;
    CLUuid                 m_uuid;
    FPLKeepAlive::UUID     m_serverUuid;
    bool                   m_serverAvailable;
    std::chrono::time_point<std::chrono::system_clock>
        m_lastValueDTFromServer;

    DOSelectedFlightPlan   m_currentSelectedFplPB;

    PThread*               m_receivethread;
    PThread*               m_sendthread;
    FlightDataCollection&  m_fdCol;

    Mutex                  m_mutexServerStatus;
    Mutex                  m_mutexSelFPLData;

    static void*           receiveThread(PThread&, void* par);
    static void*           sendThread(PThread&, void* par);
    void                   receive();
    void                   sendkeepalive();
    void                   handleKeepAliveMsg(std::string msg_str);
    void                   send(std::string msg);
    bool                   serverAvailable();
    std::string            getSelFplKey() const;

public:
    void                   notifyServerStatus();
    void                   processSelectedFlightplanMessage();
    void                   sendNewSelectedFlightPlan(std::string
        wpid, std::string fplKey);

    CLSignal1<bool>         m_sfServerSignal;
```

```
    CLSignal1<std::string> m_selectedFplKeySignal;

    SFManager(FlightDataCollection& fdCol, std::string wpid);
    ~SFManager();
};

#endif
```

D.2 SFManager - Implementation

Listing 36: SFManager Implementation

```
#include <google/protobuf/message.h>
#include <google/protobuf/util/time_util.h>

#include <Mutex.h>
#include <AutoLocker.h>

#include "SFManager.h"
#include "FlightDataCollection.h"

SFManager::SFManager(FlightDataCollection& fdCol, std::string wpid)
    : m_sendcontext(),
      m_sendsocket(),
      m_mywpid(wpid),
      m_uuid(CLUuid::createUuid()),
      m_serverUuid(),
      m_serverAvailable(false),
      m_lastValueDTFromServer(),
      m_currentSelectedFplPB(),
      m_receivethread(0),
      m_sendthread(0),
      m_fdCol(fdCol)
{
    GOOGLE_PROTOBUF_VERIFY_VERSION;
    std::cout << "[SFManager] ****** Initializing SFManager" << std
        ::endl;

    // initial selected flight is processed => not handled in
        process..() method
    m_currentSelectedFplPB.set_processed(true);

    m_serverUuid.set_value("NOT SET");

    m_sendcontext = new zmq::context_t(1);
    if( m_sendcontext == NULL )
        std::cout << "[SFManager] [ERROR] Unable to create zmq
            context" << std::endl;

    m_sendsocket  = new zmq::socket_t(*m_sendcontext, ZMQ_PUB);
    if( m_sendsocket == NULL )
        std::cout << "[SFManager] [ERROR] Unable to create zmq
            socket" << std::endl;
    const int rate = 1000000;                                  // 1Gb
        TX- and RX- rate
    m_sendsocket->setsockopt(ZMQ_RATE, &rate, sizeof(rate));
    std::stringstream connstr;
    connstr << "epgm://lanB;239.10.10.10:3106";
    std::cout << "[SFManager] " << connstr.str() << std::endl;
    m_sendsocket->connect(connstr.str());
```

```
    m_sendsocket->connect("ipc:///tmp/SelectedFPLPub");

    m_receivethread = new PThread(&SFManager::receiveThread,
         reinterpret_cast<void*>(this),
         PThread::NORMAL, 0);
    m_sendthread = new PThread(&SFManager::sendThread,
         reinterpret_cast<void*>(this),
         PThread::NORMAL, 0);

    PThread::yield();
}

SFManager::~SFManager()
{
    delete m_receivethread;
    delete m_sendthread;
}

void* SFManager::receiveThread(PThread&, void* par)
{
    SFManager* sfreceiver = reinterpret_cast<SFManager*>(par);
    sfreceiver->receive();
    return 0;
}

void* SFManager::sendThread(PThread&, void* par)
{
    SFManager* sfsend = reinterpret_cast<SFManager*>(par);
    sfsend->sendkeepalive();
    return 0;
}

void SFManager::send(std::string msg)
{
    if( not serverAvailable() )
        return;

    zmq::message_t selfpl_msg(msg.length());
    memcpy( selfpl_msg.data(), msg.data(), msg.length() );

    m_sendsocket->send(selfpl_msg);
}

void SFManager::sendkeepalive()
{
    FPLKeepAlive      kamsg;

    // Static keepalive data
    // UUID
    FPLKeepAlive::UUID uuid;
    uuid.set_value(m_uuid.toString().toStdString());
    kamsg.set_allocated_uuid(&uuid);

    // hostname
    char hostname[256];
    int ret = gethostname(hostname,sizeof(hostname));
    if(ret == -1)
    {
        perror("[ERROR] gethostname() failed");
        exit(1);
    }
    // hostname
```

```
    kamsg.set_hostname(hostname);

    // processname
    kamsg.set_processname("TODO");

    // commstate
    kamsg.set_commstate(FPLKeepAlive::Ready);

    while(true)
    {
        // Timestamp
        google::protobuf::Timestamp timestamp;
        struct timeval tv;
        gettimeofday(&tv, NULL);
        timestamp.set_seconds(tv.tv_sec);
        timestamp.set_nanos(tv.tv_usec * 1000);
        kamsg.set_allocated_timestamp(&timestamp);

        // Serialize
        std::string kastr;
        kamsg.SerializeToString(&kastr);
        std::string msg = "KA";
        msg += '\0';
        msg += kastr;

        send(msg);

        // Free object
        kamsg.release_timestamp();

        sleep(1);

        // Check if server is available
        auto now = std::chrono::system_clock::now();
        auto elapsed = std::chrono::duration_cast<std::chrono::
            seconds>(now - m_lastValueDTFromServer);
        if( elapsed.count() > 3.0 )
        {
            std::cout << "[SFManager] No connection to server." <<
                std::endl;
            AutoLocker lock(m_mutexServerStatus);
            m_serverUuid.set_value("NOT SET");
        }
    }
}

bool SFManager::serverAvailable()
{
    AutoLocker lock(m_mutexServerStatus);

    return (m_serverUuid.value() != "NOT SET");
}

void SFManager::handleKeepAliveMsg(std::string msg_str)
{
    FPLKeepAlive      ka;
    ka.ParseFromString(msg_str);

    // Check timestamp - only accept 0.5 sec latency
    struct timeval tv;
    gettimeofday(&tv, NULL);
    google::protobuf::Timestamp timestamp;
```

```
    timestamp.set_seconds(tv.tv_sec);
    timestamp.set_nanos(tv.tv_usec * 1000);
    double now = tv.tv_sec + tv.tv_usec/1000000.0;
    double then = ka.timestamp().seconds() + ka.timestamp().nanos()
        /1000000000.0;
    if( now - then > 0.5 )
    {
        std::cout << "[SFManager] Update from " << ka.hostname() <<
             ", " << ka.processname() << " was too late! (" << now
            - then << " secs) Ignoring message." << std::endl;
        return;
    }

    AutoLocker lock(m_mutexServerStatus);

    if( m_serverUuid.value() == "NOT SET" )
    {
        m_serverUuid = ka.uuid();
        std::cout << "[SFManager] First connection to server! Uuid
            = " << m_serverUuid.value() << std::endl;
    }
    else if( m_serverUuid.value() != ka.uuid().value() )
    {
        std::cout << "[SFManager] Received keep alive message from
            new server. Will ignore!" << std::endl;
        return;
    }

    m_lastValueDTFromServer = std::chrono::system_clock::now();
}

void SFManager::receive()
{
    zmq::context_t   *recvcontext;
    zmq::socket_t    *recvsocket;

    recvcontext = new zmq::context_t(1);

    recvsocket  = new zmq::socket_t(*recvcontext, ZMQ_SUB);
    recvsocket->setsockopt( ZMQ_SUBSCRIBE, "", 0 );
    std::stringstream connstr;
    connstr << "epgm://lanB;239.10.10.10:3105";
    std::cout << "[SFManager] " << connstr.str() << std::endl;
    recvsocket->connect(connstr.str());
    recvsocket->connect("ipc:///tmp/SelectedFPLSub");

    int rc;
    zmq::message_t resultset;
    while (1)
    {
        try {
            if( (rc = recvsocket->recv(&resultset)) == true) {
                std::string msg_str(static_cast<char*>(resultset.
                    data()), resultset.size());

                std::string::size_type pos = msg_str.find('\0');
                std::string type;
                if (pos != std::string::npos)
                {
                    type = msg_str.substr(0, pos);
                    msg_str = msg_str.substr(pos+1, msg_str.length
                        ());
```

```
                }
                if( type == "KA" )
                {
                    handleKeepAliveMsg(msg_str);
                    continue;
                }

                DOSelectedFlightPlan selFplPB;
                selFplPB.ParseFromString(msg_str);

                AutoLocker lock(m_mutexSelFPLData);

                auto curfplkey = getSelFplKey();

                // Only last one important, so just overwrite
                m_currentSelectedFplPB = selFplPB;

                auto newfplkey = getSelFplKey();

                // Check if it is necessary to send signal of
                    updated fplkey
                if( curfplkey != newfplkey )
                    m_currentSelectedFplPB.set_processed(false);
            }
        }
        catch(zmq::error_t& e) {
            std::cout << "[SFManager] " << zmq_strerror(errno) <<
                std::endl;
        }
    }
}

std::string SFManager::getSelFplKey() const
{
    auto sel = m_currentSelectedFplPB.selfpl().find(m_mywpid);
    if( sel != m_currentSelectedFplPB.selfpl().end() )
      return sel->second;
    return std::string("");
}

void SFManager::sendNewSelectedFlightPlan(std::string wpid, std::
    string fplKey)
{
    AutoLocker lock(m_mutexSelFPLData);

    auto& map = *m_currentSelectedFplPB.mutable_selfpl();
    map[wpid] = fplKey;

    google::protobuf::Timestamp timestamp;
    struct timeval tv;
    gettimeofday(&tv, NULL);
    timestamp.set_seconds(tv.tv_sec);
    timestamp.set_nanos(tv.tv_usec * 1000);
    m_currentSelectedFplPB.set_allocated_timestamp(&timestamp);

    std::cout << "[SFManager] Operator Selected Flightplan: Map<
        WpID, fplKey>: ";
    for( auto it = m_currentSelectedFplPB.selfpl().begin() ; it !=
        m_currentSelectedFplPB.selfpl().end() ; it++ )
        std::cout << "<" << it->first << ", " << it->second << "> "
            ;
    std::cout << "" << std::endl;
```

```
    std::string protomsg;
    m_currentSelectedFplPB.SerializeToString(&protomsg);
    send(protomsg);

    m_currentSelectedFplPB.release_timestamp();
}

void SFManager::notifyServerStatus()
{
    m_sfServerSignal.sigemit(serverAvailable());
}

void SFManager::processSelectedFlightplanMessage()
{
    AutoLocker lock(m_mutexSelFPLData);

    if( m_currentSelectedFplPB.processed() )
        return;

    std::cout << "[SFManager] Processing new selected flightplan:
        Map<WpID, fplKey>: ";
    for( auto it = m_currentSelectedFplPB.selfpl().begin() ; it !=
        m_currentSelectedFplPB.selfpl().end() ; it++ )
        std::cout << "<" << it->first << ", " << it->second << "> "
            ;
    std::cout << "" << std::endl;

    m_currentSelectedFplPB.set_processed(true);
    m_selectedFplKeySignal.sigemit(getSelFplKey());
}
```

References

[1] B. Adamson, C. Bormann, M. Handley, and J. Macker. Negative-acknowledgment (nack)-oriented reliable multicast (norm) building blocks. RFC 3941, RFC Editor, November 2004.

[2] B. Adamson, C. Bormann, M. Handley, and J. Macker. Negative-acknowledgment (nack)-oriented reliable multicast (norm) protocol. RFC 3940, RFC Editor, November 2004.

[3] B. Adamson, C. Bormann, M. Handley, and J. Macker. Multicast negative-acknowledgment (nack) building blocks. RFC 5401, RFC Editor, November 2008.

[4] B. Adamson, C. Bormann, M. Handley, and J. Macker. Nack-oriented reliable multicast (norm) transport protocol. RFC 5740, RFC Editor, November 2009.

[5] Avro c++. https://avro.apache.org/docs/1.8.2/api/cpp/html/index.html. Accessed: 2018-May-01.

[6] D. Crockford. The application/json media type for javascript object notation (json). RFC 4627, RFC Editor, July 2006. http://www.rfc-editor.org/rfc/rfc4627.txt.

[7] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. http://www.icir.org/floyd/srm-paper.html. Accessed: 2017-Oct-11, Last Modified: August 1999.

[8] Yaron Y. Goland. Writing backwards compatible xml schema 1.0 schemas. https://www.goland.org/xmlschemabackwardscompat/. Accessed: 2018-May-01, Created: 2004-Aug-09.

[9] Google. Pgm implementations/compatibility. http://code.google.com/p/openpgm/. Accessed: 2017-Oct-11.

[10] Google. Serialization/deserialization framework benchmarks. http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking. Accessed: 2014-May-13.

[11] A high performance open-source universal rpc framework. https://grpc.io/. Accessed: 2018-May-01.

[12] IBM. Websphere amqp 1.0 support. https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.amqp.doc/amqp_support.htm. Accessed: 2018-Apr-26.

[13] IETF. Ietf reliable multicast working group. http://datatracker.ietf.org/wg/rmt/charter/. Accessed: 2017-Oct-11.

[14] Steven McCoy. Pgm implementations/compatibility. https://github.com/steve-o/openpgm. Accessed: 2018-Apr-23.

[15] Nack-oriented reliable multicast (norm). https://www.nrl.navy.mil/itd/ncs/products/norm. Accessed: 2018-Apr-25.

[16] Data distribution service specification v1.4. https://www.omg.org/spec/DDS/1.4/. Created: March 2015, Accessed: 2018-May-01.

[17] OpenLogic. Comparisons of open source esb solutions. https://www.roguewave.com/events/on-demand-webinars/comparison-open-source-software-esb. Accessed: 2014-Jun-01, Created: 2010-Aug-04.

[18] M. Pullen, F. Zhao, and D. Cohen. Selectively reliable multicast protocol (srmp). RFC 4410, RFC Editor, February 2006.

[19] Fernando Crespo Sanches. Xtypes: Taking type evolution to the next level. https://www.rti.com/blog/2014/02/12/xtypes-taking-type-evolution-to-the-next-level/. Created: 2014-Feb-11, Accessed: 2018-May-01.

[20] Eishay Smith. Jvm serializers. https://github.com/eishay/jvm-serializers/wiki. Accessed: 2014-May-13.

[21] Jim Kurose Sneha K. Kasera and Don Towsley. Scalable reliable multicast using multiple multicast groups. www.cs.utah.edu/~kasera/myPapers/Kasera99-Scalable.pdf.

[22] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano. Pgm reliable transport protocol specification. RFC 3208, RFC Editor, December 2001. http://www.rfc-editor.org/rfc/rfc3208.txt.

[23] J. Widmer and M. Handley. Tcp-friendly multicast congestion control (tfmcc): Protocol specification. RFC 4654, RFC Editor, August 2006.

[24] Wikipedia. Reed solomon error correction. http://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction. Accessed: 2017-Sep-27.

[25] Kai Wähner. Choosing the right esb. http://www.infoq.com/articles/ESB-Integration. Accessed: 2014-Jun-01, Created: 2013-Apr-02.

[26] Zeromq. http://zeromq.org. Accessed: 2018-Apr-23.

[27] Zeromq latency/throughput tests. http://zeromq.org/results:0mq-tests-v03. Accessed: 2018-Apr-23.