TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Ingmar Trump 155202IAPB

IMPLEMENTING TTÜ SATELLITE COMMUNICATION PROTOCOL

Bachelor's thesis

Supervisor: Evelin Halling
MSc

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Ingmar Trump 155202IAPB

TTÜ SATELLIIDI KOMMUNIKATSIOONIPROTOKOLLI IMPLEMENTEERIMINE

bakalaureusetöö

Juhendaja: Evelin Halling
MSc

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Ingmar Trump

21.05.2018

Abstract

The goal of this thesis is to improve the TTÜ satellite communication protocol implementation in ground station. It includes writing new code, refactoring already existing code, designing high level communication protocol and its possible implementation and redesigning flawed systems.

As the thesis is a part of a bigger project, a lot of explaining of existing code and design choices is done. Many of the decisions in the systems are heavily guided by the limitations in satellite technology. These limitations need to be well understood to make most decisions in the project.

Throughout the thesis, many flaws were discovered in the existing implementations. Once the flaws were noticed, they were further analyzed and possible solutions were proposed. Some of the fixes were applied right away.

High level commands got developed during the thesis too. The development is nowhere near done, but some design work got done. This part did not get too much attention because there were more urgent fixes to be done on low level commands and their implementation.

This thesis is written in English and is 31 pages long, including 8 chapters, 7 figures and 4 tables.

Annotatsioon

TTÜ satlliidi kommunikatsiooniprotokolli implementeerimine

Selle lõputöö eesmärk on implementeerida TTÜ satelliidi kommunikatsiooniprotokoll maajama poolel. Täpsemalt oli vaja arendada antud protokolli siini ja transpordi taseme käskude teostust.

Töö käigus kirjutatakse uut ning muudetakse vanat koodi. Lisaks disainitakse kõrge taseme protokolle kui ka muudetakse ümber olemasoleva missioonijuhtimis tarkvara disaini.

Terve satelliidi projekt on väga suur ning on arenduses olnud juba aastaid. Seetõttu on antud lõputöö käigus palju kirjeldatud erinevaid olemasolevaid lahendusi süsteemis. Kirjeldatud on satelliidi kommunikatsiooni limitatsioone ning nende mõju teatud lähenemistele.

Missioonijuhtimis tarkvara oli juba eelnevalt arenduses olnud. Peamiselt oli eelnevalt rõhutud algse prototüübi valmimisele, millega saaks lihtsamaid käske satelliidile saata. See tekitas aga antud lõputöö käigus mitmeid probleeme. Tekkisid valed eeldused, et tehtud disaini peaks järgima. Antud disain vajas aga suuri muudatusi, takistades peamise eesmärgini jõudmist.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 31 leheküljel, 8 peatükki, 7 joonist, 4 tabelit.

List of abbreviations and terms

TTÜ	Tallinn University of Technology
AX.25	Amateur X.25
MCS	Mission Control Software
UHF	Ultra High Frequency
MCS	Mission Control System
UI	User Interface
LED	Light Emitting Diode
IDE	Integrated Development Environment
OBC	On Board Computer
Backoffice	Front-end of MCS
Core	Back-end of MCS
RF	Radio Frequency
TDD	Test Driven Development

Table of contents

List of figures

List of tables

1 Introduction

TTÜ100 is a nano satellite program which has been in the development for a few years by primarily TTÜ (Tallinn University of Technology) students. It is also known as TTÜ-Mektory Student Satellite Space System, but got changed recently. As a lot of the mechanical and electrical engineering tasks are getting completed and the hardware side of the satellite is nearing its final form, a lot of the focus gets shifted onto software side.

The subject of the thesis is designing and implementing communication protocol on the ground station side. More specifically focus is on the creation of higher level commands for the communication. Most of the lower level commands were designed and partly confirmed too before the start of this work. The satellite side software implementation is out of the scope of this thesis, but there is a lot of cooperation with developers working on it to keep the design consistent.

The communication protocol consists of 4 layers. Connection is made over radio communication with 2 different types of antennas. AX.25 (Amateur X.25) is used to handle the radio links and it would be the second layer. The third layer in the protocol is bus command layer. It has to move commands between different boards in the satellite via a common bus. The fourth layer is a transport layer which enables moving bigger files over a connection that is available for less than an hour every day.

To enable communication between satellite and people on the ground, there needs to be functional ground station. Two most vital elements there are antennas with their controllers and MCS (Mission Control Software) with some UI (User Interface) (Figure 1). UI enables planning the mission and queuing commands. MCS handles most of the communication logic, including bus command and transport layers. Antenna controllers handle radio link and AX.25 part of the communication protocol.

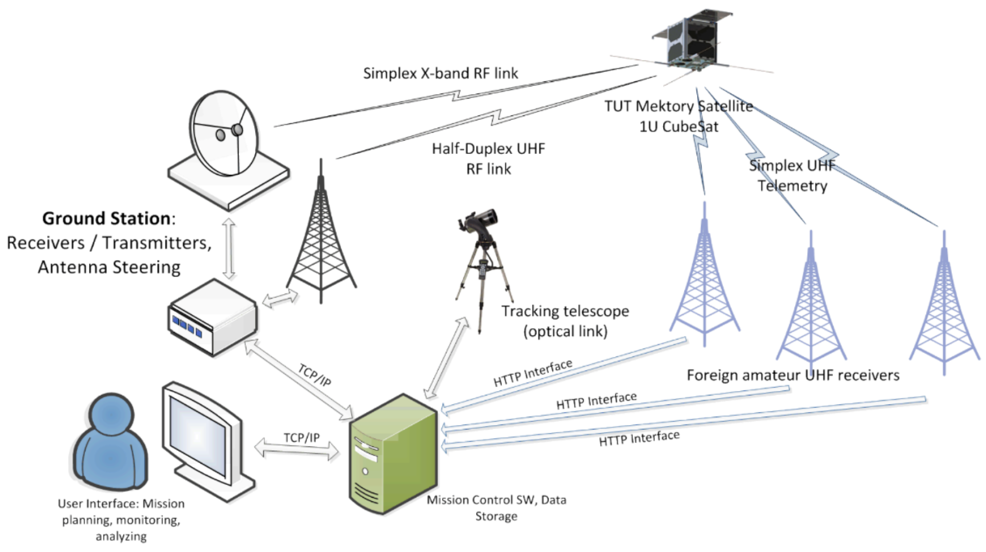
Simplex X-band RF link
TUT Mektory Satellite 1U CubeSat
Half-Duplex UHF RF link
Simplex UHF Telemetry
Ground Station: Receivers / Transmitters, Antenna Steering
Tracking telescope (optical link)
TCP/IP
HTTP Interface
Foreign amateur UHF receivers
HTTP Interface
HTTP Interface
TCP/IP
User Interface: Mission planning, monitoring, analyzing
Mission Control SW, Data Storage

Figure 1. Overview of planned space system [1].

The goal of the thesis is to implement the bus protocol layer and the transport layer in MCS. A lot of work has already been done there, mainly on UI and connection between different ground station parts. Some early prototype on bus commanding is working too, but it is not usable in the final version. This means that some or all of the communication protocol related code needs to be redesigned.

All of the coding work is done in Java as it is the MCS's primary programming language. Considerable parts of the thesis are actually not about the code creation itself but instead on the process of software development as a whole. Analyzing existing code in the project, considering its importance and checking for alternatives. All these are important parts and possibly even higher in the priority list than just writing code based on given design.

2 Low level communication protocol

The communication between the satellite and ground station is planned to be done over 2 channels. One of them is amateur radio band at 435-438 MHz and other is X-band at 10 GHz. X-band is more experimental and has a lower chance of working properly. One of the main reasons is that X-band is harder to test on and a lot depends on the big parabola antenna aiming in the right direction.

As there is a chance that the X-band will not work at all, the whole communication system needs to be able to work reliably on only amateur radio band's slow bitrates. This constraint is a key factor in defining how the whole protocol is built up. It does not play much of a role in a lower level communication protocol. Few hundred bytes of data making its way over from satellite to ground station or vice versa will happen almost instantly on even slower connections. It does play a tremendous role in higher level commands where hundreds of megabytes need to be sent.

In either case, AX.25 or also known as Amateur Packet-Radio Link-Layer Protocol is used to package the messages for radio communication. This works in both cases because both of the communication channels are over radio signals.

2.1 Importance of good rules

In communication of any kind it is important that all sides understand the info in the same way. The most crucial element would probably be the protocol followed. If the sides do not follow the same rules for communication, they are bound to fail to some degree. Even if the sides follow the protocol exactly, it does not necessarily mean that the sides will understand each other. That is because just having some rules does not mean that they are good rules.

A good communication protocol often depends on multiple agreed on methods to let the other side know that they are going to send a message or end it. In written text it usually is in form of a period which is added in the end of the sentence. It lets the reader know that there is nothing else coming and the sentence is now complete.

Computer communication protocols have a similar solution for a similar problem. The listening side needs to know when the message has ended. It is also important to know, if the message is just a string of text or a piece of code that needs to be run. It can have a big impact, if a piece of text is sent between systems and one of them considers it as a code that needs running. Executing such code can have absolutely no effect if the system notices that the code does not have a proper syntax for execution. In a worst case scenario though, it can cause billions of euros worth of damage by deleting a whole database.

One way to avoid such misunderstandings between systems is to define a way in protocol to notify the other side on how to interpret the message. An example of this would be a file format after the name of the file. Text_file.txt, image.png or code.java are all files and could even have the exact same data inside. However, the computer starts dealing with these files differently because of the file format after the dot.

2.2 Frames

One problem with passing any kind of data over an analogue channel is bits getting out of sync. It happens in both metal wires and wireless transmission. The reason being that real physical world is not as perfect as digital world. The sender might be sending data on a slightly higher bitrate than the receiver. On a short transmissions it does not really matter, but on longer sessions, the sender might be already sending 10000th bit while the receiving side is still expecting 9999th bit. The issue is worsened by the imperfections in channels themselves, possibly causing some noise and messing up some bits.

There are many options for trying to fix that issue. One obvious one would be to be more accurate with bitrates and make sure that the channels are clean. It clearly is one options, but in real world, it is a rather difficult task to keep all kinds of different channels clean for hours on end. For example, there could be a 1GB (gigabyte) file which needs to be transmitted. 1GB file would be 8 000 000 000 bits in total. Firstly, it is really likely that bit counters of receiver and sender are out of sync. Secondly, it is even more likely, that at least one of these bits out of 8 billion is wrong.

There are possibilities of figuring out some mistakes in the received data by checking the checksum. In case of a 1GB file it could become rather challenging.

The main idea is, that no matter how good the transmission quality is, it would not be enough to comfortably send large quantities of data, for example images or software updates for satellite. Even just one faulty bit there can cause a lot of problems. In case of an image, it can cause a bright spot, which could be interpreted as a bright fire on ground or just a glitch in the file. In case of a satellite software update, it can totally break the satellite and make it never respond to ground station again.

Another solution for the problem of bits getting out of sync is splitting up the data. If data is split up, each subsection of data can be checked individually with a checksum for example. It is also likely that in the smaller blocks of data there is no sync problem at all. Main reason being that the bit counting gets synchronized before each block of data.

For these reasons, most communication protocols rely on data splitting of some sorts. In the satellite project these lowest level blocks of data are called data frames.

2.3 Flags and escaping

In programming, pretty much every single character in the code has a meaning and an effect on the way it is executed. Exception for this would be if the characters would be between quotation marks. In that case, most programming languages consider it some sort of string of characters and does not consider the characters in there the same way.

There are cases where even between quotation marks, some characters have a special meaning for a code. There are two main options in this scenario. One would be that the characters are just forbidden and will always have an impact on how the code works. Second option would be to add a special escape character that can make an exception for such cases. Escape characters are usually made in a similar pattern. If there is for any reason an escape character somewhere, the next character after that is made into an exception. If there is a need to have a regular escape character in the text, then another escape is added in front of an existing one.

It happens to be that satellite communication cannot only rely on pure messages without some sort of extra layer of complexity with special characters. One could even say that in case of satellite communication it is even more important to have complexity to win on the bandwidth and other resource use.

In case of the satellite it is not too different. There is what is called a flag which usually is just some special character or set of characters. In the case of project such flag is defined as 7E in hex. It is 0111 1110 in binary form. It is always used as a first thing in the message to let the other side know that they should be listening and it is always used as a last part of the message to notify the listener that the message is over.

This all ties back to the idea that if there are some special characters in the messaged, then there also needs to be a way to escape them. Escaping the special characters is increasingly important for hex characters and even more so in case of defined binary pattern. Binary pattern used as a flag would need to be really long to ensure that it never is used inside the message. It is also really hard to check from higher levels if there is such pattern. One obvious reason is that it binary looks really different from hex or regular text strings and humans do not have intuitive understanding on how to convert them.

Another reason would be that binary patterns do not have clear boundaries, so the flag checks cannot be done by just checking each character’s or hex’s binary form. It is important to check everything continuously. All this leads to a need for a different way of escaping flags.

2.3.1 Bit stuffing

In one approach, bit stuffing as a concept is a bit harder to wrap one’s head around. If the problem is looked at from top to down, then it is tempting to start thinking that there is a need for a complex system that takes into consideration a lot of special cases where a flag is triggered by two characters being next to each other. Coding with all these edge cases in mind could be a nightmare.

For these reasons it is important to look at bit stuffing from the perspective of the really low level code that is already working with the data in its binary form. This code might not even know what the binary code was used for or from what kind of data it derived from. Often times it does not even see the whole file, but instead sees it all as a stream that comes in from one end, is modified a bit and then passed over to the other side without much saved in memory.

If the problem of escaping some binary pattern is looked at from this rather simple perspective, one can start to understand how to solve the issue. All this low level code has to do is to find patterns in binary code. It could be some complex pattern, but in most cases its kept rather simple to keep the code and design more simple too. One of the more common patterns for a flag in binary is to just have some amount of ones or zeroes in a row. For example it could be 6 ones in a row.

Now writing a code to count how many ones there have been in a row is rather simple. It just need to keep one number in memory and a bit of logic. Logic could get in a number and then it determines if is 0 or 1. If it is zero, then the code can set the one counter to zero. If the input is 1, then the code can increase the counter by one. A bit more complicated thing to do is to figure out how to escape a pattern. The sending side counts the ones and zeroes the same way as the receiving side, so it also finds the pattern. Now it can decide, if the pattern it found needs to be escaped or not. If it does not need to be escaped, then it is left as is. If it does need to be escaped though, the code has to just do something that would break the pattern. In current example, it could add a zero.

One might wonder now, how the receiving side understands if escape of pattern was planned, or if there was no pattern at all and no zero was added by sender. This is where following the same protocol comes in again. The protocol could define, that the sender always escapes the patterns that are just 1 bit away from being complete. This way the receiver can also always take away that zero. Once it does remove it, the code can see if there was a complete pattern being escaped or not. In either case, it does not really matter as the message got from one side to another without any confusion. All this is called bit stuffing as extra bits are added even in cases where there is no escaping happening.

There is one case still not looked at. It would be a case where there is a pattern of a flag and it is not escaped because it is meant to convey some other kind of information. In this case, the sender does not put the zero there to break the pattern. The receiver does the same pattern matching and tries to remove the extra zeroes added. If it finds that it cannot remove any zero, because there is a complete pattern without any stuffing in it, then the code can know that it is an intentional flag.

In most real cases, the pattern checking and bit stuffing happens to the main data and only then there are flags added around it all, this ensures that flags are not accidentally bit stuffed too. It is a bit of a limitation, because there is not much room for different patterns that do not also get too long. Also, if input data needs to have flags in-between, that further complicates the matter.

Bit stuffing is primarily used in radio communication, because bit stuffing works better if the data is coming in bit by bit. In radio communication, all the bits come in in a sequence one after another. It is not too common to send bits in any parallel form, even though it is possibly if multiple channels at different bandwidths are open.

2.3.2 Byte stuffing

Bit stuffing is not always the best way to escape flagged data. One reason could be that there is a need for multiple different flag types. Maybe even a different flag for start and end of the message. This reason is even more true when a programmer is considered. Making a code to deal with multiple different binary patterns at once can get a bit difficult and defeats the purpose of bit stuffing's simple nature.

It is often much easier for programmers to understand escaping special characters by adding some escape character in front of the special one. It might have something to do with how people start learning programming and making comments in their code that does not affect at all how the code works. In most languages, comments can be added by adding some special character in front of whatever the comment is.

Byte stuffing is in many cases a better option for escaping special characters in a rather low level messages. It is done by adding an escape character in front of a special character. The escape character itself is a special character as well so it would require the same treatment which means it needs to be escaped too.

With an example all this might be simpler to understand. Byte stuffing is possibly the easiest to understand with an example image (Figure 2) of how it works.

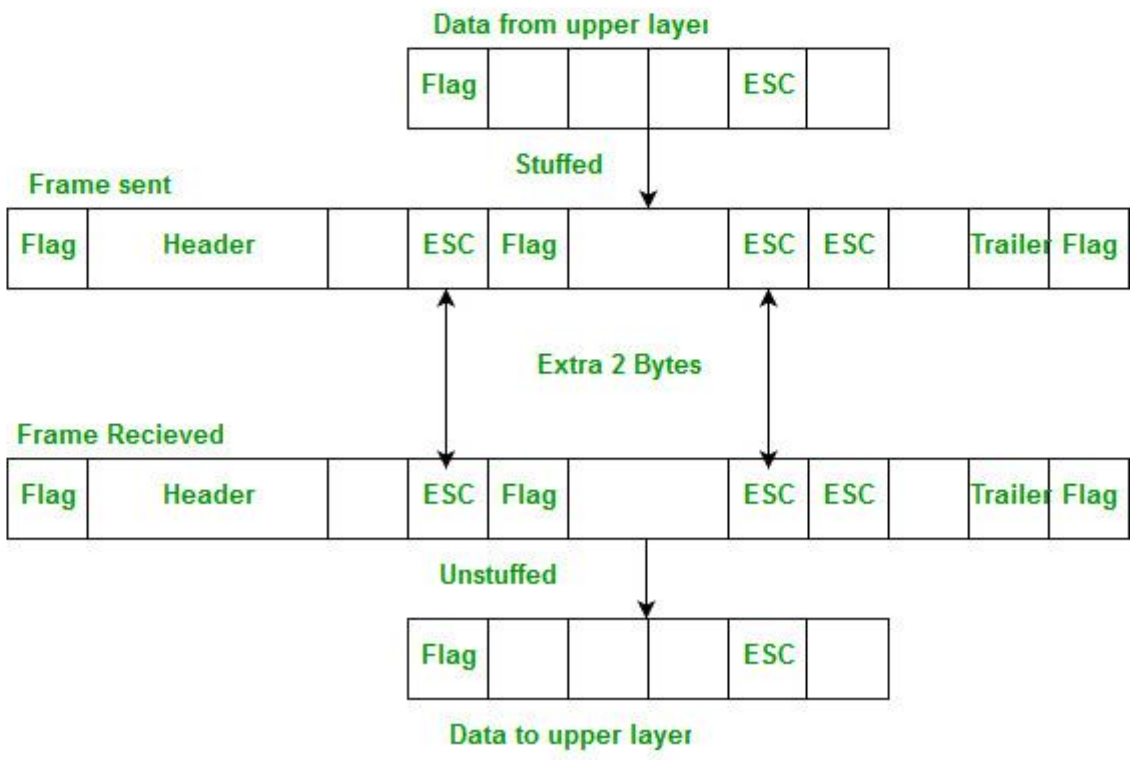

Figure 2. Byte stuffing example.

On image (Figure 2) it can be seen how escape character is added before both flag and escape character. The stuffing is done before real flags are added to avoid them from getting stuffed too.

2.3.3 Satellite project state

Throughout the life of the project, flags and escaping has caused a lot of confusion among people working on communication protocol or areas related to it. At the time when the work on this thesis started, there was an understanding among some of the team that every frame of data needs to be surrounded by 7 one bits in a row from both sides. Others did not even think about this. It is understandable, because most of the team's work is done on higher level like storing info in database, converting images or making a comfortable to use website.

There did not seem to be much understanding for why there was a need for these 7 bits, just that it was needed for communication to work and so these flags got added pretty much everywhere with a "to do" comment mentioning that it probably needs a better design. Not long after the start of the work on the thesis, it became clear that 7 bits in a row was not actually a flag. The flag in the project and in the AX.25 standard in general is actually 7E in hex which is 0111 1110 in binary. The flag only has 6 one bits in a row, but it is understandable where the confusion might have originated from, the seven in the 7E probably stuck to people’s mind better.

In the beginning, pretty much no one even thought about why there are flags in the project. As it was the goal of the thesis to further improve the communication protocol and its code, it was discovered that there are few bits of code which do add flags onto the commands passed over to the satellite, but without much context. It was soon pretty clear that there is absolutely no bit stuffing or byte stuffing done anywhere in the project.

The goal of this thesis was to mainly work on higher level protocols that allow transferring bigger files. It was definitely not to implement the lowest level systems for the communication protocol. Bit stuffing and byte stuffing both are really low level elements and without them, the communication in many cases would just break. For that reason, no significant work could be done on the higher level communication and the focus shifted towards the investigation of what exactly was the structure at the time.

It did not take too long to figure out that the flags added in the code were just temporary placeholders added in during a hackaton few months prior. This meant that bit stuffing, byte stuffing or both need to be implemented somewhere by someone. There were multiple documents collected in the wiki of the project’s repository, multiple of which were talking about bit stuffing. At the time it was not totally clear for what the bit stuffing was important, but it seemed like it had something to do with how radio communication works.

After multiple meetings with the team, there was an agreement that the bit stuffing will be done by someone who already works close to the radio communication. The task moved from core of ground station code to an independent and rather isolated code at the antenna. This separated code primarily dealt with converting raw analog signals into bits. As it already has data in its binary form, it seemed most fitting to also do any kind of bit stuffing there.

The plan was set and a few months of work got done on other parts of the project with an assumption that the bit stuffing will work at some point, it is dealt with by some lower level code that in a totally different place.

During the last few weeks of the work on the thesis it came out that the understanding of the whole bit stuffing and byte stuffing among the team has been rather inaccurate. First signs came out when work on other part of the project was conducted. Software uploading systems, they rely heavily on a properly implemented communication protocol. It was noticed that the uploading system fails to upload, because of what seemed to be bit stuffing problem.

This bothersome discovery turned the attention of this thesis back to flags, escaping and bit stuffing. On investigation with some fresh info about the requirements and with a better overview of what these systems are for, the team reached out to people who originally designed the protocol in the project. After a lot of discussion, it became clear how the systems should work instead and what still needs to be implemented.

It came out that there is a need for both bit stuffing and byte stuffing. The reason being that bit stuffing is used in a radio communication while byte stuffing is used when different parts of satellite communicate to each other over a serial bus. During this thesis, one important thing to do was to clear out misunderstandings of the communication protocol. Continuous pressure was applied to the whole team to dig deeper on the problem and in the end it seems it managed to catch some dangerous problems before they were able to cause any harm.

2.4 AX.25

AX.25 is the lowest level protocol used in the satellite project. It does not define which channel it is sent through, because amateur radio band and X-band are both radio frequencies. The AX.25 protocol is a bit modified to fit the requirements of the project more accurately.

Flag	**Address**	**Control**	**PID**	**Data**	**FCS**	**Flag**
1 (0x7E) Byte	14 Bytes	1 Byte	1 Byte	Up to 256 Bytes	2 Bytes	1 (0x7E) Byte

Table 1. AX.25 frame structure.

Lower level protocol will be responsible for transporting higher level packages. It is often but not necessarily always done by cutting a higher level package into smaller parts.

AX.25 was initially thought to be finished. A bit after the work on the thesis started, it became clear that AX.25 is nowhere near as complete as it would need to be. The frame structure (Table 1) and the protocol around it was really solid. Probably because it is not originally designed by the satellite team and is not done for this project. The main issue was that it was not really implemented or even designed anywhere in MCS.

Part of this thesis was to figure out where and why the AX.25 should be implemented in. First plan was that it should be done in the core of the system and be together with all of the other command building. In a short term, it might have been an easier solution and some work even got done on it. This coding work was a totally wasted time, but at least it brought some more attention to it.

After some analysing and discussion a better solution was reached. AX.25 should not be a part of any command inherently. Only reason for AX.25 to exist in the project is to help with moving data from ground station antenna to satellite antenna, so radio communication. There is no reason for this layer to be present in anywhere else in the system. Databases should not store AX.25 wrappers with the command. User interface should not even know AX.25 exists. No core communication logic should have to deal with such a low level layer.

The task of implementing AX.25 got moved to antenna system which keeps listening for bus commands in hex form from the ground station side. It makes the command into a binary form, wraps it in AX.25, bit stuffs the whole thing and then adds flags for radio communication. After this, the system is ready to send it all as a radio signal over to satellite. Core of MCS should not even know that AX.25 is used somewhere in the middle.

2.5 Bus protocol

Satellite is not a single complicated piece of technology that does absolutely everything it needs to. Instead it is made out of multiple boards with each doing one dedicated task. It obviously makes it much easier to design and manufacture any of these parts independently. It does require some extra work on making all these systems talk to each other, but it is small in comparison to the single board design.

Solution used in the satellite project is to have a bus to which all different boards are connected to. Each of the boards has its own address and if some command is written on the bus they know to which it was meant for. This means one more layer of communication protocol though.

The bus never gets an AX.25 frame written on it. Only one of the boards even knows what AX.25 is. It is a communication board which is responsible for radio communication. Rest of the boards see it as a ground station which is giving new commands. This analogue goes the other way around too. MCS can port bus commands to the antenna system as if it was the bus connecting ground station and satellite boards.

Bus command structure is shown in Table 2 [2].

Source	**Destination**	**Control**	**Code**	**Data**	**IFCS**	**Auth**
1 Byte	1 Byte	2 Bytes	2 Bytes	Up to 250 Bytes	2 Bytes	2 Bytes

Table 2. Bus command frame structure.

Two similar looking commands written on bus can have really different meanings. To make the task of distinguishing between command types easier, a command code is used (Table 3). As part of this thesis was to implement burst control, a new code “L3” got added to this list.

Bus command	**Request hex**	**Response hex**
L3	01	41
Parameter read	03	43
Parameter write	06	46
Multiple Parameter write	10	50
PIC memory update	10	50
Read memory checksum	-	-

Table 3. Bus command types.

Just like protocol of radio communication with AX.25 did not change during this thesis, the bus protocol did not really change either. A lot of work got done on its implementation side though (p 24).

3 Mission Control Software

MCS (Mission Control Software) is directly worked on by around 10 people. It is a rather new part of the whole satellite project, because usually work on the software can begin once some or all of the work on the hardware has been done. Some of the sub-systems in MCS are not too critical for the life of the satellite itself. The satellite can probably orbit the earth and keep itself alive for a while without MCS at all. For the whole project to be considered successful though, the satellite has to be able to communicate with the ground station as well. That is where the main responsibility of the MCS lies.

3.1 Initial observations

The state of the MCS before the work on this thesis seemed rather good on the first glance. The java project was already massive, compared to most projects done during the years in school. The project needed excessive setup with VMs (Virtual Machines) and database running in Docker [3]. Just getting the project open and running in Intellij [4] took around an hour at best. That was in cases where the person doing the setup already knew what the potential problems are and how to solve them. For a person unexperienced in what Docker is or what it is used for, it could easily take days of messing around.

Once everything was up and running, the user was welcomed by a long list of modules and packages, most of which did not make any sense. During this thesis, a somewhat decent understanding of the project was constructed. One or two months are definitely too short of a time to get an understanding of the different parts of the code and why exactly they are even necessary to have. It could be enough of a time if the person working on it is learning the code full time or is already really familiar with frameworks and applications used.

A possibly really big mistake done during the first few months of the work on the thesis was to just jump in and start implementing systems based on the understanding at the time. It caused a waste of countless hours on systems and code that in the end were not doing the required tasks too well and often times did not even do the right thing in the correct place. One could say that people should learn from their mistakes and the time was still well spent because it have a better understanding of the whole structure.

One good example would probably be work from the very first weeks when lower level commands were still being implemented. One of these commands was LED (Light Emitting Diode) command. Its goal was to blink a LED light on the satellite. It is a rather handy command to have for testing if the communication with the satellite is even working and if it even is receiving any messages.

The problem with working on the LED command was not that the command itself was not useful, but rather that how it was implemented. At the very beginning, it was not simple to understand what part of codes could be touched and which should stay as they are. Safe approach would be to just not change any existing code and just assume everything already written is good and well justified. Issue was that neither of it was true.

Firstly, the code was a bit difficult to read. Field and class names did not make much sense at the time. That is not really much of an issue though because after getting a better understanding of the project and its concepts, names start making much more sense. Refactoring the naming is not a difficult task at all nowadays with modern IDEs (Integrated Development Environment) so this not really a big issue needing much attention.

Secondly, the code was not so well structured. Many of the design choices were rather questionable. It is not easy to notice these problems early on though, because odd design decisions can be totally justified with a proper explanation. If these problems are not addressed early enough, they can start piling up as more and more code is written to fit in with the existing bad code, effectively spreading the "bad smells" of the code.

Thirdly, the code was not even following the already written documentation. For one reason or another, the code written was with no regards to what the design documents had stated. Some of really core principles were disregarded, passing issues further into the future.

3.2 Weak structure

Few hints for a bad structure were noticed early on, mainly due to the bit stuffing and byte stuffing questions. There were 3 primary areas in project where work was done to get the LED command working. One of it was bus protocol module, other was backoffice and last was socket to serial module.

Bus protocol module contained most of the logic for the commanding. It had classes which were used to turn input data into hex commands in string form. For example, an integer was passed into a parameter read class. It was a number of register to read. The class made a whole command around it. It added necessary source and destination addresses, the code of the command, registry number to read and a check sum. The class also enabled to get a hex in string form. The class itself did everything right.

Backoffice was one of the more problematic one though. Its goal is to deal with most of the front end side. For example, it takes in user input which wants to send a command to the satellite. On paper, backoffice should have no idea on how to build actual commands. It should only pass the user input to some system which deals with hex command building. Problem was that backoffice used bus protocol module really directly and even added parts of hex to the commands. Another problem was that the backoffice posted the commands to the satellite system directly. All this gives a really messy system where a lot of logic is piled in one part and also causes a possible security risk as front end connects to satellite rather directly.

Socket to serial module was a part of the project which is now moved to antenna system. It actually did not have much wrong with it. The module was just removed from the core of MCS as it started to really grow feature wise and it really did not do much of the higher level logic. It only passed hex commands from ActiveMQ to the bus of the satellite or into a radio. It was more of a temporary solution to get something working before this thesis started.

Largest issues with the code design were related to front-end and back-end not being sufficiently separated. After discovering the problems fixing them became one of the main goals of this thesis.

3.3 Documentation and implementation mismatch

One could say that the system is actually well designed and it is just understood in a wrong way. After all, the whole system was built by a much more experienced developers. On the other hand, it was mainly just one developer, which means that it is harder to notice some issues.

Most of the problems with MCS mentioned this far would not really get too much attention if the design documents were supporting the actual implementation. It is not the case as a lot of the documentation is different from what is actually done. It is especially true on parts that were noticed earlier as problem areas.

This gave motivation and reason to dig deeper and see what could be fixed. This lead to a lot of refactoring and redesigning work. It also meant that the high level commanding had to be pushed further into the future. With that much unexpected fixing to be done in different parts of the project it was clear that burst control and especially large file uploading and downloading systems would not be operational by the end of the thesis.

4 Redesigning

Throughout the work it became more clear that a lot of effort needs to be put into refactoring and redesigning different parts of the project. At first it seemed as if low level frame management needed to be done. Bit stuffing and Byte stuffing seemed to be missing. Some of these tasks got pushed to other areas of the project for other developers to deal with them.

While trying to implement higher lever communication protocol, a lot of the lower level design problems started getting in the way. As it all was still part of the communication protocol, it seemed really fitting to deal with the problems in this thesis too.

4.1 Frames in layers

One of the first things needing attention and causing issues was the layering of frames. It was known that AX.25 is the lowest level wrapper for data, but not much else was too concrete after that. Bus commands had somewhat solid design too, even though there was already some confusion between bus commands and AX.25. L3 had a command protocol designed, but that was it. Anything higher than that had only some faint ideas floating around.

The frame structuring got quite a lot of attention early on and after multiple meetings and discussions, a design was agreed on (Figure 3). The example in the figure is done as a part of this thesis.

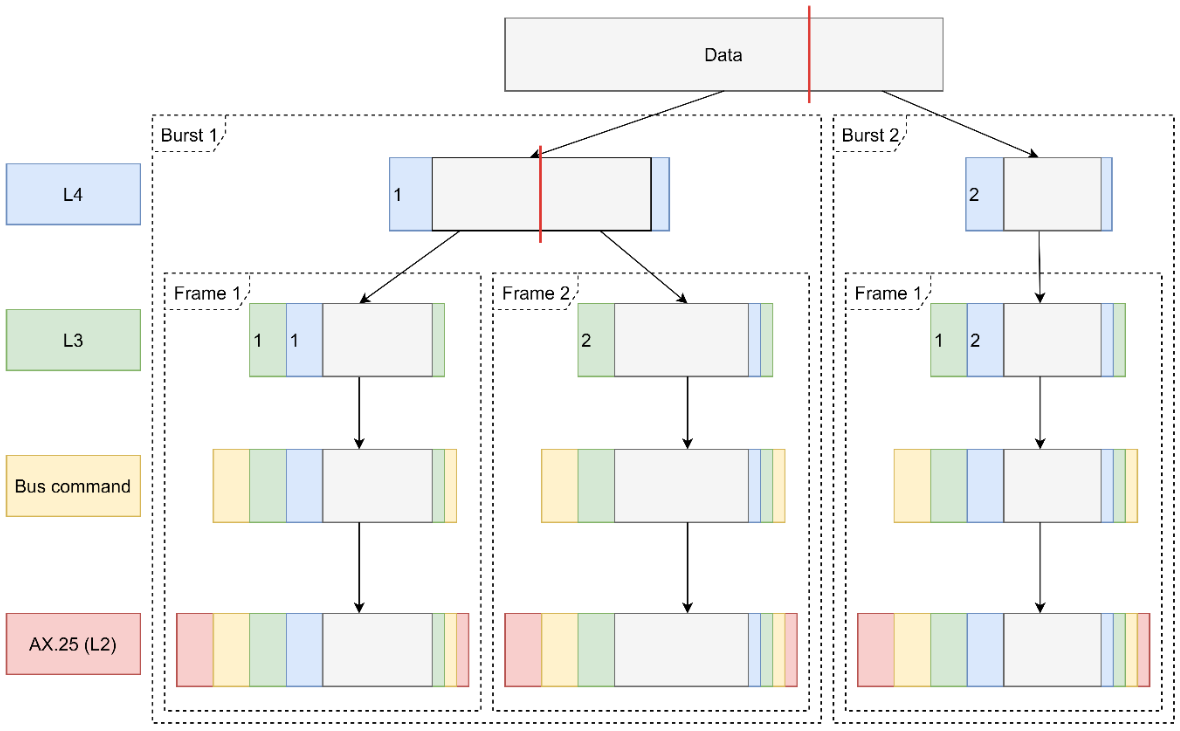

Figure 3. Frame build-up through layers.

The figure is just to give an example of how in a more final version data is split up among frames. The red line indicates, where the data is split up to fit it in lower level frames or bursts. Lower level frames have a limit on how much data they can have in their body, meaning that higher levels have to consider it too.

Bus command has to be complete in every single frame as it is almost like an extension of the radio communication layer AX.25. Only complete frames can move over the bus.

L3 is actually a part of bus command layer as it also needs to have its frames complete when traveling over bus. The L3 frames are only moved between OBC (On Board Computer) and radio board. It is used to move data between ground station and satellite OBC in larger bursts. Up to 64 kB of data can be moved from ground station to satellite in one burst, meaning that the satellite only has to respond in the end. This means that a lot of back and forward communication can be avoided as a report is made in the end on how many of the frames actually made it and which ones should be resent. L3 design is to some extent done during this thesis.

Many files are bigger than 64 kB and due to the possible connection speed limitation, even this much can be a problem at times. This problem is solved by L4 communication protocol layer (p 34).

4.2 Front-end and back-end separation

Possibly the biggest issue in the communication protocol implementation in ground station was that the front end was pretty much communication with the antenna to the satellite directly. This is not only a possible security threat but also weak and definitely incorrect design compared to some provided documents. Before any real implementation work on the high level commands could be done, the low level commands need to be implemented well.

4.2.1 All commands through core

The first step on fixing the design was to make sure that backoffice never sees the antenna system. All of the communication had to go through what is called core. The separation itself is not too complicated (Figure 4).

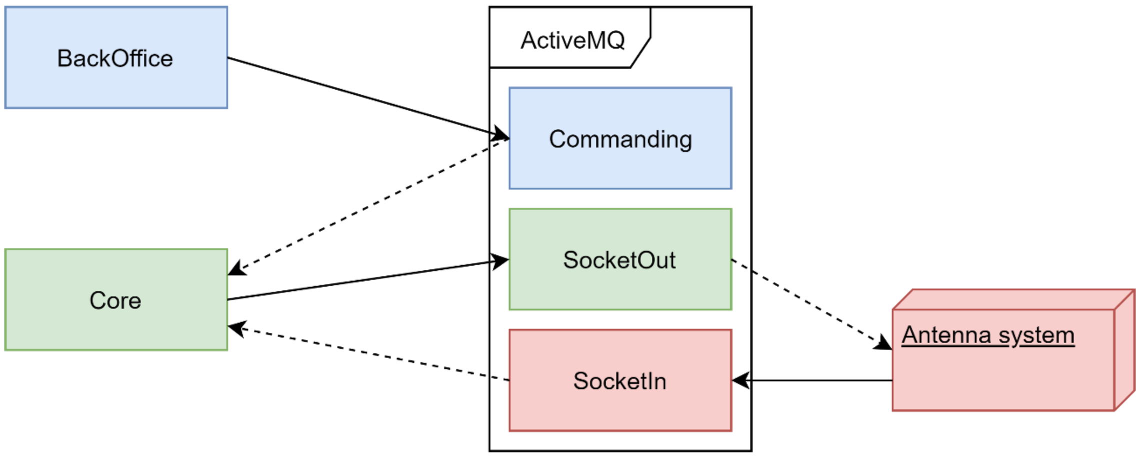

Figure 4. Data flow through ActiveMQ topics.

In old system, the backoffice made a hex command out of user input and then posted it in ActiveMQ topic. Antenna system was listening to this particular topic and just passed anything from there to antenna to send it.

In the new system, the backoffice takes the user input and passes it to another topic. This topic is only listened by core and not by antenna system. This ensures that front-end does not reach the satellite that directly and also enables multiple different systems to give commands to core. For example there could be two different front end solutions, both of which connect to the same back-end solution, meaning that the commands end up being same. Also some higher level commands, probably even L4, need to use this path to give out commands.

4.2.2 Command types

The second step was to define types of commands that the core can take in. This depends a lot on what kinds of commands are sent to the satellite. As the project is not final yet, it cannot be said what commands exactly are needed. It can be known which ones definitely already need to exist (Figure 5).

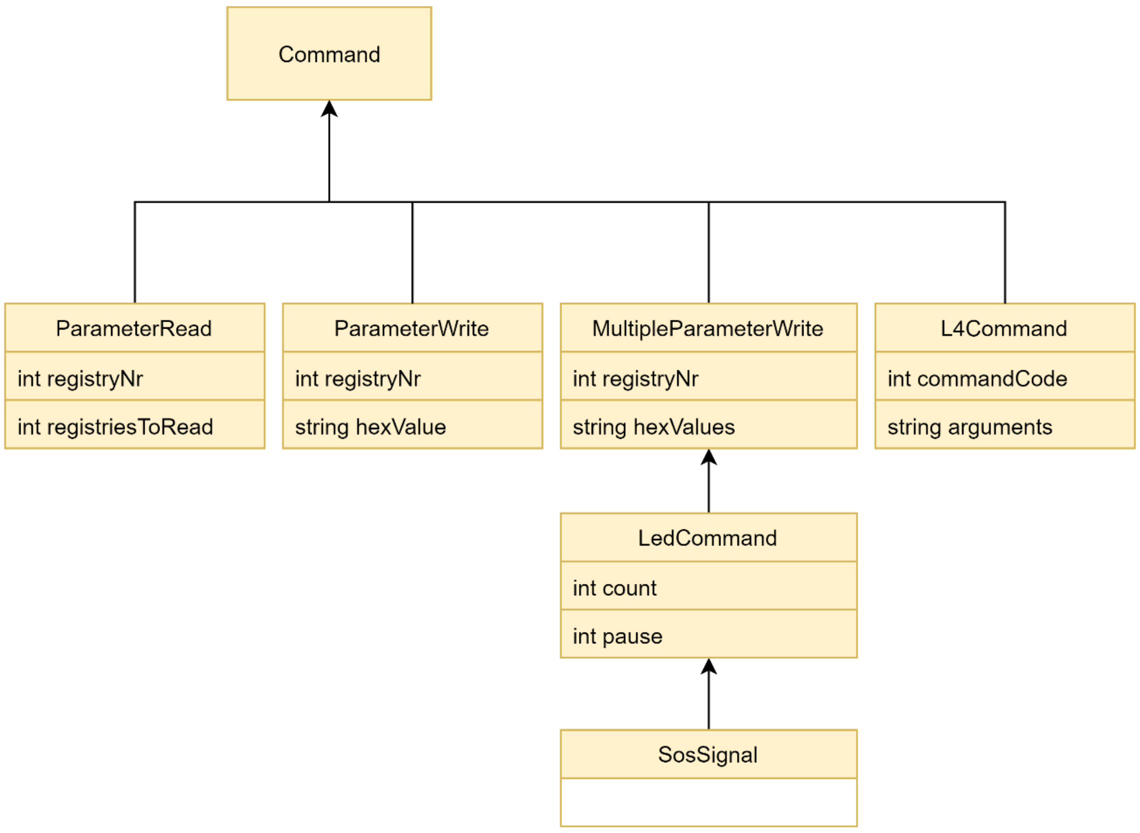

Figure 5. Command types.

The different command types can be defined in a class. Command base class can have some more general data like send time in it, but it is not yet decided on. It is not too difficult to add these fields later on without breaking anything, because the definition of commands is kept in one place only.

Parameter read, write and multiple parameter write are all commands which are already implemented, but due to weak design, they need some refactoring. However, it is clear what these commands do and what inputs they need.

LED command is generalizing from multiple parameter write because in reality LED command actually writes some predetermined registers. It does not really affect how front-end passes command info to back-end, but it was insisted by the team that it would follow this structure to reflect how things work in actual satellite.

SOS signal command is just a special case of LED command. Hex values that are written in the registers are already predetermined in a way that if the command is triggered, LED light starts blinking in morse to signal SOS.

L4 command is a bit less defined, but it is clear that at some point it is needed. L4 commands have different sub-commands to it (p 34). As more focus is on fixing the actual systems, less time got invested on detailing out how higher level commands are presented in command types.

4.2.3 Spliting up core

Separation of front-end and back-end is not the only thing that can increase the readability of the code. By this point, there is still a huge deal of tasks done inside core. These tasks could be broken away and made into their own module of code.

One thing which already was separated from core in the initial implementation was bus command module. It was actually called but protocol module. This module already had code to make a hex bus command from some input values.

The plan is to improve the simplicity of core to not really convert the incoming command objects into some arguments. It can be done by making bus command take command objects in as arguments (Figure 6). The mind-set is that core should not really know how hex commands are built or what exactly is needed for that. For this reason a decision got made that bus command should be only able to take in command objects as arguments. All that core needs to know is which commands should go to bus command and which should not.

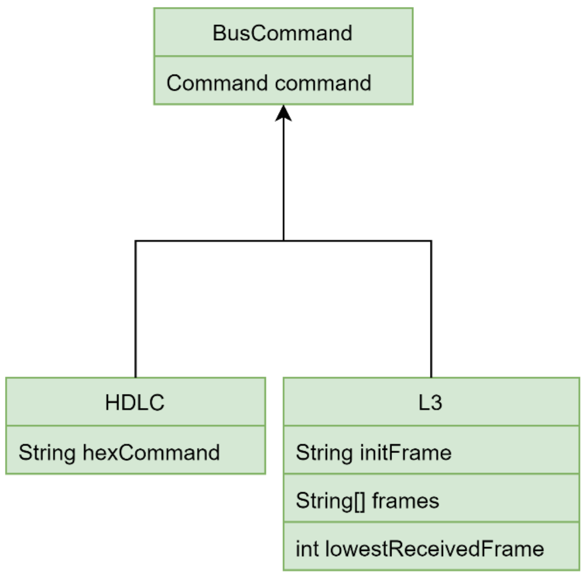

Figure 6. Bus command types.

There are ideas of maybe breaking up the core even more in the future. A name “core” is not too descriptive of what it actually does. It also is possibly that it still does too many tasks in itself. The most important task in current core is to send and receive bus command messages over to antenna system. Not all commands are bus commands, but if anything is sent to antenna to be sent to satellite, it needs to be inside a bus command of some sort. In most cases the commands will be transported in L3 or L4 if they are bigger.

The core should not deal with L4 package (Figure 7) or even L3 burst frame management. It should only sort the incoming and outgoing commands. To make any further decisions on the design, L4 protocol would need to have a somewhat solid design. Otherwise a lot of design work on one or another side needs to be thrown away.

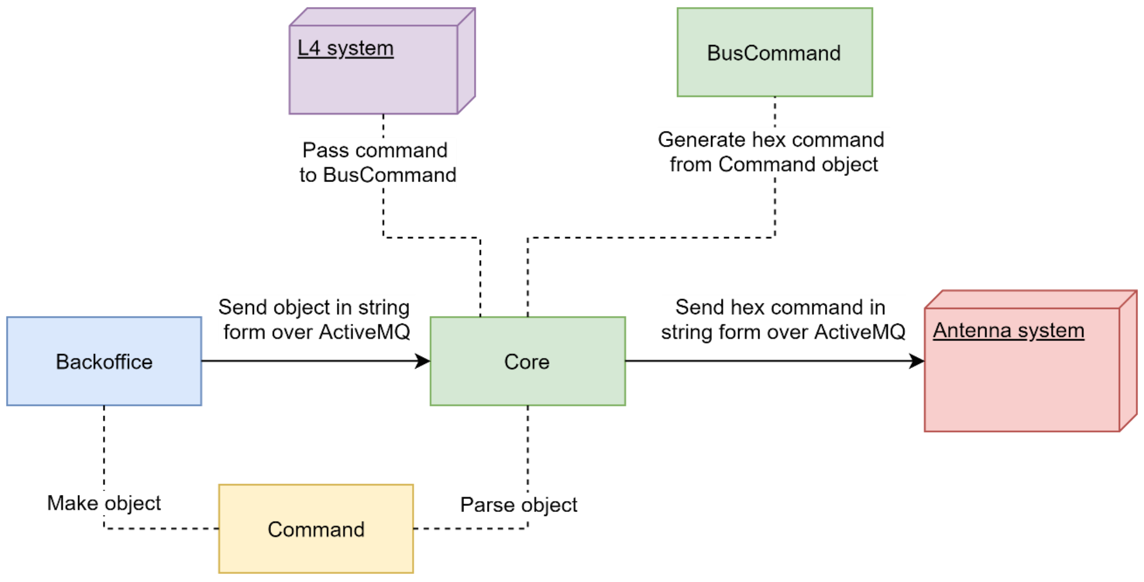
L4 system
BusCommand
Pass command to BusCommand
Generate hex command from Command object
Send object in string form over ActiveMQ
Send hex command in string form over ActiveMQ
Backoffice
Core
Antenna system
Make object
Command
Parse object

Figure 7. MCS communication protocol design proposal.

One possible change to the proposed design is that L4 commands do not go through core at all. Backoffice would send the command directly to L4 system and it would make L3 commands accordinly. Only these L3 commands are sent to core which in turn makes the these commands into L3 bus commands. As mentioned earlier, L3 is just a variation of bus command (p 27).

5 L4 design

L4 is a communication layer that has to be able to manage even bigger data flows than L3 bursts. It has to be able to take a file in any size and make sure the file gets to the satellite or to ground station. Making this layer take in however massive files does not sound too wise of a decision, but it does not have to. The layer has to be able to move files that are just bigger than the communication would ever need. The size limit is not yet decided on.

All layers up until L4 have to get their task of transporting data done within one fly over. This means that if the satellite moves behind the horizon and the connection is lost, the L3 has to have successfully moved the burst. If it has not, then on the next fly over, the data transmission has to start over from zero.

L4 on the other hand is meant to deal with this exact problem. It splits given data in small enough packages that they fit in L3 bursts and can be passed over to the connection in one fly over. The layer keeps track of packages and is able to continue after a long pause.

Command	**Hex**	**Binary**
List	0	0000
Remove	1	0001
Get	2	0010
Put	3	0011
Continue	4	0100

Table 4. L4 command codes

5.1 Example scenario

Ground station sends out L4 command "get Some_File.png" with an ID of 1. The command is short enough that it should fit within one L3 burst. The response is too long to come in one burst. For that reason the file needs to be sent over multiple L3 bursts. In this example, the file needs 3 bursts.

As L4 might not be able to send all of these 3 bursts in sequence over one pass over, the Ground Station should also tell the satellite, how long it can talk or how many bytes it is allowed to send. (Maybe L3 already knows how long it is allowed to respond.)

In current example, the L4 can only send one burst as a response before the Ground Station gives it further instructions. Before sending a request for next L4 package for the file Ground Station decides that it needs to get more important command done first.

Ground Station makes a new L4 command "lists" with an ID of 2. The command is sent to satellite, it executes the command and sends the response with the ID of 2 as well. Ground Station received the L4 response and knows it is not for the file but for the "list" command because of the ID.

Now as Ground Station has nothing urgent to do, it sends another L4 request out. It sends a "continue" command with the ID of 1 and the received package amount which is 1. Now satellite knows, that it needs to continue sending an already existing L4 command's response. It also knows that the L4 command in question has an ID of 1 so it needs to continue sending “Some_File.png”. It also knows that 1 package is received so it can send next chunk of data through L3 burst.

Should satellite be able to send multiple bursts without being told to do so? It might be a bit simpler solution if for each next burst Ground Station needs to actually send out and L4 "continue" command. It also allows the data transfers to be stopped more easily as the satellite cannot talk for too long. Down side is that this would require more bandwidth for the continue commands.

Once 2nd and 3rd burst have arrived in Ground Station, they are a complete file and a notification is sent out. Each of the L4 packages is contained in a database and can be accessed before the whole file is received. Some file formats can have meta data in the beginning of the file. Some file formats can be partially read, even if the whole file is not there yet.

6 Deleting already done work

During this thesis, a lot of code got thrown away. It might not seem like the redesigning process was a hard one. The diagrams and solutions presented might sound reasonable and make one wonder, what took so long to make these.

First step on getting these redesign proposals done was to even understand, what the whole project and MCS is about. Like mentioned earlier (p 23) learning about all this was rather time consuming. Weekly meetings helped to make the process faster, without them the thesis would not have made any progress probably. All that was not enough though. Often times work had to be stopped until next meeting to ask for some help from other developers. In the beginning it was a bit taunting to change anything without consulting with others, because there was always a fear that something will be totally broken after the change.

This problem got eleviated after some time and work started to speed up. L3 protocol was implemented in to a point where it was able to make proper frames and pack them into a burst. The burst handling was thoroughly discussed with other team members too, especially with people working on antenna and satellite communication systems. The L3 code was able to receive awknowledgement messages and know how to respond.

It was developed to the point that some testing almost started. Plan was to start testing the newly written code against the L3 code from satellite end [5]. Idea was that the 2 implementations should be able to communicate with each other in a test environment and simulate a scenario where they would talk to each other through radio and bus.

The plan was to start small and just test what the satellite L3 code responds to ground station's L3 code. Few modifications were made on the new code side to better match with the code on the other side. The newly written implementation output hex strings while the satellite version handled everything in binary boolean array. This was not really much work to do though.

Some problems started surfacing on the satellite side though. Problems that are not really in focus of this thesis. Their affect was rather big though. The plan was to get first tests going 2 to 3 months before the end of the thesis. By the time this thesis got finished up, the tests still did not get going. On one hand it is a really bad thing, because the implementation got no confirmation for its usefulness and all further development on that exact side stopped.

On another hand, it is possibly a really good thing. It redirected focus from improving an isolated system towards making the system actually work with rest of the code in the project. On first glance it seemed like a task that could be done in a week at most. That was a huge over-estimation.

Not long after that it became more and more obvious that integrating the written L3 code into existing MCS core code is not an easy task. The problem was not in the L3 implementation, even though it could possibly have been better too. The issue originated from the code that existed previously (p 24).

Throughout the process of trying to make L3 work with the existing code, a lot of extra code and refactoring got done. Each time it ended in a failure as more problems were noticed. With every attemt it became clearer that simple refactoring would not do the trick and actual redesign of existing systems needs to be done (p 27). Most of these half-measure attempts are still somewhere in the code repository, either in unfinished branches, commented out somewhere or only accessible in older commits.

Rougly a few weeks worth of code got even deleted from the repository. At the time the code seemed really useless and was going in totally wrong direction. It was true, but it was not bothering anyone there, being in a branch that never merged to master branch. A few months later a bit of that code could have possibly saved a teammate from some extra work.

Bit stuffing and byte stuffing were parts of code that caught attention early on. For that reasaon there were multiple attempts to do that. Each try ended up with some useless code and a bit better understanding of what needed to be done any why. This could have been possibly avoided if more time was invested early on into research of the problems.

7 Validation

In the beginning of the work there was a lot of testing done on the code written. It was not quite pure TDD (Test Driven Development), but the attempt to be close to that was there. In many more complicated scenarios, expected outcome was defined before the implementation was actually made.

This approach proved to be quite effective for implementation of extra bus commands like LED command. The structure and expected output were well defined. Burst control got some testing too. It was making sure that data was well split and if the input data got too big, it threw an exception.

At some point, issues about bit and byte stuffing appeared and that is when the written tests started being less useful and in many cases more of a drag. Problem was that almost all of the written tests suddenly had a different expected outcome. This meant that even if the change in the actual code was simple, the required changes in the tests to pass were quite numerous. Just changing the expected value to what the actual value was after the change is pretty much defeating the purpose of TDD.

Not long later it became more and more clear that the whole structure of bus commands in the MCS needs to change. One change after another the written tests started becoming totally obsolete. Many of the tests did not make any sense in the new design, this means that they just got commented out and soon after deleted.

This lead to two important things to learn from this. One is that it is probably not too wise to make really simple tests. Sometimes it might be a better idea to mix a few really simple tests into one a bit more complex test, just to save time. An example would be a tests for null string, empty string and too long string, all of these could be in one test together, because the programmer would not have much trouble figuring out, what is causing the issue from there.

Another thing to take away is that it is no point in testing code written on flawed or rapidly changing design. The tests can check if the written code is following one exact requirement. If the requirements themselves change often then all of the tests require constant updating too. It seems like a better idea to just spend more time on figuring out what is needed and then start implementing anything. It is hard to implement an unknown.

Not all validation was done with regular unit tests though. There was a plan for quite a while to test the L3 implementation against already completed satellite L3 implementation. It never got to it because there were several problems on the satellite code side. Once one problem got solved, another appeared and this has been going on for a month or two. If at some point the code is running well though, some tests can be done to validate the correctness of the L3 implementation written as a part of this thesis.

Hardest part to validate is probably the overall structure of the MCS and its new proposed design. Even a bad design can be made to work, it might not be too optimized or maybe safe. If the requirements do not include priorities for them then it is not necessarily a failing. Sometimes even the requirements themselves change and it is even harder to validate if something is required for a higher level goal.

Before the start of redesigning process for the MCS, a few high priority requirements were made. One was to follow the general design shown in some of the previous system requirements documents for the satellite project [1]. Separation of front-end from back-end was the most important one. It was the part that was wrong in the old prototype design and was causing the most issues. As the new design was done, this requirement was the first one to consider and so got achieved rather early. As new elements got added, the whole system design was double checked to make sure that the front-end does not send any commands directly to the antenna. This requirement has succeeded.

Another requirement was that there should not be any duplicate code in the new design. This requirement was more targeted towards fixing an issue where the bus commands had at least 2 classes that did pretty much the same things, but were positioned in different parts of the project. This got fixed in the end too as back-end provided a list possible commands for any other module.

The most important requirement was to get the satellite and ground station communicate in bursts or share massive files with L4. This requirement is not filled. Only a few smaller bus protocol commands got added. L3 got some implementation as well but due to difficulties with MCS structure the implementation was never able to communicate totally automatically.

8 Summary

The thesis did not achieve its initial goal which was to implement L3 and L4 communication protocol layers in MCS. However, it possibly managed to do something much more important and save a lot of work hours in the future. If the high level commands were somehow implemented into the system, then it would have caused some even bigger issues later on.

Throughout the work on the thesis, several major issues were noticed. They were not noticed right in the beginning, but early enough that they could still be dealt with within this work. This of course meant that some of the work done was not too useful to the whole project. These possibly meaningless tasks brought more attention to the flaws in the project and also were a great learning tool on how the whole system is built up.

Bit stuffing and byte stuffing have been a complicated topic in the team for a long time. Code in multiple parts of the project was written in a way that would have broken communication between satellite and ground station. A lot of work was done to research the topic and clear out most of the confusion around the topic in the team.

Most of the existing communication protocol implementation in MCS had to be redesigned as the support for higher level commands was lacking. Furthermore, the system was hard to work with due to lack of some clean code principles like separation of front-end and back-end. It could have even caused security holes in the system.

Main focus of this thesis was to fix the issues with the MCS design. Most critical problems got addressed, researched and fixed on a design level. Actual coding based on the new design still needs to be done, but that should not start before some further designing of higher level commands. Once there is good enough design done, a decently good developer should not require more than a few weeks to implement it all.

Some work got done on L4 and L3 command protocol layers too. L3 burst control got full attention while designing the new systems. L4 was considered too, but as it does not have a final command protocol yet, it cannot be expected that the new system considers all of the details.

There is still a lot work to be done on command protocol before it is good enough to use in real life scenario. However, this thesis should provide a good continuation point for any further work.

References

[1] "MCS general requiements documentation," Tallinn, 2015.

[2] TAPR, “AX.25 Amateur Packet-Radio Link-Layer Protocol,” [Online]. Available: https://www.tapr.org/pub_ax25.html. [Accessed 02 05 2018].

[3] Docker, "What is a container," [Online]. Available: https://www.docker.com/what-container. [Accessed 18 05 2018].

[4] Jetbrains, "IntelliJ IDEA," [Online]. Available: https://www.jetbrains.com/idea/. [Accessed 20 05 2018].

[5] D. Rodionov, "Implementing TTÜ Nanosatellite Communication Protocol Using TASTE," 2017. [Online]. Available: https://digi.lib.ttu.ee/i/?8026. [Accessed 28 02 2018].

[6] "The Cyclic Redundancy Check (CRC) for AX.25," [Online]. Available: http://practicingelectronics.com/articles/article-100003/article.php. [Accessed 02 05 2018].

[7] "Computer Network | Difference between Byte stuffing and Bit stuffing," [Online]. Available: https://www.geeksforgeeks.org/computer-network-difference-byte-stuffing-bit-stuffing/. [Accessed 15 05 2018].

[8] S. Romanov, “Kuupsateliidi missioonijuhtimistarkvara arhitektuur,” 2017. [Online]. Available: https://digi.lib.ttu.ee/i/?8028. [Accessed 28 02 2018].

[9] Docker, "Docker: Get started," [Online]. Available: https://docs.docker.com/get-started/. [Accessed 18 05 2018].

[10] Apache, "ActiveMQ," [Online]. Available: http://activemq.apache.org/. [Accessed 18 05 2018].

[11] Wikipedia, "Hyper-V," [Online]. Available: https://en.wikipedia.org/wiki/Hyper-V. [Accessed 18 05 2018].

[12] Wikipedia, "AX.25," [Online]. Available: https://en.wikipedia.org/wiki/AX.25. [Accessed 18 05 2018].

[13] The TCP/IP Guide, "IPv6 Fragmentation Example," [Online]. Available: http://www.tcpipguide.com/free/t_IPv6DatagramSizeMaximumTransmissionUnitMTUFragment-4.htm. [Accessed 18 05 2018].