GPS Time Spoofing

A detection and mitigation system for GPS timing

Aril Schultzen

Thesis submitted for the degree of
Master in Informatikk: programmering og nettverk
60 credits

Institutt for informatikk
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2016

GPS Time Spoofing

A detection and mitigation system for GPS timing

Aril Schultzen

GPS Time Spoofing

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

Printed: Reprosentralen, University of Oslo

Abstract

Abstract goes here.

Foreword

Here goes foreword

Contents

Chapter 1

Introduction

1.1 Global Positioning System: A short introduction

The Global Positioning System (GPS) is a utility owned by the United States that provides its user with positioning, navigation and timing services. At the end of 60's, the U.S Navy was developing the Polaris missile, a missile capable of being launched from a submarine. One of the requirements for launching the Polaris missile was exact knowledge of the submarines position. The problem led the Navy and The Applied Physics Laboratory at Hopkins to develop the Transit system, the earliest predecessor to the GPS system [10].

Today, roughly 40 years later we are surrounded by GPS technology. In fields like emergency response, search and rescue, fleet management and even agriculture, it has become a vital tool of utmost importance to everyday operation. Satellite navigation can be found in most new cars and few phones are today sold without an internal GPS receivers. The European Space Agency estimated that there were 2 billion GPS enabled devices by 2012 [1]. What started out as a navigation tool for the U.S navy is now used by millions, if not billions of users both civilian and military all over the globe. A common misconception (that is often reinforced by

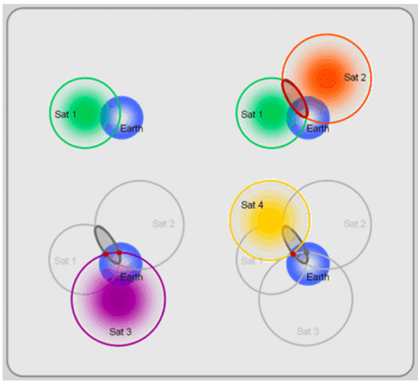


Figure 1.1: Figure showing how GPS satellites are used to trilaterate to determine a GPS receivers position. Source: [2]

Hollywood action movies) is that the GPS satellites track *you* by communicating with your GPS receiver. It actually works the other way around. You are, with your GPS receiver, tracking a set of satellites in order to establish your own position. At any given time, there are at least 24 GPS satellites each in its own orbit at about 11,000 nautical miles above your head [14]. In order for a GPS receiver to determine its position and obtain correct time, it will need 4 GPS satellites within line of sight [1]. The method used by your GPS receiver to determine its position is called *trilateration*. Trilateration is used in geometry as a process of determining the location (absolute or relative) of point by measuring distance. It is often confused with triangulation which instead of distance, uses angles. Measuring the distance from the GPS satellites to a given position on earth is quite simple when using the equation:

$$Distance = Rate \times Time \tag{1.1}$$

The equation is simple to solve, first we need the rate. In this context, the *rate* is how fast the signals travel. This is equal to the speed of light (299,792,458 m/s). The time the signal has used traveling from the satellite to earth can be obtained by analyzing the signal itself. A simple and slightly inaccurate description is the signal contains a "time stamp" of when the signal was sent. By comparing this time stamp with the current time, one can calculate the age of the signal and therefore how long it has spent traveling. This is explained in greater detail under (1.3) [15].

1.2 Clocks

What does a $10 wristwatch and a $100 000 atomic clock have in common? They don't stay accurate forever. This phenomena known as *frequency drift*, is when a clock no longer runs at the exact same speed as a reference clock and they drift apart.
This property is a result of how they track time. In essence, all clocks work in the same way. They have a part that oscillates, a way to count the number of oscillations and a way to show the count. If we transfer this analogy to the typical "grandfather clock", the pendu-

Figure 1.2: High pure Caesium crystals in ampule under argon. Source: [18]

[1]The line of sight requirement might seem unreasonable, but by the time the signal has reached earth, is has degraded to a minimum of -160 dBW [7]

lum would be the oscillator, the counting mechanism the clockwork and the clock face and dials would be the display. In a typical wristwatch, the oscillator is a quartz crystal powered by a battery. The frequency of which the crystal oscillates is then divided down to a single Hertz by simple electronics. The purity of the crystal is among the decisive factors determining the accuracy of the clock. [19]. Although a completely different beast, the same principles apply to the atomic clock which uses the microwave radiation that electrons in atoms emit when they change energy levels. One of the most commonly used elements in atomic clocks, is *caesium-133*, an isotope of caesium.[2] [16]. Bear in mind that this is of course an extremely limited explanation.

1.3 GPS signals and Time

During the introduction of this essay the properties of GPS as a tool for navigation was made apparent. This is however not the only use of GPS, it is also used for timing. The GPS satellites transmits a *Coarse/Acquisition (C/A)* code and a restricted *Precision (P)* code. The C/A code is freely open for everyone and is transmitted at the L1 carrier frequency (1575.42 MHz) and the P code is transmitted at both L1 and L2 (1227.60 MHz) and is reserved for the military. The C/A code is a 1023 bit pseudo random code that is transmitted at 1.023 Mbit/s, which means it repeats itself every millisecond. Each satellite transmits a different pseudo random code, codes that does not correlate well with each other. This is important because it makes it possible to separate the satellites from each other. The way the receiver calculates its position was briefly mentioned during the introduction and is better explained here. The receiver calculates the distance from itself to the satellites by comparing the pseudo random code received from the satellite with an identical one it generates itself. The receiver "slides" these codes over each other further and further until they match up. The signals travel time is determined by how far the codes had to be slided before the matched. This is what is called *Code-phase GPS* and it has got some problems. Since the codes have a wide cycle width, almost a microsecond, there is a lot of slop and at the speed of light, a microsecond wrong is roughly 300 meters wrong. What many receivers do is that they start with the code-phase and moves on to using measurements based on the carrier frequency. Since the frequency

[2] 1 second equals 9,192, 631,770 cycles of the Cs-133 transition

is much higher, the slop decreases and the accuracy increases dramatically. This is whats known as *Carrier-phase GPS.*

Alright, but what about time? We have already established that the key to GPS is measuring the travel time of a radio signal, but considering the consequences of a couple of microseconds of slack when dealing with light-speed, it is really putting some pressure on the GPS receivers internal clocks. As previously mentioned, all your receiver needs to do to find its position in a three dimensional space, is three GPS satellites. If the GPS receivers internal clocks where perfect, the three satellite ranges would intersect at a single point, your position. But in the real world our clocks is everything but perfect. One could use atomic clocks in the receivers but that would make the receivers too expensive (even though chip scale atomic clocks (CSAC) are becoming increasingly affordable 2) for anyone to buy. The solution is to make a fourth measurement from a fourth satellite. This measurement will not intersect with the first three when using an imperfect clock. The receiver can then try to find a correction factor it can subtract from its timing measurement in order to make the measurements intersect. By doing this, it also brings the receivers clock back to sync with universal time. With the correct time, it can also make correct and precise positioning. [12]

1.4 Phasor Measurement Units

An example of an application relying on GPS derived time is a PMU (phasor measurement unit).A PMU analyzes the waves on the electrical grid and uses a common time source for synchronization. This synchronization allows for real-time measurements between multiple points in the grid by multiple PMU's. The common time source (and why PMU's are relevant) is often obtained by using GPS. [13] The value of such a device is understood clearer by recognizing that the power grid is a complex, interconnected, interdependent network. In other words, errors and abnormalities in one part of the grid will have an effect on operation elsewhere and in some cases lead to whole spread blackouts [17].

1.5 Threat Models and countermeasures

The thread models and countermeasures presented in this paper are based on the article *Reliable GPS-Based Timing for Power Systems: A Multi-Layered, Multi Receiver* by L. Heng, D. Chou and G. Xingxin Gao (2012). The only exception is our proposed countermeasure under 2.

1.5.1 Threats

Jamming

By emitting a high-power signal at the frequencies used by GPS satellites, one can interfere with the signals received by the GPS receiver, effectively denying GPS receivers use of these signals. These signals are already weak considering their travel from space. Such an "attack", although effective, is pretty naive and easily recognized by the jammed party. If your equipment is operational and you don't have a signal, you are probably being jammed.

Signal-level Spoofing

Signal-level spoofing is when an attacker causes a receiver to loose lock on an authentic GPS signal by overpowering it with a false signal. This can be achieved by using a GPS simulator that matches the authentic signals phase, code delay and encoded data [9]. Knowing the signal that the victim is receiving is important in order to successfully spoof it. To anyone with access to the military-grade GPS signals, this is less of an issue since their signals are encrypted and harder to spoof, the civilian frequencies on the other hand are publicly known and readily predictable. Shepard, Humphreys and Fansler (2012)[17] describes in their paper *Evaluation of the Vulnerability of Phasor Measurement Units to GPS Spoofing Attacks*, a way to successfully spoof a GPS signal used by a PMU. They describe how they "introduce" the counterfeit signal to the victim by adjusting the power of the signal below the victim receivers noise floor and then gradually raises it until it surpasses the authentic signals strength. Once the victims receiver locks on, the attacker has gained full control.

Data-level Spoofing

In data-level spoofing, the contents (data) of the GPS signal are manipulated. GPS signals includes ephemeris data used to solve the positions of each satellite in orbit and also the time and status of the satellite constellation. By altering this data, the receiver solves incorrect velocity, location and most important in this context, clock offset.[9]

Replay spoofing

Replay spoofing (or *meaconing*[3]) is a technique where GPS signals are intercepted and rebroadcasted. The rebroadcast can be delayed and used to

[3] *Meacon* is portmanteau of *Masking Beacon*

confuse navigation or to cause delay in applications relying on GPS signals for time.

Malfunctions

Just like any tool or device, a GPS receiver is prone to failure. This threat may not be posed by an external party, but is still a threat to normal operation. The ability to differentiate between an attack and a malfunction is important when deciding how to respond to such an event.

1.5.2 Countermeasures

Monitoring Signal Power

In any kind of attack, jamming or spoofing, a counterfeit signal must overpower the authentic signal in order for the receiver to lock onto it or in the case with jamming, denying access to the authentic signal. By monitoring the strength of the signal and detecting a spike or rise in signal power, a possible attack can be identified. This is a low-cost, low-complexity and independent (in contrast to for example using other receivers as a reference) countermeasure. It is however because of the unpredictable nature of signals, not considered to be a detection confident countermeasure and should therefore only be used along side other countermeasures.[11]

Checking solved position against known position

By checking the position solution against the known position of the receiver, both receiver errors (1.5.1) and a replay spoofing (1.5.1) attack can be detected. It does however fall short when more sophisticated techniques like Data and Signal-level spoofing (1.5.1,1.5.1) are used. These kind of attacks when done properly (unless it's done with intention), will not alter the solved position. It is important to note that this only relevant when only using *one* receiver. If the position solution from multiple receivers deployed in the same area are cross-checked, this countermeasure can still be considered effective. Consider the following scenarios when using 3 receivers:

- **None of the receivers are spoofed:** Each receivers solved position matches their respective known position. They all solve the same time.
- **One or two receivers are spoofed:** The spoofed receiver(s) solve(s) different time compared to the receiver(s) not being spoofed.

- **All the receivers are spoofed:** As long as they are spoofed by the same spoofer, they will solve the same time but also the same position which again makes it possible to detect the attack.

A possible way to for a attacker to avoid detection would be to use one spoofer per receiver. These spoofers would need to be synchronized and their signal power fine tuned to make sure that they only spoof their respective receiver. It is believed that such an attack would be too complex and costly to be considered practical. [11]

Checking time solutions against receiver clock statistics

By comparing statistics created by monitoring the receivers clock with the time solution, one can detect spoofing (1.5.1,1.5.1) as well as malfunctions (1.5.1). This is because the time solution is unlikely to be consistent with the statistics in event of an attack. Since this countermeasure relies on the receivers clock which can be described as both unpredictable and stochastic, it should only be used along side other countermeasures.[11]

Cross-checking navigation data among receivers

When under a data-level spoofing attack (1.5.1), the navigation data is modified. By comparing one GPS receivers navigation data with another, both data-level spoofing and malfunctions (1.5.1) can be detected. This countermeasure can also prove useful during jamming attacks (1.5.1) because a jammed receiver could use the data from other receivers in the event that is unable to correctly decode navigation, but still able to track satellites. This may enable the receiver to continue operation during an attack. [11]

Comparing navigation data and reverse-calculated satellite positions

The PMU GPS receivers are never moved and their position is known. By using their pseudorange measurements, the satellites positions can be reverse calculated by using trilateration. Since the reverse-calculated positions only match the positions calculated from the navigation data when both pseudorange and navigation data is correct, one can effectively detect replay spoofing (1.5.1) and malfunctions (1.5.1). Its also worth noting that this countermeasure increases the difficulty of both signal and data-level spoofing (1.5.1,1.5.1) because it narrows down the possible valid (seemingly) spoofing signals. [11]

Cross-correlating P(Y) code

This countermeasure assumes two receivers with at least 1 km distance from each other that tracks a signal from a satellite visible to them both. It is also based on the assumption that the encrypted military P(Y) code cannot be forged by a spoofer. The receivers use the C/A code phase and timing relationship to the P(Y) code to obtain two samples from the same time frame of the received P(Y) code and then correlate the two samples. Even though the samples will be encrypted, noisy and perhaps distorted by narrow-band RF front-ends, a high correlation peak should be created when a cross-correlation is conducted as long as the receivers are not spoofed. A key conclusion of the research made by L. Heng (2013) as referenced by L. Heng *et alia* (2014) was that the probability of detection errors using this method decreased exponentially with the length of the samples made from the P(Y) code and the number of receivers used as reference. This method has therefore proved itself effective against spoofing attacks (1.5.1,1.5.1), but ineffective against replay spoofing because the rebroadcast uses authentic GPS signals with correct P(Y) code. It is important to note that the implementation of this countermeasure relies on the GPS receivers ability to output baseband samples and these samples ability to be transfered over a data network. Because the sampling rate of the samples are fairly high, it is recommended that the spoofing detection is done periodically instead of continuously. [11]

Position Aided (PIA) Tracking loops

Vector tracking is a receiver architecture that combine the tasks of signal tracking and position/velocity estimation into one algorithm. This is a contrast to the traditional way where the tracking methods track satellites independently as well as the position/velocity solution independently. Even though this requires more computing power, it increases immunity to interference and jamming. The vector tracking is aided by the fact the we know the PMU GPS receivers true location. The tracking robustness can be further improved by using a Kalman filter. Since a PMU and its GPS receiver remain stationary, the parameters of the tracking loops can be chosen to narrow the loop filter bandwidth which reduces noise and the effective radius of a potential jamming attack (1.5.1). Replay spoofing attacks will also fail since the PIA vector tracking depends on the knowledge of the GPS receivers true position. In the event of such an attack, the result would be that the vector tracking will fail to function. [11]

Multi-receiver tracking loops

Building on the idea from *PIA Tracking loops*(1.5.2) one can benefit from the networked nature of the GPS-timed PMU. In a multi-receiver vector tracking loop, many receivers process information in collaboration. A key conclusion of the research made by A. Soloviev *et alia* as referenced by L. Heng *et alia* (2014) showed that acquisition and tracking performance under low signal-to-noise ratio conditions was improved under multi-receiver signal accumulation. Multi-receiver phased arrays also improved the robustness against both jamming (1.5.1) and spoofing attacks (1.5.1,1.5.1) by *"Forming beams to satellites and steering nulls in the direction of attacking transmitters"* (L. Heng *et alia* (2014), p.41). In addition to the increase robustness, it increases the ability to detect malfunction (1.5.1). A faulty receiver will usually not be consistent with other correctly functioning receivers. As with the countermeasure based on cross-correlating P(Y) code (1.5.2), this implementation also requires that the GPS receivers are able to output baseband samples. In this implementation, the samples need to be transmitted continuously among the receivers which requires a capable data network such as a typical LAN. [11]

1.5.3 Summary

The table (1.1) shows the different threat models and the effect of the countermeasures discussed.

Table 1.1: The table shows the effectiveness of the covered countermeasures against threat models.

Counter Measures	Threat Models				
	JAM[4]	SLS[5]	DLS[6]	RS[7]	MF[8]
Monitoring Signal Power (1.5.2)	N	X	X	X	N
Check pos. solution (1.5.2)	N	Y	Y	Y	Y
Check time solutions (1.5.2)	N	X	X	X	X
Checking nav. data (1.5.2)	X	N	Y	N	Y
Reverse calculated sat. pos. (1.5.2)	N	X	X	Y	Y
Cross-correlating P(Y) (1.5.2)	N	Y	Y	N	N
PIA TL (1.5.2)	Y	N	N	Y	N
Multi-receiver TL (1.5.2)	Y	X	X	X	X

Table 1.2: Legend for table (1.1)

Y	Effective	N	Ineffective	X	Auxiliary

[4] Jamming (1.5.1)
[5] Signal-level Spoofing (1.5.1)
[6] Data-level Spoofing (1.5.1)
[7] Replay Spoofing (1.5.1)
[8] Malfunctions (1.5.1)

Chapter 2

Our Proposal: Spoof proof CSAC SMACC

We propose to construct and use what we call a *Spoof proof chip scale atomic clock smart miniature atomic clock controller (CSAC SMACC)*. This is in essence just a piece of software running on a computer controlling a GPS disciplined chip scale atomic clock. The SMACC software will be connected to the CSAC and perform the following trivial tasks:

Figure 2.1: Symmetricom SA.45s CSAC. Courtesy Symmetricom.

- Checking time solutions a clock model of the CSAC(1.5.2)
- Checking solved position against known position (1.5.2)
- Monitoring Signal Power(1.5.2)

In the event of spoofing or jamming attack, actions can me made depending on the situation. For example, during a spoofing attack, the SMACC upon detecting an attack, could simply disable the disciplining of the CSAC thus minimizing the damage done to the CSAC's stability caused by the spoofing attack. The SMACC could also do "one better" by steering the CSAC based on a model of the CSAC's oscillator, thus ensuring stability for a longer period of time than by just disabling the disciplining.

2.0.4 Notes

It is important to note that this approach doesn't really do anything with the fact that you are being attacked, it simply tries to eliminate the effects of it. In a scenario where you are under attack weeks at a time, you will have to address the fact that you are under attack at some point. It is also important to notes that this countermeasure mostly apply to applications using GPS as a source of time. Having a stable clock during a jamming attack will not help you determine your position once you move (given that you are fully jammed).

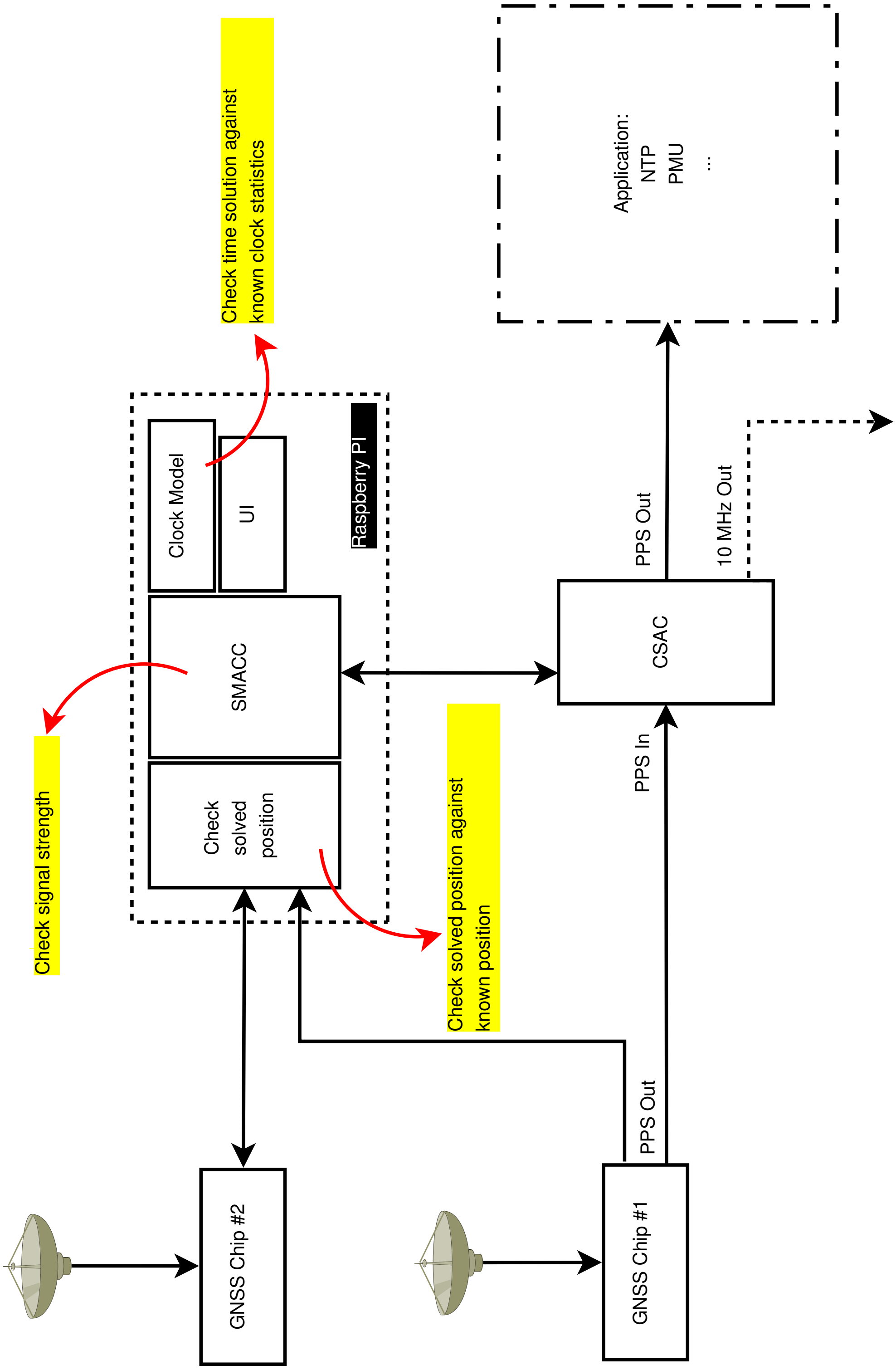


Figure 2.2: A block diagram of our proposed solution

Chapter 3

Hardware

3.1 Chip Scale Atomic Clock

I propose to use the Symmetricom SA.45 as the CSAC. This is a CSAC measuring only 16cc with 1 pulse per second (PPS) output and 1 PPS input (for disciplining). The SA.45's strength is it's low power consumption (less than 120mW) and low price [20]. The SA.45 also uses a built-in controller which can be communicated with over a RS-232 serial interface. The ability to communicate with the CSAC, issue commands and collect data, is paramount for the feasibility of our proposal. It's worth mentioning that any atomic clock such as Cesium standard or even a Rubidium standard could be used given that they have a means to communicate basic telemetry as used by the SMACC software.

3.2 SMACC platform

I propose to use the Raspberry Pi 3 Model B (RASPI3) in the role as the host running the SMACC software. The RASPI3 is an interesting piece of equipment with an impressive list of specifications. It is a single board computer with a 1.2GHz 64-bit quad-core ARMv8 CPU, 1 GB of RAM, built in 802.11n Wireless LAN and four USB ports ([5]) (just to mention some). As with the Symmetricom SA.45, the RASPI3 is very affordable and retailed at about 35 USD when this report was written. We also propose to use Raspbian ([21]), a Debian derived flavor of Linux optimized for the Raspberry Pi as the operating system.

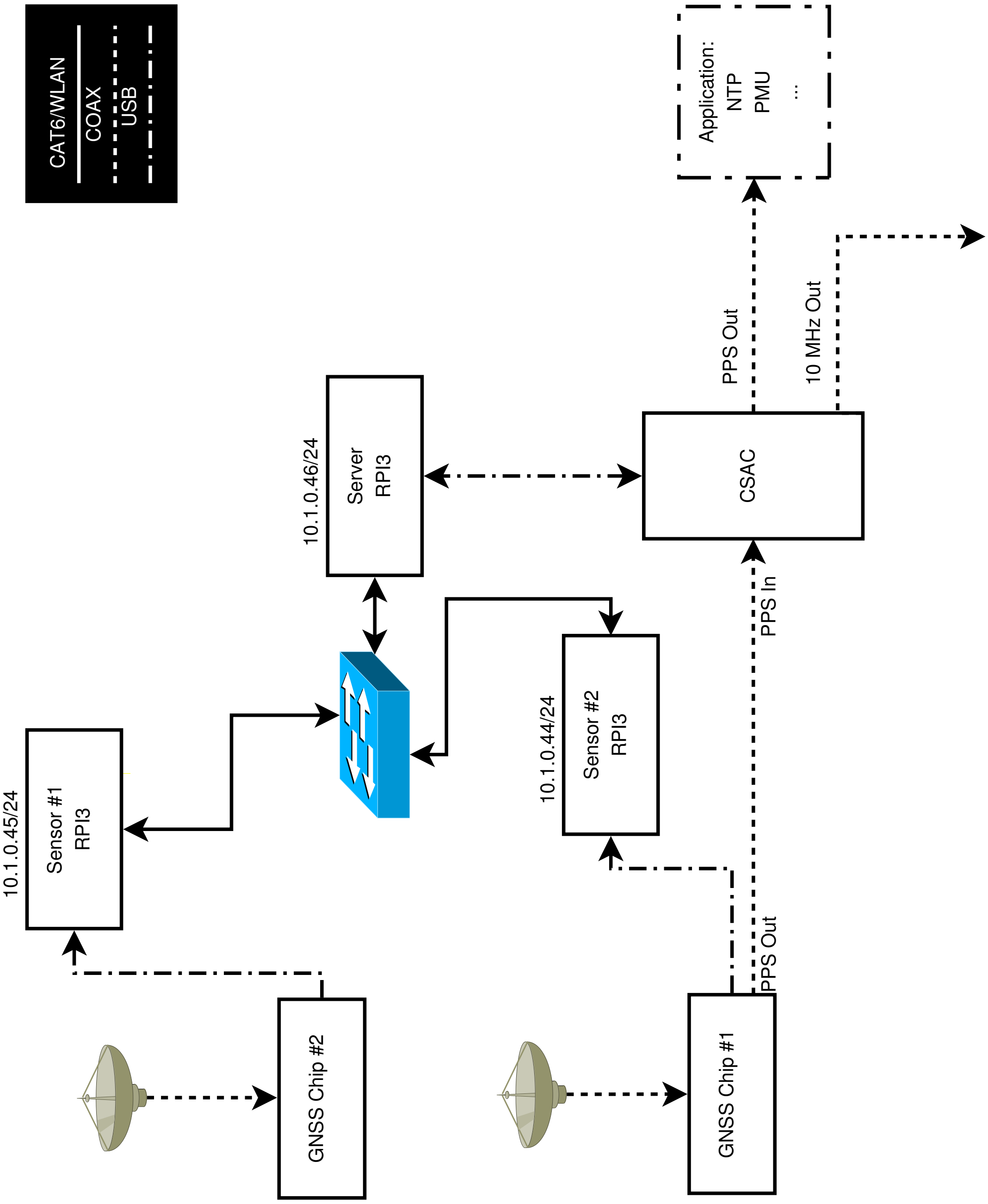


Figure 3.1: A block diagram showing the tested implementation

3.3 GNSS receiver

I propose to use at least two GNSS receivers. Both of the receivers should simply collect data and feed it to the SMACC but one of the receivers should also double as a 1 PPS disciplining source for the CSAC. Considering that need for a stable 1 PPS source, I propose to use the u-blox M8T. This is a relativity affordable GNSS receiver with a temperature compensated crystal oscillator (TCXO), 3 concurrent GNSS reception and an external antenna ([22]). Currently, only strings of NMEA data is collected from the GNSS receiver. However in the future it might be beneficial to collect and process raw data from the receivers as well. Since most GNSS receivers today follow the NMEA standard (to some extent) and raw data currently isn't required, common and popular receivers like the u-blox NEO series should be more that sufficient to use in an implementation if this proposal.

Chapter 4

Software

4.1 The Sensor Server/Client model

Numerous approaches where considered when planning the implementation of the SMACC software (See 6.2 for alternate approaches). The approach that was chosen is a Client/Server model which I have named the "Sensor Server". The Sensor Server model is based on the idea that a GNSS receiver and a computer can be viewed abstractly as a single device, a Sensor. The Server and Sensor communicates over an IP network. The Sensor runs a trivial program that receives data from a GNSS receiver, formats the data correctly and sends the data to a Server (more about the client 4.2). The Server on the other hand, is responsible for the heavy lifting. The data gathered from the Sensors are applied to what we call *filters*. The filters are just algorithms that are able to detect anomalies in the data, thus making it possible to react to a spoofing or jamming attempt. Server tasks include:

- Handle connections to all Sensors.
- Update structures as Sensor status changes (Disconnects, kick request)
- Communication with the CSAC and CSAC model updating
- Sensor data analysis and filter updates
- Raising alarms based on filter status

By using already existing network infrastructure, it becomes a lot easier to distribute Sensors and cover more area. This makes spoofing attacks harder to implement and easier to detect(1.5.2).However, every router and switch between a Sensor and the Server imposes a delay on the stream of packets between the two, especially when compared with a directly cabled approach. This might make the Sensor Server approach less responsive. It is of our understanding that whatever increase in complexity the Client/Server introduces to our approach, the Sensor Server makes up for by eliminating the need for potential miles of signal cables and signal amplifiers.

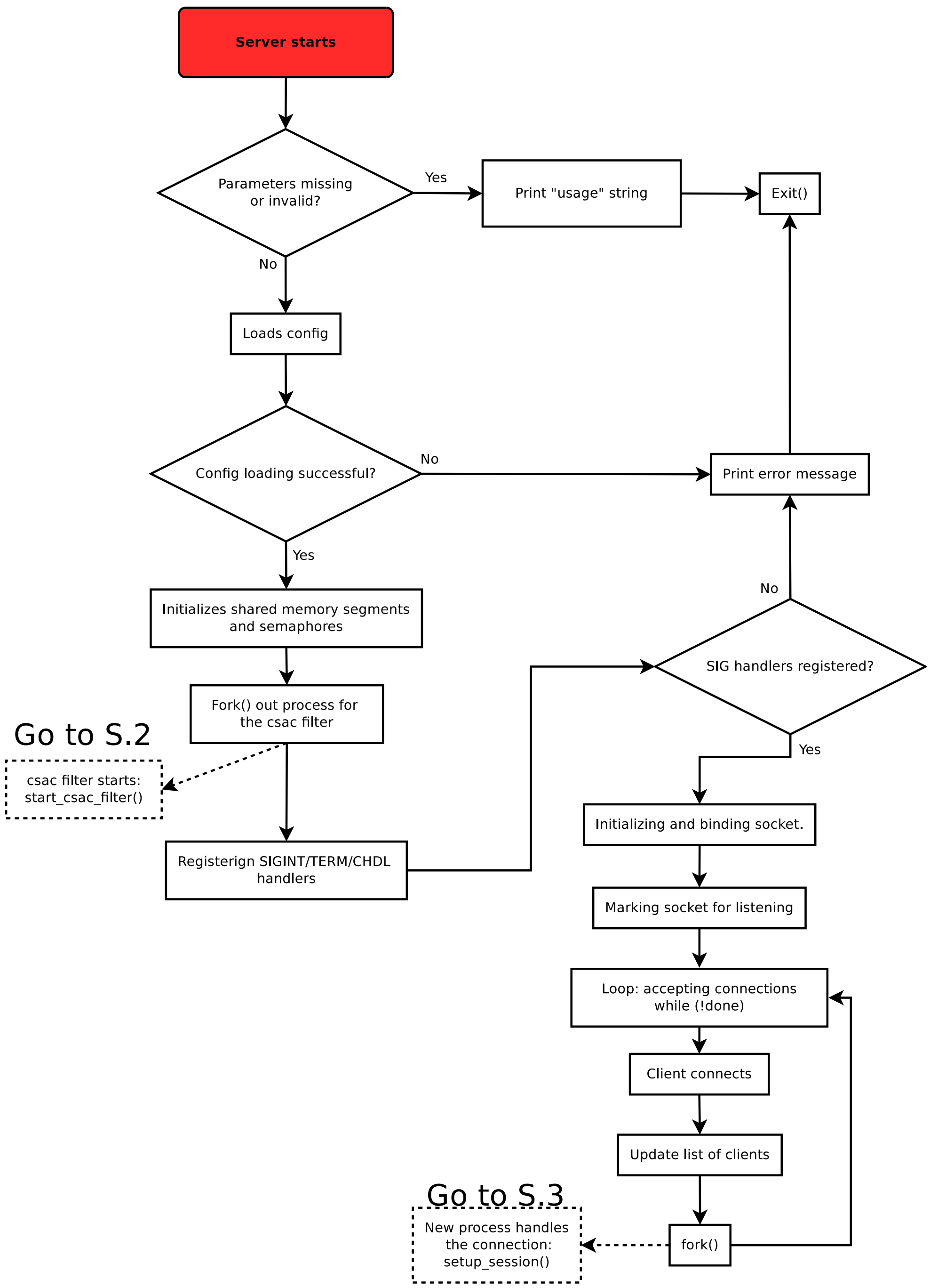


Figure 4.1: The block diagram shows an abstracted view of the Sensor Server.

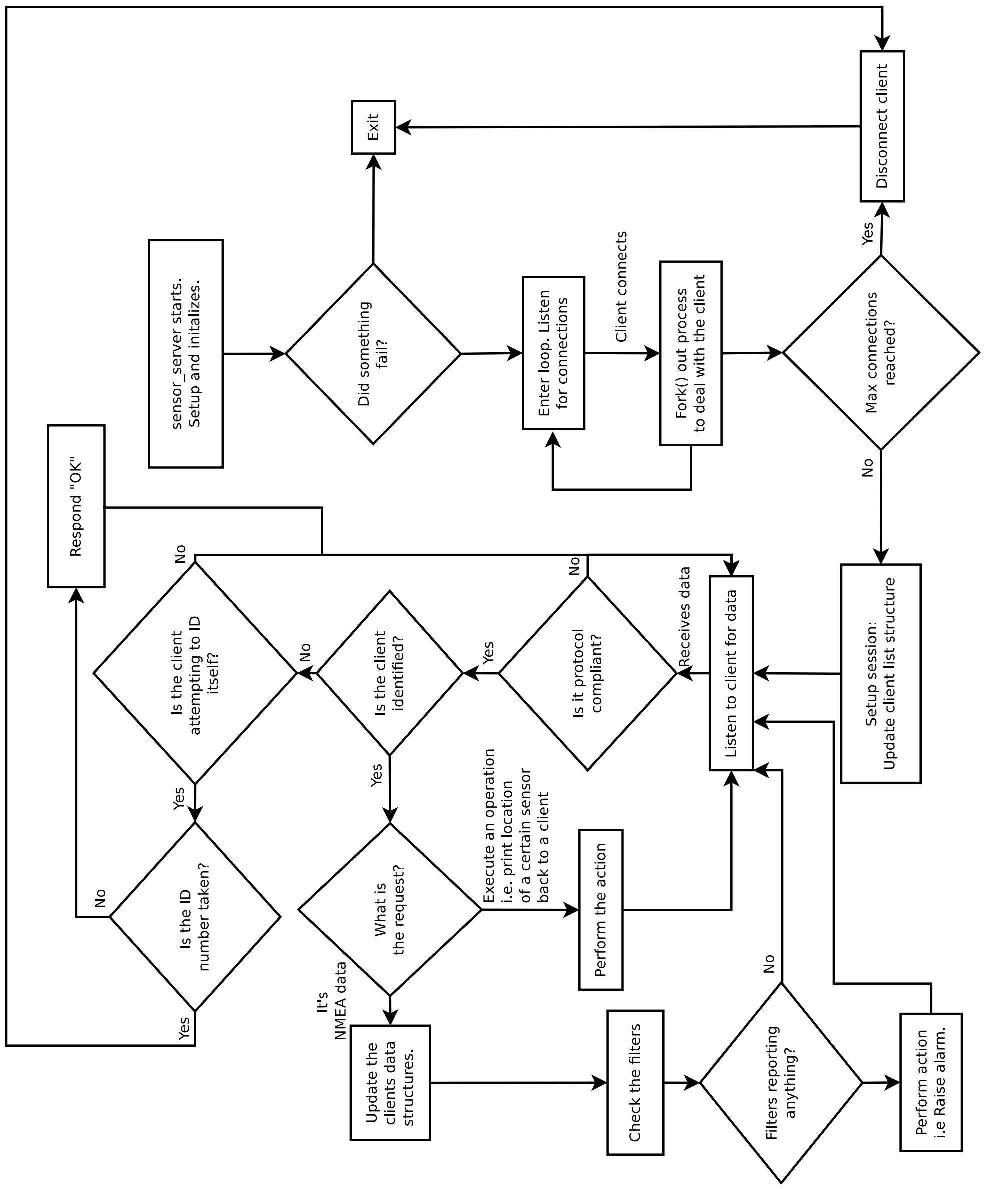


Figure 4.2: The block diagram shows an abstracted view of execution after a client has connected to the server and a `fork()` has been performed.

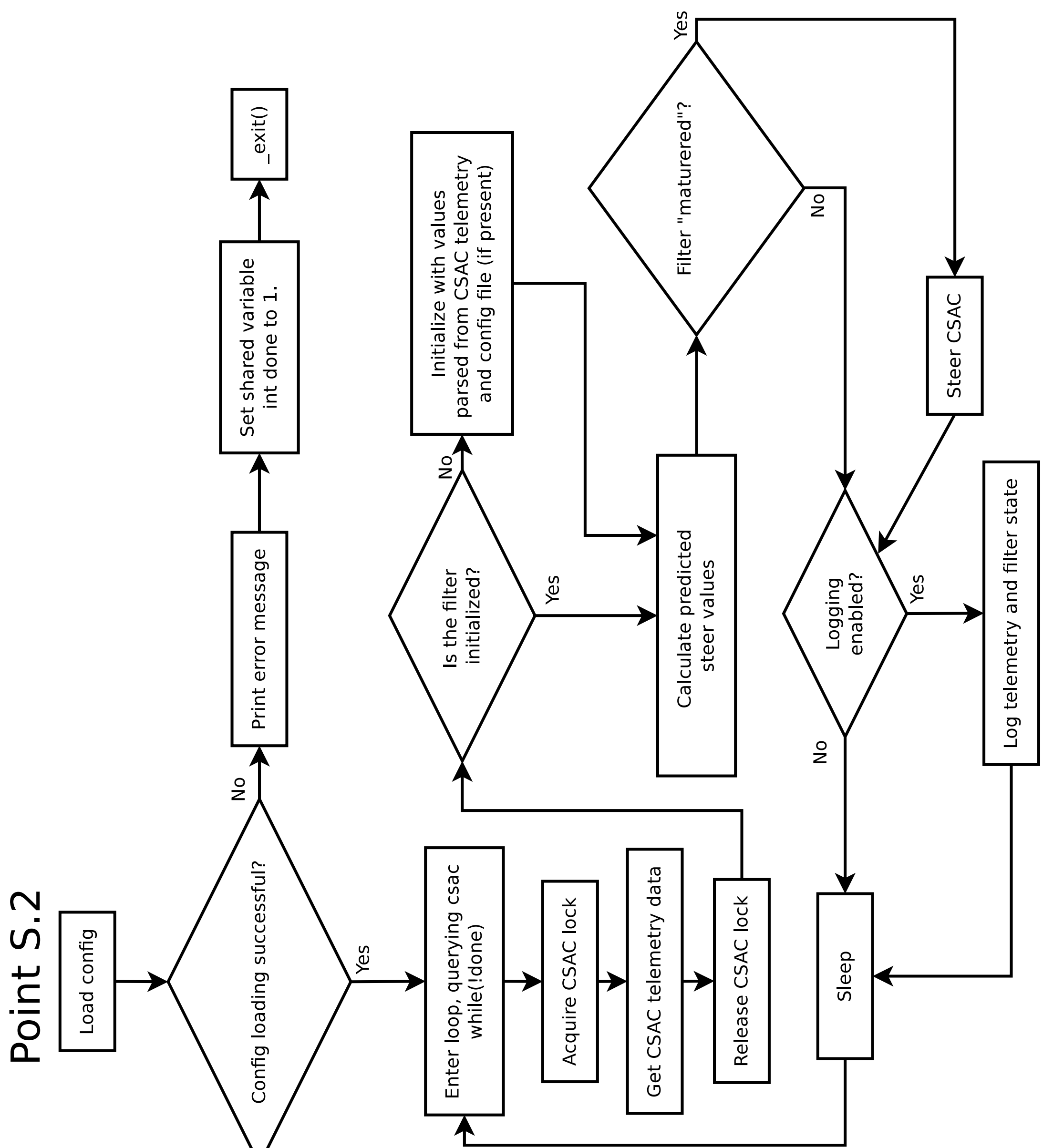


Figure 4.3: The block diagram shows the execution flow of the CSAC filter.

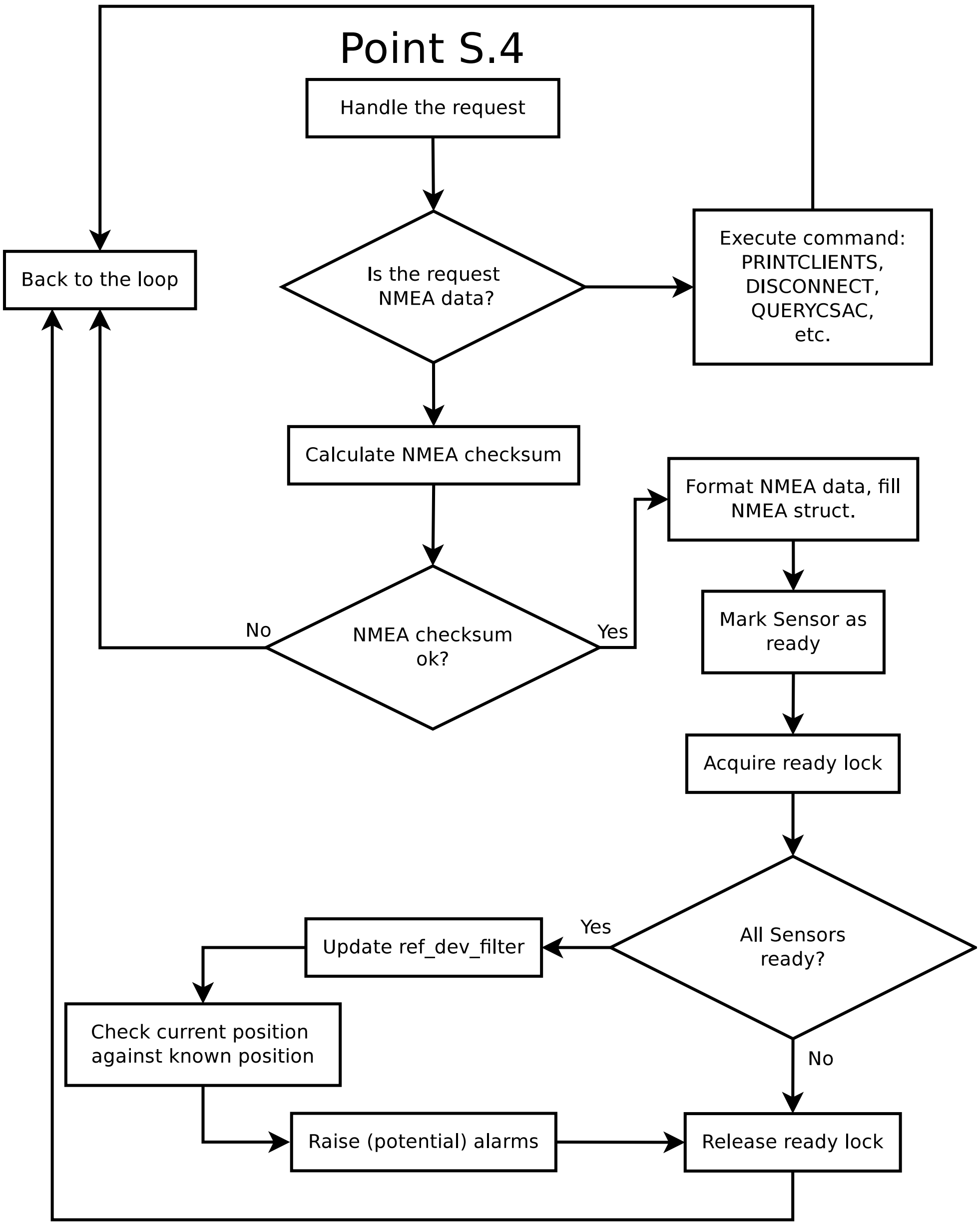


Figure 4.4: The block diagrams shows and abstracted view of the execution after data has been received from a client.

4.1.1 Roles

A client connected to the server can have two roles; It can either be a Sensor or a Monitor. The Sensor role is already explained, but the Monitor role was added in order for a user of the Sensor Server to connect to the server and check status or issue commands. For a client to assume the role of a Monitor, the client has to pick a negative integer as ID number. This way, the Server does not expect you as a client to report any NMEA data the way it would with a Sensor. As a Monitor, you can issue the following commands:

Table 4.1: Sensor Server available commands

Command	Short	Parameter	Description
HELP	?	NONE	Prints this table
IDENTIFY	ID	ID	Clients ID is set to PARAM
DISCONNECT	EXIT	NONE	Disconnect from the server
PRINTCLIENTS	PC	NONE	Prints an overview of connected clients
PRINTSERVER	PS	NONE	Prints server state and config
PRINTTIME		ID	Prints time solved from GNSS data received from Sensor <ID>
PRINTAVGDIFF	PAD	NONE	Prints the difference between current solved position and the average reported for all Sensors
PRINTLOC	PL	ID	Print solved position for Sensor <ID>
LISTDATA	LSD	NONE	List all dump files stored by the server
DUMPDATA	DD	ID & FILE	Dumps state of Sensor <ID>into a file named <FILE>
LOADDATA	LD	ID & FILE	Load state stored in file called <FILE>into Sensor <ID>
QUERYCSAC	QC	COMMAND	Queries the CSAC with COMMAND.
LOADRFDATA	LRFD	ID	Load reference location data into Sensor <ID>
PRINTCFD	PFD	NONE	Prints CSAC filter data

4.1.2 Sockets

In order to implement the Server/Client model, the Sensor Server is implemented using the Linux Socket API. The API is based on BSD sockets and are available in in almost all Unix like operating system ([4], p.610). Listing 1 shows a sample of code taken from one of the Sensor Server's source file. The sample shows the following:

- Line 4: The server waits for a connection. `accept()` is a blocking function. The code does not continue past this point before a client has connected.
- Line 12: The code has been executed way past the blocking `accept` function. Someone must have connected! The server forks out a new process from this point in the execution with the function `fork()`.
- Line 13: Upon entering the if statements regarding it's process identification (PID), the parent ends back at the top in the while loop. The child on the other hand, matches the criteria for the if sentence at line 15.

- Line 16: The child process closes it's parent's socket file descriptor and continues to setup the session at the next line.

Listing 1: Sample of code taken from `sensor_server.c`*(B.1.1, line 356). The sample has been edited for clarity purposes.*

```
1    listen(server_sockfd,SOMAXCONN);
2    int session_fd = 0;
3    while (!done) {
4        session_fd = accept(server_sockfd,0,0);
5        if (session_fd==-1) {
6            if (errno==EINTR) continue;
7            t_print(ERROR_CONNECTION_ACCEPT,errno);
8        }
9        if(number_of_clients == max_clients) {
10           close(session_fd);
11       } else {
12           pid_t pid=fork();
13           if (pid==-1) {
14               printf(ERROR_FAILED_FORK, errno);
15           } else if (pid==0) {
16               close(server_sockfd);
17               setup_session(session_fd, new_client);
18               close(session_fd);
19               _exit(0);
20           } else {
21               close(session_fd);
22           }
23       }
24   }
```

Even though the `accept()` function in the sockets API is blocking, CPU cycles are not wasted. If a socket call cannot be completed immediately, the process who issued the call will be put to sleep thus enabling the scheduler to schedule other processes for execution until conditions are right for the sleeping process. ([24], p.435). It is also possible to use non-blocking socket calls, and while this often increases performance, it also increases complexity, and was therefor not chosen for this approach. It's also worth mentioning that one could create *threads* instead of forking out processes for new connections. The creation of threads are typically less expensive in terms of CPU cycles than the creation of processes. Processes on the other hand, always have their own virtual address space as opposed to threads who share their address space with the other threads withing the process. This makes programming with threads more complex and the result of a crash more severe as it affects the other threads as well.

4.1.3 Shared memory & Semaphores

The sensor server architecture uses several shared memory segments. This is necessary because as mentioned earlier(**??**), each process has got it's own virtual address space. The pointers to the shared memory segments are declared as *extern* in `sensor_server.h`. The extern keyword means the the variable has an external linkage, making it visible from other files than the one in which it is defined. Listing 2 shows a code sample taken from `sensor_server.h` where the shared memory segments are declared.

Listing 2: Sample of code from `sensor_server.h`*(B.1.2, line 356) where shared memory segments are declared.*

```
    extern volatile sig_atomic_t done;
    extern struct client_table_entry *client_list;
    extern struct server_data *s_data;
    extern struct server_synchro *s_synch;
    extern struct server_config *s_conf;
    extern struct csac_filter_data *cfd;
```

Every process that forks out from the server is given access to these memory segment. One might make the point that this voids the idea of processes, and one might be correct (see 6). The shared memory is created using the GNU library's Memory Mapped I/O (MMAP). Although typically used to map files to a region of memory, MMAP can also be used to create an anonymous map which is not connected to file but rather for sharing data between tasks without using files.

Listing 3: Listing shows the use of MMAP to create an anonymous map of memory to be used as a shared memory segment

```
  client_list = mmap(NULL,
                     (s_conf->max_clients * sizeof(struct client_table_entry)),
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED | MAP_ANONYMOUS,
                     -1, 0);
```

Having shared memory segments comes with a price. Whenever two or more processes are working on the same data set, they are prone to create race conditions, deadlocks and data corruption. Therefore, semaphores where used to lock the segments during read and write operations at the shared memory segments.

Listing 4: Function for removing disconnected clients from list of clients

```
void remove_client_by_id(int id)
{
    struct client_table_entry* client_list_iterate;
    struct client_table_entry* temp_remove;

    sem_wait(&(s_synch->client_list_mutex));
    list_for_each_entry_safe(client_list_iterate,
                             temp_remove,&client_list->list,
                             list) {
        if(client_list_iterate->client_id == id) {
            list_del(&client_list_iterate->list);
        }
    }
    s_data->number_of_clients--;
    sem_post(&(s_synch->client_list_mutex));
}
```

Figure 4 shows a typical example of a function locking down access to the shared memory segment containing the list of connected clients, by using a semaphore. In the example (4) a client has been disconnected from the server and the the list of connected clients are being updated. The semaphore is necessary to make sure that another process is not attempting to read or write to the segment while the data is deleted. If another process had attempted to execute the `sem_wait()` on the semaphore, it would have been put in a queue. Depending on the operating system, it would most likely signal the scheduler to do a context switch since the resource was busy anyway and it therefor should relinquish control of the CPU. Once the semaphores is raised, it can be lowered again by another process. It is important to note that the semaphores are not a function of or related to the memory segments by anything other the name. The semaphores are just "flags" used to control access to a resource. There is no automatic raising or lowering of the associated semaphores by reading or writing the shared memory segments. All functions in the sensor server does however use semaphores when dealing with shared memory segments in order to avoid deadlock and race conditions.

4.1.4 Data structures

In the C programming language, a "struct" is a complex data type that defines a list of variables to be placed under the structs given name in a block of memory. This makes it possible for multiple variables to be accessed via a single pointer. Before delving deeper into the code base of the sensor server, some crucial and often used structs will be explained in this section.

Linked list

Since the C standard does not provide data structures like linked lists, I had to choose between reinventing the wheel or finding some implementation to drop into the project. While studying another subject, I found a guide on how to use the linked list implementation from the linux kernel source code ([25]) in a user space program. Since the implementation was extremely solid, well tested and had many useful functions, i decided to use it. The modified header file containing all the code, is GPL licensed.

Listing 5: Sample of code taken from `list.h` line 70

```
struct list_head {
  struct list_head *next, *prev;
};
\label{struct_client_table}
```

The fields of the struct is pretty self explanatory. There is a pointer to previous node and one the next. By using these, the list can traversed.

client_table_entry

The client_table_entry struct is what the name suggests, it's an entry in a list of clients. Every client connected to the server, no matter the purpose, has an entry in the client list. Listing **??** shows the complete struct.

Listing 6: Sample of code taken from `sensor_server_common.h` line 99

```
struct client_table_entry {
    struct list_head list;
    struct transmission_s transmission;
    struct timeval heartbeat_timeout;
    struct command_code cm;
    struct nmea_container nmea;
    pid_t pid;
    time_t timestamp;
    int client_id;
    int client_type;
    int ready;
    int marked_for_kick;
    char ip[INET_ADDRSTRLEN];
    struct filters fs;
};
\label{struct_client_table}
```

The lient_table_entry struct is probably the type of the most commonly passed pointers in the program.

server_config

Listing 7: Sample of code taken from `sensor_server.h` *line 23*

```
struct server_config {
    int max_clients;
    int warm_up_seconds;
    int human_readable_dumpdata;
    char csac_path[PATH_LENGTH_MAX];
    int logging;
    char log_path[PATH_LENGTH_MAX];
    int csac_logging;
    char csac_log_path[PATH_LENGTH_MAX];
};
\label{struct_server_config}
```

server_data

Listing 8: Sample of code taken from `sensor_server_common.h` *line 116*

```
struct server_synchro {
    sem_t ready_mutex;
    sem_t csac_mutex;
    sem_t client_list_mutex;
    volatile int ready_counter;
};
\label{server_data}
```

server_synchro

Listing 9: Sample of code taken from `sensor_server_common.h` *line 125*

```
struct server_synchro {
    sem_t ready_mutex;
    sem_t csac_mutex;
    sem_t client_list_mutex;
    volatile int ready_counter;
};
\label{server_synchro}
```

command_code

Listing 10: Sample of code taken from `sensor_server_common.h` *line 34*

```
struct command_code {
    int code;
    char parameter[MAX_PARAMETER_SIZE];
    int id_parameter;
};
\label{command_code}
```

NMEA container

Listing 11: Sample of code taken from `nmea.h` line 21

```c
struct nmea_container {
    /* Raw data */
    char raw_gga[SENTENCE_LENGTH];
    char raw_rmc[SENTENCE_LENGTH];

    /* Latitude */
    double lat_current;
    double lat_average;
    double lat_avg_diff;
    double lat_total;
    int lat_disturbed;

    /* Longitude */
    double lon_current;
    double lon_average;
    double lon_avg_diff;
    double lon_total;
    int lon_disturbed;

    /* Altitude */
    double alt_current;
    double alt_average;
    double alt_avg_diff;
    double alt_total;
    int alt_disturbed;

    /* Speed */
    double speed_current;
    double speed_average;
    double speed_avg_diff;
    double speed_total;
    int speed_disturbed;

    /* CHECKSUM */
    int checksum_passed;

    /* COUNTER FOR AVERAGE */
    int n_samples;
};
```

4.2 The Sensor Client

The sensor client software is a simple program written in C99 whose only task is to relay information read from the GNSS receivers. Summed up shortly:

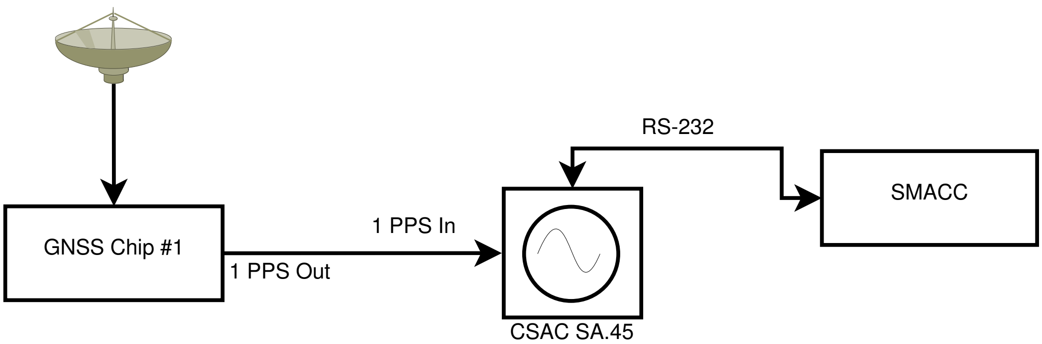
- The client software takes two parameters to start, the servers IP and port. If parameters are missing, the program exits.
  - Example: `./sensor_client -p 10000 -i 192.168.1.5`
- Initializes and loads configuration from configuration file. The configuration file includes path to the GNSS receiver, the sensors ID number and a binary value for whether or not logging of NMEA should be done as well as path to the log file. If the loading of the configuration file fails, default values are used instead:
  - The ID number is chosen at random but within legal limits.
  - Logging is disabled.
  - Maximum of server connection attempts are set to 10.
  - Path to GNSS receiver is set to `/dev/ttyACM0`. This should be the path to the receiver unless another similar device is connected to the computer and given it is a Raspberry Pi running Raspbian.
- Establishes communication with GNSS receiver, exits if it fails.
- Attempts to establish communication with the server, retries for a configurable amount of times at 1 second intervals.
- Identifies the client for the server according to protocol.
- Reads from the GNSS receiver, scans for lines starting with either `$GNRMC` or `$GNGGA`. When both lines are found, the data is stored in a buffer.
- Sends the GNSS data to the server according to protocol.
- Repeats.

Table 4.2: Commands for the SA.45 CSAC

Shortcut	Description	Command
6	Return telemetry headers as comma-delimited string	!6[CRLF]
ˆ	Return telemetry as comma-delimited string	!ˆ[CRLF]
F	Adjust frequency	!F?[CRLF]
M	Set operating mode register bits	!M?[CRLF]
S	Sync CSAC 1 PPS to external 1 PPS	!S[CRLF]
D	Set 1 PPS disciplining time constant	!D?[CRLF]
U	Set ultra-low power mode parameters	!U?[CRLF]
T	Set/report time-of-day	!T?[CRLF]

Source: [3]

4.3 CSAC Communication



The SA.45 CSAC includes a serial interface that enables communication with a PC by using a COM port. As mentioned earlier, our approach relies heavily on the ability to communicate with the CSAC. Information can be queried by sending commands to the CSAC. These commands are explained in table 4.2.

4.4 Detection algorithms: Filters

4.4.1 Data acquisition

In order to create an accurate clock-model of the CSAC, it was necessary to log data from it while it was running in a disciplined mode. In the disciplined mode, the CSAC will correct it's frequency based on either a 1 PPS (Pulse per second) signal or a 10 MHz signal. A similar approach was used in order to collect GPS data. Data from two u-blox M8T was gathered over the same time period as the data gathered from the CSAC. By gathering the data over the same period, it was possible to detect any correlation between the time solved by the GPS receivers and any frequency adjustments done by the CSAC. It also provided valuable data that could be used to tune the spoofing detection algorithms in the CSAC SMACC. The data gathering was done by simple Python scripts (A.1 and A.2) running on a computer connected to the receivers and the CSAC (B.3)

Clock Model

Write about clock model

Probability filter

The "probability filter" is implemented by using collected data to create a model of expected data. The current data is then compared in order to evaluate whether or not it is abnormal.

Chapter 5

Testing

5.1 Software performance

How was the performance of the server? Slow? Buggy?

5.2 Preliminary test

Write about the simple test where I moved the antennas about.

5.3 Spoof test

5.3.1 Challenges

What if their clocks sucks?

5.3.2 The test

Write about the test

Chapter 6

Results and discussion

6.1 Choice of programming language

The SMACC software was originally planned to be written in Java since this was my most fluent programming language. Java is great language, it's object oriented, it has a garbage collector and a lot of useful libraries. As development started, it quickly became apparent that some parts of the code would be performance critical and that portability really wasn't that important anyway. The platform was already decided and there was no reason to believe that it would change in the near future. As we all know, premature optimization is the root of all evil. Being reluctant to commit a deadly programming sin, i decided to look at other languages. Since performance was a concern, Python was also quickly dismissed as an option. C++ would probably have been the best choice, but having never written anything in C before made it sound more exciting and like a nice opportunity to learn something new. During the planning phase of SMACC development, raspbian-2015-05-07 was the latest build. It came with GCC 4.6.3 which only had experimental support for C11([6]). With C11 no longer considered an option, C99 was the obvious choice given it's attractive features like:

- Variable-length arrays.
- Single line comments.
- snprintf() as standard ([8]).

6.2 Alternative approaches

When planning on how to execute our proposal, these where among the ideas that came up.

Single computer, many GNSS receivers

A single computer is used to run the SMACC software. The SMACC does not include a Server/Cient model, but the receivers used to collect data are all connected to to the computer through whatever USB ports available or made available by the use of USB hubs. With this approach, you are not dependent on a network, but it limits the number of GNSS receivers you could connect as the USB specification limits the number possible endpoints to an absolute 127([23, pp. 3]) because of addressing. This does not mean that 127 devices can be connected, a single device might use more than one endpoint. It's also worth mentioning that a USB hub might "reserve" multiple endpoints. Depending on the GNSS receivers and how they are made, this number might be reduced even further by the power usage of the connected devices. Depending on how far each GNSS receiver is distanced from the SMACC, a signal amplifier might be necessary to compensate for the signal attenuation. In some cases where a network is absent, this might be only option.

Store in database and analyze

With this approach, the idea of a GNSS receiver and RASPI as a single "sensor" unit is the same as with Client-server approach. The difference is that it with this approach, each sensor stores the collected data in a database. The SMACC software monitors the clock directly as with the Client-server approach, but the data in the database is routinely queried and analyzed. The strength with this approach is that data is easily stored, shared and maintained by a single entity. The complexity of the client software would be the same as with Client-server approach, but the SMACC software could be implemented with less complexity as no Client-server architecture or shared memory schemes would be necessary. During planning, this approach seemed promising but was rejected because it was thought that it might not be time-sensitive enough. It was also some doubt concerning whether or not the ability to store data to a database actually was important. Once the different filters and algorithms was in place, it turned that the database functionality would have been nice, but not of any real importance for the SMACC to perform it's tasks, and would have been overkill anyway.

Chapter 7

Conclusion

You should have done things different.

Appendices

Appendix A

Data acquisition

A.1 CSAC Logger source code

```
'''
:Author: Aril Schultzen
:Email: aschultzen@gmail.com
'''

import ctypes
import fileinput
import sys
import datetime
import time
import io
import os
import serial
import jdutil


def get_today_mjd():
    today = datetime.datetime.utcnow()
    return jdutil.jd_to_mjd(jdutil.datetime_to_jd(today))


def t_print(message):
    current_time = datetime.datetime.now().time()
    complete_message = "[" + str(
        current_time.isoformat(
        )) + "] " + "[" + message + "]"
    print(complete_message)


def main_routine():
    log_file = open("dp.txt", "a+")
    t_print("Started CSAC logging script")
    ser = serial.Serial("/dev/ttyUSB0", 57600, timeout=0.1)
    sio = io.TextIOWrapper(
        io.BufferedRWPair(ser,
                          ser),
        encoding='ascii',
     newline="\r")

```

```
    while(True):
        log_file = open("dp.txt", "a+")
        ser.write(b'^')
        time.sleep(0.1)
        telemetry = sio.readline()
        output = str(get_today_mjd()) + "," + telemetry
        log_file.write(output)
        log_file.close()
        time.sleep(1)

if __name__ == '__main__':
    main_routine()
```

A.2 GPS Logger source code

```
'''
:Author: Aril Schultzen
:Email: aschultzen@gmail.com
'''

"""
GPS Logger requires:
- Python v.2.7
- python-mysqldb

EXPECTED TABLE
---------

create table gprmc (
    id INT NOT NULL AUTO_INCREMENT,
    sensorID INT ,
    fix_time TIME,
    recv_warn VARCHAR(5),
    latitude DECIMAL(10,5),
    la_dir VARCHAR(5),
    longitude DECIMAL(10,5),
    lo_dir VARCHAR(5),
    speed DECIMAL(10,5),
    course DECIMAL(5,2),
    fix_date DATE,
    variation DECIMAL(5,2),
    var_dir VARCHAR(5),
    faa VARCHAR(5),
    checksum VARCHAR(5),
    mjd VARCHAR(50),
    alt DECIMAL(5,2),
    PRIMARY KEY (id) );
"""

import ctypes
import MySQLdb as mdb
import ConfigParser
import fileinput
import sys
import datetime
import time
import io
import os
import serial
```

```
import jdutil
from subprocess import call

config = ConfigParser.ConfigParser()


def dbConnect():
    con = mdb.connect(config.get('db', 'ip'), config.get('db', 'user'),
                      config.get('db', 'password'), config.get('db', 'database'))
    return con


def dbClose(dbConnection):
    dbConnection.close()
    t_print("Connection to database closed")


def initConfig():
    configFile = "config.ini"
    config.read(configFile)


def t_print(message):
    current_time = datetime.datetime.now().time()
    complete_message = "[" + str(
        current_time.isoformat(
        )) + "] " + "[" + message + "]"
    print(complete_message)


def format_date_string(date_s):
    split = date_s.split(".")
    split = split[::-1]
    split = ''.join(split)
    return split


def insert(con, data):
    st = data
    temp = st[12]
    checksum = temp[1] + temp[2] + temp[3]
    faa = temp[0]
    x = con.cursor()
    date = st[9][4:6] + st[9][2:4] + st[9][0:2]
    st[9] = date

    try:
        query = ("INSERT INTO " + config.get('db', 'table') +
        " (sensorID, fix_time, recv_warn, latitude, la_dir, longitude, lo_dir, ) " +
        "(speed, course, fix_date, variation, var_dir, faa, checksum, mjd, alt) VALUES " +
        "(" + config.get('general', 'sensorID') + "," + st[1] + ",'" + st[2] +
        "'," + st[3] + ",'" + st[4] + "'," + st[5] + ",'" + st[6] + "','" + st[7] +
        "','" + st[8] + "','" + st[9] + "','" + st[10] + "','" + st[11] +
        "','" + faa + "','" + checksum + "'," + st[14] + ",'" + st[13] + "');")
        x.execute(query)
        con.commit()
    except:
        con.rollback()

# Function used to reset the serial configuration
# in Linux in case its mangled by something'

```

```python

def reset_serial():
    call("stty -F " + config.get('gps', 'port') + " icanon", shell=True)


def get_today_mjd():
    today = datetime.datetime.utcnow()
    return jdutil.jd_to_mjd(jdutil.datetime_to_jd(today))


def main_routine():
    initConfig()
    t_print("GPS logger started!")
    reset_serial()
    con = dbConnect()
    counter = 0
    data = ""

    while(True):
        ser = serial.Serial(
            config.get('gps',
                       'port'),
            config.get('gps',
                       'baud'),
            timeout=0.1)
        sio = io.TextIOWrapper(io.BufferedRWPair(ser, ser), newline="\r")
        time.sleep(1)
        while True:
            temp = sio.readline()
            if(temp.find("GNRMC") == 1):
                data = temp
                data = data.split(",")
                sio.readline()  # Reading forward manually
                temp = sio.readline()
                temp = temp.split(",")
                data.append(str(temp[9]))
                data.append(str(get_today_mjd()))
                counter = counter + 1
                if(counter == int(config.get('general', 'discard_interval'))):
                    insert(con, data)
                    counter = 0
    dbClose(con)

if __name__ == '__main__':
    main_routine()
```

Appendix B

Sensor server software

B.1 Client

B.1.1 sensor_client.c

```c
#include "sensor_client.h"

/* CONFIG */
#define CONFIG_SERIAL_INTERFACE "serial_interface:"
#define CONFIG_CLIENT_ID "client_id:"
#define CONFIG_LOG_NAME "log_file_name:"
#define CONFIG_LOG_NMEA "log_nmea:"
#define CONFIG_FILE_PATH "client_config.ini"
#define DEFAULT_SERIAL_INTERFACE "/dev/ttyACM0"
#define CONFIG_CONNECTION_ATTEMPTS_MAX "connection_attempts_max:"
#define CONFIG_ENTRIES 5

struct config_map_entry conf_map[1];

static int identify(int session_fd, int id);
static int create_connection(struct sockaddr_in *serv_addr, int *session_fd,
                             char *ip, int portno);
static void receive_nmea(int gps_serial, struct raw_nmea_container *nmea_c);
static int format_nmea(struct raw_nmea_container *nmea_c);
static void initialize_config(struct config_map_entry *conf_map,
                              struct config *cfg);
static int start_client(int portno, char* ip);
static int usage(char *argv[]);


/* Identify the client for the server */
static int identify(int session_fd, int id)
{
    /* Converting from int to string */
    char id_str[5];
    bzero(id_str, 5);
    sprintf(id_str, " %d", id); //Notice the space in the second parameter.
    int read_status = 0;

    /* Declaring message string */
    char identify_message[sizeof(PROTOCOL_IDENTIFY) + sizeof(id_str) + 1];
```

```c

    /* copying */
    memcpy(identify_message, PROTOCOL_IDENTIFY, sizeof(PROTOCOL_IDENTIFY));
    memcpy(&identify_message[8],id_str, sizeof(id_str));

    write(session_fd, identify_message, sizeof(identify_message));

    char buffer[100];
    while ( (read_status = read(session_fd, buffer, sizeof(buffer)-1)) > 0) {
        if(strstr((char*)buffer, PROTOCOL_OK ) == (buffer)) {
            /* ID not used. Accepting. */
            t_print("ID %d accepted by server.\n", id);
            return 0;
        } else {
            /* ID in use. Rejected. */
            t_print("ID %d rejected by server, already in use.\n", id);
            return -1;
        }
    }
    /* Something happened during read. read() returns -1 at error */
    return read_status;
}

/* Create connection to server */
static int create_connection(struct sockaddr_in *serv_addr, int *session_fd,
                             char *ip, int portno)
{
    if((*session_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        t_print("Could not create socket\n");
        return -1;
    }

    memset(serv_addr, '0', sizeof(*serv_addr));

    serv_addr->sin_family = AF_INET;
    serv_addr->sin_port = htons(portno);

    if(inet_pton(AF_INET, ip, &(serv_addr->sin_addr))<=0) {
        t_print("inet_pton error occured!\n");
        return 1;
    }

    if( connect(*session_fd, (struct sockaddr *)serv_addr,
                sizeof(*serv_addr)) < 0) {
        return 1;
    }

    return 0;
}

/* Get chosen NMEA from GPS receiver */
static void receive_nmea(int gps_serial, struct raw_nmea_container *nmea_c)
{
    char buffer[SENTENCE_LENGTH * 2];
    int position = 0;
    memset(buffer, '\0',sizeof(buffer));

    bool rmc = false;
    bool gga = false;

    /* Get a load of THIS timebomb!! */
    while(1) {
```

```
        while(position < 100) {
            read(gps_serial, buffer+position, 1);
            if( buffer[position] == '\n' ) break;
            position++;
        }

        if(strstr(buffer, RMC ) != NULL) {
            memcpy(nmea_c->raw_rmc, buffer, position+1);
            nmea_c->raw_rmc[position + 2] = '\0';
            rmc = true;
        }

        if(strstr(buffer, GGA ) != NULL) {
            memcpy(nmea_c->raw_gga, buffer, position+1);
            nmea_c->raw_rmc[position + 2] = '\0';
            gga = true;
        }

        if(rmc && gga) {
            break;
        }
        position = 0;
    }
}

/* Send received NMEA data to server */
static int format_nmea(struct raw_nmea_container *nmea_c)
{
    int nmea_prefix_length = 6;
    memcpy(nmea_c->output, "NMEA \n", nmea_prefix_length);
    int total_length = 0;
    int newline_length = 1;

    /* RMC */
    int rmc_length = strlen(nmea_c->raw_rmc);
    memcpy( nmea_c->output+nmea_prefix_length, nmea_c->raw_rmc, rmc_length );
    //nmea_c->output[nmea_prefix_length + rmc_length + newline_length] = '\n';

    /* Updating total length */
    total_length = rmc_length + nmea_prefix_length; //+ newline_length;

    /* GGA */
    int gga_length = strlen(nmea_c->raw_gga);
    memcpy( nmea_c->output+total_length, nmea_c->raw_gga, gga_length );
    nmea_c->output[total_length + gga_length + newline_length] = '\n';

    /* Updating total length */
    total_length += gga_length + newline_length;

    return total_length;
}

static int make_log(struct raw_nmea_container *nmea_c, int id, char* log_name)
{
    /* Allocating memory for filename buffer */
    int filename_length = strlen(log_name) + 100;
    char filename[filename_length];

    /* Clearing buffer */
    memset(filename,'\0' ,filename_length);

    /* Copying name from loaded config */
```

```c
    strcpy(filename, log_name);

    /* Casting int to string */
    char id_string[10];
    memset(id_string,'\0', 10);
    sprintf(id_string, "%d", id);

    /* Concating filename and ID */
    strcat(filename, id_string);

    char log_buffer[SENTENCE_LENGTH * 2];
    memset(log_buffer, '\0', SENTENCE_LENGTH * 2);
    strcat(log_buffer, nmea_c->raw_rmc);
    log_buffer[strlen(log_buffer)-2] = '\0';
    log_buffer[strlen(log_buffer)-1] = ',';

    strcat(log_buffer, nmea_c->raw_gga);

    return log_to_file(filename, log_buffer, 1);
}

/* Setting up the config structure specific for the server */
static void initialize_config(struct config_map_entry *conf_map,
                              struct config *cfg)
{
    conf_map[0].entry_name = CONFIG_SERIAL_INTERFACE;
    conf_map[0].modifier = FORMAT_STRING;
    conf_map[0].destination = &cfg->serial_interface;

    conf_map[1].entry_name = CONFIG_CLIENT_ID;
    conf_map[1].modifier = FORMAT_INT;
    conf_map[1].destination = &cfg->client_id;

    conf_map[2].entry_name = CONFIG_LOG_NAME;
    conf_map[2].modifier = FORMAT_STRING;
    conf_map[2].destination = &cfg->log_name;

    conf_map[3].entry_name = CONFIG_LOG_NMEA;
    conf_map[3].modifier = FORMAT_INT;
    conf_map[3].destination = &cfg->log_nmea;

    conf_map[4].entry_name = CONFIG_CONNECTION_ATTEMPTS_MAX;
    conf_map[4].modifier = FORMAT_INT;
    conf_map[4].destination = &cfg->con_attempt_max;
}

static int start_client(int portno, char* ip)
{
    struct termios tty;
    memset (&tty, 0, sizeof tty);

    struct sockaddr_in serv_addr;
    int session_fd = 0;
    int connection_attempts = 1;
    int con_status;

    struct raw_nmea_container nmea_c;
    memset(&nmea_c, 0, sizeof(nmea_c));

    struct config cfg;

    initialize_config(conf_map, &cfg);
```

```
    int load_config_status = load_config(conf_map, CONFIG_FILE_PATH,
                                         CONFIG_ENTRIES);
    if(!load_config_status) {
        t_print("Failed to load the config, using default values\n");
        memcpy(cfg.serial_interface, DEFAULT_SERIAL_INTERFACE,
               strlen(DEFAULT_SERIAL_INTERFACE)*sizeof(char));

        /* Picking ID number for client at random */
        cfg.client_id = rand() % ID_MAX;
        t_print("Picked ID %d at random\n", cfg.client_id);

        /* Disabling logging */
        cfg.log_nmea = 0;

        /* Setting retry times to 10 */
        cfg.con_attempt_max = 10;
    } else {
        if(cfg.client_id == 0 || cfg.client_id > ID_MAX) {
            t_print("Client ID can not be less than 1 or more than %d!\n", ID_MAX);
            exit(0);
        }
    }

    /* Establishing connection to GPS receiver */
    int gps_serial = open_serial(cfg.serial_interface, GPS);
    if(gps_serial == -1) {
        t_print("Connection to GPS receiver failed! Exiting...\n");
        exit(0);
    } else {
        t_print("Connection to GPS receiver established!\n");
    }

    /* Establishing connection to server */
    while(connection_attempts <= cfg.con_attempt_max) {
        con_status = create_connection(&serv_addr, &session_fd, ip, portno);
        if(con_status == 0) {
            t_print("Connected to server!\n");
            break;
        }
        t_print("Connection attempt %d failed. Code %d\n", connection_attempts,
                con_status);
        sleep(1);
        connection_attempts++;
    }

    /* Identifying client for server */
    if( identify(session_fd, cfg.client_id) == -1 ) {
        exit(0);
    }

    if(cfg.log_nmea) {
        t_print("NMEA data logging enabled\n");
    }

    while (1) {
        receive_nmea(gps_serial, &nmea_c);
        int trans_length = format_nmea(&nmea_c);
        /* Writing to socket (server) */
        write(session_fd, nmea_c.output, trans_length);
        if(cfg.log_nmea) {
            make_log(&nmea_c, cfg.client_id, cfg.log_name);
        }
```

```c
    }
    return 0;
}

static int usage(char *argv[])
{
    t_print("Usage: %s -s <SERVER IP> -p <SERVER PORT>\n", argv[0]);
    return 0;
}

int main(int argc, char *argv[])
{
    char *ip_address = NULL;
    char *port_number = NULL;

    if(argc < 5) {
        usage(argv);
        return 0;
    }

    while (1) {
        char c;

        c = getopt (argc, argv, "s:p:");
        if (c == -1) {
            break;
        }
        switch (c) {
        case 's':
            ip_address = optarg;
            break;
        case 'p':
            port_number = optarg;
            break;
        default:
            usage(argv);
        }
    }

    if(ip_address == NULL || port_number == NULL) {
        t_print("Missing parameters!\n");
        exit(0);
    }

    start_client(atoi(port_number), ip_address);
    return 0;
}
```

B.1.2 sensor_client.h

```c
#ifndef SENSOR_CLIENT_H
#define SENSOR_CLIENT_H

// Mine
#include "net.h"
#include "utils.h"
#include "protocol.h"
#include "nmea.h"
#include "utils.h"
#include "serial.h"
```

```
struct config {
    char serial_interface[100];
    int client_id;
    char log_name[100];
    int log_nmea;
    int con_attempt_max;
};

/* Used by the client */
struct raw_nmea_container {
    /* Raw data */
    char raw_gga[SENTENCE_LENGTH];
    char raw_rmc[SENTENCE_LENGTH];
    char output[SENTENCE_LENGTH * 2];
};

#endif /* !SENSOR_CLIENT_H */
```

B.1.3 client_config.ini

```
serial_interface: /dev/ttyACM0
client_id: 1
log_nmea: 1
log_file_name: log_sensor
connection_attempts_max: 10
```

B.1.4 query_csac.py

```
import ctypes
import fileinput, sys
import datetime
import time
import io
import os
import serial

def main_routine():
    # Opening serial stream, use ASCII
    ser = serial.Serial("/dev/ttyUSB0",57600, timeout=0.1)
    sio = io.TextIOWrapper(io.BufferedRWPair(ser, ser),encoding='ascii',newline="\r\n")

    # Open log file, mostly used for debug
    log_file = open("query_csac.txt", "a+")

    # The query to use
    query = sys.argv[1].strip("\r\n")

    # How long to sleep between read from serial con.
    sleep_time = 0.2

    # The minimum length of the answer
    # for the given query.
    minimum_len = 0

    if(query == '^' or query == '6'):
        minimum_len = 80
    elif(query == 'F'):
        sleep_time = 0.5
```

```python
            minimum_len = 10
        elif(query == 'M'):
            minimum_len = 6
        elif (query == 'S'):
            sleep_time = 3
            minimum_len = 2
        else:
            minimum_len = 1

        response_len = 0

        if(len(query) > 1):
            query = "!" + query + "\r\n"

        retry_count = 0

        while (response_len < minimum_len):
            ser.write(bytes(query))
            time.sleep(sleep_time)
            response = sio.readline()
            response = response.strip("\r\n\x00")
            response_len = len(response)
            retry_count = retry_count + 1

        print(response)
        ser.close()
        query = query.strip("\r\n")
        log_string = ("Issued query " + "'" +  query + "' " + str(retry_count) + " times\n")
        log_file.write(log_string)
if __name__ == '__main__':
        main_routine()
```

B.2 Server

B.2.1 sensor_server.c

```c
#include "sensor_server.h"

/* VERSION */
#define PROGRAM_VERSION "0.8c"

/* ERRORS */
#define ERROR_MAX_CLIENTS_REACHED "CONNECTION REJECTED: MAXIMUM NUMBER OF CLIENTS REACHED\n"
#define ERROR_CONFIG_LOAD_FAILED "CONFIG LOAD FAILED: CONFIG FILE CORRUPTED\n"
#define ERROR_SEMAPHORE_CREATION_FAILED "SEMAPHORE CREATION FAILED\n"
#define ERROR_SOCKET_OPEN_FAILED "ERROR: FAILED TO OPEN SOCKET\n"
#define ERROR_SOCKET_BINDING "ERROR: FAILED TO BIND ON %d\n"
#define ERROR_CONNECTION_ACCEPT "ERROR: FAILED TO ACCEPT CONNECTION (%d)\n"
#define ERROR_FAILED_FORK "ERROR: FORK FAILED (%d)\n"
#define ERROR_MISSING_PARAMS "MISSING PARAMETERS!\n"

/* GENERAL STRINGS */
#define PROCESS_REAPED "Process %d reaped. Status: %d Signum: %d\n"
#define SIGTERM_RECEIVED "[%d] SIGTERM received!\n"
#define SIGINT_RECEIVED "[%d] SIGINT received!\n"
#define STOPPING_SERVER "Stopping server...\n"
#define CONFIG_LOADED "Config loaded!\n"
#define SERVER_RUNNING "Server is running. Accepting connections.\n"
```

```c
#define WAITING_FOR_CONNECTIONS "Waiting for connections...\n"
#define CON_ACCEPTED "Connection accepted\n"
#define CLIENT_DISCONNECTED "[%d] Disconnected\n"
#define SERVER_STOPPED "Server STOPPED!\n"
#define SERVER_STARTING "Sensor server starting...\n"
#define CLIENT_KICKED "Client was kicked\n"

/* USAGE() STRINGS */
#define USAGE_DESCRIPTION "Required argument:\n\t -p <PORT NUMBER>\n\n"
#define USAGE_PROGRAM_INTRO "Sensor_server: Server part of GPS Jamming/Spoofing system\n\n"
#define USAGE_USAGE "Usage: %s [ARGS]\n\n"

/* CONFIG CONSTANTS*/
#define CONFIG_FILE_PATH "config.ini"
#define CONFIG_SERVER_MAX_CONNECTIONS "max_clients:"
#define CONFIG_SERVER_WARM_UP "warm_up:"
#define CONFIG_SERVER_HUMANLY_READABLE "humanly_readable_dumpdata:"
#define CONFIG_CSAC_PATH "csac_serial_interface:"
#define CONFIG_LOGGING "logging:"
#define CONFIG_LOG_PATH "log_path:"
#define CONFIG_CSAC_LOG_PATH "csac_log_path:"
#define CONFIG_CSAC_LOGGING "csac_logging:"
#define SERVER_CONFIG_ENTRIES 8

/* Server data and stats */
struct server_data *s_data;

/* Shared synchro elements */
struct server_synchro *s_synch;

/* Used by sig handlers */
volatile sig_atomic_t done;

/* Pointer to shared memory containing the client list */
struct client_table_entry *client_list;

/* Pointer to shared memory containing config */
struct server_config *s_conf;

/* Pointer to shared CSAC_filter data */
struct csac_filter_data *cfd;

static void remove_client_by_pid(pid_t pid);
void remove_client_by_id(int id);
static struct client_table_entry* create_client(struct client_table_entry* ptr);
static void handle_sigchld(int signum);
static void handle_sig(int signum);
static void initialize_config(struct config_map_entry *conf_map,
                              struct server_config *s_conf);
static void start_server(int port_number);
static int usage(char *argv[]);

/* Prints a formatted string containing server info to monitor */
void print_server_data(struct client_table_entry *monitor)
{
    char buffer [1000];
    int snprintf_status = 0;
    struct tm *loctime_started;
    loctime_started = localtime (&s_data->started);

    s_write(&(monitor->transmission), SERVER_TABLE_LABEL,
            sizeof(SERVER_TABLE_LABEL));
```

```
    s_write(&(monitor->transmission), HORIZONTAL_BAR, sizeof(HORIZONTAL_BAR));

    snprintf_status = snprintf( buffer, 1000,
                                "PID: %d\n" \
                                "Number of clients: %d\n" \
                                "Number of sensors: %d\n" \
                                "Max clients: %d\n" \
                                "Sensor Warm-up time: %ds\n" \
                                "Dump humanly readable data: %d\n" \
                                "Started: %s" \
                                "Version: %s\n",
                                s_data->pid,
                                s_data->number_of_clients,
                                s_data->number_of_sensors,
                                s_conf->max_clients,
                                s_conf->warm_up_seconds,
                                s_conf->human_readable_dumpdata,
                                asctime (loctime_started),
                                s_data->version);

    s_write(&(monitor->transmission), buffer, snprintf_status);
    s_write(&(monitor->transmission), HORIZONTAL_BAR, sizeof(HORIZONTAL_BAR));
}

struct client_table_entry* get_client_by_id(int id)
{
    struct client_table_entry* client_list_iterate;
    struct client_table_entry* temp;
    int found = 0;

    sem_wait(&(s_synch->client_list_mutex));
    list_for_each_entry_safe(client_list_iterate, temp, &client_list->list, list) {
        if(client_list_iterate->client_id == id) {
            found = 1;
            break;
        }
    }
    sem_post(&(s_synch->client_list_mutex));
    if(found) {
        return client_list_iterate;
    } else {
        return NULL;
    }
}

/* Removes a client with the given PID */
static void remove_client_by_pid(pid_t pid)
{
    struct client_table_entry* client_list_iterate;
    struct client_table_entry* temp_remove;

    sem_wait(&(s_synch->client_list_mutex));
    list_for_each_entry_safe(client_list_iterate, temp_remove,&client_list->list,
                             list) {
        if(client_list_iterate->pid == pid) {
            if(client_list_iterate->client_id > 0) {
                s_data->number_of_sensors--;
            }
            list_del(&client_list_iterate->list);
        }
    }
    s_data->number_of_clients--;
```

```c
    sem_post(&(s_synch->client_list_mutex));
}

/* Removes a client with the given ID */
void remove_client_by_id(int id)
{
    struct client_table_entry* client_list_iterate;
    struct client_table_entry* temp_remove;

    sem_wait(&(s_synch->client_list_mutex));
    list_for_each_entry_safe(client_list_iterate, temp_remove,&client_list->list,
                             list) {
        if(client_list_iterate->client_id == id) {
            list_del(&client_list_iterate->list);
        }
    }
    s_data->number_of_clients--;
    sem_post(&(s_synch->client_list_mutex));
}

/* Creates an entry in the client list structure and returns a pointer to it*/
static struct client_table_entry* create_client(struct client_table_entry* ptr)
{
    sem_wait(&(s_synch->client_list_mutex));
    s_data->number_of_clients++;
    struct client_table_entry* tmp;
    tmp = (client_list + s_data->number_of_clients);
    list_add_tail( &(tmp->list), &(ptr->list) );
    sem_post(&(s_synch->client_list_mutex));

    return tmp;
}

/* SIGCHLD Handler */
static void handle_sigchld(int signum)
{
    pid_t pid;
    int   status;
    while ((pid = waitpid(-1, &status, WNOHANG)) != -1) {
        if(pid == 0) {
            break;
        }

        if(pid > 0) {
            remove_client_by_pid(pid);
            t_print(PROCESS_REAPED, pid, status, signum);
        }
    }
}

/* SIGTERM/INT Handler */
static void handle_sig(int signum)
{
    if(signum == 15) {
        t_print(SIGTERM_RECEIVED, getpid());
    }
    if(signum == 2) {
        t_print(SIGINT_RECEIVED, getpid());
    }
    t_print(STOPPING_SERVER, getpid());
    done = 1;
}
```

```c

/* Setting up the config structure specific for the server */
static void initialize_config(struct config_map_entry *conf_map,
                              struct server_config *s_conf)
{
    conf_map[0].entry_name = CONFIG_SERVER_MAX_CONNECTIONS;
    conf_map[0].modifier = FORMAT_INT;
    conf_map[0].destination = &s_conf->max_clients;

    conf_map[1].entry_name = CONFIG_SERVER_WARM_UP;
    conf_map[1].modifier = FORMAT_INT;
    conf_map[1].destination = &s_conf->warm_up_seconds;

    conf_map[2].entry_name = CONFIG_SERVER_HUMANLY_READABLE;
    conf_map[2].modifier = FORMAT_INT;
    conf_map[2].destination = &s_conf->human_readable_dumpdata;

    conf_map[3].entry_name = CONFIG_CSAC_PATH;
    conf_map[3].modifier = FORMAT_STRING;
    conf_map[3].destination = &s_conf->csac_path;

    conf_map[4].entry_name = CONFIG_LOGGING;
    conf_map[4].modifier = FORMAT_INT;
    conf_map[4].destination = &s_conf->logging;

    conf_map[5].entry_name = CONFIG_LOG_PATH;
    conf_map[5].modifier = FORMAT_STRING;
    conf_map[5].destination = &s_conf->log_path;

    conf_map[6].entry_name = CONFIG_CSAC_LOG_PATH;
    conf_map[6].modifier = FORMAT_STRING;
    conf_map[6].destination = &s_conf->csac_log_path;

    conf_map[7].entry_name = CONFIG_CSAC_LOGGING;
    conf_map[7].modifier = FORMAT_INT;
    conf_map[7].destination = &s_conf->csac_logging;
}

/*
* Main loop for the server.
* Forks everytime a client connects and calls setup_session()
*/
static void start_server(int port_number)
{
    /* Initializing variables */
    int server_sockfd;
    struct sockaddr_in serv_addr;
    struct config_map_entry conf_map[SERVER_CONFIG_ENTRIES];

    /* Initializing config structure */
    s_conf = mmap(NULL, sizeof(struct server_config), PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    initialize_config(conf_map, s_conf);

    /* Loading config */
    int load_config_status = load_config(conf_map, CONFIG_FILE_PATH,
                                         SERVER_CONFIG_ENTRIES);

    /* Falling back to default if load_config fails */
    if(load_config_status) {
        t_print(CONFIG_LOADED);
        client_list = mmap(NULL,
```

```
                            (s_conf->max_clients * sizeof(struct client_table_entry)),
                            PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    } else {
        t_print(ERROR_CONFIG_LOAD_FAILED);
        exit(0);
    }

    INIT_LIST_HEAD(&client_list->list);

    /* Create and initialize shared memory for server data */
    s_data = mmap(NULL, sizeof(struct server_data), PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    bcopy(PROGRAM_VERSION, s_data->version,4);
    s_data->pid = getpid();
    s_data->started = time(NULL);

    /* Init shared semaphores and sync elements */
    s_synch = mmap(NULL, sizeof(struct server_synchro), PROT_READ | PROT_WRITE,
                   MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    sem_init(&(s_synch->ready_mutex), 1, 1);
    sem_init(&(s_synch->client_list_mutex), 1, 1);
    sem_init(&(s_synch->csac_mutex), 1, 1);

    /* Init pointer to shared CSAC_filter data */
    cfd = mmap(NULL, sizeof(struct csac_filter_data), PROT_READ | PROT_WRITE,
               MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if( &(s_synch->ready_mutex) == SEM_FAILED
            || &(s_synch->client_list_mutex) == SEM_FAILED) {
        t_print(ERROR_SEMAPHORE_CREATION_FAILED);
        sem_close(&(s_synch->ready_mutex));
        sem_close(&(s_synch->client_list_mutex));
        exit(1);
    }


    pid_t f_pid;
    f_pid = fork();
    if(f_pid == 0) {
        t_print("Forked out CSAC filter [%d]\n", getpid());
        start_csac_filter(cfd);
        _exit(0);
    }

    /* Registering the SIGINT handler */
    struct sigaction sigint_action;
    memset(&sigint_action, 0, sizeof(struct sigaction));
    sigint_action.sa_handler = handle_sig;
    sigaction(SIGINT, &sigint_action, NULL);
    if (sigaction(SIGCHLD, &sigint_action, 0) == -1) {
        perror(0);
        exit(1);
    }

    /* Registering the SIGTERM handler */
    struct sigaction sigterm_action;
    memset(&sigterm_action, 0, sizeof(struct sigaction));
    sigterm_action.sa_handler = handle_sig;
    sigaction(SIGTERM, &sigterm_action, NULL);
    if (sigaction(SIGCHLD, &sigterm_action, 0) == -1) {
        perror(0);
        exit(1);
```

```c
    }

    /* Registering the SIGCHLD handler */
    struct sigaction child_action;
    child_action.sa_handler = &handle_sigchld;
    sigemptyset(&child_action.sa_mask);
    child_action.sa_flags = SA_RESTART | SA_NOCLDSTOP;
    if (sigaction(SIGCHLD, &child_action, 0) == -1) {
        perror(0);
        exit(1);
    }

    /* Initialize socket */
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sockfd < 0) {
        die(62,ERROR_SOCKET_OPEN_FAILED);
    }

    /*
    * Initializing the server address struct:
    * AF_INET = IPV4 Internet protocol
    * INADDR_ANY = Accept connections to all IPs of the machine
    * htons(port_number) = Endianess: network to host long(port number).
    */
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(port_number);

    /*
    * Assigns the address (serv_addr) to the socket
    * referred to by server_sockfd.
    */
    if (bind(server_sockfd, (struct sockaddr *) &serv_addr,
             sizeof(serv_addr)) < 0) {
        t_print(ERROR_SOCKET_BINDING, port_number);
        exit(1);
    }

    /* Marking the connection for listening*/
    listen(server_sockfd,SOMAXCONN);

    int session_fd = 0;
    t_print(SERVER_RUNNING);
    while (!done) {
        t_print(WAITING_FOR_CONNECTIONS);
        session_fd = accept(server_sockfd,0,0);
        if (session_fd==-1) {
            if (errno==EINTR) continue;
            t_print(ERROR_CONNECTION_ACCEPT,errno);
        }
        if(s_data->number_of_clients == s_conf->max_clients) {
            write(session_fd, ERROR_MAX_CLIENTS_REACHED, sizeof(ERROR_MAX_CLIENTS_REACHED));
            close(session_fd);
        } else {
            struct client_table_entry *new_client = create_client(client_list);
            pid_t pid=fork();
            if (pid==-1) {
                t_print(ERROR_FAILED_FORK, errno);
                /* WHAT HAPPENS WITH THE LIST WHEN FORK FAILS? DEAL WITH IT.*/
            } else if (pid==0) {
                close(server_sockfd);
```

```c
                setup_session(session_fd, new_client);
                close(session_fd);
                if(new_client->marked_for_kick) {
                    t_print(CLIENT_KICKED, getpid());
                }
                t_print(CLIENT_DISCONNECTED, getpid());
                _exit(0);
            } else {
                t_print(CON_ACCEPTED);
                close(session_fd);
            }
        }
    }

    /* Destroying semaphores */
    sem_destroy(&(s_synch->csac_mutex));
    sem_destroy(&(s_synch->ready_mutex));
    sem_destroy(&(s_synch->client_list_mutex));

    /* Freeing */
    munmap(client_list, sizeof(struct client_table_entry));
    munmap(s_data, sizeof(struct server_data));
    munmap(cfd, sizeof(struct csac_filter_data));
    munmap(s_synch, sizeof(struct server_synchro));

    /* Closing server FD */
    close(server_sockfd);
    t_print(SERVER_STOPPED);
}

static int usage(char *argv[])
{
    printf(USAGE_USAGE, argv[0]);
    printf(USAGE_PROGRAM_INTRO);
    printf(USAGE_DESCRIPTION);
    return 0;
}

int main(int argc, char *argv[])
{
    char *port_number = NULL;

    /* getopt silent mode set */
    opterr = 0;

    if(argc < 3) {
        usage(argv);
        return 0;
    }

    while (1) {
        char c;

        c = getopt (argc, argv, "p:");
        if (c == -1) {
            break;
        }

        switch (c) {
        case 'p':
            port_number = optarg;
            break;
```

```c
        }
    }

    if(port_number == NULL) {
        printf(ERROR_MISSING_PARAMS);
    }

    t_print(SERVER_STARTING);
    start_server(atoi(port_number));
    exit(0);
}
```

B.2.2 sensor_server.h

```c
/**
 * @file sensor_server.h
 * @author Aril Schultzen
 * @date 13.04.2016
 * @brief File containing function prototypes, structs and includes for sensor_server.c
 */

#ifndef SENSOR_SERVER_H
#define SENSOR_SERVER_H

#define PATH_LENGTH_MAX 1000

#include <fcntl.h>
#include <sys/stat.h>
#include "session.h"
#include "serial.h"
#include "sensor_server_common.h"
#include "csac_filter.h"

/*!@struct*/
/*!@brief Contains configuration values for the server
*/
struct server_config {
    int max_clients;
    int warm_up_seconds;
    int human_readable_dumpdata;
    char csac_path[PATH_LENGTH_MAX];
    int logging;
    char log_path[PATH_LENGTH_MAX];
    int csac_logging;
    char csac_log_path[PATH_LENGTH_MAX];
};

/*
* Made extern because the sessions should
* exit if the server is given a SIGINT/TERM
*/
extern volatile sig_atomic_t done;

/* Also used by session and action */
extern struct client_table_entry *client_list;
extern struct server_data *s_data;
extern struct server_synchro *s_synch;
extern struct server_config *s_conf;
extern struct csac_filter_data *cfd;

```

```c
/** @brief Removes a client whose ID matches parameter
 *
 * Iterates through the linked list and removes the
 * node containing the client whose ID matches the parameter.
 * @param id ID for the client
 * @return Void
 */
void remove_client_by_id(int id);

/** @brief Returns a client whose ID matches parameter
 *
 * Iterates through the linked list and returns
 * a pointer to the client_table_entry struct in the
 * list that corresponds with the parameter.
 * @param id ID for the client
 * @return client_table_entry *
 */
struct client_table_entry* get_client_by_id(int id);

/** @brief Prints information about the server.
 *
 * Transmits info about the server:
 * Time when started, PID, number of clients,
 * number of sensors, max number of clients,
 * sensor warm-up time and version.
 *
 * @param client MONITOR who made the request.
 * @return Void
 */
void print_server_data(struct client_table_entry *monitor);

#endif /* !SENSOR_SERVER_H */
```

B.2.3 config.ini

```
humanly_readable_dumpdata: 1
max_clients: 10
warm_up: 24000
csac_serial_interface: /dev/ttyUSB0
logging: 1
log_path: server_log.txt
csac_logging: 1
csac_log_path: csac_log.txt
```

B.2.4 sensor_server_common.h

```c
/**
 * @file sensor_server_common.h
 * @author Aril Schultzen
 * @date 13.04.2016
 * @brief File containing structs and defines used by session.c, analyzer.c, sensors_server.c and actions.c
 */

#ifndef SENSOR_SERVER_COMMON_H
#define SENSOR_SERVER_COMMON_H

#include <semaphore.h>
#include "net.h"
#include "colors.h"
```

```c

/* General */
#define SERVER_TABLE_LABEL "SERVER DATA\n"
#define HORIZONTAL_BAR "=================================================================================\n"
#define ERROR_NO_CLIENT "ERROR: No such client\n"
#define ERROR_NO_FILENAME "ERROR: No FILENAME specified\n"
#define MAX_FILENAME_SIZE 30
#define ID_AS_STRING_MAX 10

/* Errors */
#define ERROR_CODE_NO_FILE -1
#define ERROR_CODE_READ_FAILED -2
#define ERROR_NO_FILE "ERROR:No such file\n"
#define ERROR_READ_FAILED "ERROR:Failed to read file\n"

/*
* command_code struct is used by the parser
* to convey an easy to compare command code, as well
* as any parameter belonging to that command
*/
struct command_code {
    int code;
    char parameter[MAX_PARAMETER_SIZE];
    int id_parameter;
};

/*!@struct*/
/*!@brief Data used by the red_dev_filter.
* Read from file.
*/
struct ref_dev_data {
    double alt_ref;
    double lon_ref;
    double lat_ref;
    double speed_ref;
    double alt_dev;
    double lon_dev;
    double lat_dev;
    double speed_dev;
};

struct disturbed_values {
    int lat_disturbed;
    int lon_disturbed;
    int alt_disturbed;
    int speed_disturbed;
};

struct ref_dev {
    struct ref_dev_data rdd;
    int moved;
    int was_moved;
    struct disturbed_values dv;
};

struct filters {
    struct ref_dev rdf;
};

/*
* CLIENT TABLE STRUCT
*
```

```c
* list_head list: The head in the list of clients
* pid: Process ID for the client connection (See "fork")
* session_fd: The file descriptor for the session.
* client_id: The connected clients ID
* iobuffer: A general purpose buffer for in and output
* heartbeat_timeout: Number of seconds of inactivity before disconnect
* ip: Clients IP Address.
* cm: Command code. Used for quick comparison after commands
* are parsed by command parser.
*/

/*!@struct*/
/*!@brief Contain information about every client that is connected.
*/
struct client_table_entry {
    struct list_head list;              /* The head of the client list */
    struct transmission_s transmission; /* Everything needed for socket com. */
    struct timeval heartbeat_timeout;   /* Timeout in seconds if not activity */
    struct command_code cm;             /* See command code */
    struct nmea_container nmea;         /* All NMEA data associated with the client */
    pid_t pid;                          /* The process ID */
    time_t timestamp;                   /* When last analyzed */
    int client_id;                      /* Clients ID */
    int client_type;                    /* Client type, SENSOR or MONITOR */
    int ready;                            /* Ready status */
    int marked_for_kick;                /* Marked for kicked at next opportunity */
    char ip[INET_ADDRSTRLEN];           /* Clients IP address */
    struct filters fs;
};

/* Server info shared with processes */
struct server_data {
    int number_of_clients;    /* Number of clients currently connected */
    int number_of_sensors;    /* Number of sensors, subset of clients */
    time_t started;            /* When the server was started */
    pid_t pid;               /* Servers PID */
    char version[4];         /* Version of server software */
};

/* Synchronization elements shared with processes */
struct server_synchro {
    sem_t ready_mutex;
    sem_t csac_mutex;
    sem_t client_list_mutex;
    volatile int ready_counter;
};

/*
* Roles of client, either SENSOR or MONITOR.
* A monitor is only used to monitor the programs state.
*/
enum client_type {
    SENSOR,
    MONITOR
};

#endif /* !SENSOR_SERVER_COMMON_H */
```

B.2.5 session.c

```c
#include "session.h"

#define CLIENT_TIMEOUT 5
#define MONITOR_TIMEOUT 1000
#define UNIDENTIFIED_TIMEOUT 100

/* ERRORS*/
#define ERROR_ILLEGAL_COMMAND "ERROR:Illegal command\n"
#define ERROR_NO_ID "ERROR:Client not identified\n"
#define ERROR_ID_IN_USE "ERROR:ID in use\n"
#define ERROR_ILLEGAL_MESSAGE_SIZE "\rERROR:Illegal message size\n"
#define ERROR_WARMUP_NOT_SENSOR "ERROR:Warm-up only applies to sensors\n"
#define ERROR_DUMPDATA_FAILED "ERROR:Failed to dump data\n"
#define ERROR_LOADDATA_FAILED "ERROR:Failed to load data\n"
#define ERROR_NO_COMMAND  "ERROR:No command specified\n"
#define ERROR_LRFD_LOAD_FAILED "ERROR:Failed to laod REF_DEV_FILTER data from file\n"

static int nmea_ready();
static void extract_nmea_data(struct client_table_entry *cte);
static void calculate_nmea_average(struct client_table_entry *cte);
static void calculate_nmea_diff(struct client_table_entry *cte);
static int set_timeout(struct client_table_entry *target,
                       struct timeval h_timeout);
static int parse_input(struct client_table_entry *cte);
static int respond(struct client_table_entry *cte);

/*
* Used by spawned client processes to "mark" that their NMEA
* data is ready for processing. Works as a barrier in a way.
*/
static int nmea_ready()
{
    struct client_table_entry* client_list_iterate;
    struct client_table_entry* temp;
    int ready = 0;

    list_for_each_entry_safe(client_list_iterate, temp, &client_list->list, list) {
        if(client_list_iterate->ready == 1) {
            ready++;
        }
    }
    if(ready == s_data->number_of_sensors) {
        return 1;
    } else {
        return 0;
    }
}

/* Extract position data from NMEA */
static void extract_nmea_data(struct client_table_entry *cte)
{
    int buffsize = 100;
    char buffer[buffsize];
    memset(&buffer, 0, buffsize);

    /* Extracting latitude */
    substring_extractor(LATITUDE_START,LATITUDE_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_rmc, strlen(cte->nmea.raw_rmc));
    cte->nmea.lat_current = atof(buffer);

```

```
    /* Extracting longitude */
    substring_extractor(LONGITUDE_START,LONGITUDE_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_rmc, strlen(cte->nmea.raw_rmc));
    cte->nmea.lon_current = atof(buffer);

    /* Extracting altitude */
    substring_extractor(ALTITUDE_START,ALTITUDE_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_gga, strlen(cte->nmea.raw_gga));
    cte->nmea.alt_current = atof(buffer);

    /* Extracting speed */
    substring_extractor(SPEED_START,SPEED_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_rmc, strlen(cte->nmea.raw_rmc));
    cte->nmea.speed_current = atof(buffer);
}

/* Calculate the average NMEA values */
static void calculate_nmea_average(struct client_table_entry *cte)
{
    /* Updating number of samples */
    cte->nmea.n_samples++;

    /* Updating total */
    cte->nmea.lat_total = cte->nmea.lat_total + cte->nmea.lat_current;
    cte->nmea.lon_total = cte->nmea.lon_total + cte->nmea.lon_current;
    cte->nmea.alt_total = cte->nmea.alt_total + cte->nmea.alt_current;
    cte->nmea.speed_total = cte->nmea.speed_total + cte->nmea.speed_current;

    cte->nmea.lat_average = ( cte->nmea.lat_total / cte->nmea.n_samples );
    cte->nmea.lon_average = ( cte->nmea.lon_total / cte->nmea.n_samples );
    cte->nmea.alt_average = ( cte->nmea.alt_total / cte->nmea.n_samples );
    cte->nmea.speed_average = ( cte->nmea.speed_total / cte->nmea.n_samples );
}

/*
* Calculate the diff between current
* NMEA values and the average values.
*/
static void calculate_nmea_diff(struct client_table_entry *cte)
{
    cte->nmea.lat_avg_diff = (cte->nmea.lat_current - cte->nmea.lat_average);
    cte->nmea.lon_avg_diff = (cte->nmea.lon_current - cte->nmea.lon_average);
    cte->nmea.alt_avg_diff = (cte->nmea.alt_current - cte->nmea.alt_average);
    cte->nmea.speed_avg_diff = (cte->nmea.speed_current - cte->nmea.speed_average);
}

static int set_timeout(struct client_table_entry *target,
                       struct timeval h_timeout)
{
    /* setsockopt return -1 on error and 0 on success */
    target->heartbeat_timeout = h_timeout;
    if (setsockopt (target->transmission.session_fd, SOL_SOCKET,
                    SO_RCVTIMEO, (char *)&target->heartbeat_timeout, sizeof(struct timeval)) < 0) {
        t_print("an error: %s\n", strerror(errno));
        return 0;
    }
    return 1;
}

/*
* Parses input from clients. Return value indicates status.
* Uses the command_code struct to convey parameter and command code.
```

```c
*
* Returns -1 if size is wrong
* Returns 0 if protocol is not followed
* Returns 1 if all is ok
*/

static int parse_input(struct client_table_entry *cte)
{
    char *incoming = cte->transmission.iobuffer;

    /* INPUT TO BIG */
    if(strlen(incoming) > (MAX_PARAMETER_SIZE + MAX_COMMAND_SIZE) + 2) {
        return -1;
    }

    /* INPUT TO SMALL */
    if(strlen(incoming) < (MIN_PARAMETER_SIZE + MIN_COMMAND_SIZE) + 2) {
        return -1;
    }

    /* ZEROING COMMAND CODE */
    cte->cm.code = 0;
    /* ZEROING ID_PARAMETER */
    cte->cm.id_parameter = 0;

    /* NMEA */
    if(strstr((char*)incoming, PROTOCOL_NMEA ) == (incoming)) {
        cte->cm.code = CODE_NMEA;
    }

    /* IDENTIFY */
    else if(strstr((char*)incoming, PROTOCOL_IDENTIFY ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_IDENTIFY) );
        memcpy(cte->cm.parameter, (incoming)+(strlen(PROTOCOL_IDENTIFY)*(sizeof(char))),
               length);
        cte->cm.code = CODE_IDENTIFY;
    }

    /* IDENTIFY SHORT */
    else if(strstr((char*)incoming, PROTOCOL_IDENTIFY_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_IDENTIFY_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_IDENTIFY_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_IDENTIFY;
    }

    /* DUMPDATA */
    else if(strstr((char*)incoming, PROTOCOL_DUMPDATA ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_DUMPDATA) );
        memcpy(cte->cm.parameter, (incoming)+(strlen(PROTOCOL_DUMPDATA)*(sizeof(char))),
               length);
        cte->cm.code = CODE_DUMPDATA;
    }

    /* DUMPDATA_SHORT */
    else if(strstr((char*)incoming, PROTOCOL_DUMPDATA_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_DUMPDATA_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_DUMPDATA_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_DUMPDATA;
    }

```

```
    /* PRINT_LOCATION */
    else if(strstr((char*)incoming, PROTOCOL_PRINT_LOCATION ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_PRINT_LOCATION) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_PRINT_LOCATION)*(sizeof(char))), length);
        cte->cm.code = CODE_PRINT_LOCATION;
    }

    /* PRINT_LOCATION_SHORT */
    else if(strstr((char*)incoming, PROTOCOL_PRINT_LOCATION_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_PRINT_LOCATION_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_PRINT_LOCATION_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_PRINT_LOCATION;
    }

    /* PRINTTIME */
    else if(strstr((char*)incoming, PROTOCOL_PRINTTIME ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_PRINTTIME) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_PRINTTIME)*(sizeof(char))), length);
        cte->cm.code = CODE_PRINTTIME;
    }

    /* PRINTCLIENTS */
    else if(strstr((char*)incoming, PROTOCOL_PRINTCLIENTS ) == (incoming) ||
            strstr((char*)incoming, PROTOCOL_PRINTCLIENTS_SHORT ) == (incoming)) {
        cte->cm.code = CODE_PRINTCLIENTS;
    }

    /* PRINTSERVER */
    else if(strstr((char*)incoming, PROTOCOL_PRINTSERVER ) == (incoming) ||
            strstr((char*)incoming, PROTOCOL_PRINTSERVER_SHORT ) == (incoming)) {
        cte->cm.code = CODE_PRINTSERVER;
    }

    /* KICK */
    else if(strstr((char*)incoming, PROTOCOL_KICK ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_KICK) );
        memcpy(cte->cm.parameter, (incoming)+(strlen(PROTOCOL_KICK)*(sizeof(char))),
               length);
        cte->cm.code = CODE_KICK;
    }

    /* EXIT */
    else if(strstr((char*)incoming, PROTOCOL_EXIT ) == (incoming)) {
        cte->cm.code = CODE_DISCONNECT;
    }

    /* DISCONNECT */
    else if(strstr((char*)incoming, PROTOCOL_DISCONNECT ) == (incoming) ||
            strstr((char*)incoming, PROTOCOL_DISCONNECT_SHORT ) == (incoming)) {
        cte->cm.code = CODE_DISCONNECT;
    }

    /* HELP */
    else if(strstr((char*)incoming, PROTOCOL_HELP ) == (incoming) ||
            strstr((char*)incoming, PROTOCOL_HELP_SHORT ) == (incoming)) {
        cte->cm.code = CODE_HELP;
    }

    /* PRINTAVGDIFF */
```

```
    else if(strstr((char*)incoming, PROTOCOL_PRINTAVGDIFF ) == (incoming) ||
            strstr((char*)incoming, PROTOCOL_PRINTAVGDIFF_SHORT ) == (incoming)) {
        cte->cm.code = CODE_PRINTAVGDIFF;
    }

    /* LISTDUMPS */
    else if(strstr((char*)incoming, PROTOCOL_LISTDUMPS ) == (incoming) ||
            strstr((char*)incoming, PROTOCOL_LISTDUMPS_SHORT ) == (incoming)) {
        cte->cm.code = CODE_LISTDUMPS;
    }

    /* LOADDATA */
    else if(strstr((char*)incoming, PROTOCOL_LOADDATA ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_LOADDATA) );
        memcpy(cte->cm.parameter, (incoming)+(strlen(PROTOCOL_LOADDATA)*(sizeof(char))),
               length);
        cte->cm.code = CODE_LOADDATA;
    }

    /* LOADDATA_SHORT */
    else if(strstr((char*)incoming, PROTOCOL_LOADDATA_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_LOADDATA_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_LOADDATA_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_LOADDATA;
    }

    /* QUERYCSAC */
    else if(strstr((char*)incoming, PROTOCOL_QUERYCSAC ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_QUERYCSAC) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_QUERYCSAC)*(sizeof(char))), length);
        cte->cm.code = CODE_QUERYCSAC;
    }

    /* QUERYCSAC_SHORT */
    else if(strstr((char*)incoming, PROTOCOL_QUERYCSAC_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_QUERYCSAC_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_QUERYCSAC_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_QUERYCSAC;
    }

    /* PRINT_LOADRFDATA */
    else if(strstr((char*)incoming, PROTOCOL_LOADRFDATA ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_LOADRFDATA) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_LOADRFDATA)*(sizeof(char))), length);
        cte->cm.code = CODE_LOADRFDATA;
    }

    /* PRINT_LOADRFDATA_SHORT */
    else if(strstr((char*)incoming, PROTOCOL_LOADRFDATA_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_LOADRFDATA_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_LOADRFDATA_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_LOADRFDATA;
    }

    /* PROTOCOL_PRINTCFD */
    else if(strstr((char*)incoming, PROTOCOL_PRINTCFD ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_PRINTCFD) );
```

```
        memcpy(cte->cm.parameter, (incoming)+(strlen(PROTOCOL_PRINTCFD)*(sizeof(char))),
               length);
        cte->cm.code = CODE_PRINTCFD;
        printf("PRINTCFD\n");
    }

    /* PROTOCOL_PRINTCFD_SHORT */
    else if(strstr((char*)incoming, PROTOCOL_PRINTCFD_SHORT ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_PRINTCFD_SHORT) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_PRINTCFD_SHORT)*(sizeof(char))), length);
        cte->cm.code = CODE_PRINTCFD;
    }

    else {
        return 0;
    }

    /* Attempting to retrive ID */
    sscanf(cte->cm.parameter, "%d", &cte->cm.id_parameter);

    return 1;
}

/* Responds to client action */
static int respond(struct client_table_entry *cte)
{
    bzero(cte->cm.parameter, MAX_PARAMETER_SIZE);
    /* Only print ">" if client is monitor */
    if(cte->client_id < 0) {
        s_write(&(cte->transmission), ">", 1);
    }

    int read_status = s_read(&(cte->transmission)); /* Blocking */
    if(read_status == -1) {
        t_print("[ CLIENT %d ] Read failed or interrupted!\n", cte->client_id);
        return 0;
    }

    if(cte->marked_for_kick) {
        return 0;
    }

    int parse_status = parse_input(cte);

    if(parse_status == -1) {
        s_write(&(cte->transmission), ERROR_ILLEGAL_MESSAGE_SIZE,
                sizeof(ERROR_ILLEGAL_MESSAGE_SIZE));
    } else if(parse_status == 0) {
        s_write(&(cte->transmission), ERROR_ILLEGAL_COMMAND,
                sizeof(ERROR_ILLEGAL_COMMAND));
    }
    /* PARSING OK, CONTINUING */
    else {
        /* Comparing CODES to determine the correct action */
        if(cte->cm.code == CODE_DISCONNECT) {
            t_print("Client %d requested DISCONNECT.\n", cte->client_id);
            s_write(&(cte->transmission), PROTOCOL_GOODBYE, sizeof(PROTOCOL_GOODBYE));
            return 0;
        }

        else if(cte->cm.code == CODE_HELP) {
```

```
            print_help(cte);
        }

        else if(cte->cm.code == CODE_IDENTIFY) {
            if(cte->cm.id_parameter == 0) {
                s_write(&(cte->transmission), ERROR_ILLEGAL_COMMAND,
                        sizeof(ERROR_ILLEGAL_COMMAND));
                return 0;
            }

            /* Checking to see if the ID is in use */
            struct client_table_entry* client_list_iterate;
            list_for_each_entry(client_list_iterate, &client_list->list, list) {
                if(client_list_iterate->client_id == cte->cm.id_parameter) {
                    cte->client_id = 0;
                    t_print("[%s] bounced! ID %d already in use.\n", cte->ip,cte->cm.id_parameter);
                    s_write(&(cte->transmission), "ID in use!\n", 11);
                    return 0;
                }
            }

            /* Determining role */
            if(cte->cm.id_parameter < 0) {
                cte->client_type = MONITOR;
                struct timeval timeout = {MONITOR_TIMEOUT, 0};
                set_timeout(cte, timeout);

            } else {
                cte->client_type = SENSOR;
                sem_wait(&(s_synch->client_list_mutex));
                s_data->number_of_sensors++;
                sem_post(&(s_synch->client_list_mutex));
            }
            /* Everything is good, setting id and responding*/
            s_write(&(cte->transmission), PROTOCOL_OK, sizeof(PROTOCOL_OK));
            cte->client_id = cte->cm.id_parameter;
            t_print("[%s] ID set to: %d\n", cte->ip,cte->client_id);

            if(cte->client_type == SENSOR) {
                if(load_ref_def_data(cte)) {
                    s_write(&(cte->transmission), PROTOCOL_OK, sizeof(PROTOCOL_OK));
                    t_print("Loaded filter data for client %d\n", cte->client_id);
                } else {
                    s_write(&(cte->transmission),ERROR_LRFD_LOAD_FAILED,
                            sizeof(ERROR_LRFD_LOAD_FAILED));
                }
            }

            return 1;
        }

        /* Stop here if client is unidentified */
        else if(cte->client_id == 0) {
            s_write(&(cte->transmission), ERROR_NO_ID, sizeof(ERROR_NO_ID));
            return 1;
        }

        else if(cte->cm.code == CODE_NMEA) {
            /* Fetching data from buffer */
            char *rmc_start = strstr(cte->transmission.iobuffer, RMC);
            char *gga_start = strstr(cte->transmission.iobuffer, GGA);
            memcpy(cte->nmea.raw_rmc, rmc_start, gga_start - rmc_start);
```

```c
            memcpy(cte->nmea.raw_gga, gga_start,
                   ( strlen(cte->transmission.iobuffer) - (rmc_start - cte->transmission.iobuffer)
                     - (gga_start - rmc_start)));

            /* Checking NMEA checksum */
            int rmc_checksum = calculate_nmea_checksum(cte->nmea.raw_rmc);
            int gga_checksum = calculate_nmea_checksum(cte->nmea.raw_gga);

            /* Continue to filters if ok */
            if(rmc_checksum && gga_checksum) {
                cte->timestamp = time(NULL);
                cte->nmea.checksum_passed = 1;
                extract_nmea_data(cte);
                calculate_nmea_average(cte);
                calculate_nmea_diff(cte);

                /* Checksums where OK, client marked ready */
                cte->ready = 1;

                /* Acquiring ready-lock */
                sem_wait(&(s_synch->ready_mutex));

                /* Checking if the other clients are ready as well*/
                int ready = nmea_ready();

                /* If everyone is ready, process data */
                if(ready) {
                    /* Last process ready gets the job of analyzing the data */
                    ref_dev_filter();

                    /* Check the results of the filters */
                    raise_alarm();
                }
                /* Releasing ready-lock */
                sem_post(&(s_synch->ready_mutex));
            } else {
                cte->nmea.checksum_passed = 0;
                t_print("RMC and GGA received from %d , checksum failed!\n", cte->client_id);
            }
        }

        else if(cte->cm.code == CODE_PRINT_LOCATION) {
            struct client_table_entry* candidate = get_client_by_id(cte->cm.id_parameter);
            if(candidate == NULL) {
                s_write(&(cte->transmission), ERROR_NO_CLIENT, sizeof(ERROR_NO_CLIENT));
            } else {
                print_location(cte, candidate);
            }
        }

        else if(cte->cm.code == CODE_LOADRFDATA) {
            struct client_table_entry* candidate = get_client_by_id(cte->cm.id_parameter);
            if(candidate == NULL) {
                s_write(&(cte->transmission), ERROR_NO_CLIENT, sizeof(ERROR_NO_CLIENT));
            } else {
                if(load_ref_def_data(candidate)) {
                    s_write(&(cte->transmission), PROTOCOL_OK, sizeof(PROTOCOL_OK));
                } else {
                    s_write(&(cte->transmission),ERROR_LRFD_LOAD_FAILED,
                            sizeof(ERROR_LRFD_LOAD_FAILED));
                }
            }
```

```
        }

        else if(cte->cm.code == CODE_PRINTCLIENTS) {
            print_clients(cte);
        }

        else if(cte->cm.code == CODE_PRINTSERVER) {
            print_server_data(cte);
        }

        else if(cte->cm.code == CODE_PRINTTIME) {
            struct client_table_entry* candidate = get_client_by_id(cte->cm.id_parameter);
            if(candidate != NULL) {
                print_client_time(cte, candidate);
            } else {
                s_write(&(cte->transmission), ERROR_NO_CLIENT, sizeof(ERROR_NO_CLIENT));
            }
        }

        else if(cte->cm.code == CODE_KICK) {
            struct client_table_entry* candidate = get_client_by_id(cte->cm.id_parameter);
            if(candidate == NULL) {
                s_write(&(cte->transmission), ERROR_NO_CLIENT, sizeof(ERROR_NO_CLIENT));
            } else {
                kick_client(candidate);
            }
        }

        else if(cte->cm.code == CODE_DUMPDATA) {
            int filename_buffer_size = MAX_FILENAME_SIZE;
            char filename[filename_buffer_size];
            int target_id;
            char id_buffer[ID_AS_STRING_MAX];
            bzero(id_buffer, ID_AS_STRING_MAX);
            bzero(filename, filename_buffer_size);

            /* Attempting to extract filename */
            substring_extractor(2,3, ' ', filename, filename_buffer_size,cte->cm.parameter,
                                MAX_FILENAME_SIZE);

            /* If length of filename = 0 (no filename specified).. */
            if(strlen(filename) == 0) {
                /* ...Cast to int without a care */
                target_id = atoi(cte->cm.parameter);
            }
            /* Else, extract ID */
            else {
                substring_extractor(1,2, ' ', id_buffer, ID_AS_STRING_MAX,cte->cm.parameter,
                                    ID_AS_STRING_MAX);
                target_id = atoi(id_buffer);
            }

            if(!target_id) {
                s_write(&(cte->transmission), ERROR_ILLEGAL_COMMAND,
                        sizeof(ERROR_ILLEGAL_COMMAND));
            } else {
                struct client_table_entry* candidate = get_client_by_id(target_id);
                if(candidate != NULL) {
                    if(!datadump(candidate,filename, s_conf->human_readable_dumpdata)) {
                        s_write(&(cte->transmission), ERROR_DUMPDATA_FAILED,
                                sizeof(ERROR_DUMPDATA_FAILED));
                    }
```

```
                } else {
                    s_write(&(cte->transmission), ERROR_NO_CLIENT, sizeof(ERROR_NO_CLIENT));
                }
            }
        }

        else if(cte->cm.code == CODE_LOADDATA) {
            int filename_buffer_size = MAX_FILENAME_SIZE;
            char filename[filename_buffer_size];
            int target_id;
            char id_buffer[ID_AS_STRING_MAX];
            bzero(id_buffer, ID_AS_STRING_MAX);
            bzero(filename, filename_buffer_size);

            substring_extractor(2,3, ' ', filename, filename_buffer_size,cte->cm.parameter,
                                MAX_FILENAME_SIZE);

            /* No filename specified, abort */
            if(strlen(filename) == 0) {
                s_write(&(cte->transmission), ERROR_NO_FILENAME, sizeof(ERROR_NO_FILENAME));
                return 1;
            }
            /* Extract target id and move on */
            else {
                substring_extractor(1,2, ' ', id_buffer, ID_AS_STRING_MAX,cte->cm.parameter,
                                    ID_AS_STRING_MAX);
                target_id = atoi(id_buffer);
            }

            if(!target_id) {
                s_write(&(cte->transmission), ERROR_ILLEGAL_COMMAND,
                        sizeof(ERROR_ILLEGAL_COMMAND));
            } else {
                struct client_table_entry* candidate = get_client_by_id(target_id);
                if(candidate != NULL) {
                    int load_status = loaddata(candidate,filename);
                    if(load_status == ERROR_CODE_NO_FILE) {
                        s_write(&(cte->transmission), ERROR_NO_FILE, sizeof(ERROR_NO_FILE));
                    } else if(load_status == ERROR_CODE_READ_FAILED) {
                        s_write(&(cte->transmission), ERROR_READ_FAILED, sizeof(ERROR_READ_FAILED));
                    }
                } else {
                    s_write(&(cte->transmission), ERROR_NO_CLIENT, sizeof(ERROR_NO_CLIENT));
                }
            }
        }

        else if(cte->cm.code == CODE_PRINTAVGDIFF) {
            print_avg_diff(cte);
        }

        else if(cte->cm.code == CODE_LISTDUMPS) {
            listdumps(cte);
        }

        else if(cte->cm.code == CODE_QUERYCSAC) {
            if(strlen(cte->cm.parameter) < 3) {
                s_write(&(cte->transmission), ERROR_NO_COMMAND, sizeof(ERROR_NO_COMMAND));
                return 1;
            }
            client_query_csac(cte, cte->cm.parameter);
        } else if(cte->cm.code == CODE_PRINTCFD) {
```

```c
            print_cfd(cte, cte->cm.id_parameter);
        }

        else {
            t_print("No action made for this part of the protocol\n");
        }
    }
    return 1;
}

/* Setups the clients structure and initializes data */
void setup_session(int session_fd, struct client_table_entry *new_client)
{
    /* Setting the IP adress */
    char ip[INET_ADDRSTRLEN];
    get_ip_str(session_fd, ip);

    /* Setting the PID */
    new_client->pid = getpid();
    new_client->timestamp = time(NULL);
    strncpy(new_client->ip, ip, INET_ADDRSTRLEN);

    /* Initializing structure, zeroing just to be sure */
    new_client->client_id = 0;
    new_client->transmission.session_fd = session_fd;

    /* Zeroing out filters */
    new_client->fs.rdf.moved = 0;
    new_client->fs.rdf.was_moved = 0;

    new_client->marked_for_kick = 0;
    new_client->ready = 0;

    /* Setting timeout */
    struct timeval timeout = {UNIDENTIFIED_TIMEOUT, 0};
    if(!set_timeout(new_client, timeout)) {
        t_print("Failed to set timeout for client\n");
    }

    memset(&new_client->transmission.iobuffer, '0', IO_BUFFER_SIZE*sizeof(char));
    memset(&new_client->cm.parameter, '0', MAX_PARAMETER_SIZE*sizeof(char));

    /*
    * Entering child process main loop
    * (Outer) breaks if server closes.
    * (Inner) Breaks (disconnects the client) if
    * respond < 0
    */
    while(!done) {
        if(!respond(new_client)) {
            break;
        }
    }
}
```

B.2.6 session.h

```c
/**
 * @file session.h
 * @author Aril Schultzen
```

```c
 * @date 13.04.2016
 * @brief File containing function prototypes and includes for session.c
 */

#ifndef SESSION_H
#define SESSION_H

#include "sensor_server_common.h"
#include "filters.h"
#include "actions.h"
#include "sensor_server.h"

/** @brief Sets up and starts the session with the client
 *
 * Initializes and prepares the session and calls respond().
 *
 * @return Void
 */
void setup_session(int session_fd, struct client_table_entry *new_client);

#endif /* !SESSION_H */
```

B.2.7 actions.c

```c
#include "actions.h"

/* GENERAL */
#define CLIENT_TABLE_LABEL "CLIENT TABLE\n"
#define NEW_LINE "\n"
#define PRINT_LOCATION_HEADER "      CURRENT         MIN           MAX           AVG\n"
#define DUMPDATA_HEADER "CURRENT        MIN            MAX       AVERAGE     AVG_DIFF      TOTAL       DISTURBED\n"
#define PRINT_AVG_DIFF_HEADER "ID      LAT         LON        ALT       SPEED\n"
#define DATADUMP_EXTENSION ".bin"
#define DATADUMP_HUMAN_EXTENSION ".txt"
#define RDF_HEADER "\nREF_DEV_FILTER DATA\n"
#define CSAC_SCRIPT_COMMAND "python query_csac.py "

/* ERRORS */
#define ERROR_APPEND_TOO_LONG "ERROR: TEXT TO APPEND TOO LONG\n"
#define ERROR_NO_SENSORS_CONNECTED "NO SENSORS CONNECTED\n"
#define ERROR_FCLOSE "Failed to close file, out of space?\n"
#define ERROR_FWRITE "Failed to write to file, aborting.\n"
#define ERROR_FREAD "Failed to read file, aborting.\n"
#define ERROR_FOPEN "Failed to open file, aborting.\n"
#define ERROR_UPDATE_WARMUP_ILLEGAL "Warm-up time value has to be greater than 0!\n"
#define ERROR_CSAC_FAILED "Communication with CSAC failed!\n"

/* LOAD_REF_DEV_DATA */
#define REF_DEV_FILENAME "ref_dev_sensor"
#define ALT_REF "alt_ref:"
#define LON_REF "lon_ref:"
#define LAT_REF "lat_ref:"
#define SPEED_REF "speed_ref:"
#define ALT_DEV "alt_dev:"
#define LON_DEV "lon_dev:"
#define LAT_DEV "lat_dev:"
#define SPEED_DEV "speed_dev:"
#define LOAD_REF_DEV_DATA_ENTRIES 8

/* HELP */
```

```c
#define HELP "\n"\
" COMMAND      | SHORT | PARAM     | DESCRIPTION\n"\
"-------------------------------------------------------------------------------\n"\
" HELP         | ?     | NONE      | Prints this table\n"\
"-------------------------------------------------------------------------------\n"\
" IDENTIFY     | ID    | ID        | Your ID is set to PARAM ID\n"\
"-------------------------------------------------------------------------------\n"\
" DISCONNECT   | EXIT  | NONE      | Disconnects\n"\
"-------------------------------------------------------------------------------\n"\
" PRINTCLIENTS | PC    | NONE      | Prints a list of connected clients\n"\
"-------------------------------------------------------------------------------\n"\
" PRINTSERVER  | PS    | NONE      | Prints server state and config\n"\
"-------------------------------------------------------------------------------\n"\
" PRINTTIME    |       | ID        | Prints time solved from Sensor <ID>\n"\
"-------------------------------------------------------------------------------\n"\
" PRINTAVGDIFF | PAD   | NONE      | Prints all average diffs for all clients\n"\
"-------------------------------------------------------------------------------\n"\
" PRINTLOC     | PL    | ID        | Prints solved location for Sensor <ID>\n"\
"-------------------------------------------------------------------------------\n"\
" LISTDATA     | LSD   | NONE      | Lists all dump files in server directory\n"\
"-------------------------------------------------------------------------------\n"\
" DUMPDATA     | DD    | ID & FILE | Dumps state of Sensor <ID> into FILE\n"\
"-------------------------------------------------------------------------------\n"\
" LOADDATA     | LD    | ID & FILE | Loads NMEA of FILE into sensor ID\n"\
"-------------------------------------------------------------------------------\n"\
" QUERYCSAC    | QC    | COMMAND   | Queries the CSAC with parameter COMMAND\n"\
"-------------------------------------------------------------------------------\n"\
" LOADRFDATA   | LRFD  | ID        | Load reference location data into Sensor<ID>\n"\
"-------------------------------------------------------------------------------\n"\
" PRINTCFD     | PFD   |           | Prints CSAC filter data\n"\
"-------------------------------------------------------------------------------\n"\

/* SIZES */
#define DUMPDATA_TIME_SIZE 13
#define MAX_APPEND_LENGTH 20

void kick_client(struct client_table_entry* client)
{
    sem_wait(&(s_synch->client_list_mutex));
    sem_wait(&(s_synch->ready_mutex));
    client->marked_for_kick = 1;
    sem_post(&(s_synch->ready_mutex));
    sem_post(&(s_synch->client_list_mutex));
}

/* Prints client X's solved time back to monitor */
void print_client_time(struct client_table_entry *monitor,
                       struct client_table_entry* client)
{
    int buffsize = 100;
    char buffer[buffsize];
    memset(&buffer, 0, buffsize);

    substring_extractor(RMC_TIME_START,RMC_TIME_START + 1,',',buffer, buffsize,
                        client->nmea.raw_rmc, strlen(client->nmea.raw_rmc));
    s_write(&(monitor->transmission), buffer, 12);
    s_write(&(monitor->transmission), "\n", 1);
}

/* Prints a formatted string containing info about connected clients to monitor */
void print_clients(struct client_table_entry *monitor)
{
```

```
    char buffer [1000];
    int snprintf_status = 0;
    char *c_type = "SENSOR";
    char *modifier = "";

    struct client_table_entry* client_list_iterate;
    s_write(&(monitor->transmission), CLIENT_TABLE_LABEL,
            sizeof(CLIENT_TABLE_LABEL));
    s_write(&(monitor->transmission), HORIZONTAL_BAR, sizeof(HORIZONTAL_BAR));
    list_for_each_entry(client_list_iterate,&client_list->list, list) {

        if(client_list_iterate->client_type == MONITOR) {
            c_type = "MONITOR";
        } else {
            c_type = "SENSOR";
        }

        if(monitor->client_id == client_list_iterate->client_id) {
            modifier = BOLD_GRN_BLK;
        } else {
            modifier = RESET;
        }
        snprintf_status = snprintf( buffer, 1000,
                                    "%sID: %d " \
                                    "IP:%s, " \
                                    "PID: %d, " \
                                    "TYPE: %s, " \
                                    "NMEA age %d%s",
                                    modifier,
                                    client_list_iterate->client_id,
                                    client_list_iterate->ip,
                                    client_list_iterate->pid,
                                    c_type,
                                    (int)difftime(time(NULL),client_list_iterate->timestamp),
                                    RESET);

        s_write(&(monitor->transmission), buffer, snprintf_status);
    }
    s_write(&(monitor->transmission), HORIZONTAL_BAR, sizeof(HORIZONTAL_BAR));
}

/*
* Prints a string containing simple description
* of the different implemented commands back
* to the monitor.
*/
void print_help(struct client_table_entry *monitor)
{
    s_write(&(monitor->transmission), HELP, sizeof(HELP));
}

/*
* Prints MAX, MIN, CURRENT and AVERAGE position
* for client X back to the monitor
*/
void print_location(struct client_table_entry *monitor,
                    struct client_table_entry* client)
{
    char buffer [1000];
    int snprintf_status = 0;

    char *lat_modifier;
```

```c
    char *lon_modifier;
    char *alt_modifier;
    char *speed_modifier;
    char *reset = RESET;

    struct nmea_container nc;

    nc = client->nmea;
    s_write(&(monitor->transmission), PRINT_LOCATION_HEADER,
            sizeof(PRINT_LOCATION_HEADER));

    /*Determining colors*/
    if(!nc.lat_disturbed) {
        lat_modifier = BOLD_GRN_BLK;
    } else if(nc.lat_disturbed > 0) {
        lat_modifier = BOLD_RED_BLK;
    } else {
        lat_modifier = BOLD_CYN_BLK;
    }

    if(!nc.lon_disturbed) {
        lon_modifier = BOLD_GRN_BLK;
    } else if(nc.lon_disturbed > 0) {
        lon_modifier = BOLD_RED_BLK;
    } else {
        lon_modifier = BOLD_CYN_BLK;
    }

    if(!nc.alt_disturbed) {
        alt_modifier = BOLD_GRN_BLK;
    } else if(nc.alt_disturbed > 0) {
        alt_modifier = BOLD_RED_BLK;
    } else {
        alt_modifier = BOLD_CYN_BLK;
    }

    if(!nc.speed_disturbed) {
        speed_modifier = BOLD_GRN_BLK;
    } else if(nc.speed_disturbed > 0) {
        speed_modifier = BOLD_RED_BLK;
    } else {
        speed_modifier = BOLD_CYN_BLK;
    }

    snprintf_status = snprintf( buffer, 1000,
                               "LAT: %s%f%s %f\n" \
                               "LON: %s%f%s %f\n" \
                               "ALT: %s %f%s  %f\n" \
                               "SPD: %s   %f%s  %f\n",
                               lat_modifier, nc.lat_current,reset, nc.lat_average,
                               lon_modifier, nc.lon_current,reset, nc.lon_average,
                               alt_modifier, nc.alt_current,reset, nc.alt_average,
                               speed_modifier, nc.speed_current,reset, nc.speed_average);
    s_write(&(monitor->transmission), buffer, snprintf_status);
}

/*
* Prints the difference between the calculated
* average values for location and the current value
*/
void print_avg_diff(struct client_table_entry *client)
{
```

```c
    char buffer [1000];
    int snprintf_status = 0;
    struct nmea_container nc;

    if(s_data->number_of_sensors > 0) {
        s_write(&(client->transmission), PRINT_AVG_DIFF_HEADER,
                sizeof(PRINT_AVG_DIFF_HEADER));
        struct client_table_entry* client_list_iterate;
        list_for_each_entry(client_list_iterate,&client_list->list, list) {
            if(client_list_iterate->client_id > 0) {
                nc = client_list_iterate->nmea;
                snprintf_status = snprintf( buffer, 1000, "%d   %f  %f  %f  %f\n",
                                            client_list_iterate->client_id, nc.lat_avg_diff, nc.lon_avg_diff,
                                            nc.alt_avg_diff, nc.speed_avg_diff);
                s_write(&(client->transmission), buffer, snprintf_status);
            }
        }
    } else {
        s_write(&(client->transmission), ERROR_NO_SENSORS_CONNECTED,
                sizeof(ERROR_NO_SENSORS_CONNECTED));
    }
}

static int get_pfd_string(char *buffer, int buf_len)
{
    memset(buffer, '\0',buf_len);
    int snprintf_status = snprintf( buffer, 1000,
                                    "Phase:                     %lf\n\n" \
                                    "T current:                 %lf\n" \
                                    "T current (smooth):        %lf\n" \
                                    "T previous (smooth):       %lf\n" \
                                    "T today (smooth):          %lf\n" \
                                    "T yesterday (smooth):      %lf\n\n" \
                                    "Steer current:             %lf\n" \
                                    "Steer current (smooth):    %lf\n" \
                                    "Steer previous (smooth):   %lf\n\n" \
                                    "Steer today (smooth):      %lf\n" \
                                    "Steer yesterday (smooth):  %lf\n\n" \
                                    "Steer prediction:          %lf\n\n" \
                                    "MJD today:                 %lf\n" \
                                    "Days passed since startup: %d\n\n" \
                                    "Discipline status:         %d\n" \
                                    "Fast timing filter status  %d\n" \
                                    "Freq corr. filter status   %d\n\n",
                                    cfd->phase_current,
                                    cfd->t_current,
                                    cfd->t_smooth_current,
                                    cfd->t_smooth_previous,
                                    cfd->t_smooth_today,
                                    cfd->t_smooth_yesterday,
                                    cfd->steer_current,
                                    cfd->steer_smooth_current,
                                    cfd->steer_smooth_previous,
                                    cfd->steer_smooth_today,
                                    cfd->steer_smooth_yesterday,
                                    cfd->steer_prediction,
                                    cfd->today_mjd,
                                    cfd->days_passed,
                                    cfd->discok,
                                    cfd->ftf_status,
                                    cfd->fqf_status);
    return snprintf_status;
```

```
}

void print_cfd(struct client_table_entry *monitor, int update_count)
{
    int buf_len = 1000;
    char buffer [buf_len];
    int counter = 0;

    if(update_count == 0) {
        update_count = 1;
    }

    while(counter < update_count) {
        get_pfd_string(buffer, buf_len);
        s_write(&(monitor->transmission), buffer, strlen(buffer));
        counter++;
        sleep(1);
    }
}

int dump_cfd(char *path)
{
    int buf_len = 1000;
    char buffer[buf_len];

    /* Formating string with CSAC filter data */
    get_pfd_string(buffer, buf_len);

    /* Opening and writing to file */
    FILE *cfd_file;
    cfd_file = fopen(path, "w+");

    if(!cfd_file) {
        t_print("dump_cfd: %s: %s",ERROR_FOPEN, path);
        return 0;
    }

    if(!fprintf(cfd_file,"%s", buffer) ) {
        t_print(ERROR_FWRITE);
        return 0;
    }

    if(fclose(cfd_file)) {
        t_print(ERROR_FCLOSE);
    }
    return 1;
}

/* Dumps data location data for client X into a file */
int datadump(struct client_table_entry* client, char *filename,
             int dump_human_read)
{
    FILE *bin_file;
    char bin_name[strlen(filename) + strlen(DATADUMP_EXTENSION)];
    strcpy(bin_name, filename);
    strcat(bin_name, DATADUMP_EXTENSION);

    bin_file=fopen(bin_name, "wb");

    if(!bin_file) {
        t_print(ERROR_FOPEN);
        return 0;
```

```
    }

    if(!fwrite(&client->nmea, sizeof(struct nmea_container), 1, bin_file)) {
        t_print(ERROR_FWRITE);
        return 0;
    }

    if(fclose(bin_file)) {
        t_print(ERROR_FCLOSE);
    }

    if(dump_human_read) {
        /* Dumping humanly readable data */
        FILE *h_dump;
        char h_name[strlen(filename) + strlen(DATADUMP_HUMAN_EXTENSION)];
        strcpy(h_name, filename);
        strcat(h_name, DATADUMP_HUMAN_EXTENSION);

        h_dump = fopen(h_name, "wb");

        fprintf(h_dump, "Sensor Server dumpfile created for client %d\n",
                client->client_id);

        /*
        * Dumping all from NMEA container
        * after raw_rmc and including speed_disturbed
        */
        int inner_counter = 0;
        int outer_counter = 0;
        double *data = &client->nmea.lat_current;

        fprintf(h_dump,DUMPDATA_HEADER);
        while(outer_counter < 4) {
            while(inner_counter < 7) {
                fprintf(h_dump, "%f  ",*data);
                data++;
                inner_counter++;
            }
            fprintf(h_dump, "%f", *data);
            inner_counter = 0;
            outer_counter++;
        }

        /*
        * Dumping ref_dev_data
        */
        fprintf(h_dump,DUMPDATA_HEADER);
        inner_counter = 0;
        double *rdf = &client->fs.rdf.rdd.alt_ref;
        while(inner_counter < 8) {
            fprintf(h_dump, "%lf \n",*rdf);
            rdf++;
            inner_counter++;
        }

        if(fclose(h_dump)) {
            t_print(ERROR_FCLOSE);
        }
    }
    return 1;
}
```

```
/* Print list of dumped data */
int listdumps(struct client_table_entry* monitor)
{
    DIR *dp;
    struct dirent *ep;

    dp = opendir ("./");
    if(dp != NULL) {
        while ( (ep = readdir(dp)) ) {
            if(strstr(ep->d_name,DATADUMP_EXTENSION) != NULL) {
                s_write(&(monitor->transmission),ep->d_name, strlen(ep->d_name));
                s_write(&(monitor->transmission),NEW_LINE, sizeof(NEW_LINE));
            }
        }
        closedir (dp);
    } else {
        return 0;
    }

    return 1;
}

/* Load dumped data into the client */
int loaddata(struct client_table_entry* target, char *filename)
{
    FILE *dump_file;
    int file_len = 0;


    dump_file=fopen(filename, "rb");

    if(!dump_file) {
        t_print(ERROR_FOPEN);
        return ERROR_CODE_NO_FILE;
    }

    /* Checking file length */
    fseek(dump_file, 0, SEEK_END);
    file_len=ftell(dump_file);
    fseek(dump_file, 0, SEEK_SET);

    int f_s = fread( &target->nmea,1,sizeof(struct nmea_container), dump_file);

    t_print("Read %d/%d bytes successfully from %s\n", f_s, file_len,filename);

    if(f_s == 0) {
        t_print(ERROR_FREAD);
        return ERROR_CODE_READ_FAILED;
    }

    if(fclose(dump_file)) {
        t_print(ERROR_FCLOSE);
    }

    return 1;
}

int query_csac(char *query, char *buffer)
{
    /* Building command */
    int command_size = MAX_PARAMETER_SIZE + sizeof(CSAC_SCRIPT_COMMAND);
    char command[command_size];
```

```
    memset(command,'\0', command_size);
    strcat(command, CSAC_SCRIPT_COMMAND);
    strcat(command, query);

    /* Acquiring lock*/
    sem_wait(&(s_synch->csac_mutex));

    /* Running command */
    if(!run_command(command, buffer)) {
        /* Releasing lock */
        sem_post(&(s_synch->csac_mutex));
        return 0;
    }

    /* Releasing lock */
    sem_post(&(s_synch->csac_mutex));
    return 1;
}


int client_query_csac(struct client_table_entry *monitor, char *query)
{
    char buffer[MAX_PARAMETER_SIZE];
    memset(buffer, '\0', MAX_PARAMETER_SIZE);

    if(!query_csac(query, buffer)) {
        return 0;
    }

    if(!s_write(&(monitor->transmission), buffer, strlen(buffer))) {
        return 0;
    }
    return 1;
}

/*
* Load ref_dev data into the client struct.
* Re-using the config loader.
* This whole function needs some work! Magic numbers beware.
*/
int load_ref_def_data(struct client_table_entry* target)
{
    /* Request lock */
    sem_wait(&(s_synch->client_list_mutex));
    sem_wait(&(s_synch->ready_mutex));
    struct config_map_entry conf_map[LOAD_REF_DEV_DATA_ENTRIES];

    int filename_length = strlen(REF_DEV_FILENAME) + 10;
    char filename[filename_length];
    memset(filename,'\0' ,filename_length);
    strcpy(filename, REF_DEV_FILENAME);

    /* Way overkill for int to string, but still. */
    char id[10];
    memset(id,'\0' ,10);
    sprintf(id, "%d", target->client_id);
    strcat(filename, id);

    conf_map[0].entry_name = ALT_REF;
    conf_map[0].modifier = FORMAT_DOUBLE;
    conf_map[0].destination = &target->fs.rdf.rdd.alt_ref;

```

```c
    conf_map[1].entry_name = LON_REF;
    conf_map[1].modifier = FORMAT_DOUBLE;
    conf_map[1].destination = &target->fs.rdf.rdd.lon_ref;

    conf_map[2].entry_name = LAT_REF;
    conf_map[2].modifier = FORMAT_DOUBLE;
    conf_map[2].destination = &target->fs.rdf.rdd.lat_ref;

    conf_map[3].entry_name = SPEED_REF;
    conf_map[3].modifier = FORMAT_DOUBLE;
    conf_map[3].destination = &target->fs.rdf.rdd.speed_ref;

    conf_map[4].entry_name = ALT_DEV;
    conf_map[4].modifier = FORMAT_DOUBLE;
    conf_map[4].destination = &target->fs.rdf.rdd.alt_dev;

    conf_map[5].entry_name = LON_DEV;
    conf_map[5].modifier = FORMAT_DOUBLE;
    conf_map[5].destination = &target->fs.rdf.rdd.lon_dev;

    conf_map[6].entry_name = LAT_DEV;
    conf_map[6].modifier = FORMAT_DOUBLE;
    conf_map[6].destination = &target->fs.rdf.rdd.lat_dev;

    conf_map[7].entry_name = SPEED_DEV;
    conf_map[7].modifier = FORMAT_DOUBLE;
    conf_map[7].destination = &target->fs.rdf.rdd.speed_dev;

    t_print("Loading filter data from: %s\n", filename);

    int load_config_status = load_config(conf_map, filename,
                                         LOAD_REF_DEV_DATA_ENTRIES);

    /* releasing lock */
    sem_post(&(s_synch->ready_mutex));
    sem_post(&(s_synch->client_list_mutex));
    return load_config_status;
}
```

B.2.8 actions.h

```c
/**
 * @file actions.h
 * @brief File containing function prototypes and includes for actions.c
 *
 * Function prototypes for functions that implements different
 * actions that a MONITOR or the system can use to manipulate the
 * state of the SENSORS or print stats or similar.
 *
 * Be advised that any reference to MONITOR in this file means
 * a client connected to the server who's role is that of a
 * monitor of the system and not a monitor like a peripheral
 * connected to a computer. The names of these roles are under
 * discussion and will probably be changed to avoid misunderstanding.
 *
 * @author Aril Schultzen
 * @date 9.11.2015
 */

#ifndef ACTIONS_H
```

```c
#define ACTIONS_H

#include "sensor_server.h"
#include "serial.h"
#include <dirent.h>

/** @brief Kicks a client (both MONITOR or SENSOR)
 *
 * Marks the client so respond() in session.c can
 * disconnect it the next time that client transmits
 * data. The kick is in other words not instant, this
 * is however an easy way to gracefully disconnect a
 * client.
 *
 * @param client Pointer to the client_table_entry for the candidate to be kicked.
 * @return Void
 */
void kick_client(struct client_table_entry* client);

/** @brief Prints clients solved time to MONITOR
 *
 * Extracts the time solved by the GPS receiver, transmitted
 * via NMEA and stored in the client_table_struct at the server,
 * and transmits it to the MONITOR that requested it.
 *
 * @param monitor Pointer to MONITOR who made the request.
 * @param client Pointer to SENSOR whose time was requested.
 * @return Void
 */
void print_client_time(struct client_table_entry *monitor,
                       struct client_table_entry* client);

/** @brief Prints a table of clients to the MONITOR
 *
 * Transmits a table of the connected clients to the MONITOR.
 *
 * @param monitor Pointer to MONITOR who made the request.
 * @return Void
 */
void print_clients(struct client_table_entry *monitor);

/** @brief Prints table of available commands to requesting MONITOR.
 *
 * @param monitor Pointer to MONITOR who made the request.
 * @return Void
 */
void print_help(struct client_table_entry *monitor);

/** @brief Prints location of SENSOR to requesting MONITOR.
 *
 * Prints a overview of current as well as MIN, MAX and AVERAGE
 * values of LAT, LON, ALT and SPEED recovered from NMEA.
 *
 * @param monitor Pointer to MONITOR who made the request.
 * @param client Pointer to SENSOR whose location is requested.
 * @return Void
 */
void print_location(struct client_table_entry *monitor,
                    struct client_table_entry* client);

/** @brief Prints difference between current position and average.
 *
```

```c
 * Prints the difference between the current position values
 * recorded from NMEA, and the calculated averages.
 * Two sensors in close proximity (100m >) should be
 * subjected to the same noise. If the difference between
 * sensor A (current-avg) and sensor B (current-avg) changes,
 * this could mean that one of them is being spoofed.
 *
 * @param monitor Pointer to MONITOR who made the request.
 * @return Void
 */
void print_avg_diff(struct client_table_entry *monitor);

/** @brief Restarts the warm-up procedure for the given SENSOR
 *
 * Sets the SENSORs warmup_started to NOW, warmup to 1 and ready to 0.
 * This "triggers" a restart of the warm-up procedure.
 *
 * @param client Pointer to client whose warm-up procedure to restart.
 * @return Void
 */
void restart_warmup(struct client_table_entry* client);

/** @brief Dumps NMEA data to file for given client
 *
 * @param client Pointer to client whose data should be dumped.
 * @param filename Pointer to filename.
 * @param human_readable Switch to determine if humanly readable data should be made as well.
 * @return 1 if success, 0 if fail.
 */
int datadump(struct client_table_entry* client, char *filename,
             int human_readable);

/** @brief Restore NMEA data from file
 *
 * @param client Pointer to client whose data should be restored from file
 * @param filename Pointer to filename.
 * @return 1 if success, 0 if fail.
 */
int datarestore(struct client_table_entry* client, char *filename);

/** @brief List files in folder
 *
 * @param monitor Pointer to requesting monitor
 * @return 1 if success, 0 if fail.
 */
int listdumps(struct client_table_entry* monitor);

/** @brief Sets a new warm-up time for a given SENSOR.
 *
 * @param client Pointer to client whose warm-up time to be given new value.
 * @param value New warm-up time in seconds.
 * @return Void
 */
void set_warmup(struct client_table_entry *client, int value);

/** @brief Loads NMEA data into the NMEA struct of a given client (target).
*
* @param target Pointer to the client whose NMEA data should be loaded
* from file.
* @param filename Pointer to the filename of the data file.
*/
int loaddata(struct client_table_entry* target,  char *filename);
```

```c

/** @brief Uses the query_csac.py to communicate with the CSAC.
*           Stores the response in a buffer.
*
* @param buffer Buffer to store the response
* @param query Command (query) to send to the CSAC.
*/
int query_csac(char *query, char *buffer);

/** @brief Uses the query_csac.py to communicate with the CSAC
*           Prints the response from the CSAC back to the client
*
* @param monitor Monitor who made the request
* @param query Command (query) to send to the CSAC.
*/
int client_query_csac(struct client_table_entry *monitor, char *query);

/** @brief Loads data for the REF_DEV_FILTER into the client.
*
* @param target Client to load the data into
*/
int load_ref_def_data(struct client_table_entry* target);

/** @brief Prints the current state of the CSAC filter.
*
* @param monitor Monitor to print the data to.
* @return Status of sprintf() used to build string.
*/
void print_cfd(struct client_table_entry *monitor, int update_count);

/** @brief Dumps the state of the CSAC filter to file.
*
* @param Path to desired file to use.
* @return 1 if successful, 0 else.
*/
int dump_cfd(char *path);
#endif /* !ACTIONS_H */
```

B.2.9 utils.c

```c
#include "utils.h"

/* These are also in action.c, duplicates are no solution */
#define ERROR_FCLOSE "Failed to close file, out of space?\n"
#define ERROR_FWRITE "Failed to write to file, aborting.\n"
#define ERROR_FREAD "Failed to read file, aborting.\n"
#define ERROR_FOPEN "Failed to open file, aborting.\n"

#define MJD_SCRIPT_PATH "./get_mjd.py"

void die (int line_number, const char * format, ...)
{
    va_list vargs;
    va_start (vargs, format);
    fprintf (stderr, "%d: ", line_number);
    vfprintf (stderr, format, vargs);
    fprintf (stderr, ".\n");
    exit(1);
}

```

```c
/*
* Extracts IP address from sockaddr struct.
* Supports both IPV4 and IPV6
*/
void extract_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
    case AF_INET:
        inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr),
                  s, maxlen);
        break;

    case AF_INET6:
        inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
                  s, maxlen);
        break;

    default:
        strncpy(s, "Unknown AF", maxlen);
    }
}

/*
* Extracts IP from session file descriptor
*/
void get_ip_str(int session_fd, char *ip)
{
    struct sockaddr addr;
    addr.sa_family = AF_INET;
    socklen_t addr_len = sizeof(addr);
    if(getpeername(session_fd, (struct sockaddr *) &addr, &addr_len)) {
        die(44,"getsocketname failed\n");
    }
    extract_ip_str(&addr,ip, addr_len);
}

/*
* Print with timestamp:
* Example : [01.01.01 - 10:10:10] [<Some string>]
*/
void t_print(const char* format, ...)
{
    char buffer[100];
    time_t rawtime;
    struct tm *info;
    time(&rawtime);
    info = gmtime(&rawtime);
    strftime(buffer,80,"[%x - %X] ", info);
    va_list argptr;
    va_start(argptr, format);
    fputs(buffer, stdout);
    vfprintf(stdout, format, argptr);
    va_end(argptr);
}

/*
* Loads config.
* Returns: 0 fail | 1 success
*/
int load_config(struct config_map_entry *cme, char *path, int entries)
{
    FILE *config_file;
```

```c
    int file_len;
    char *input_buffer;

    int status = 0;

    config_file=fopen(path, "r");
    if(!config_file) {
        t_print("config_loader(): Failed to load config file, aborting.\n");
        return 0;
    }

    fseek(config_file , 0L , SEEK_END);
    file_len = ftell(config_file);
    rewind(config_file);

    char temp_buffer[file_len];

    /* Alocating memory for the file buffer */
    input_buffer = calloc( file_len, sizeof(char));
    if(!input_buffer) {
        fclose(config_file);
        t_print("config_loader(): Memory allocation failed, aborting.\n");
        return 0;
    }

    /* Get the file into the buffer */
    if(fread( input_buffer , file_len, 1 , config_file) != 1) {
        fclose(config_file);
        free(input_buffer);
        t_print("config_loader(): Read failed, aborting\n");
        return 0;
    }

    int counter = 0;
    while(counter < entries) {
        memset(temp_buffer, '\0',file_len);
        char *search_ptr = strstr(input_buffer,cme->entry_name);
        if(search_ptr != NULL) {
            int length = strlen(search_ptr) - strlen(cme->entry_name);
            memcpy(temp_buffer, search_ptr+(strlen(cme->entry_name)*(sizeof(char))),
                   length);
            status = sscanf(temp_buffer, cme->modifier, cme->destination);
            if(status == EOF || status == 0) {
                fclose(config_file);
                free(input_buffer);
                return -1;
            }
        }
        counter++;
        cme++;
    }

    fclose(config_file);
    free(input_buffer);
    return 1;
}

int calculate_nmea_checksum(char *nmea)
{
    char checksum = 0;
    int i;
    int received_checksum = 0;
```

```
    int calculated_checksum = 0;


    /* Substring to iterate over */
    char substring[100] = {0};

    /* Finding end (*) and calculate length */
    char *substring_end = strstr(nmea, "*");
    int length = substring_end - (nmea+1);

    /* Copying the substring */
    memcpy(substring, nmea+1, length);

    /* Calculating checksum */
    for(i = 0; i < length; i++) {
        checksum = checksum ^ substring[i];
    }

    /* Reusing substring buffer */
    sprintf(substring, "%x\n", checksum);

    /* Converting calculated checksum to int */
    sscanf(substring, "%d", &calculated_checksum);

    /* Fetching received checksum */
    memcpy(substring, substring_end+1, strlen(nmea));

    /* Converting received checksum to int*/
    sscanf(substring, "%d", &received_checksum);

    /* Comparing checksum */
    if(received_checksum == calculated_checksum) {
        return 1;
    } else {
        return 0;
    }

}

/*
* Used to extract words from between two delimiters
* delim_num_1 -> The number of the first delimiter, ex.3
* delim_num_2 -> The number of the second delimiter, ex.5
* delimiter -> The character to be used as a delimiter
* string -> Input
* buffer -> To transport the string
*/
int substring_extractor(int start, int end, char delimiter, char *buffer,
                        int buffsize, char *string, int str_len)
{
    int i;
    int delim_counter = 0;
    int buffer_index = 0;

    const int carriage_return = 13;

    bzero(buffer, buffsize);

    for(i = 0; i < str_len; i++) {
        /* Second delim (end) reached, stopping. */
        if(delim_counter == end || (int)string[i] == carriage_return) {
            return 1;
```

```c
        }

        if(string[i] == delimiter) {
            delim_counter++;
        } else {
            /* The first delim is reached */
            if(delim_counter >= start) {
                buffer[buffer_index] = string[i];
                buffer_index++;
            }
        }
    }
    /* Reached end of string without encountering end delimit */
    return 0;
}

int str_len_u(char *buffer, int buf_len)
{
    int i;
    char prev = 'X';
    for(i = 0; i < buf_len; i++) {
        if(buffer[i] == 0x0a && prev == 0x0a) {
            return i;
        }
        prev = buffer[i];
    }
    return -1;
}

/* Mega hackish code for getting MJD */
int get_today_mjd(char *buffer)
{
    int status = run_command(MJD_SCRIPT_PATH, buffer);
    /* Removing newline */
    buffer[strcspn(buffer, "\n")] = 0;
    return status;
}

int run_command(char *path, char *output)
{
    FILE *fp;
    int buffer_size = 1000;
    char buffer[buffer_size];
    memset(buffer, '\0', buffer_size);

    /* Open the command for reading. */
    fp = popen(path, "r");
    if (fp == NULL) {
        t_print("Failed to run command\n");
        return 0;
    }

    /* Read the output a line at a time - output it. */
    while (fgets(buffer, sizeof(buffer)-1, fp) != NULL) {
        strcat(output,buffer);
    }

    /* close */
    pclose(fp);
    return strlen(output);
}

```

```c
int log_to_file(char *path, char *content, int stamp_switch)
{
    FILE *log_file;
    log_file = fopen(path, "a+");

    /* Open file */
    if(!log_file) {
        t_print(ERROR_FOPEN);
        return 0;
    }

    /* Add MJD timestamp */
    if(stamp_switch == 1) {
        int timestamp_size = 50;
        char timestamp[timestamp_size];
        memset(timestamp,'\0', timestamp_size);

        get_today_mjd(timestamp);
        if(!fprintf(log_file,"%s,",timestamp)) {
            t_print(ERROR_FWRITE);
            return 0;
        }
    }

    /* Just stamp with regular time */
    if(stamp_switch == 2){
        char timestamp[100];
        memset(timestamp, '\0', 100);
        time_t rawtime;
        struct tm *info;
        time(&rawtime);
        info = gmtime(&rawtime);
        strftime(timestamp,80,"[%x - %X] ", info);

        if(!fprintf(log_file,"%s,",timestamp)) {
            t_print(ERROR_FWRITE);
            return 0;
        }
    }

    /* Write content to file */
    if(!(fprintf(log_file,"%s",content))) {
        t_print(ERROR_FWRITE);
        return 0;
    }

    /* Close file */
    if(fclose(log_file)) {
        t_print(ERROR_FCLOSE);
    }
    return 1;
}
```

B.2.10 utils.h

```c
/**
 * @file utils.h
 * @author Aril Schultzen
 * @date 13.04.2016
 * @brief File containing function prototypes and includes for utils.c
```

```
 */

#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <string.h>
#include <time.h>

#include "list.h"
#include "config.h"

/** @brief Terminates program and prints the line
 *           number where die was called from.
 *
 *  @param line_number Line number where die() was written
 *  @param format String with error description.
 *    @return Void
 */
void die (int line_number, const char * format, ...);

/** @brief Extracts IP address from file descriptor
 *
 *    @param session_fd file descriptor for the session
 *    @param ip Buffer to store the IP address as string.
 */
void get_ip_str(int session_fd, char *ip);

/** @brief Extracts IP address from sockaddr struct
 *
 *    Used by get_ip_str() to extract IP address from
 *    sockaddr struct.
 *
 *    @param session_fd file descriptor for the session
 *    @param ip Buffer to store the IP address as string.
 *    @return Void
 */
void extract_ip_str(const struct sockaddr *sa, char *s, size_t maxlen);

/** @brief Print function with time-stamp
 *
 *    Print function like printf() but with time-stamp
 *  in square bracket appended before the String.
 *     Example: [04/13/16 - 08:50:41] Waiting for connections..
 *
 *    @param format String to print
 *    @return Void
 */
void t_print(const char* format, ...);

/** @brief Loads config from file using config_map_entry struct
 *
 *    Uses the config_map_entry struct to find the correct entry
 *     in the config file, cast it to correct type and fill the
 *     respective memory area (pointer).
 *
 *    @param cme Pointer to the config_map_entry struct
 *    @param path Path to config file
 *    @param entries Entries in the config file
```

```c
 *    @return 1 if success, 0 if fail.
 */
int load_config(struct config_map_entry *cme, char *path, int entries);

/** @brief Calculates the checksum of a given string of NMEA data.
 *
 *  Used to check the integrity of NMEA data from the
 *    GPS receiver before potential analysis.
 *
 *    @param nmea String containing NMEA data to check
 *    @return 1 if success, 0 if fail.
 */
int calculate_nmea_checksum(char *s);

/** @brief Extracts words from a String
 *
 *    Used to extract a substring from a string by using a
 *  delimiter. The from and to parameters defines which
 *    occurrence of the delimiter in the parent string to
 *    use as start and end for the substring.
 *
 *    @param start The delimiter number to start from
 *    @param end The delimiter number to stop
 *    @param delimiter Symbol/character to use as delimit
 *    @param buffer Buffer to store the word(s)
 *    @param buffsize Size of buffer
 *    @param string Pointer to parent string
 *    @param str_len Length of parent string
 *    @return 1 if success, 0 if no string within the delimits was found.
 */
int substring_extractor(int start, int end, char delimiter, char *buffer,
                        int buffsize, char *string, int str_len);

/** @brief Counts bytes from start to first occurence of null character
 *
 *    @param buffer Buffer to search through
 *    @param buf_len Length of the buffer in bytes
 */
int str_len_u(char *buffer, int buf_len);

/** @brief Calls a script using run_command to get mjd(now).
 *
 *    @param buffer Buffer to store response
 */
int get_today_mjd(char *buffer);

/** @brief Run a script or a program through the shell
 *
 *    @param path Path to program
 *    @param output Buffer to store response
 */
int run_command(char *path, char *output);

/** @brief Log to file
 *
 *    @param content Data to log
 *    @param path Path to the log file to log to
 *    @param stamp_switch 0 if no timestamp, 1 if MJD.
 */
int log_to_file(char *path, char *content, int stamp_switch);
#endif /* !UTILS_H */
```

B.2.11 net.c

```c
#include "net.h"

int s_read(struct transmission_s *tsm)
{
    bzero(tsm->iobuffer,IO_BUFFER_SIZE);
    return read(tsm->session_fd, tsm->iobuffer,IO_BUFFER_SIZE);
}

int s_write(struct transmission_s *tsm, char *message, int length)
{
    return write(tsm->session_fd, message, length);
}
```

B.2.12 net.h

```c
#ifndef NET_H
#define NET_H

#define _GNU_SOURCE 1
#include <unistd.h>
#include <sys/mman.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <stdarg.h>
#include <signal.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <stdbool.h>

/* My own header files */
#include "utils.h"
#include "protocol.h"
#include "nmea.h"

/* GENERAL */
#define IO_BUFFER_SIZE MAX_PARAMETER_SIZE + MAX_COMMAND_SIZE

struct transmission_s {
    int session_fd;
    char iobuffer[IO_BUFFER_SIZE];
};

int s_read(struct transmission_s *tsm);
int s_write(struct transmission_s *tsm, char *message, int length);

#endif /* !NET_H */
```

B.2.13 csac_filter.c

```c
#include "csac_filter.h"

/* PATH TO CONFIG FILE */
#define CSAC_FILTER_CONFIG_PATH "cfilter_config.ini"

/* CONFIG CONSTANTS */
#define CONFIG_PRED_LOGGING "pred_logging: "
#define CONFIG_PRED_LOG_PATH "pred_log_path: "
#define CONFIG_CFD_PATH "cfd_state_path: "
#define CONFIG_INIT_FROM_FILE "init_cfd_from_file: "
#define CONFIG_INIT_SSC "init_cfd_steer_smooth_current: "
#define CONFIG_INIT_SST "init_cfd_steer_smooth_today: "
#define CONFIG_INIT_SSP "init_cfd_steer_smooth_previous: "
#define CONFIG_INIT_SSY "init_cfd_steer_smooth_yesterday: "
#define CONFIG_PHASE_LIMIT "phase_limit: "
#define CONFIG_STEER_LIMIT "steer_limit: "
#define CONFIG_PRED_LIMIT "pred_limit: "
#define CONFIG_TIME_CONSTANT "time_constant: "
#define CONFIG_WARMUP_DAYS "warmup_days: "
#define CSAC_FILTER_CONFIG_ENTRIES 13


static float mjd_diff_day(double mjd_a,
                          double mjd_b)
{
    float diff = mjd_a - mjd_b;
    return diff;
}

static int load_telemetry(struct csac_filter_data
                          *cfd, char *telemetry)
{
    const int BUFFER_LEN = 100;
    char buffer[BUFFER_LEN];

    /* Checking discipline mode of the CSAC */
    if(!substring_extractor(13,14,',',buffer,100,
                            telemetry,strlen(telemetry))) {
        printf("Failed to extract DiscOK from CSAC data\n");
        return 0;
    } else {
        if(sscanf(buffer, "%d", &cfd->discok) == EOF) {
            return 0;
        }
        /* CSAC is in holdover or acquiring */
        if(cfd->discok == 2) {
            return 0;
        }
    }

    if(!substring_extractor(12,13,',',buffer,100,
                            telemetry,strlen(telemetry))) {
        printf("Failed to extract Phase from CSAC data\n");
        return 0;
    } else {
        if(sscanf(buffer, "%lf",
                  &cfd->phase_current) == EOF) {
            return 0;
        }
    }
```

```c

    if(!substring_extractor(10,11,',',buffer,100,
                            telemetry,strlen(telemetry))) {
        printf("Failed to extract Steer from CSAC data\n");
        return 0;
    } else {
        if(sscanf(buffer, "%lf",
                  &cfd->steer_current) == EOF) {
            return 0;
        }
    }

    double mjd_today = 0;
    memset(buffer, '\0', BUFFER_LEN);
    if(!get_today_mjd(buffer)) {
        printf("Failed to calculate current MJD\n");
        return 0;
    } else {
        if(sscanf(buffer, "%lf", &mjd_today) == EOF) {
            return 0;
        } else {
            if(mjd_diff_day(mjd_today, cfd->today_mjd) >= 1
                    && cfd->t_current != 0) {
                cfd->new_day = 1;
                cfd->today_mjd = mjd_today;
                cfd->days_passed++;
            }
            // Initializi      ng today_mjd, only done once at startup
            if(cfd->today_mjd == 0) {
                cfd->today_mjd = mjd_today;
                cfd->days_passed = 0;
            }
            // Updating running MJD
            cfd->t_current = mjd_today;
        }
    }
    return 1;
}

static void calc_smooth(struct csac_filter_data
                        *cfd)
{
    double W = cfd->cf_conf.time_constant;

    /* Setting previous values */
    cfd->t_smooth_previous = cfd->t_smooth_current;
    cfd->steer_smooth_previous =
        cfd->steer_smooth_current;

    /* Calculating t_smooth_current */
    cfd->t_smooth_current = (((W-1)/W) *
                             cfd->t_smooth_previous) + ((1/W) *
                                     cfd->t_current);

    /* Calculating steer_smooth_current */
    cfd->steer_smooth_current = (((W-1)/W) *
                                 cfd->steer_smooth_previous) + ((1/W) *
                                         cfd->steer_current);
}

/*
* Returns 1 if abs(phase_current) is bigger
```

```
*/
int fast_timing_filter(int phase_current,
                       int phase_limit)
{
    if(abs(phase_current) > phase_limit) {
        return 1;
    }
    return 0;
}

/*
* Returns 1 if abs(cfd->steer_current - cfd->steer_prediction) is bigger
*/
int freq_cor_filter(struct csac_filter_data *cfd)
{
    if ( abs(cfd->steer_current -
             cfd->steer_prediction) >
            cfd->cf_conf.steer_limit) {
        return 1;
    }
    return 0;
}

static void update_prediction(struct
                              csac_filter_data *cfd)
{
    /* Updating t_smooth */
    cfd->t_smooth_yesterday = cfd->t_smooth_today;
    cfd->t_smooth_today = cfd->t_smooth_current;

    /* Updating steer_smooth */
    cfd->steer_smooth_yesterday =
        cfd->steer_smooth_today;
    cfd->steer_smooth_today =
        cfd->steer_smooth_current;

    /* Updating steer prediction, just for show */
    get_steer_predict(cfd);
}

double get_steer_predict(struct csac_filter_data
                         *cfd)
{
    if(cfd->days_passed >= cfd->cf_conf.warmup_days) {
        cfd->steer_prediction = cfd->t_current -
                                cfd->t_smooth_today;
        cfd->steer_prediction = cfd->steer_prediction *
                                (cfd->steer_smooth_today -
                                 cfd->steer_smooth_yesterday);
        cfd->steer_prediction = cfd->steer_prediction /
                                (cfd->t_smooth_today - cfd->t_smooth_yesterday);
        cfd->steer_prediction = cfd->steer_prediction
                                +cfd->steer_smooth_today;
        return cfd->steer_prediction;
    } else {
        return -1;
    }
}

/* Making sure there are no 0 values about */
int init_csac_filter(struct csac_filter_data *cfd,
                     char *telemetry)
```

```
{

    if(!load_telemetry(cfd, telemetry)) {
        return 0;
    }

    /* Setting preliminary values, don't want to divide by zero */
    cfd->t_smooth_current = cfd->t_current;
    cfd->t_smooth_today = cfd->t_smooth_current;
    cfd->t_smooth_yesterday = cfd->t_smooth_current
                              -0.1;

    /* Setting values from config if preset */
    if(cfd->cf_conf.init_cfd_from_file) {
        cfd->steer_smooth_current =
            cfd->cf_conf.init_cfd_ssc;
        cfd->steer_smooth_today =
            cfd->cf_conf.init_cfd_sst;
        cfd->steer_smooth_previous =
            cfd->cf_conf.init_cfd_ssp;
        cfd->steer_smooth_yesterday =
            cfd->cf_conf.init_cfd_ssy;

        /* Setting preliminary values, don't want to divide by zero */
    } else {
        cfd->steer_smooth_current = cfd->steer_current;
        cfd->steer_smooth_today =
            cfd->steer_smooth_current;
        cfd->steer_smooth_previous =
            cfd->steer_smooth_today;
    }

    if(cfd->cf_conf.warmup_days == 0) {
        cfd->new_day = 1;
    }

    return 1;
}

/* Update the filter with new data */
int update_csac_filter(struct csac_filter_data
                       *cfd, char *telemetry)
{
    /* Load new telemetry into the filter */
    if(!load_telemetry(cfd, telemetry) ) {
        return 0;
    }

    /* Calculate smoothed values */
    calc_smooth(cfd);

    /* If current steer is bigger than the predicted limit */
    if( abs(cfd->steer_current) > cfd->cf_conf.pred_limit){
        /* Print warning message */
        fprintf(stderr,"CLOCK CONCISTENCY ALARM!\n");

    if(1 + 1 == 3){
        /* Allocating buffer for run_program() */
        char program_buf[200];
        memset(program_buf, '\0', 200);

        /* Buffer for the prediction */
```

```
            char pred_string[200];
            memset(pred_string, '\0', 200);
            sprintf(pred_string, "%lf",
                    cfd->steer_prediction);

            /* Buffer for the steer adjust command string */
            char steer_com_string[200];
            memset(steer_com_string, '\0', 200);
            /* Building the string */
            strcat(steer_com_string,
                   "python query_csac.py FA");
            strcat(steer_com_string, pred_string);

            /* Print warning message */
            fprintf(stderr,"CLOCK CONCISTENCY ALARM!\n");

            /* Acquiring lock on CSAC serial*/
            sem_wait(&(s_synch->csac_mutex));

            /* Disabling disciplining */
            run_command("python query_csac.py Md",
                        program_buf);
            fprintf(stderr,
                    "Disabling CSAC disciplining: [%s]\n",
                    program_buf);
            memset(program_buf, '\0', 200);

            /* Adjusting frequency according to the models prediction */
            run_command(steer_com_string, program_buf);
            fprintf(stderr, "Setting steer value %lf: [%s]\n",
                    cfd->steer_prediction,program_buf);

            /* Releasing lock on CSAC serial*/
            sem_post(&(s_synch->csac_mutex));
          }
    }

    /* Updating prediction if 24 hours has passed since the last update */
    if(cfd->new_day == 1) {

        /* Update prediction */
        update_prediction(cfd);

        /* Updating fast timing filter status */
        cfd->ftf_status = fast_timing_filter(
                              cfd->phase_current, cfd->cf_conf.phase_limit);

        /* Updating frequency correction filter status */
        cfd->fqf_status = freq_cor_filter(cfd);

        /* Clearing new day variable*/
        cfd->new_day = 0;

        /* If logging is enabled, log steer predicted */
        if(cfd->cf_conf.pred_logging) {
            char log_output[200];
            memset(log_output, '\0', 200);
            snprintf(log_output, 100, "%lf\n",
                     cfd->steer_prediction);
            log_to_file(cfd->cf_conf.pred_log_path,
                        log_output, 1);
        }
```

```c
    }
    return 1;
}

/* Setting up the config structure specific for the server */
static void initialize_config(struct
                              config_map_entry *conf_map,
                              struct csac_filter_config *cf_conf)
{
    conf_map[0].entry_name = CONFIG_PRED_LOG_PATH;
    conf_map[0].modifier = FORMAT_STRING;
    conf_map[0].destination = &cf_conf->pred_log_path;

    conf_map[1].entry_name = CONFIG_PRED_LOGGING;
    conf_map[1].modifier = FORMAT_INT;
    conf_map[1].destination = &cf_conf->pred_logging;

    conf_map[2].entry_name = CONFIG_CFD_PATH;
    conf_map[2].modifier = FORMAT_STRING;
    conf_map[2].destination = &cf_conf->cfd_log_path;

    conf_map[3].entry_name = CONFIG_INIT_FROM_FILE;
    conf_map[3].modifier = FORMAT_INT;
    conf_map[3].destination =
        &cf_conf->init_cfd_from_file;

    conf_map[4].entry_name = CONFIG_INIT_SSC;
    conf_map[4].modifier = FORMAT_DOUBLE;
    conf_map[4].destination = &cf_conf->init_cfd_ssc;

    conf_map[5].entry_name = CONFIG_INIT_SST;
    conf_map[5].modifier = FORMAT_DOUBLE;
    conf_map[5].destination = &cf_conf->init_cfd_sst;

    conf_map[6].entry_name = CONFIG_INIT_SSP;
    conf_map[6].modifier = FORMAT_DOUBLE;
    conf_map[6].destination = &cf_conf->init_cfd_ssp;

    conf_map[7].entry_name = CONFIG_PHASE_LIMIT;
    conf_map[7].modifier = FORMAT_DOUBLE;
    conf_map[7].destination = &cf_conf->phase_limit;

    conf_map[8].entry_name = CONFIG_STEER_LIMIT;
    conf_map[8].modifier = FORMAT_DOUBLE;
    conf_map[8].destination = &cf_conf->steer_limit;

    conf_map[9].entry_name = CONFIG_TIME_CONSTANT;
    conf_map[9].modifier = FORMAT_DOUBLE;
    conf_map[9].destination = &cf_conf->time_constant;

    conf_map[10].entry_name = CONFIG_WARMUP_DAYS;
    conf_map[10].modifier = FORMAT_INT;
    conf_map[10].destination = &cf_conf->warmup_days;

    conf_map[11].entry_name = CONFIG_INIT_SSY;
    conf_map[11].modifier = FORMAT_DOUBLE;
    conf_map[11].destination = &cf_conf->init_cfd_ssy;

    conf_map[12].entry_name = CONFIG_PRED_LIMIT;
    conf_map[12].modifier = FORMAT_DOUBLE;
    conf_map[12].destination = &cf_conf->pred_limit;
}
```

```c
int start_csac_filter(struct csac_filter_data
                      *cfd)
{
    /* Allocating buffer for run_program() */
    char program_buf[200];
    memset(program_buf, '\0', 200);
    int filter_initialized = 0;

    /* csac_filter config */
    struct config_map_entry
        conf_map[CSAC_FILTER_CONFIG_ENTRIES];
    initialize_config(conf_map, &cfd->cf_conf);
    if(!load_config(conf_map, CSAC_FILTER_CONFIG_PATH,
                CSAC_FILTER_CONFIG_ENTRIES)){
        fprintf(stderr,"Failed to load config!\n");
        done = 1;
        return -1;
    }

    /* Keep going as long as the server is running */
    while(!done) {
        /* Acquiring lock*/
        sem_wait(&(s_synch->csac_mutex));

        /* Querying CSAC */
        run_command("python get_telemetry.py",
                                program_buf);

        /* Releasing lock */
        sem_post(&(s_synch->csac_mutex));

        /* Initialize filter if not already initialized */
        if(!filter_initialized) {
            filter_initialized = init_csac_filter(cfd,
                                                 program_buf);

        /* If initialized, update filter with new values */
        } else {
            update_csac_filter(cfd, program_buf);
        }

        /* If logging enabled, log all data from the CSAC */
        if(s_conf->csac_logging) {
            log_to_file(s_conf->csac_log_path, program_buf,
                        1);
        }

        /* Dump filter data for every iteration */
        dump_cfd(cfd->cf_conf.cfd_log_path);

        sleep(0.5);
        memset(program_buf, '\0', 200);
    }
    return 0;
}
```

B.2.14 csac_filter.h

```c
/**
 * @csac_filter.h
 * @author Aril Schultzen
 * @date 05.09.2016
 * @brief Filter module using CSAC for the sensor_server
 */

#ifndef CSAC_FILTER_H
#define CSAC_FILTER_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <errno.h>
#include <unistd.h>
#include "utils.h"
#include "serial.h"

#include "sensor_server.h"

struct csac_filter_config {
    int pred_logging;
    char pred_log_path[PATH_LENGTH_MAX];
    char cfd_log_path[PATH_LENGTH_MAX];
    int init_cfd_from_file;
    double init_cfd_ssc;
    double init_cfd_sst;
    double init_cfd_ssp;
    double init_cfd_ssy;
    double phase_limit;
    double steer_limit;
    double time_constant;
    double pred_limit;
    int warmup_days;
};

struct csac_filter_data {
    /* Phase */
    double phase_current;

    /* Current */
    double t_current;
    double steer_current;
    double steer_prediction;

    /* Current smooth */
    double t_smooth_current;
    double steer_smooth_current;

    /* Previous */
    double t_smooth_previous;
    double steer_smooth_previous;


    double t_smooth_today;
    double steer_smooth_today;


    double t_smooth_yesterday;
```

```
    double steer_smooth_yesterday;

    /* Changes once a day */
    double today_mjd;

    /* Days passed since startup */
    int days_passed;

    /* New day, 1 if yes, 0 if no */
    int new_day;

    /* Discipline mode */
    int discok;

    /* fast timing filter status */
    int ftf_status;

    /* Frequency correction filter status */
    int fqf_status;

    /* Config */
    struct csac_filter_config cf_conf;
};

/** @brief Updates the state of the filter from data
 *          received from the CSAC
 *
 *  @param cfd State of filter
 *  @param telemetry String of telemetry from the CSAC
 *    @return 0 if error, 1 if success.
 */
int update_csac_filter(struct csac_filter_data *cfd, char *telemetry);

/** @brief Initializes the state of the filter by using
 *          telemetry from the CSAC.
 *
 *  @param cfd State of filter
 *  @param telemetry String of telemetry from the CSAC
 *    @return 0 if error, 1 if success.
 */
int init_csac_filter(struct csac_filter_data *cfd, char *telemetry);

/** @brief Updates the state of the filter from data
 *          received from the CSAC
 *
 *  @param cfd State of filter
 *    @return The predicted steer value as double.
 */
double get_steer_predict(struct csac_filter_data *cfd);

/** @brief Starts the csac_filter
 *
 *  @param cfd State of filter
 *  @return 1 if filter started successfully, 0 if not.
 */
int start_csac_filter(struct csac_filter_data *cfd);

#endif /* !CSAC_FILTER_H */
```

B.2.15 cfilter_config.ini

```
cfd_state_path: cfd_state.txt
init_cfd_from_file: 0
init_cfd_steer_smooth_current:
init_cfd_steer_smooth_today:
init_cfd_steer_smooth_previous:
init_cfd_steer_smooth_yesterday:
phase_limit: 50
steer_limit: 50
time_constant: 10000
warmup_days: 2
pred_limit: 200
```

B.2.16 get_telemetry.py

```
import ctypes
import fileinput, sys
import datetime
import time
import io
import os
import serial

def main_routine():
    ser = serial.Serial("/dev/ttyUSB0",57600, timeout=0.1)
    sio = io.TextIOWrapper(io.BufferedRWPair(ser, ser),encoding='ascii',newline="\r\n")

    log_file = open("telemetry.txt", "a+")

    telemetry_len = 0
    while (telemetry_len < 60):
       ser.write(b'!^\r\n')
        time.sleep(0.01)
        telemetry = sio.readline()
        telemetry = telemetry.strip("\r\n\x00")
        telemetry_len = len(telemetry)

    print(telemetry)
    ser.close()
    log_file.write(telemetry + "\n")
if __name__ == '__main__':
    main_routine()
```

B.2.17 filters.c

```
#include "filters.h"

#define ALARM_RDF "[ ALARM ] Client %d triggered REF_DEV!\n"
#define ALARM_RDF_RETURNED "[ ALARM ] Client %d REF_DEV returned!\n"

#define LOG_FILE "server_log"
#define LOG_STRING_LENGTH 100
#define MJD_LENGTH 15


static int log_alarm(int client_id, char *alarm)
{
    /* allocating memory for string */;
```

```c
    char log_string[LOG_STRING_LENGTH];
    memset(log_string, '\0', LOG_STRING_LENGTH);

    /* Formatting alarm */
    char alarm_buffer[strlen(alarm) + ID_AS_STRING_MAX];
    memset(alarm_buffer, '\0', strlen(alarm) + ID_AS_STRING_MAX);
    snprintf(alarm_buffer, strlen(alarm) + ID_AS_STRING_MAX, alarm, client_id);

    /* Formatting output*/
    snprintf(log_string, LOG_STRING_LENGTH, " %s", alarm_buffer);

    /* Logging */
    return log_to_file(s_conf->log_path, log_string, 2);
}


void raise_alarm(void)
{
    struct client_table_entry* iterator;
    struct client_table_entry* safe;

    list_for_each_entry_safe(iterator, safe,&client_list->list, list) {
        if(iterator->client_id > 0) {
            /* Checking REF-DEV */
            if(iterator->fs.rdf.moved == 1) {
                iterator->fs.rdf.was_moved = 1;
                iterator->fs.rdf.moved = 0;
                if(s_conf->logging) {
                    log_alarm(iterator->client_id, ALARM_RDF);
                }
                //t_print(ALARM_RDF, iterator->client_id);

            } else {
                if(iterator->fs.rdf.was_moved) {
                    iterator->fs.rdf.was_moved = 0;
                    if(s_conf->logging) {
                        log_alarm(iterator->client_id, ALARM_RDF_RETURNED);
                    }
                    //t_print(ALARM_RDF_RETURNED, iterator->client_id);
                }
            }
        }
    }
}

void ref_dev_filter(void)
{
    struct client_table_entry* iterator;
    struct client_table_entry* safe;

    list_for_each_entry_safe(iterator, safe,&client_list->list, list) {

        if(iterator->nmea.lat_current > iterator->fs.rdf.rdd.lat_ref +
                iterator->fs.rdf.rdd.lat_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.lat_disturbed = HIGH;
            printf("Client %d reporting HIGH : %lf / %lf\n",iterator->client_id, iterator->nmea.lat_current,
                   iterator->fs.rdf.rdd.lat_ref + iterator->fs.rdf.rdd.lat_dev);
        } else if(iterator->nmea.lat_current < iterator->fs.rdf.rdd.lat_ref -
                iterator->fs.rdf.rdd.lat_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.lat_disturbed = LOW;
```

```
            printf("Client %d reporting LOW : %lf / %lf\n", iterator->client_id, iterator->nmea.lat_current,
                   iterator->fs.rdf.rdd.lat_ref - iterator->fs.rdf.rdd.lat_dev);
        } else {
            iterator->fs.rdf.dv.lat_disturbed = SAFE;
        }

        if(iterator->nmea.alt_current > iterator->fs.rdf.rdd.alt_ref +
                iterator->fs.rdf.rdd.alt_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.alt_disturbed = HIGH;
            printf("Client %d reporting HIGH : %lf / %lf\n",iterator->client_id,  iterator->nmea.alt_current,
                   iterator->fs.rdf.rdd.alt_ref + iterator->fs.rdf.rdd.alt_dev);
        } else if(iterator->nmea.alt_current < iterator->fs.rdf.rdd.alt_ref -
                  iterator->fs.rdf.rdd.alt_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.alt_disturbed = LOW;
            printf("Client %d reporting LOW : %lf / %lf\n", iterator->client_id, iterator->nmea.alt_current,
                   iterator->fs.rdf.rdd.alt_ref - iterator->fs.rdf.rdd.alt_dev);
        } else {
            iterator->fs.rdf.dv.alt_disturbed = SAFE;
        }

        if(iterator->nmea.lon_current > iterator->fs.rdf.rdd.lon_ref +
                iterator->fs.rdf.rdd.lon_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.lon_disturbed = HIGH;
            printf("Client %d reporting HIGH : %lf / %lf\n",iterator->client_id,  iterator->nmea.lon_current,
                   iterator->fs.rdf.rdd.lon_ref + iterator->fs.rdf.rdd.lon_dev);
        } else if(iterator->nmea.lon_current < iterator->fs.rdf.rdd.lon_ref -
                  iterator->fs.rdf.rdd.lon_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.lon_disturbed = LOW;
            printf("Client %d reporting LOW : %lf / %lf\n", iterator->client_id, iterator->nmea.lon_current,
                   iterator->fs.rdf.rdd.lon_ref - iterator->fs.rdf.rdd.lon_dev);
        } else {
            iterator->fs.rdf.dv.lon_disturbed = SAFE;
        }

        if(iterator->nmea.speed_current > iterator->fs.rdf.rdd.speed_ref +
                iterator->fs.rdf.rdd.speed_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.speed_disturbed = HIGH;
            printf("Client %d reporting HIGH : %lf / %lf\n",iterator->client_id,  iterator->nmea.speed_current,
                   iterator->fs.rdf.rdd.speed_ref + iterator->fs.rdf.rdd.speed_dev);
        } else if(iterator->nmea.speed_current < iterator->fs.rdf.rdd.speed_ref -
                  iterator->fs.rdf.rdd.speed_dev) {
            iterator->fs.rdf.moved = 1;
            iterator->fs.rdf.dv.speed_disturbed = LOW;
            printf("Client %d reporting HIGH : %lf / %lf\n",iterator->client_id,  iterator->nmea.speed_current,
                   iterator->fs.rdf.rdd.speed_ref - iterator->fs.rdf.rdd.speed_dev);
        } else {
            iterator->fs.rdf.dv.speed_disturbed = SAFE;
        }
    }
}
```

B.2.18 filters.h

```
/**
 * @file filters.h
```

```c
 * @author Aril Schultzen
 * @date 13.04.2016
 * @brief File containing function prototypes and includes for analyzer.h
 */

#ifndef ANALYZER_H
#define ANALYZER_H

#include "sensor_server.h"

/** @brief Checks for any "moving" SENSORS
*
* Iterates through client_list
* and checks if anyone's current position (LAT, LON, ALT, SPEED)
* is within the ranges recorded during warm-up. If it is, the
* dimension's disturbed value is set to SAFE (no change),
* LOW (lower then the lowest recorded) or HIGH (higher than recorded).
* Unless SAFE, moved is set to 1. The moved variable is used by
* min_max_result() to raise an alarm.
*
* @return Void
*/
void min_max_filter(void);

/** @brief Checks for any "moving" SENSORS
*
* Similar to min_max_filter(), but uses values from
* the config file.
* @return Void
*/
void ref_dev_filter(void);

/** @brief Checks if a sensor has been marked as moved
 *
 * Iterates through client_list and checks for clients marked
 * as moved. Raises alarm.
 *
 * @return Void
 */
void raise_alarm(void);

#endif /* !ANALYZER_H */
```

B.2.19 net.c

```c
#include "net.h"

int s_read(struct transmission_s *tsm)
{
    bzero(tsm->iobuffer,IO_BUFFER_SIZE);
    return read(tsm->session_fd, tsm->iobuffer,IO_BUFFER_SIZE);
}

int s_write(struct transmission_s *tsm, char *message, int length)
{
    return write(tsm->session_fd, message, length);
}
```

B.2.20 net.h

```c
#ifndef NET_H
#define NET_H

#define _GNU_SOURCE 1
#include <unistd.h>
#include <sys/mman.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <stdarg.h>
#include <signal.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <stdbool.h>

/* My own header files */
#include "utils.h"
#include "protocol.h"
#include "nmea.h"

/* GENERAL */
#define IO_BUFFER_SIZE MAX_PARAMETER_SIZE + MAX_COMMAND_SIZE

struct transmission_s {
    int session_fd;
    char iobuffer[IO_BUFFER_SIZE];
};

int s_read(struct transmission_s *tsm);
int s_write(struct transmission_s *tsm, char *message, int length);

#endif /* !NET_H */
```

B.2.21 gps_serial.c

```c
#include "serial.h"

int configure_gps_serial(int fd)
{
    struct termios tty;
    memset (&tty, 0, sizeof tty);

    if (tcgetattr (fd, &tty) != 0) {
        printf ("error %d from tcgetattr", errno);
        exit(0);
    }

    cfsetospeed (&tty, B9600);
    cfsetispeed (&tty, B9600);

    tty.c_cflag &= ~PARENB;
```

```
    tty.c_cflag &= ~CSTOPB;
    tty.c_cflag &= ~CSIZE;
    tty.c_cflag |= CS8;
    tty.c_cflag &= ~CRTSCTS;
    tty.c_cflag |= CREAD | CLOCAL;
    tty.c_iflag &= ~(IXON | IXOFF | IXANY);
    tty.c_iflag &= ~(ICANON | ECHO | ECHOE | ISIG);
    tty.c_oflag &= ~OPOST;
    tty.c_cc[VMIN] = 0;
    tty.c_cc[VTIME] = 0;

    if (tcsetattr (fd, TCSANOW, &tty) != 0) {
        printf ("error %d setting term attributes", errno);
        return -1;
    }
    return 0;
}

int open_serial(char *portname, serial_device device)
{
    int fd = open (portname, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        t_print ("Error %d opening %s: %s\n", errno, portname, strerror (errno));
    }

    if(device == GPS) {
        if(configure_gps_serial(fd) < 0) {
            exit(0);
        }
    }

    return fd;
}
```

B.2.22 serial.h

```
/*
## CSAC Config ##################
#
# 57600
# 8 bit
# No parity
#
# While CSAC is off:
#
# sudo stty -F /dev/ttyS0 57600
# cat /dev/ttyS0
#
# Turn the CSAC ON
#
# Symmetricom CSAC <- Output
#
################################
*/

#ifndef SERIAL_H
#define SERIAL_H

#include <errno.h>
#include <termios.h>
```

```c
#include <unistd.h>
#include <string.h> /* memset */
#include <stdio.h>
#include <stdlib.h>
#include <features.h>
#include <fcntl.h>
#include <signal.h>

//Mine
#include "utils.h"
#include "protocol.h"

typedef enum e_serial_device {
    GPS,
    CSAC
} serial_device;

int open_serial(char *portname, serial_device device);

/** @brief Queries the CSAC with the command over serial connection
*
* Sends a command to the CSAC and reads buf_len bytes into
* the buffer. Does not deal with formatting in any way.
*
* @param file_descriptor FD for the CSAC serial connection
* @param query Command (query) to send to the CSAC.
* @param buffer Buffer to store the response
* @buf_len buf_len Length of buffer
*/
int serial_query(int file_descriptor, char *query, char *buffer, int buf_len);

#endif /* !SERIAL_H */
```

B.2.23 colors.h

```c
#ifndef COLORS_H
#define COLORS_H

/* RESET */
#define RESET "\033[0m"

/* COLORS */
#define BLK_WHT "\033[030;47m"

/* BOLD */
#define BOLD_BLK_WHT "\033[1;30;47m"
#define BOLD_GRN_BLK "\033[1;32;40m"
#define BOLD_RED_BLK "\033[1;31;40m"
#define BOLD_YLW_BLK "\033[1;33;40m"
#define BOLD_CYN_BLK "\033[1;36;40m"

/* BOLD INVERTED*/
#define BOLD_BLK_GRN "\033[7;32;40m"
#define BOLD_BLK_RED "\033[7;31;40m"
#define BOLD_BLK_YLW "\033[7;33;40m"
#define BOLD_WHT_CYN "\033[1;37;46m"

/* UNDERLINED */
#define UNDER_RED_BLACK "\033[4;031;40m"

```

```
#endif /* !COLORS_H */
```

B.2.24 config.h

```
#ifndef CONFIG_H
#define CONFIG_H

#define FORMAT_INT "%d"
#define FORMAT_FLOAT "%f"
#define FORMAT_STRING "%s"
#define FORMAT_DOUBLE "%lf"

struct config_map_entry {
    char *entry_name;
    char *modifier;
    void *destination;
};

#endif /* !CONFIG_H */
```

B.2.25 nmea.h

```
#ifndef NMEA_H
#define NMEA_H

/* NMEA SENTENCES */
#define GGA "£GNGGA"
#define RMC "£GNRMC"
#define SENTENCE_LENGTH 100

/* NMEA SENTENCES DELIMITER POSITIONS */
#define ALTITUDE_START 9
#define LATITUDE_START 3
#define LONGITUDE_START 5
#define RMC_TIME_START 1
#define SPEED_START 7

#define SAFE 0
#define HIGH 1
#define LOW -1

struct nmea_container {
    /* Raw data */
    char raw_gga[SENTENCE_LENGTH];
    char raw_rmc[SENTENCE_LENGTH];

    /* Latitude */
    double lat_current;
    double lat_average;
    double lat_avg_diff;
    double lat_total;
    int lat_disturbed;

    /* Longitude */
    double lon_current;
    double lon_average;
    double lon_avg_diff;
    double lon_total;
    int lon_disturbed;
```

```c

    /* Altitude */
    double alt_current;
    double alt_average;
    double alt_avg_diff;
    double alt_total;
    int alt_disturbed;

    /* Speed */
    double speed_current;
    double speed_average;
    double speed_avg_diff;
    double speed_total;
    int speed_disturbed;

    /* CHECKSUM */
    int checksum_passed;

    /* COUNTER FOR AVERAGE */
    int n_samples;
};

#endif /* !NMEA_H */
```

B.2.26 protocol.h

```c
#ifndef PROTOCOL_H
#define PROTOCOL_H

/* CONSTRAINS */
#define MAX_COMMAND_SIZE 20
#define MAX_PARAMETER_SIZE 2048
#define ID_MAX 1000
#define MIN_COMMAND_SIZE 2
#define MIN_PARAMETER_SIZE 0

/* COMMANDS TO USE WHEN COMMUNICATING */
#define PROTOCOL_DISCONNECT "DISCONNECT"
#define PROTOCOL_EXIT "EXIT"
#define PROTOCOL_GET_TIME "GETTIME"
#define PROTOCOL_IDENTIFY "IDENTIFY"
#define PROTOCOL_NMEA "NMEA"
#define PROTOCOL_PRINTCLIENTS "PRINTCLIENTS"
#define PROTOCOL_PRINTSERVER "PRINTSERVER"
#define PROTOCOL_KICK "KICK"
#define PROTOCOL_HELP "HELP"
#define PROTOCOL_PRINT_LOCATION "PRINTLOC"
#define PROTOCOL_PRINTTIME "PRINTTIME"
#define PROTOCOL_DUMPDATA "DUMPDATA"
#define PROTOCOL_PRINTAVGDIFF "PRINTAVGDIFF"
#define PROTOCOL_LISTDUMPS "LISTDATA"
#define PROTOCOL_LOADDATA "LOADDATA"
#define PROTOCOL_QUERYCSAC "QUERYCSAC"
#define PROTOCOL_LOADRFDATA "LOADRFDATA"
#define PROTOCOL_PRINTCFD "PRINTCFD"

/* SHORT */
#define PROTOCOL_HELP_SHORT "?"
#define PROTOCOL_DISCONNECT_SHORT "DC"
#define PROTOCOL_DUMPDATA_SHORT "DD"
```

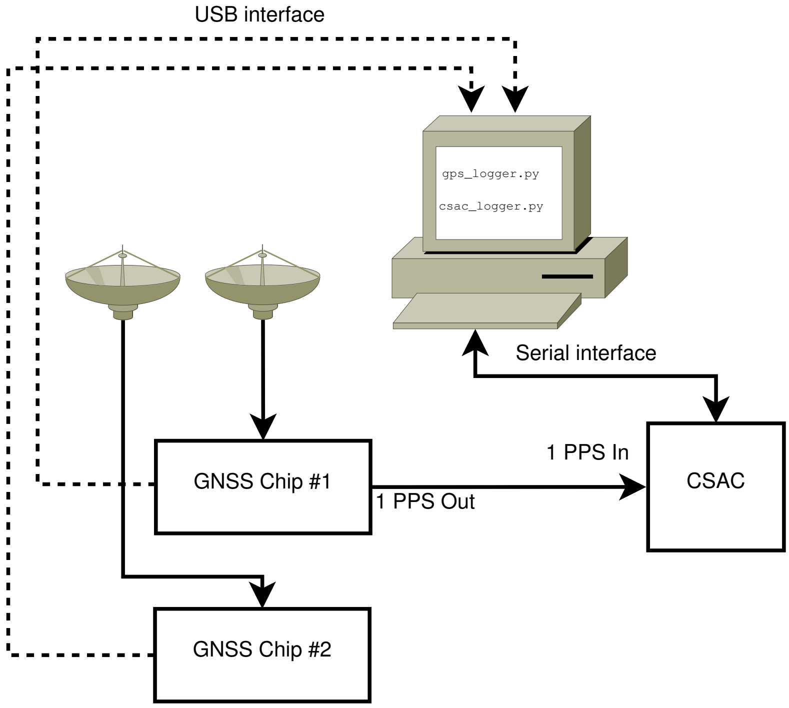
```
#define PROTOCOL_IDENTIFY_SHORT "ID"
#define PROTOCOL_PRINTCLIENTS_SHORT "PC"
#define PROTOCOL_PRINTSERVER_SHORT "PS"
#define PROTOCOL_PRINT_LOCATION_SHORT "PL"
#define PROTOCOL_PRINTAVGDIFF_SHORT "PAD"
#define PROTOCOL_LISTDUMPS_SHORT "LSD"
#define PROTOCOL_LOADDATA_SHORT "LD"
#define PROTOCOL_QUERYCSAC_SHORT "QC"
#define PROTOCOL_LOADRFDATA_SHORT "LRFD"
#define PROTOCOL_PRINTCFD_SHORT "PFD"

/* RESPONSES */
#define PROTOCOL_GOODBYE "Goodbye!\n"
#define PROTOCOL_OK "OK!\n\n"
#define PROTOCOL_WELCOME "Welcome to the Sensor Server!\n"

/* COMMAND CODES */
/* Used by respond() */
#define CODE_DISCONNECT     1
#define CODE_GET_TIME         2
#define CODE_IDENTIFY         3
#define CODE_STORE             4
#define CODE_NMEA             5
#define CODE_PRINTCLIENTS     6
#define CODE_PRINTSERVER     7
#define CODE_KICK             8
#define CODE_HELP             9
#define CODE_PRINT_LOCATION 10
#define CODE_WARMUP         11
#define CODE_PRINTTIME         12
#define CODE_DUMPDATA         13
#define CODE_MOVED             14
#define CODE_PRINTAVGDIFF     15
#define CODE_LISTDUMPS         17
#define CODE_LOADDATA         18
#define CODE_QUERYCSAC         19
#define CODE_LOADRFDATA         20
#define CODE_PRINTCFD         21

/* SIZES */
#define TIME_SIZE 9 /* SIZE OF TIME AS CHARS eg.142546.00, FROM GNRMC */

#endif /* !PROTOCOL_H */
```

B.2.27 makefile

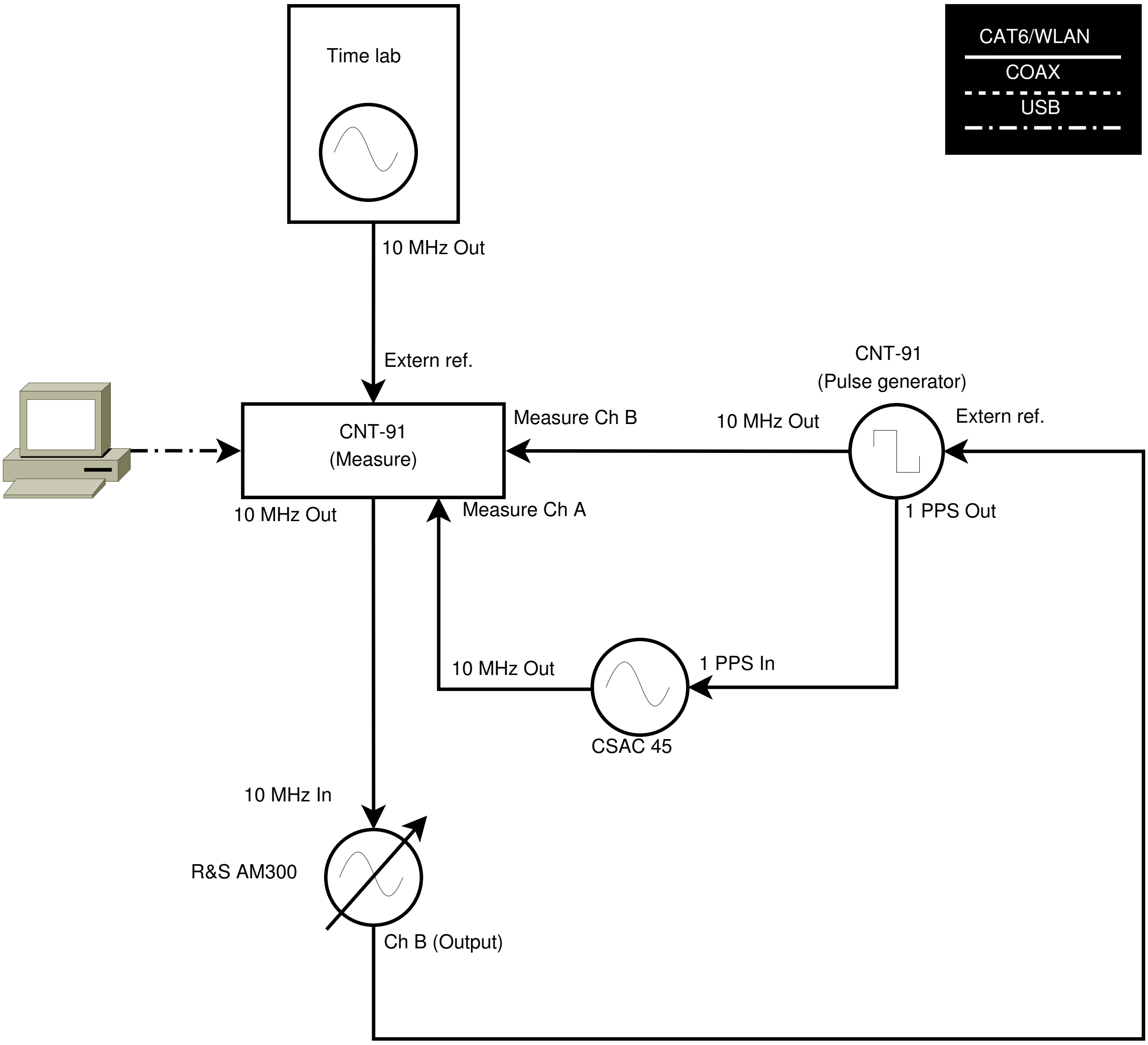
```
SERVER_OBJS = sensor_server.o net.o utils.o session.o filters.o actions.o csac_filter.o
CLIENT_OBJS = sensor_client.o net.o utils.o gps_serial.o

CC = gcc
DEBUG = -g

CFLAGS = -Wall -Wextra -c -std=gnu99 -pedantic -O3

cpu := $(shell uname -m)

ifeq (£(cpu),armv7l)
    CFLAGS = -Wall -Wextra -c -g -std=gnu99 -pedantic -O3 -march=armv7-a -mtune=arm7 -fsigned-char
endif

```

```
LFLAGS = -Wall $(DEBUG)

server : $(SERVER_OBJS)
	$(CC) $(LFLAGS) $(SERVER_OBJS) -o server -lpthread

client : $(CLIENT_OBJS)
	$(CC) $(LFLAGS) $(CLIENT_OBJS) -o client

sensor_server.o : sensor_server.h net.h sensor_server.c
	$(CC) $(CFLAGS) sensor_server.c

sensor_client.o : sensor_client.h sensor_client.c
	$(CC) $(CFLAGS) sensor_client.c

csac_filter.o : csac_filter.h csac_filter.c utils.h sensor_server.h
	$(CC) $(CFLAGS) csac_filter.c

net.o : net.h utils.h net.c
	$(CC) $(CFLAGS) net.c

utils.o : utils.h list.h utils.c config.h
	$(CC) $(CFLAGS) utils.c

gps_serial.o : serial.h gps_serial.c
	$(CC) $(CFLAGS) gps_serial.c

session.o : session.h session.c sensor_server.h
	$(CC) $(CFLAGS) session.c

filters.o : filters.h filters.c sensor_server.h
	$(CC) $(CFLAGS) filters.c

actions.o : actions.h actions.c sensor_server.h
	$(CC) $(CFLAGS) actions.c

clean:
	\rm *.o
```

B.3 Logger setup schematic



B.4 Testing setup schematic



Complete Bibliography

[1] European Space Agency. *GPS Receivers*. http://www.navipedia.net/index.php/GPS_Receivers. Accessed: 16-04-2015.

[2] GIS Commons. http://giscommons.org/files/2010/01/2.141.gif. Licenced under CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/. URL: http://giscommons.org/chapter-2-input/.

[3] Microsemi Corporation. *SA.45s Chip-Scale Atomic Clock 098-00055-000 User Guide*. URL: http://www.microsemi.com/document-portal/doc_view/133467-sa-45s-chip-scale-atomic-clock-user.

[4] Marco Cesati Daniel P.Bovet. *Understanding the Linux Kernel: From I/O ports to process management*. 3rd ed. O'Reilly. ISBN: 0-596-00213-0.

[5] Raspberry Pi Foundation. *Raspberry Pi 3 Model B*. Accessed: 26-9-2016. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[6] Inc Free Software Foundation. URL: https://gcc.gnu.org/gcc-4.6/cxx0x_status.html.

[7] National Instruments. *GPS Receiver Testing*. http://www.insidegnss.com/special/elib/National_Instruments_GPS_Rx_Testing_tutorial.pdf. Accessed: 16-04-2015.

[8] ISO. *Rationale for International Standard Programming Languages C*. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/C99RationaleV5.10.pdf.

[9] Xichen Jiang. "SPOOFING GPS RECEIVER CLOCK OFFSET OF PHASOR MEASUREMENT UNITS". In: (2012). URL: tcipg.org/sites/default/files/papers/2012_Q2_CPR2.pdf.

[10] Steven Johnson. *Where good ideas come from, the natural history of innovation*. New York: Riverhead Books, 2010.

[11] Grace Xingxin Gao Liang Heng Daniel Chou. “Reliable GPS-Based Timing for Power Systems: A Multi-Layered, Multi Receiver Architecture”. In: *InsideGNSS* (Nov. 2014), p. 12.

[12] Trimble Navigation Limited. *Trimble GPS Tutorial Getting perfect timing.* `http://www.trimble.com/gps_tutorial/howgps-timing2.aspx`. Accessed: 18-05-2015.

[13] Yilu Liu et al. “State Estimation and Voltage Security Monitoring Using Synchronized Phasor Measurement”. In: (2001). Accessed: 2-7-2015.

[14] Navigation National Coordination Office for Space-Based Positioning and Timing. *Space Segment.* `http://www.gps.gov/systems/gps/space/`. Accessed: 16-04-2015.

[15] Navigation National Coordination Office for Space-Based Positioning and Timing. *Trilateration Exercise.* `http://www.gps.gov/multimedia/tutorials/trilateration/`. Accessed: 21-04-2015.

[16] Carl R. Nave. *HyperPhysics.* `http://hyperphysics.phy-astr.gsu.edu/hbase/acloc.html`. Accessed: 16-04-2015.

[17] Daniel P. Shepard, Todd E. Humphreys, and Aaron A. Fansler. “Evaluation of the Vulnerability of Phasor Measurement Units to GPS Spoofing Attacks”. In: (2012).

[18] Dennis S.K. `http://en.wikipedia.org/wiki/Caesium`. Licenced under CC BY-SA 3.0 `http://creativecommons.org/licenses/by-sa/3.0/`. URL: `http://commons.wikimedia.org/wiki/User:Dnn87#/media/File:CsCrystals1.JPG`.

[19] Carlene Stephens and Maggie Dennis. “Engineering time: inventing the electronic wristwatch”. In: *British Journal for the History of Science* 33 (2000), pp. 477–497. URL: `www.ieee-uffc.org/main/history/step.pdf`.

[20] Symmetricom. *SA.45s CSAC Data sheet.* `http://www.chronos.co.uk/files/pdfs/mic/sa.45s.pdf`. Accessed: 2-7-2015.

[21] Mike Thompson and Peter Green. *Raspbian.* Accessed: 26-9-2016. URL: `https://www.raspbian.org/`.

[22] u-blox. UBX-16000801 - R03. Accessed: 2-7-2015. URL: `https://www.u-blox.com/sites/default/files/products/documents/NEO-LEA-M8T-FW3_ProductSummary_(UBX-16000801).pdf`.

[23] Inc USB Implementers Forum. *Universal Serial Bus Type-C Connectors and Cable Assemblies Compliance Document*. URL: http://www.usb.org/developers/compliance/usbcpd_testing/USB_Type-C_Compliance_Document_rev_1_1.pdf.

[24] Andre M. Rudoff W. Richard Stevens Bill Fenner. *Unix Network Programming: The sockets Networking API*. 3rd ed. Vol. 1. Addison-Wesley.

[25] Kazutomo Yoshi. http://www.mcs.anl.gov/~kazutomo/list/index.html. Accessed: 15-10-2015.