Spoof proof GPS timing

A detection and mitigation system for GPS time spoofing

Aril Johannes Schultzen

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

Spoof proof GPS timing
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2016

Spoof proof GPS timing

A detection and mitigation system for GPS time spoofing

Aril Johannes Schultzen

Spoof proof GPS timing

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

Printed: Reprosentralen, University of Oslo

Foreword

Five years ago I started studying for my bachelors degree in computer science at Oslo and Akershus University college of applied sciences. I had no idea what I was doing and I can clearly recall a professor telling us that "you are here because you want to learn how to program". At that point, this was news to me. Five years later I am submitting my master thesis and all I want to do is thank the people who enabled me.

Thank you, Harald Hauglin for guiding me when I doubted myself and for being patient with me and my uninformed questions. Thank you, Sverre Holm for being my supervisor and for being honest when I needed it. Thank you, Tim Dunker for providing excellent assistance and support. Thank you, Bao Chi Tran Nguyen for teaching me how to program without letting me feel utterly useless. And thank you, Ingvild Totland Pettersen for being my loving partner and for dealing with all my frustration and immaturity.

Abstract

The goal of this project was to create a prototype of a smart atomic clock controller running on commodity hardware, capable of spoof proofing a GPS controlled atomic clock, i.e harden it against GPS jamming and spoofing attacks. The project aims to use a multi-layered approach to evaluate the integrity of GPS signals and to use knowledge about the atomic clock and its behavior when GPS disciplined, to detect and mitigate a spoofing attack. Many solutions have been proposed to solve the problem of GPS time spoofing [1], but as far as we know this is an issue rarely tackled using commodity hardware and simple techniques. We aim to demonstrate the efficiency of network enabling commodity GPS receivers and connecting them to a atomic clock controller by using a client/server architecture facilitating GPS and atomic clock data analysis. The preliminary GPS manipulation tests demonstrates the feasibility of our proposal.

Contents

1 Introduction **1**

2 Background on GPS **3**
- 2.1 Clocks . . . 4
- 2.2 GPS signals and Time . . . 4
- 2.3 Threat Models and countermeasures . . . 6
  - 2.3.1 Threats . . . 6
  - 2.3.2 Countermeasures . . . 7
  - 2.3.3 Summary . . . 11
- 2.4 Benchmark clock spoofing attack . . . 12

3 Our Proposal: Spoof proof atomic clock controller **13**
- 3.1 Filtering and steering . . . 13

4 Hardware **15**
- 4.1 Atomic Clock . . . 15
- 4.2 Atomic clock controller platform . . . 15
- 4.3 GPS receiver . . . 16
  - 4.3.1 GPS receiver configuration . . . 16

5 Atomic clock controller architecture **17**
- 5.1 Client/Server model . . . 17
- 5.2 Architecture description . . . 18
  - 5.2.1 Security . . . 20

6 Atomic clock controller implementation **21**
- 6.1 The Sensor server . . . 21
- 6.2 Implementation description . . . 21
  - 6.2.1 Server core . . . 23
  - 6.2.2 Clock model implementation . . . 26
  - 6.2.3 Parser and Handler . . . 28

6.2.4 GPS filter . . . 32
6.2.5 Clock model filters . . . 33
6.2.6 GPS data . . . 34
6.2.7 Client connection . . . 34
6.2.8 Shared memory & Semaphores . . . 35
6.2.9 Client list memory segment . . . 36
6.2.10 Client linked list . . . 38
6.2.11 Client input parser . . . 39
6.2.12 Ready check algorithm . . . 40
6.3 Sensor Server Protocol . . . 41
6.4 Atomic clock Communication . . . 41
6.5 Data structures . . . 42
6.5.1 server_data . . . 43
6.5.2 server_synchro . . . 43
6.5.3 command_code . . . 43
6.5.4 nmea_container . . . 44
6.5.5 list_head . . . 45
6.5.6 transmission_s . . . 45
6.5.7 client_table_entry . . . 45
6.6 The Sensor Client . . . 47
6.7 Interfacing . . . 48

7 GPS manipulation tests **51**
7.1 Logging filter and model data . . . 51
7.2 Setup . . . 52
7.3 Test one filter limits . . . 53
7.4 Test one . . . 54
7.4.1 Goal of test one . . . 54
7.4.2 Description . . . 54
7.4.3 Observations . . . 56
7.4.4 Timing measurements . . . 66
7.4.5 Test one results . . . 68
7.5 Test two . . . 68
7.5.1 Goal of test two . . . 68
7.5.2 Change in setup . . . 68
7.5.3 Test two filter limits . . . 68
7.5.4 Description . . . 69
7.5.5 Observations . . . 70
7.5.6 Timing measurements . . . 72
7.5.7 Test two results . . . 72
7.6 Unplanned disturbance . . . 74

8 Discussion **78**
8.1 Test results . . . 78
8.2 Shortcomings in current implementation . . . 78
8.2.1 Resizing shared memory segments . . . 78
8.2.2 Atomic clock management . . . 79
8.2.3 External MJD calculation . . . 79
8.2.4 External Atomic clock communication . . . 80
8.3 Choice of programming language . . . 80
8.4 Alternative approaches . . . 81
8.4.1 Single computer, many GPS receivers . . . 81
8.4.2 Store in database and analyze . . . 81

9 Conclusion **82**

Appendices **84**

A Acknowledgments **85**
A.1 Contributions . . . 85

B Clock model **86**
B.1 Introduction . . . 86
B.2 Input data for the clock model . . . 86
B.3 Smoothing of sampled frequency steering data and estimates of the clock state . . . 88
B.4 Phase jump filter – fast timing filter . . . 89
B.5 Frequency correction filter . . . 90

C Data acquisition **92**
C.1 CSAC Logger source code . . . 92
C.2 GPS Logger source code . . . 93
C.3 GPS Logger source code . . . 95

D Sensor server software **97**
D.1 Client . . . 97
D.2 Server . . . 104

E Scripts **179**
E.1 Logger setup schematic . . . 181

F E-mails **182**
F.1 Correspondence with Mr. Davis . . . 182

G Figures **183**

Complete Bibliography **186**

Chapter 1

Introduction

Timing in the context of accurate and stable reference frequencies as used in timestamps, is both critical and valuable in the world of IT infrastructure as well as in the world of industrial control systems. The timing itself is often generated by clocks disciplined by GPS. This allows an inaccurate clock to obtain the long term accuracy and stability of a much more expensive and accurate clock. The only problem is that the GPS disciplining relies on an antenna with a clear view of the sky and uses a known civil GPS code structure. Since the code structure is known and antennas can not be hidden, a GPS receiver can be "spoofed" with a GPS-like signal generated by an attacker. GPS based timing may therefore be viewed as an unencrypted and physically unsecured port into industrial control systems.

An example of an application relying on GPS derived time is a phasor measurement unit (PMU). 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 PMUs. The common time source (and why PMUs are relevant) is often obtained by using GPS. [2] 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 in the grid, and in some cases lead to whole spread blackouts [3]. When the clocks in PMU applications are spoofed, they may end up causing damage in the network instead of protecting it [4].

GPS time is also used in telecommunication to synchronize base stations. The radio spectrum used by cellular phones is limited, making synchronization between base stations important in order to maintain efficient use of the spectrum. The ability to accurately time-stamp financial transactions is made possible all over the world using GPS and is crucial for traceability and accountability [5].

Our goal int this report was to develop a spoof proof atomic clock controller prototype by combining multiple GPS receivers, a chip scale atomic clock and a client/server architecture. This allowed us to test some strategies for spoofing detection, knowing if the GPS signals are unsuitable or unsafe, and mitigation, how to maintain accurate timing when reliable GPS synchronization is unavailable.

Chapter 2

Background on GPS

The Global Positioning System (GPS) is a utility owned by the United States that provides its users with positioning, navigation and timing services. At the end of the sixties, 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 John Hopkins University to develop the Transit system, the earliest predecessor to the GPS system [6].

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 receiver. The European Space Agency estimated that there were 2 billion GPS enabled devices by 2012 [7]. 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 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 20,372 nautical miles above your head [8]. In order for a GPS receiver to determine its position and obtain correct time, it will need four 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)

[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 [9]

of a 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:

$$Distance = Rate \times Time \tag{2.1}$$

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 that 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 therefor how long it has spent traveling. This is explained in greater detail under (2.2) [10].

2.1 Clocks

What does a 10 USD wristwatch and a 100,000 USD atomic clock have in common? They do not 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 pendulum 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. [11]. 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] [12].

2.2 GPS signals and Time

During the introduction of this chapter the properties of GPS as a tool for navigation was made apparent. This is however not the only use of GPS, it

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

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 earlier 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 that it generates itself. The receiver "slides" these codes over each other further and further in an attempt to match them up. The signals travel time is determined by how far the codes had to be slided before they matched. This is what is called *Code-phase GPS* and it is not without issues. Since the codes have a wide cycle width, almost a microsecond, there is still a significant uncertainty even though the codes match. At the speed of light a microsecond wrong translates to a roughly 300 meter error in the solved position. What many receivers do is that they start with the code-phase and move on to using measurements based on the carrier frequency. Since the frequency is much higher the uncertainty within the match decreases thus increasing the accuracy dramatically. This is what is known as *Carrier-phase GPS*. These signals can be seen as range signals, used to measure distance. This is however not the whole story. The GPS signals also include *navigation messages* like *ephemeris* as well as something called the *almanac*. The ephemeris data is information containing the orbit of every GPS satellite in the constellation. This is used by the receiver to calculate the satellites position. The almanac is a state report of the whole constellation. 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 a GPS receiver's internal clock. As previously mentioned, all your receiver needs to do to find its position in a three dimensional space, are three GPS satellites. If the GPS receivers' internal clocks were perfect, the three satellite ranges would intersect at a single point, your position. But in the real world our clocks are 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 3) 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 so, it also brings the receiver's clock back to sync with universal time. With the correct time it can also make correct and precise positioning. [13]

2.3 Threat Models and countermeasures

The threat models and countermeasures presented in this paper are based on the article *Reliable GPS-Based Timing for Power Systems: A Multi-Layered, Multi Receiver*[1]. The only exception is our own proposed countermeasure described in chapter3.

2.3.1 Threats

2.3.1.1 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 do not have a signal, you are probably being jammed.

2.3.1.2 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 [14]. 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 military-grade 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)[3] 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 receiver by adjusting the power of the signal below the victim receiver's noise floor and then gradually raise it until it surpasses the authentic signal's strength. Once the victim's receiver locks on, the attacker has gained full control.

2.3.1.3 Data-level Spoofing

In data-level spoofing the contents (data) of the GPS signal is manipulated. GPS signals include 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 [14].

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

2.3.1.5 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.

2.3.2 Countermeasures

2.3.2.1 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.[1]

2.3.2.2 Checking solved position against known position

By checking the position solution against the known position of the receiver, both receiver errors and a replay spoofing attack can be detected. It does

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

however fall short when more sophisticated techniques like Data and Signal-level spoofing are used. These types of attacks when done properly (unless it is done with intention) will not alter the solved position. It is important to note that this is 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 three receivers:

- **None of the receivers are spoofed:** Each receiver's 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 for an 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 [1].

2.3.2.3 Checking time solutions against receiver clock statistics

By comparing statistics created by monitoring the receiver's clock with the time solution, one can detect spoofing, as well as malfunctions. 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 receiver's clock which can be described as both unpredictable and stochastic, it should only be used along side other countermeasures [1].

2.3.2.4 Cross-checking navigation data among receivers

When under a data-level spoofing attack the navigation data is modified. By comparing one GPS receiver's navigation data with another, both data-level spoofing and malfunctions may be detected. This countermeasure can also prove useful during jamming attacks. The jammed receiver could use the data from other receivers in the event that it is unable to correctly decode navigation data, but still able to track satellites. This may enable the receiver to continue operation during an attack. [1]

2.3.2.5 Comparing navigation data and reverse-calculated satellite positions

The PMU's GPS receiver is never moved and its position is always 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 are correct, one can effectively detect replay spoofing and malfunctions. It is also worth noting that this countermeasure increases the difficulty of both signal and data-level spoofing, because it narrows down the possible valid (seemingly) spoofing signals. [1]

2.3.2.6 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)[1] 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, 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 receiver's ability to output baseband samples and the 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 [1].

2.3.2.7 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's GPS receiver's 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. Replay spoofing attacks will also fail since the PIA vector tracking depends on the knowledge of the GPS receiver's true position. In the event of such an attack the result would be that the vector tracking will fail to function [1].

2.3.2.8 Multi-receiver tracking loops

Building on the idea from *PIA Tracking loops* 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)[1] showed that acquisition and tracking performance under low signal-to-noise ratio conditions were improved under multi-receiver signal accumulation. Multi-receiver phased arrays also improved the robustness against both jamming (2.3.1.1) and spoofing attacks (2.3.1.2,2.3.1.3) by *"Forming beams to satellites and steering nulls in the direction of attacking transmitters"* ([1], p.41). In addition to the increased robustness, it increases the ability to detect malfunction. A faulty receiver will usually not be consistent with other correctly functioning receivers. As with the countermeasure based on cross-correlating P(Y) code, 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. [1]

2.3.3 Summary

The table (2.1) shows the different threat models and the effect of the countermeasures discussed.

Table 2.1: 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 (2.3.2.1)	N	X	X	X	N
Check pos. solution (2.3.2.2)	N	Y	Y	Y	Y
Check time solutions (2.3.2.3)	N	X	X	X	X
Checking nav. data (2.3.2.4)	X	N	Y	N	Y
Reverse calculated sat. pos. (2.3.2.5)	N	X	X	Y	Y
Cross-correlating P(Y) (2.3.2.6)	N	Y	Y	N	N
PIA TL (2.3.2.7)	Y	N	N	Y	N
Multi-receiver TL (2.3.2.8)	Y	X	X	X	X

Table 2.2: Legend for table (2.1)

Y	Effective	N	Ineffective	X	Auxiliary

[4]Jamming (2.3.1.1)
[5]Signal-level Spoofing (2.3.1.2)
[6]Data-level Spoofing (2.3.1.3)
[7]Replay Spoofing (2.3.1.4)
[8]Malfunctions (2.3.1.5)

2.4 Benchmark clock spoofing attack

In 2012 a team from The University of Texas at Austin published a paper [3] describing *The Civil GPS Spoofer*. It was a GPS spoofer and the first of its kind. The Cilvil GPS spoofer was able to precisely align both the C/A codes and navigation data in a counterfeit signal with those of an authentic GPS signal. The alignment capability allowed for sophisticated spoofing attacks that were hard to detect. The spoofer was implemented in software-defined radio with a digital signal processor. The user could control the fake GPS signal specifying both navigation and timing. By tracking and acquiring GPS signals and calculating navigation data, it could produce up to 14 spoofed L1 C/A signals by performing real-time prediction of the pseudo random C/A codes. Key features of the time spoofing attack:

- Seamless takeover: GPS spoofer tricks target into locking on to a replica of the authentic GPS signal, without loss of GPS lock or change in reported position.
- Once the target is locked on to spoofer signal, the attacker manipulates the apparent time of the target GPS clock.
- Low quality internal clocks enable aggressive manipulation of GPS clock timing, rapidly resulting in large apparent timing errors.

The sophistication of this spoofing attack is in part due to it is technical complexity, but more importantly due to the demonstrated result: The attacked GPS controlled clock had no way to detect that it was being fed false GPS signals. The study was targeted to the use of timing to phasor measurement units used in the monitoring and control of power grid transmission lines. Similar aggressive manipulation of timing may potentially cause the malicious shutdown of transmission lines within a few minutes after the onset of the attack. The team tested the spoofer against a wide variety of applications using the civilian receivers, and they were always successful [3]. When designing spoof proof GPS systems the Civil GPS spoofer can be considered as a benchmark reference threat.

Chapter 3

Our Proposal: Spoof proof atomic clock controller

We propose to construct and use what we call a *Spoof proof atomic clock controller*. The spoof proof atomic clock controller consists of custom software running on commodity computer hardware, connected to at *least* two GPS receivers and an atomic clock. With the spoof proof atomic clock controller you can perform:

- Spoofing detection.
  - Having a stable clock that can be trusted makes it possible to verify the GPS timing solution.
  - Using two or more GPS receivers makes it possible to detect whether or not the signals originate from satellites in orbit or from a spoofer on the ground. See subsection 2.3.2.2 for more about this idea.
- Mitigation.
  - A stable clock will provide accurate timing for an extended time even in the absence of valid GPS correction.

3.1 Filtering and steering

The spoof proof atomic clock controller uses what we call *filters* for detection. The filters are algorithms used to detect a spoofing attack or just general abnormalities that might affect applications relying on GPS time. The filters can be divided into two classes or types. They are either GPS based or clock model based. The GPS filters analyzes collected GPS data called NMEA data [15] and verifies their validity. This can be achieved by comparing the collected GPS data with known GPS data for the receivers. The clock model based filters are more sophisticated. By analyzing the behavior of the atomic clock used by the spoof proof atomic clock controller, a model can be built and used as a reference. This approach was actually suggested in subsection 2.3.2.3, but in that instance using the receiver's internal clock. This approach was considered auxiliary because of the receiver's clock's poor performance. This is in contrast to our proposal where we have a stable and accurate atomic clock. This makes it possible to evaluate the GPS signal used to discipline the atomic clock. The model can also be used for mitigation during an attack. Once an attack has been detected the atomic clock should no longer be disciplined by the signal. The clock should be usable for a while without disciplining, but with the clock model used for steering it will provide accurate timing for a longer time.

Chapter 4

Hardware

4.1 Atomic Clock

We decided to use the Symmetricom SA.45 as the atomic clock. This is an atomic clock measuring only 16cc with 1 pulse per second (PPS) output and 1 PPS input for disciplining. The SA.45's strength is its low power consumption (less than 120mW) and low price. 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 atomic clock, issue commands and collect data is paramount for the feasibility of our proposal. It is 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 like phase difference and steer values and can configured by wire to change modes of disciplining. For more about clock performance, review the SA.45s' user guide [16] and data sheet [17].

Figure 4.1: Symmetricom SA.45s CSAC. Courtesy Symmetricom.

4.2 Atomic clock controller platform

We chose to cast the Raspberry Pi 3 Model B (RASPI3) in the role as the host running the atomic clock controller 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 [18]. As with the Symmetricom SA.45, the RASPI3 is very affordable. The RASPI3 retailed at about 35 USD when this report was written. We also propose to use Raspbian [19], a Debian derived flavor of Linux optimized for the Raspberry Pi, as the operating system.

4.3 GPS receiver

We chose to use at least two GPS receivers. Both of the receivers should collect data and feed it to the atomic clock controller, but one of the receivers should also double as a 1 PPS disciplining source for the atomic clock. Considering the need for a stable 1 PPS source, we propose to use the u-blox NEO-M8T. This is a relativity affordable GPS receiver with a temperature compensated crystal oscillator (TCXO), 3 concurrent GPS reception and an external antenna [20]. In the current implementation of the atomic clock controller, only NMEA data is collected from the GPS receivers (see section 6.2.6 for more about NMEA data). However in the future it might be beneficial to collect and process raw data[1] from the receivers as well. Since most GPS receivers today follow the NMEA standard (to some extent) and raw data currently is not required, common and popular receivers like the u-blox NEO series should be more than sufficient for use in our implementation.

4.3.1 GPS receiver configuration

According to the u-blox NEO-M8T's manual [21], the device includes a feature called *Fixed Position*. This is a feature that must be enabled in order to put the device in *Time Mode*. This feature makes the device solve time with higher certainty even with fewer available satellites. The *Time mode* was not used, relied on or accounted for, in our solution.

[1] The ublox M8T is capable of outputting RAW data. This is for example used by RTKLIB (http://www.rtklib.com/) instead of NMEA

Chapter 5

Atomic clock controller architecture

When discussing how to implement the spoof proof atomic clock controller's software the following features and requirements were identified:

- Speed and efficiency is important in order to detect attacks as early as possible.
- Extensive logging capabilities for forensics.
- Easy and fast access to all gathered data.
- Should support as many GPS receivers as possible.
- Easy interfacing. Should be possible to operate over Telnet/SSH.
- Easy to configure.
- Graphical user interface is not a requirement

5.1 Client/Server model

In order to connect many GPS receivers to the atomic clock controller, we decided to implement a client/server model. The atomic clock controller serves the same purpose as before, but doubles as a server. The data transmitted, will still be the same, the only difference is the interface and media. A network interface will be used instead of USB and the media can be whatever is available since TCP/IP is designed to be hardware independent. This opens up the possibility for using:

- Twisted pair
- Fiber optics
- WiFi
- 4G (cellular)

Because reaction time is a concern, high latency media is not recommended [1]. The GPS receivers that we have chosen to use do not have a network interface, we instead use Raspberry PIs. A GPS receiver is connected to a Raspberry PI thus giving it a network interface. Considering how cheap single board computers have become, the price of a Raspberry PI can be justified even just to network enable a GPS receiver. The latest model of the Raspberry Pi has even got a built in wireless network interface. See 4.2 for more about the Raspberry Pi 3. The combination of the GPS receiver and the Raspberry PI are in this report abstractly seen as one device. We call this device a sensor. The sensor is responsible for reporting data to the Server and not much more. The sensor can be deployed using already existing infrastructure to communicate, thus making it easier to deploy than if each GPS receiver had to be cabled directly to the atomic clock controller. There is no denying that the implementation of a Server/Client model greatly increases the complexity of the atomic clock controller software, but it makes up for it by eliminating the need for long signal cables and amplifiers. We call this approach the *Sensor Server* model.

5.2 Architecture description

In order to connect the Sensors to the server, the atomic clock controller's software needs to be able to:

- Handle connections to clients.
- Update structures as clients' status changes (disconnects or gets kicked out).
- Receive data from clients

These are tasks that the atomic clock controller needs to be able to perform as well as controlling the clock and analyzing data. In order to implement the Server/Client model, the Server is implemented using the Linux Socket API.

[1] IP over Avian Carriers as described in RFC 1149 is highly discouraged

The API is based on BSD sockets and is available in in almost all Unix-like operating system ([22], p.610). A socket is a handle that can be passed by a program to the network API in order to use the network connection. The plan was briefly to use Glib [23], a library that provides building blocks for libraries and applications written in C, instead of using low-level sockets. This idea was scrapped because I felt that it would be overkill for this implementation.

Having decided *how* the clients should connect to the server, the next challenge becomes how the server should communicate with the clients and vice versa. We decided to use what is called *blocking I/O* and *fork()*. This is a very common approach, but not the fastest ([24], p. 188). It is however quite easy to implement. Blocking I/O means that read operations on the socket blocks the main thread in the process until data has been received. This is not as dramatic as it might seem: 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). Fork() is a system call used to create a new process. The new process is called a child and the process that called the fork() system call is called the parent. The child process is a duplicate of the parent process. The alternative to using fork() is to create a thread. The creation of threads are typically less expensive in terms of CPU cycles than the creation of processes, but presents their own challenges. Processes always have their own virtual address space as opposed to threads who share their address space with the other threads within the process. This makes programming with threads more complex. The result of a thread crashing may have a more severe impact on the other threads within the same process since they all shared the same data. The new process is used by the server to handle the newly created connection. This means that for every connected client, there is a process created. This of course means that this solution does not scale awfully well. This is however not a big problem for us since the time taken for a sensor to connect to the server really does not matter. Once all sensors in a setup have connected, they should not disconnect unless taken down for maintenance or replacement.

Using processes presented us with a new challenge, interprocess communication. A process is not aware of its neighbor. For all it knows, it is the only process running. Every time a client connects, a new process is born to take care of the communication with the new client. Since every process has its own virtual address space, the processes are isolated from each other. This posed a challenge because we wanted the atomic clock controller to collect data from sensors for processing. Inter process communication (IPC) is nothing new, and one way to accomplish it, is to use shared memory segments that are accessible for all the processes. The processes store the data they have received from their respective clients in the shared memory segment. That way it is easy for one process handling a sensor to check up on data received from another sensor. This kind of functionality was seen as valuable, but is not utilized to the extent it was envisioned. If more filters are added, the architecture would probably make more sense as the current ones do not really use the shared memory to cross check data.

Using shared memory is by no means the only way to implement interprocess communication. Another approach to inter process communication that was considered, was using a `pipe`. The pipe can be seen as a unidirectional channel where data written to end of the pipe is buffered by the operating system until the data is read from the other end of the pipe. This would however mean that each process would not only have to listen to the client connected, but also to the pipes connecting them to the other processes. This would be the case with message passing and sockets as well. In contrast, the shared memory approach allowed for the server to act as a single unit even though many processes are at work. Allowing multiple processes to share the same memory segment is like asking for trouble. At some point the program will suffer from race conditions and unexpected behavior. The shared memory segments are therefor protected with semaphores.

The atomic clock controller does not have graphical user interface. Seeing how it was going to operate much like a service, the need for a graphical user interface was not prioritized. A user can interface with the system by logging on using telnet, and issue commands. In order to separate the intention of a connected client, be it to deliver GPS data or to send commands to system, roles were made. A client connected to the server may have two roles. It can choose to either be a sensor or a monitor. The sensor role is already explained. The Monitor role was added in order for a user of the system 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. The monitor role can also be used to interface with the server. See (6.7) for more.

5.2.1 Security

Implementing good security is not an easy task and should not be taken lightly. The goal when developing the Sensor server was to produce a rough prototype that could prove our concept. Time was therefor allocated towards producing features that would realize this goal instead of attempting to implement security mechanisms that would have been flawed anyway because of time constraints.

Chapter 6

Atomic clock controller implementation

6.1 The Sensor server

Figure 6.1 shows a simplified block diagram of the sensor server and its tasks. server tasks include:

- Handle connections to clients.
- Update structures as the clients' status' changes (disconnects or gets kicked out).
- Communication with the atomic clock and updating the atomic clock model.
- Sensor data analysis and filter updates.
- Raising alarms based on filter status.
- Controlling the atomic clock.

6.2 Implementation description

In the following section, the architecture and inner workings, data structures and key components of the Sensor server will be explained. The server core (6.2.1) consists of the source code in `sensor_server.c`. The Parser and Handler spans over textttsession.c and textttactions.c. The atomic clock model and the associated filters' source code can be found in `csac_filter.c`. The filter using GPS data is in `filters.c`. Figure 6.3 shows a simplified call graph for the Sensor server.

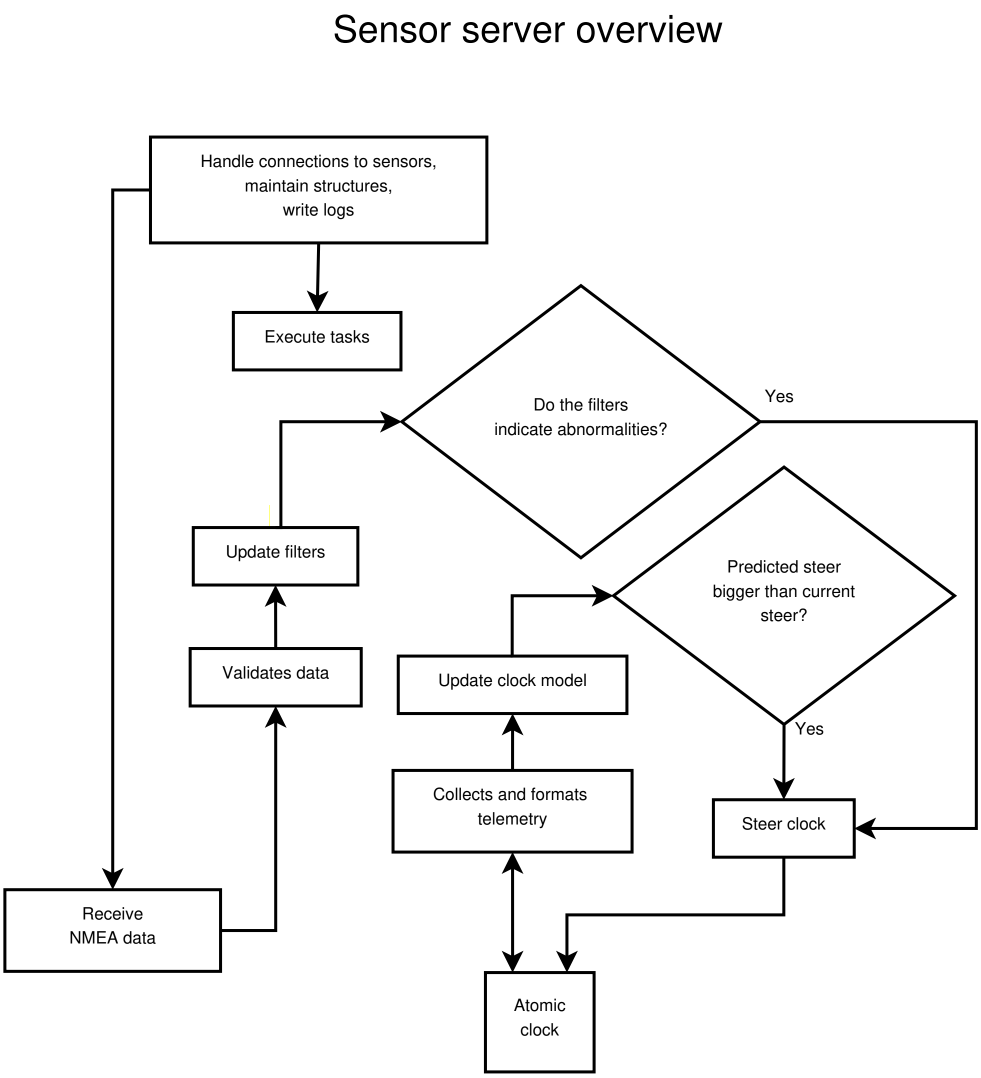


Figure 6.1: A block diagram of our proposed solution

6.2.1 Server core

The "Server core" is the main process and the parent of every process created during the life of the Sensor Server. It spawns the process maintaining the atomic clock model as well as new processes for every client that connects. Figure 6.2 is a block diagram of the Server core.

- The Server software takes one parameter to start, the port. If the parameters are missing or the parameters are illegal, a string containing usage information is printed and the program exits. Because the Server also is responsible for communication with the atomic clock, it needs the rights to access the serial port connected to the atomic clock [1].
- The configuration is initialized and loaded. If the configuration fails to load the Server prints and error message and exits.
- The `client_list`, a shared memory segment containing a linked list containing all the clients, is initialized. See section 6.2.9 and 6.2.10 for more about the client list shared memory segment and linked list implementation.
- Shared memory segments and semaphores are initialized and allocated. If the allocation fails, the Server prints an error message and exits. See section 6.2.8 for more about semaphores and shared memory.
- The process responsible for maintaining the atomic clock model and filters is forked out. If the fork fails the server prints an error message and exits. See section 6.2.2 for more about the atomic clock model.
- Handler for SIGINT (interrupt) SIGTERM (terminate) and SIGCHLD (child process is interrupted or terminated) are registered.
  - When a process exits, a SIGCHLD signal is sent to the parent. When a client disconnects from the server the process that handled the client exits. Ideally, the client disconnects from the server by sending a protocol compliant "disconnect request". This way the server can handle the disconnect in a controlled manner and update its client list when the client disconnects. However, this is not always the case and if the client disconnects abruptly and without a warning, the server still needs to handle it. Therefore,

[1] The easiest way to achieve this is to run the program as `sudo` or login as root. The best solution would probably be to add the user to a group with the right permissions. This is often the `dialout` group.

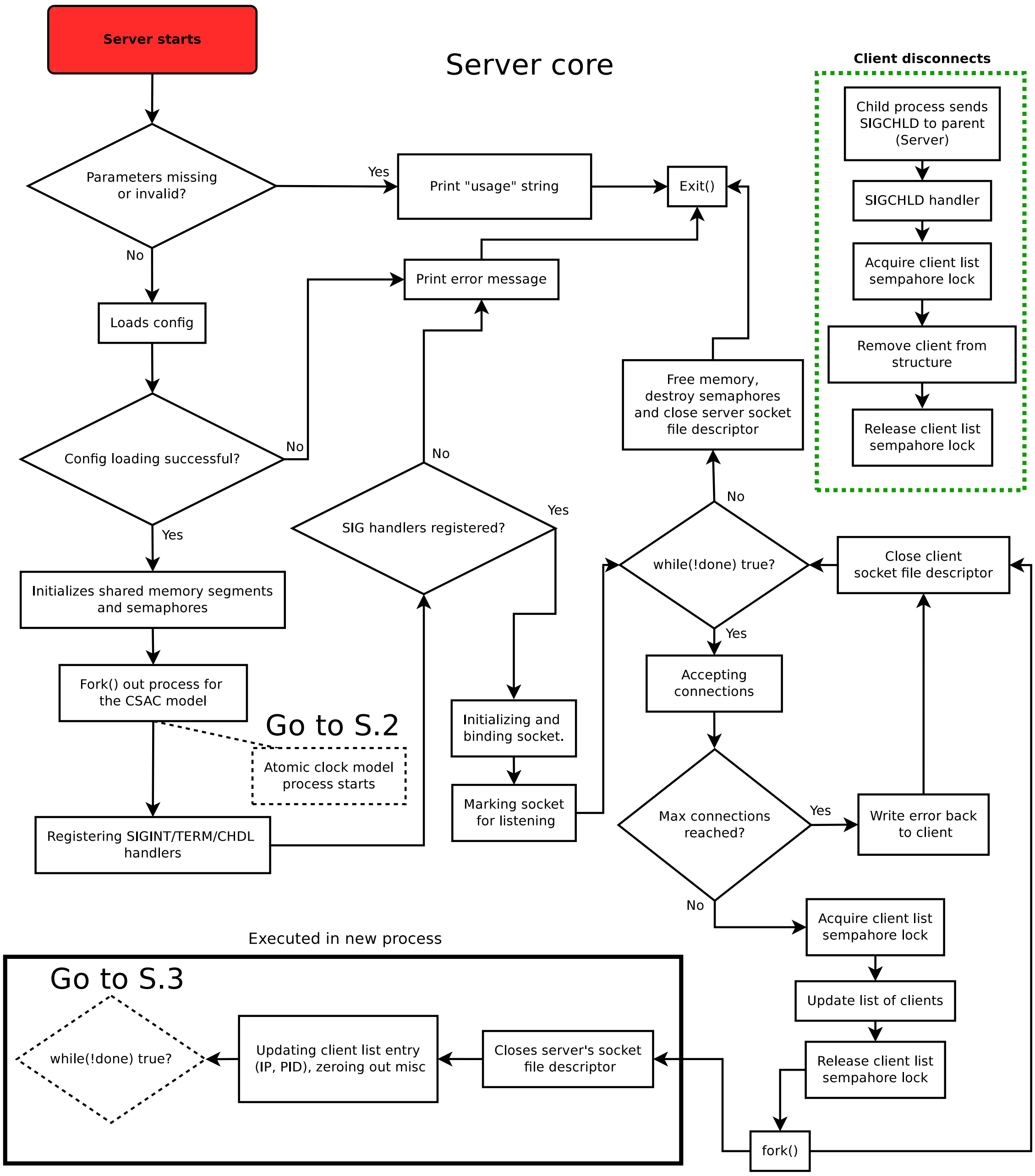


Figure 6.2: The block diagram shows an abstracted view of the Sensor Servers Core.

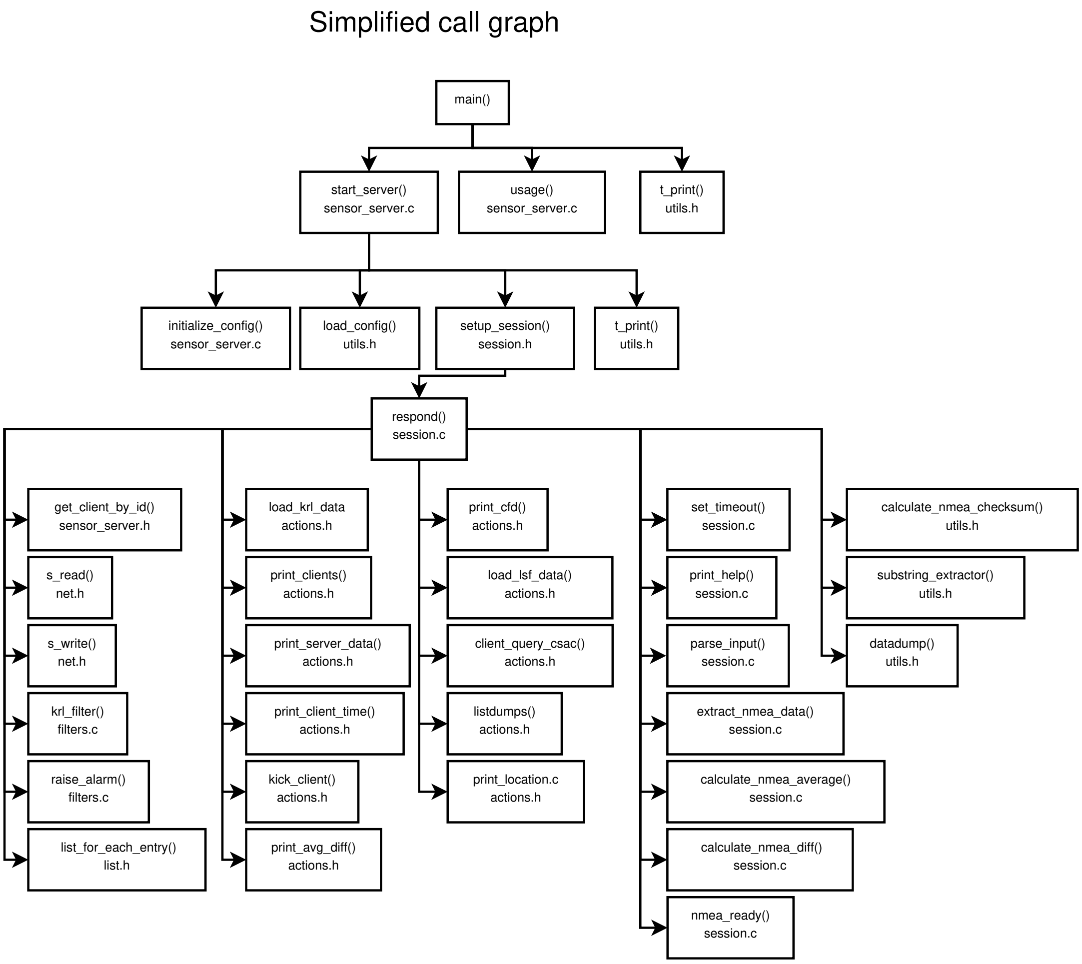


Figure 6.3: Simplified call graph for the Sensor Server.

when a SIGCHLD signal is received, the server iterates through its client list and finds the client whose PID matches the sender of the signal and removes it from the list. In figure 6.2.1 this routine is the part in the dotted box.

- The socket is initialized and marked for listening. See section 6.2.7 for more about sockets.
- A loop is entered. The loop breaks when `volatile sig_atomic_t done` equals 0. If the loop is broken, the semaphores are destroyed and all allocated memory is freed, the servers socket file descriptor is closed and the Server exits. See section 10 for more about variables used in synchronization mechanisms.
- When a client connects the server checks if the maximum number of clients has been reached. If this number has been reached an error message is written back to the client and the client's socket file descriptor is closed. If the maximum number of clients has not yet been reached, the client list is updated and a fork is performed. The parent process then resumes the loop.
- The process that just forked out for the new connection closes the server socket file descriptor it inherited from its parent.
- The new process then updates its client table entry and proceeds to the listening loop. This is covered in section 6.2.3. See section 6.5.7 for more about the client table entry data structure.

6.2.2 Clock model implementation

Conceptually, the atomic clock model and the filter that uses the model are two separate tasks. In practice however, they are intertwined as figure 6.4 suggests. See section 4.1 for more about the atomic clock and appendix B for the model itself.

- The configuration for the model is loaded. The configuration includes values defining the accepted limits for steering and phase difference. Optionally, it can also include state values from an "earlier" model [2] that can be used to initialize the model further, thus reducing the warm-up time. If the loading of the configuration fails, an error message is

[2]The functionality to restore an earlier model is not without restrictions. It is typically meant for a situation where the Server needs to be taken down for a couple hours and a new warm-up period is considered undesirable. This is better explained in section B

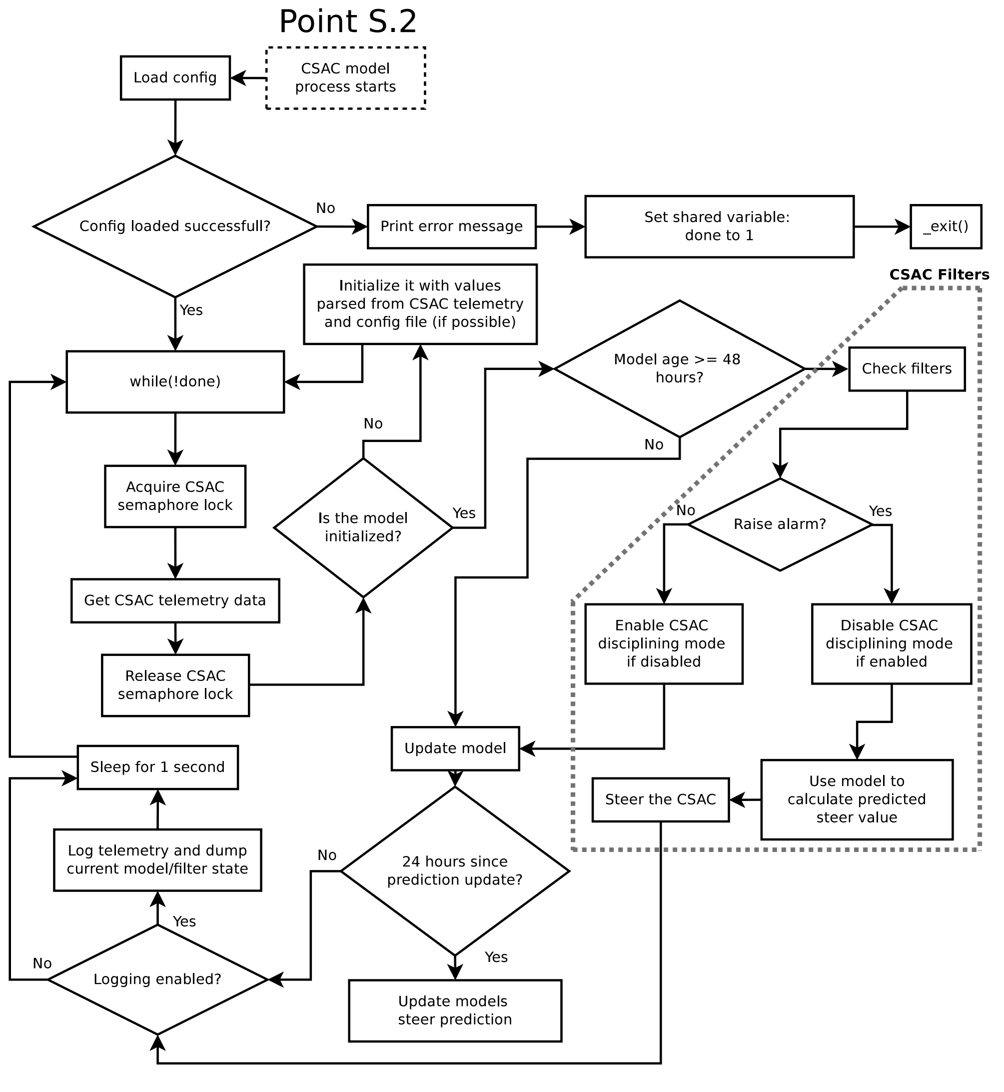


Figure 6.4: Block diagram showing the flow of execution in the atomic clock model

printed, the `volatile sig_atomic_t done` is set to 1 and the process exits. This also breaks the server core (6.2.1) loop. After all, the system needs the atomic clock to do the job it is intended for.

- A loop is entered. The loop breaks when `volatile sig_atomic_t done` equals 0.
- The atomic clock lock is acquired and once the telemetry is queried from the atomic clock it is released again.
- If the model is not initialized, telemetry data is used to initialize it. State values from earlier models will also be used if available in the earlier loaded configuration. See section 6.4 for more about communication with the atomic clock.
- A check is done to see if the model has been running for at least 48 hours. If it has, it is ready to be used as a reference for the filters. See section 6.2.5 for more about the atomic clock filters in particular.
- The telemetry received from the atomic clock is used to update the model.
- A check is done to see if 24 hours have passed since the prediction values were last calculated. The model keeps track of steer predictions for yesterday and today. By using these two points of data, an estimation of the predicted steer can be calculated.

6.2.3 Parser and Handler

The parser and handler are responsible for parsing data received from a client and making sure every request is protocol compliant. Once the goal of the request has been established, the request is executed. A function named `respond()` is called in the listening loop. Every time data is received from a client, the data is sent to the parser (`parse_input()`) to determine the validity of the request received. See subsection 6.2.6 for more information about the GPS data received from the Sensors. Figure 6.5 and 6.6 are block diagrams showing the execution flow for this part of the server. The protocol is explained in greater detail under section 6.3 and the parser is better explained under subsection 6.2.11.

- A loop is entered. The loop is broken when `volatile sig_atomic_t done` equals 0. This is the same variable that is used in the Server core 6.2.1 loop and for a good reason: If the server exits, its children should too.

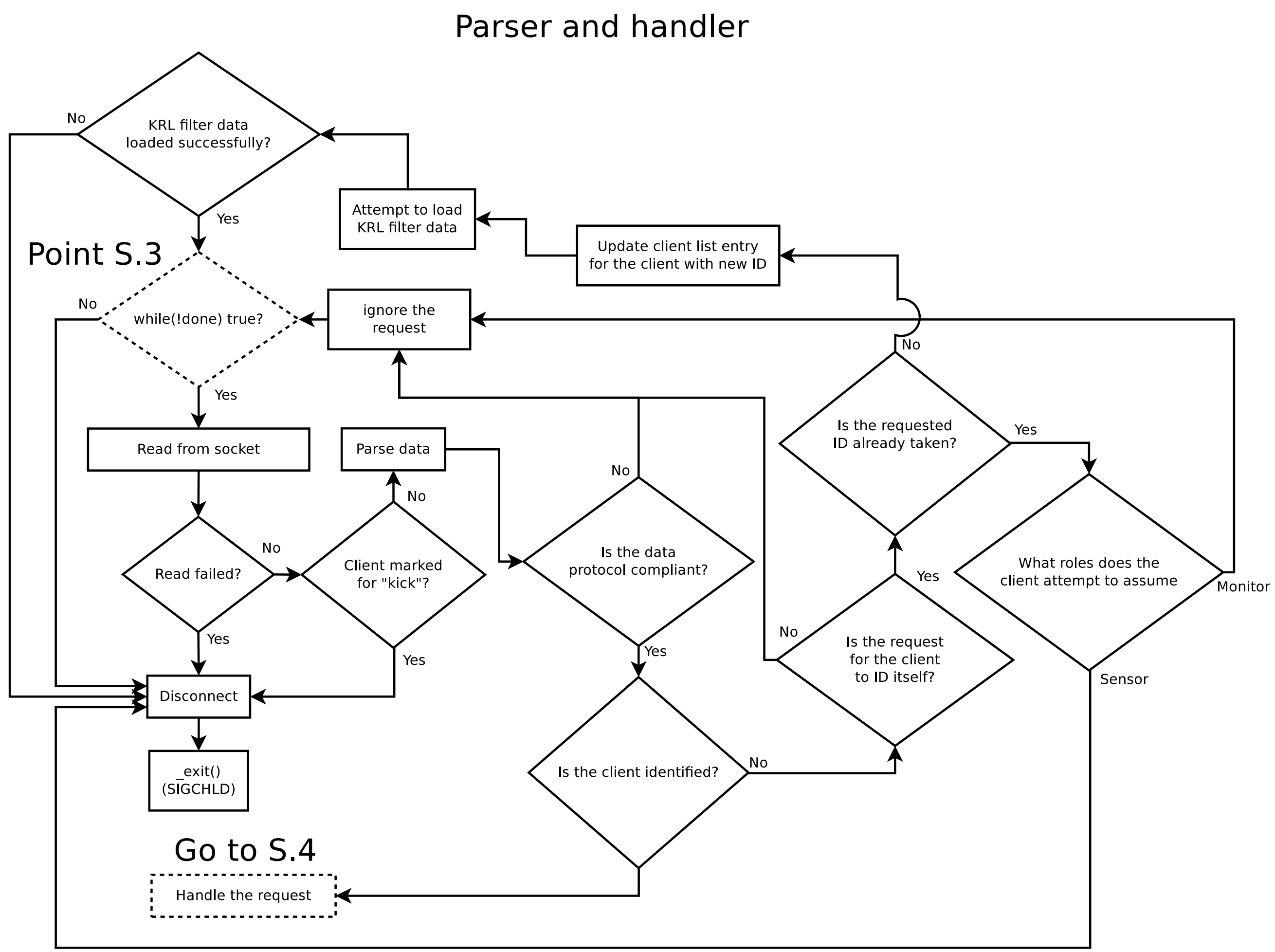


Figure 6.5: The block diagram shows an abstracted view of execution after a client has connected to the server and a `fork()` *has been performed.*

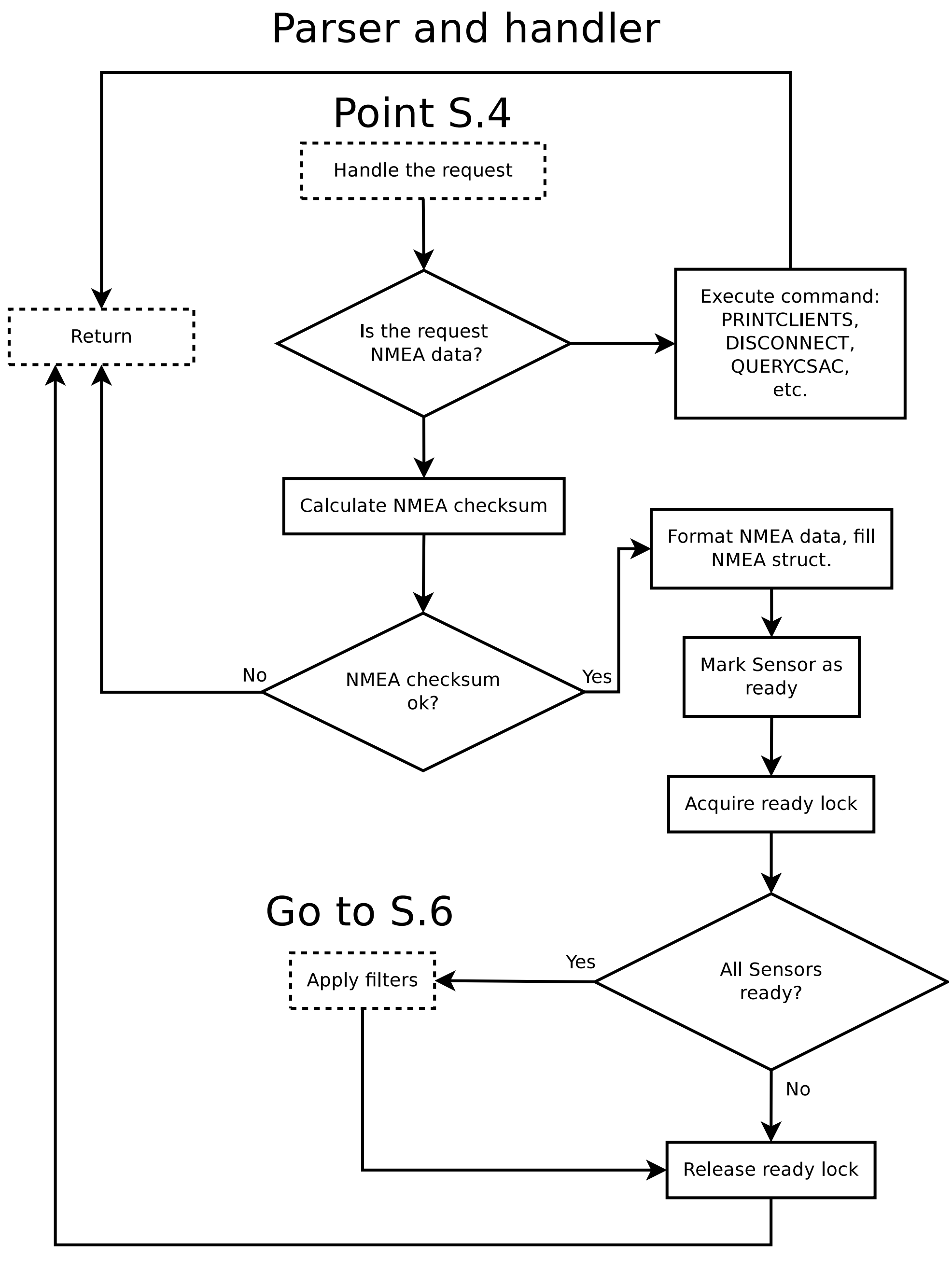


Figure 6.6: The block diagrams shows and abstracted view of the execution after data has been received from a client.

- If reading from the socket fails, the client gets disconnected. If it is successful, the client table entry for the connected client is checked to see if the client has been marked to be kicked. If so, the client is disconnected.

- The data received from the client is parsed and checked against the protocol. If the request received from the client does not match any of the supported requests or commands, the request is ignored and the loop is entered again. See section 6.3 for more information about the protocol and subsection 6.2.11 for more information about the parser.

- If the request received is valid, the clients ID is checked to determine whether or not it has identified itself. If the client has identified itself, the request gets carried out.

- If the request received is to handle NMEA data, the checksum for that NMEA data is calculated. If the checksum check fails, the data is discarded. If it succeeds, the data is copied into `nmea_container` struct (6.5.4) for easier handling.

- The client is marked as ready for processing and a ready check is commenced. The ready check is used to make sure that all the sensors have received NMEA data and are ready to have filters applied. Once the filters have been applied, the lock is released and normal execution resumes. The ready check is explained further under subsection 6.2.12.

- If the clients ID is 0, it is considered unidentified and any other requests but to identify itself, is ignored.

- If the client attempts to identify itself, the ID it attempts to use is checked against the rest of the connected clients. If the ID is already taken and the client attempts to assume the role of a Sensor, the client will get disconnected. If it attempts to assume the role of a Monitor, it will simply be notified that the chosen ID is taken and the request will be ignored. The reason why a Sensor would get disconnected and a monitor would not, is simple: When connecting the sensor to the server, it should not be any doubt whether or not it has been accepted. If something is wrong, it is better that a user of the system deals with the configuration at once rather than allowing the user to apply a faulty configuration that severely impacts the efficiency of the system. See section section 5.2 for more information about the Sensor and Monitor roles.

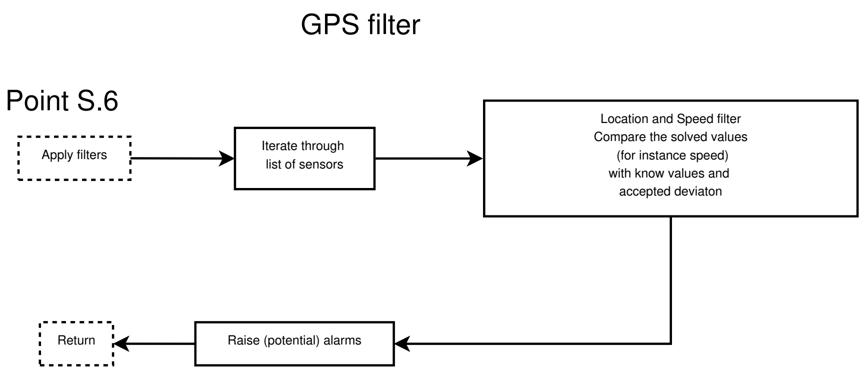


Figure 6.7: Block diagram show the order of execution in the GPS filter(s)

- If the attempted ID is available, the client table entry for the client is updated with the requested ID and the server will attempt to load data for the Location and speed filter data into the clients location and speed filter struct. The server will attempt to load this data from a file named `krl_data_sensor<ID>`. If the load fails, the client will be disconnected. It is after all useless without its reference position data. For more information about location and speed filter, see section 6.2.4.1.

6.2.4 GPS filter

There is currently only one filter using GPS data in the Sensor Server implementation to this data. This is the *Location and speed filter*. Figure 6.2.4 is a block diagram showing the filter as a part of the Sensor Server.

6.2.4.1 Location and speed filter

By configuring the Sensor Server with the location of the different Sensors, the server may trigger an action if a Sensor reports an abnormal solved position. The filter is triggered when either latitude, longitude, altitude or speed is higher or lower than the reference value minus or plus a deviation. Listing (1) shows an edited sample of code taken from `filters.c`. The code in the sample is part of the algorithm used to check whether or not the latitude part of the GPS receivers solved position is within "safe" (not spoofed) range.

Listing 1: Listing shows sample of code taken from `filters.c` line 79. The code is part of the algorithm that compares the known position of a Sensor with the Sensor resolved position. The code has been edited for clarity.

```
    if(latitude_current > latitude_reference + latitude_deviation) {
        moved = 1;
        lat_disturbed = HIGH;
    } else if(latitude_current < latitude_ref - latitude_dev) {
        moved = 1;
        latitude_disturbed = LOW;
    } else {
        latitude_disturbed = SAFE;
    }
```

When a Sensor connects to the Server and identifies itself, the Server will attempt to load a file containing the location of the Sensor as well as the accepted deviation values for the solved position. Listing 2 shows an example of such a file.

Listing 2: Location and speed filter configuration file

```
alt_ref: 123.8
lon_ref: 1102.1948
lat_ref: 5958.5448
speed_ref: 0
alt_dev: 10
lon_dev: 0.005
lat_dev: 0.005
speed_dev: 10
```

6.2.5 Clock model filters

Figure 6.4 shows both the atomic clock model and filter in one figure. As explained in section 6.2.2, the model of the atomic clock and the filters based upon, are closely related. See B for a description of the clock model.

- Once the model has been running for a configurable long time (48 hours was used during testing, see B for more about the model), it is ready to be used as a reference, filtering out abnormal data.

- The filters are checked. There are currently two filters implemented using the atomic clock model as reference:

  - *Phase jump filter*. This is a simple filter that compares the current phase reported by the atomic clock with a pre-configured limit. If the current phase is higher, the filter is triggered.
  - *Frequency correction filter*. This is a more sophisticated filter. It uses the atomic clock model to calculate a predicted steer value and compares it to the current steer value. If the current steer is too big, the filter is triggered.

- If any of the filters are triggered, the alarm is raised. The state of the alarm is expressed by an `int` that is either 0 or 1. If the alarm was not raised, the disciplining of the atomic clock is enabled if it was disabled, and the telemetry data is used to update the model.

- If the alarm is raised, the disciplining of the atomic clock is deactivated, thus protecting the atomic clock from being affected by the spoofing. The model is now used to calculate steer values that are applied to the atomic clock instead.

6.2.6 GPS data

The GPS receivers that we have chosen for our system and possibly all GPS receivers, follow the National Marine Electronics Association's 0183 standard [15]. NMEA data consists of sentences where the first word of the sentence, the type, defines how the sentence should be interpreted. The sentences used by the Sensor Server and Client, is the `GNRMC` for latitude, longitude and speed and `GNGGA` for altitude. See the u-blox manual [21] for more about the different sentences.

6.2.7 Client connection

Every time a client connects to the server, a process is forked for the new connection. Listing 3 is sample of code taken from the core of the Sensor Server, and can be explained like this:

- 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: A client has connected. The server creates a new process to handle the new connection. The new process is created using `fork()`.

- Line 13: Both the new process and its parent enters the "if statements" regarding its process identification (PID). The parent does not match the criteria of the "ifs" and returns to the top of the loop. The child on the other hand, matches the criteria for the "if sentence" at line 15.

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

Listing 3: Sample of code taken from `sensor_server.c`(D.2, line 494). The sample has been edited for clarity purposes.

```
    listen(server_sockfd,SOMAXCONN);
    int session_fd = 0;
    while (!done) {
        session_fd = accept(server_sockfd,0,0);
        if (session_fd==-1) {
            if (errno==EINTR) continue;
            t_print(ERROR_CONNECTION_ACCEPT,errno);
        }
        if(number_of_clients == max_clients) {
            close(session_fd);
        } else {
            pid_t pid=fork();
            if (pid==-1) {
                printf(ERROR_FAILED_FORK, errno);
            } else if (pid==0) {
                close(server_sockfd);
                setup_session(session_fd, new_client);
                close(session_fd);
                _exit(0);
            } else {
                close(session_fd);
            }
        }
    }
```

6.2.8 Shared memory & Semaphores

The architecture uses several shared memory segments. 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 4 shows a code sample taken from `sensor_server.h` where the shared memory segments are declared.

Listing 4: Sample of code from `sensor_server.h`(D.2, line 44) where shared memory segments are declared.

```
    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;
```

These shared memory segments have different usage. The `client_list` points to a shared segment allocated for storing a list of all connected clients. `s_data` contains information about the server, `s_synch` contains semaphores used to lock down shared resources, s_conf is the servers configuration and cfd is the data and state for the filters based on the clock model. 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 8). 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 5 shows an example of MMAP being used to map an anonymous, shared map for the client list.

Listing 5: Listing shows the use of MMAP to create an anonymous map of memory to be used as a shared memory segment. Sample of code taken from sensor_server.c(appendix D.2, line 360)

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

6.2.9 Client list memory segment

Every time a client connects, it is given a piece of shared memory where it can store its data. This piece of shared memory is used as a node in a linked list. The linked list structure, stored in the shared memory segment is available to all the processes spawned by the server. This segment is static in size and is allocated once and is never changed during the whole life of the Server. The size of the segment is determined by the maximum number of allowed clients, a configurable value read from the configuration file every time the server is started. Read more about why the client list shared segment is static in size, under subsection 8.2.1.

6.2.9.1 Using the client list memory segment

As previously explained, the list of clients are stored in a shared memory segment using a linked list structure. In order to keep tabs on what part of the shared memory segment is in use, a simple map of the memory is used (6.8):

The map is an array containing pointers to the client list segment. These pointers are offset by the size of the client_table_struct and maps 1:1 to what can be thought of as individual *pieces* of memory in the client list

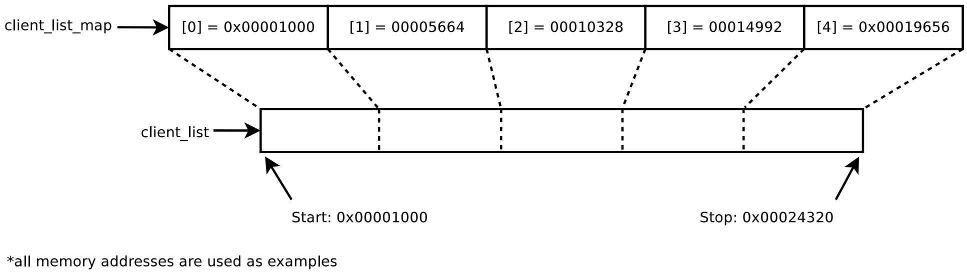


Figure 6.8: Figure is showing the relation between the shared memory segment containing the client list and the memory map

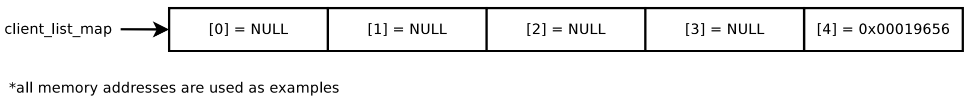


Figure 6.9: Figure is showing the relation between the shared memory segment containing the client list and the memory map after some of the pieces of shared memory has been given out.

shared memory segment. A free piece of memory is in the map represented by a pointer that is not NULL. Figure 6.9 is an illustration showing the state of the map when four of the first pieces have been given out. Listing 6 shows the function that iterates through the map to find a free piece.

Listing 6: Sample of code showing the `get_mem_piece()` function. This function is used to find a free piece of memory in the client list shared memory segment. The code is taken from line 200 in `sensor_server.c` (appendix D.2

```
static struct client_table_entry* get_mem_piece()
{
    int i;
    for(i = 1; i < s_conf->max_clients; i++){
        if(client_list_map[i] != NULL){
            struct client_table_entry *tmp = client_list_map[i];
            client_list_map[i] = NULL;
            return tmp;
        }
        i++;
    }
    return NULL;
}
```

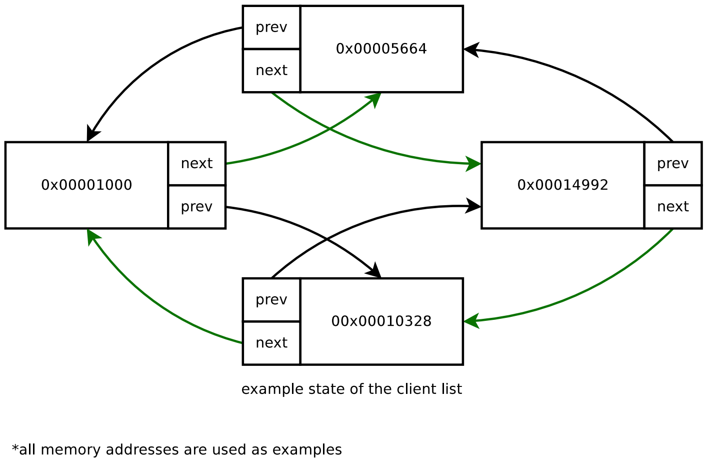


Figure 6.10: Block diagram show an example of linked list state

6.2.10 Client 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 solid, well tested and had many useful functions, I decided to use it. The modified header file containing all the code, is GPL licensed. Figure (6.10) shows pieces of the shared memory segment linked together in a linked list structure.

6.2.10.1 Alternative to linked list

The initial reason for using a linked list to organize the connected clients, was to reduce the time spent iterating over a space of memory. With a linked list it would not matter where the different pieces of memory were located, a pointer to next piece would always be readily available anyway.

6.2.10.2 Semaphores

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 are used to lock the segments during read and write operations.

Listing 7: Function for removing disconnected clients from list of clients

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

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

Listing 7 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 (7) a client has been disconnected from the server and the list of connected clients is 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 than 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 to a specific shared memory segment. All functions in the sensor server use semaphores when dealing with shared memory segments in order to avoid deadlock and race conditions.

6.2.11 Client input parser

The parser used by the Sensor Server is a simple function. As explained in section 6.2.3, every time a client sends data to the Server, the data is sent though the parser in order to detect the purpose of the request. The parser uses the protocol and compares it with the input and looks for a match. Listing 8 shows a sample of code taken from the parser.

Listing 8: Part of the parser comparing the input data to the IDENTIFY command specified by the protocol. Sample code is taken from `session.c`*(appendix D.2 line 164*

```
1   static int parse_input(struct client_table_entry *cte){
2     char *incoming = cte->transmission.iobuffer;
3   ...
4     else if(strstr((char*)incoming, PROTOCOL_IDENTIFY ) == (incoming)) {
5         int length = (strlen(incoming) - strlen(PROTOCOL_IDENTIFY) );
6         memcpy(cte->cm.parameter,
7           (incoming)+(strlen(PROTOCOL_IDENTIFY)*(sizeof(char))),length);
8         cte->cm.code = CODE_IDENTIFY;
9     }
```

The listing (8) shows how the parser attempts to find a match between the input buffer and the protocol defined IDENTIFY command. On line 6 it attempts to copy any potential parameter. The parser does not care if the parameter is missing, but it will attempt to extract it from the input buffer. If no matches are found, the function returns 0, letting the calling function (`respond()`) know that the input from the client was invalid or illegal. Comparing strings is hard work and a relatively CPU intensive task. This is why the parser sets a *command code* (6.5.3) as seen on the last line. The command code is an integer defined in the protocol and the is a command code associated with every command. The code is used in the listening loop by `respond` to determine what action to perform in case a valid request. Comparing integers are "cheaper" than comparing strings, and since the string comparison job is already done once, there is no need to do it again.

6.2.12 Ready check algorithm

Every time NMEA data is received and validated, the process that received the data will initiate a ready check. This is done by setting a flag in the client list entry called ready to 1, and calling the `nmea_ready()` function. The `nmea_ready()` function locks down the client linked list and iterates through it checking if the other processes are ready too. If all the other processes have received valid NMEA data from their respective client, the GPS filter is applied. If they are not ready, the lock is release and the process carries on. This means that the last process to receive NMEA data gets the job of applying the filters [3].

[3] Since NMEA data is generated every second by the GPS receivers, it might be more appropriate to say that the scheduler decides which process executes last. This of course also depends on the distance between the Sensor and the Client

6.3 Sensor Server Protocol

The Sensor Server protocol is quite simply a header file containing every keyword or command that the Sensor Server recognizes. The thought behind it was that a client application (or script) could import the protocol header file and easily use and recognize commands used between the Server and a Client. Some constraints are also defined in the protocol header file, like maximum parameter size, maximum and minimum command size. The *command codes* used by the parser are also defined in the protocol header file. See subsection 6.2.11 for more about the use of protocol.

6.4 Atomic clock Communication

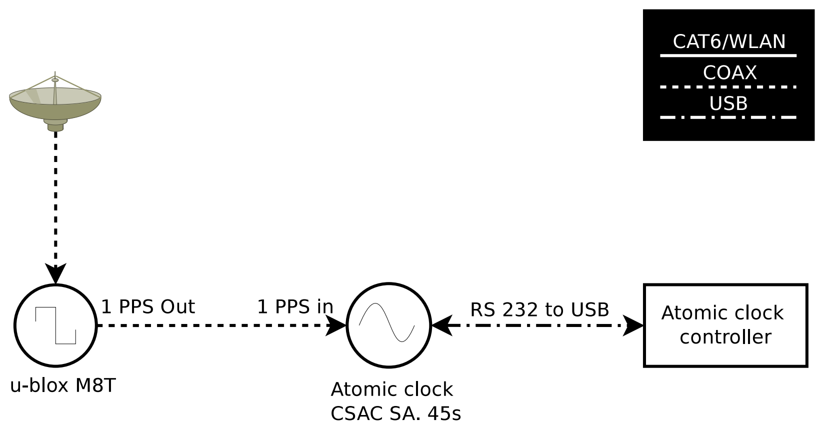


Figure 6.11: Block diagram showing the atomic clock connected to a PC

The SA.45 atomic clock 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 atomic clock. Information may be queried by sending commands to the atomic clock. These commands are explained in table 6.1.

The Sensor Server communicates with the atomic clock by invoking a script called `query_csac.py`. This script can be found in the appendix (E). The script takes an argument which it sends to the atomic clock over serial, and prints the respond to the shell which the Sensor Server grabs. In this document the word *telemetry* gets mentioned quite often in context with the atomic clock. Telemetry can be obtained by querying the atomic clock. The telemetry is a string containing a plethora of information, but we are mainly interested in the following values:

Table 6.1: Commands for the SA.45 atomic clock

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 atomic clock 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: [16]

- Phase, the difference between the atomic clock and the external signal signal at 1PPS in.
- DiscOK, the discipline status.
- Steer, the frequency adjustment.

Listing 6.1 show an example of telemetry received from the atomic clock. The fourth value from the right, is the `DiscOK`, the fifth is the `phase` and the seventh is the `steer`.

Listing 6.1: Example of a telemetry string received from the atomic clock

```
0,0x0000,1209CS00909,0x0010
    ,4381,0.86,1.573,17.62,0.996,28.26,-24,---,-1,1,1268126502,586969,1.0
```

The SA.45 uses a high-resolution phase meter to improve synchronization and to calibrate the frequency of the atomic clock. The phase meter measures the difference in time between the internal 1 PPS signal and the external reference, in our case a GPS receiver. This difference is the *phase* value. The atomic clock uses the phase value and steering algorithms to adjust the frequency of atomic clock's physics package thus simultaneously steering both the phase and frequency towards the external reference. This is called *disciplining* and is how the *steer* value is computed. The last value, *DiscOK* is simply the status of the 1 PPS disciplining routine [16].

6.5 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. Some crucial and often used structs is explained here.

6.5.1 server_data

Listing 9: Sample of code taken from `sensor_server_common.h` *(appendix D.2, line 107) showing the* `server_data` *struct.*

```
    struct server_data {
        int number_of_clients;
        int number_of_sensors;
        time_t started;
        pid_t pid;
        char version[4];
```

The `server_data` struct as shown in listing 9, contains information about the server. Some of the information like the PID, version and when the server was started, is just useful information about the server itself. The number of clients and sensors on the other hand are used to make sure that the server does not allow more connections than it can handle.

6.5.2 server_synchro

Listing 10: Sample of code taken from `sensor_server_common.h` *(appendix D.2, line 116) showing the* `server_synchro` *struct*

```
    struct server_synchro {
        sem_t ready_sem;
        sem_t csac_sem;
        sem_t client_list_sem;
        volatile sig_atomic_t done;
    };
```

The `server_synchro` as shown in listing 10 contains "flags" used by synchronization mechanisms (6.2.8) that the server and its children use to protect access to shared resources. The `csac_sem` is used to control serial access to the atomic clock, making sure that only one request is sent to atomic clock at a time. The `client_list_sem` is used by functions manipulating and reading the client list structure, and the `ready_sem` is used by ready check algorithm. See subsection 6.2.12 for more about this.

6.5.3 command_code

Listing 11: Sample of code taken from `sensor_server_common.h` *(appendix D.2, line 34) showing the* `command_code` *struct*

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

The `command_code` struct as shown in listing 11, is used by the parser to more efficiently convey protocol compliant requests and related parameters. See section 6.2.11 for more about the parser.

6.5.4 nmea_container

Listing 12: Sample of code taken from `nmea.h` *(appendix D.2, line 20) showing the* `nmea_container` *struct*

```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;
    };
```

The `nmea_container` struct as shown in listing 12 is used by the handler (see section 6.2.3) which dissects the NMEA strings and validates them before it fills the respective members of the structs. The purpose of it is just to make it easier for the other parts of the Server to use the NMEA data.

6.5.5 list_head

Listing 13: Sample of code taken from `list.h` (appendix D.2, line 70) showing the `list_head` struct

```
struct list_head {
  struct list_head *next, *prev;
};
```

The `list_head` struct is shown in listing 13. The fields of the struct are pretty self explanatory. There is a pointer to the previous node and one to the next. One of the members of the `client_table_list` is a struct of type `list_head`. This is what makes it possible to traverse the list.

6.5.6 transmission_s

The `transmission_s` struct as show in listing 14 is used by the child process forked out by the Server in the server core to communicate with the client. The struct has two members, a file descriptor for the socket and a buffer to store incoming data.

Listing 14: Sample of code taken from `net.h`(appendix D.2) line 31

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

6.5.7 client_table_entry

The client_table_entry struct is what the name suggests, it is 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 15 shows the complete struct.

Listing 15: Sample of code taken from `sensor_server_common.h`*(appendix D.2, line 90) shows the* `client_table_entry` *struct*

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

Beginning from the top:

- `list_head list` is used to traverse the linked client list. See 6.2.10 for more about the linked list implementation and 6.5.5 for more about the struct.
- `transmission_s transmission` is used by the process to network I/O with the connected client.
- `timeval timeout` is defined by the `sys/time.h` and is used in the Sensor Server implementation to hold a value defining how long a connected client can stay connected without sending any commands to the Server. When the timeout value is exceeded, the client is disconnected. The values are:
  - 5 seconds for a Sensor.
  - 1000 seconds for a Monitor.
  - 10 seconds for an unidentified client.

  These values are defined in the `sensor_server.h` file.
- `command_code cm` is used by the parser to convey protocol compliant requests. See 6.5.3 for more.
- `nmea_container` is used to contain NMEA data in a more convenient way for the filters. See 6.5.4 for more about the struct, and 6.2.6 for more about NMEA.

- `filters fs` is a struct containing data for the GPS based filters. There is currently only one GPS based filter, the Location and speed filter. It is explained in subsection 6.2.4.

- `pid_t pid` is the process ID for the process forked out for the connected client.

- `time_t timestamp` is time stamp that is stamped every time the Sensor's filter data is processed.

- `client_id` and `client_type` is the client's ID and the client's type, either "SENSOR" or "MONITOR".

- `ready` is used to indicate that a Sensor has received NMEA data and is ready for filter processing.

- `marked_for_kick` is a flag. If it is 1, the client has been marked by a monitor indicating that it should be kicked. The next time the client sends data to the Server, the client will be disconnected.

- `char ip` is the connected client's IP address.

6.6 The Sensor Client

The sensor client software is a simple program written in C99 whose only task is to relay information read from the GPS 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 GPS 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 randomly but within legal limits.
  - Logging is disabled.
  - Maximum of server connection attempts are set to 10.

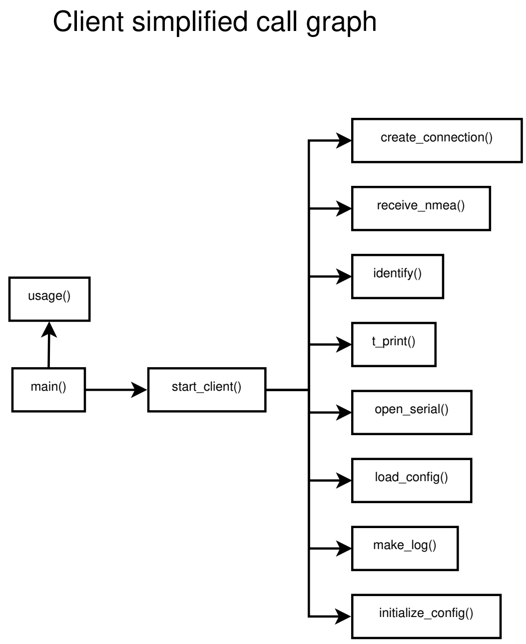


Figure 6.12: Simplified call graph for the Sensor Client.

  - Path to GPS 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 GPS 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 GPS receiver, scans for lines starting with either `$GNRMC` or `$GNGGA`. When both lines are found, the data is stored in a buffer.
- Sends the GPS data to the server according to protocol.
- Repeats.

Figure 6.12 shows a simplified call graph for the Client.

6.7 Interfacing

As mentioned earlier under section (5.2), a client can assume the role of both a monitor and a sensor. The sensor server does not differentiate between the two roles other than when it routinely checks the status of its filters. This means that one could connect to the server using the sensor client software as a monitor by configuring the sensor client software to use a negative integer as ID. At this point, this kind of functionality is not very useful as there is currently no way to change the ID of a client unless the client explicitly issues the command to do so, but it opens for the possibility to interface with the server. One way to interface with the server is to connect to it using Telnet:

```
user@machine:/$ telnet 10.1.0.46 10001
Trying 10.1.0.46...
Connected to 10.1.0.46.
Escape character is '^]'.
ID -3
OK!

>
```

It is also possible to assume the role of a sensor by connecting to the server via telnet. This can be used to debug or troubleshoot the sensor server by manually feeding it NMEA data. The only requirement is that the communication is sensor server protocol compliant:

```
user@machine:/$ telnet 10.1.0.46 10001
Trying 10.1.0.46...
Connected to 10.1.0.46.
Escape character is '^]'.
ID 2
OK!

NMEA<GNRMC part of NMEA><GNGGA part of NMEA>
OK!
```

Another possibility is of course to write scripts that communicates with the server. An example script can be found in the appendix (E).

Table 6.2: 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.
LOADLSFDATA	LLSFD	ID	Load reference location data into Sensor <ID>
PRINTCFD	PFD	NONE	Prints CSAC filter data

Chapter 7

GPS manipulation tests

In this chapter we present the tests that were conducted. Ideally these tests would have been performed by using a GPS spoofer like the "Civil GPS spoofer" that was introduced in subsection 2.4. Unfortunately, or perhaps not, this kind of hardware is hard to come by and spoofing GPS signals is not legal. What we did instead was manipulating the time solution of the GPS receivers by moving the antennas, thus naively simulating a spoofing attack. By performing these tests we where able to not only test the sensor server architecture part of the atomic clock controller, but also the filters and the clock model. We also present data gathered during an unplanned GPS disturbance that occurred in between our planned tests.

7.1 Logging filter and model data

In order to create an accurate clock model of the atomic clock, it was necessary to log data from the atomic clock while it was running in a disciplined mode. In the disciplined mode the atomic clock corrects its frequency based on either a one pulse per second (PPS) signal or a 10 MHz signal as discussed earlier under section (6.4). A similar approach was used in order to collect GPS data. Data from the two u-blox NEO-M8T receivers was gathered over the same time period as the data gathered from the atomic clock. 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 atomic clock. It also provided valuable data that could be used to tune the detection filters. The data gathering was done by simple Python scripts (appendix C.1 and C.2) running on a computer connected to the GPS receivers and the atomic clock. Figure (E.1) shows a block diagram of the setup.

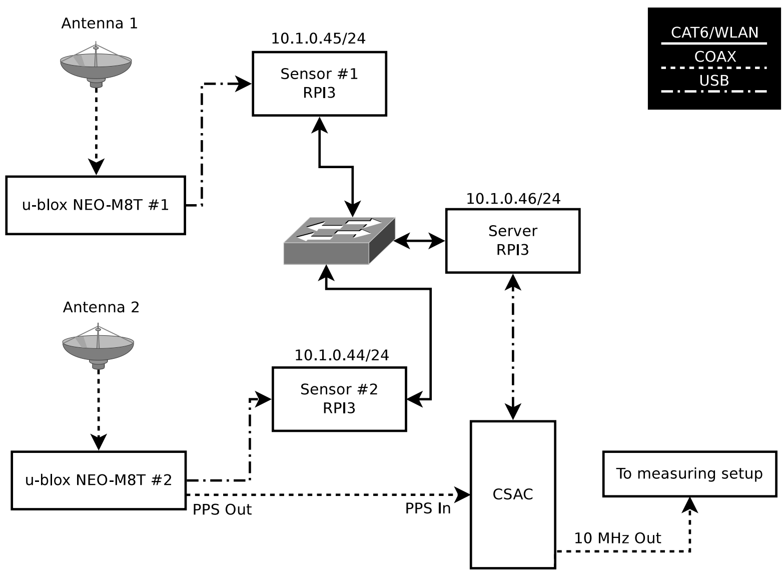


Figure 7.1: A block diagram showing the tested implementation.

7.2 Setup

Figure 7.1 shows how the server, sensors and atomic clock were physically set up. In order to assure good GPS satellite geometry, the antennas were placed at the roof of Justervesenet's office at Kjeller. Antenna one was placed at a railing about one meter above the ground, antenna two was placed at ground level. Figure 7.2 shows the placement of the antennas. Antenna one was connected to GPS receiver one, which in turn was connected to sensor one. It was the same deal with antenna two, which was connected to GPS receiver two which in turn was connected to sensor two. The distance between the two antennas was about 35 meters. The antenna connected to sensor one was not placed as far away as the cable would have allowed. This is because it proved challenging to find a suitable place to securely place the antenna and at the same time use the full length of the cable. The sensors and the server were connected to a LAN through a Gigabit Ethernet switch. The server was configured to log telemetry data received from the atomic clock and the clients were configured to log all NMEA data received from their respective GPS receiver. GPS receiver two supplied the atomic clock with a one PPS signal. Because the atomic clock model needs live data over time to mature, the system was started the 7 October 2016 and the test was performed 10 October 2016. Figure 7.3 is a block diagram showing how the measuring setup was physically configured. The measuring was done using a Spectracom CNT-91 frequency counter using a 10 MHz reference from Justervesenet's atomic clocks. The CNT-91 was configured to measure a continuous gap-less frequency at one second time-intervals. Figure G.1 is a photograph of the measuring setup. Both sensors were configured to use Justervesenet's internal NTP server and to use UTC instead of GMT+1.

7.3 Test one filter limits

Table 7.1 shows the filter thresholds used during test one.

Table 7.1: Filter thresholds used during test one.

	Sensor one	Sensor two
Altitude reference	123.8°	122.427°
Longitude reference	1102.1948°	1102.1934°
Latitude reference	5958.5448°	5958.5231°
Speed reference	0 knot	0 knot
Altitude deviation	10°	10°
Longitude deviation	0.005°	0.005°
Latitude deviation	0.005°	0.005°
Speed deviation	10 knot	10 knot

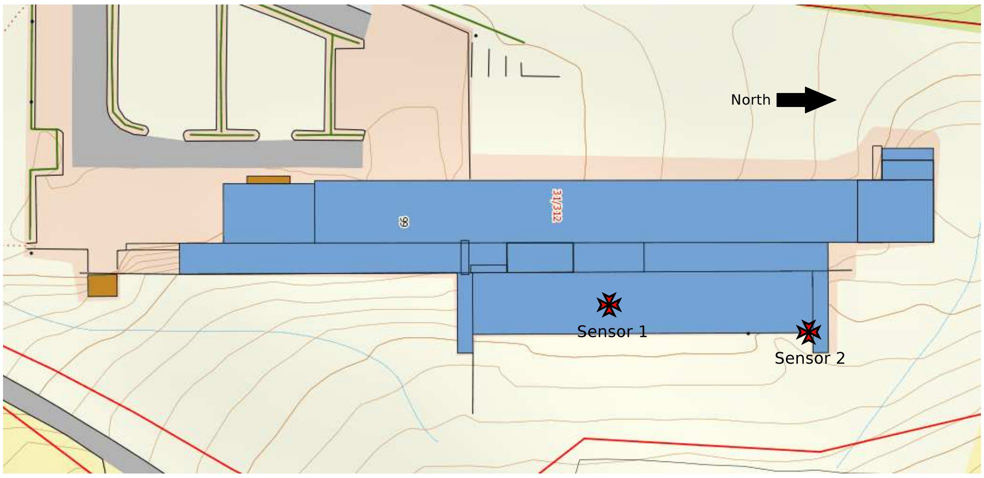


Figure 7.2: Block diagram showing the position of the antennas as reported by the receivers. Map courtesy of Kartverket [26]

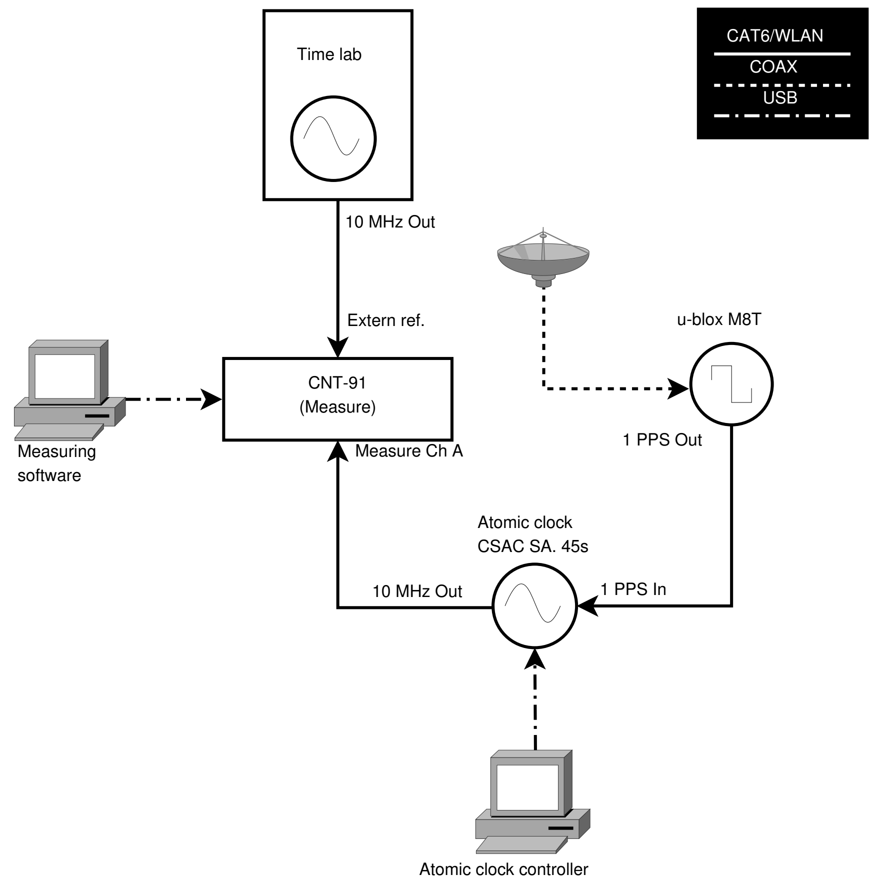


Figure 7.3: Block diagram showing the setup of the measurement equipment

7.4 Test one

7.4.1 Goal of test one

The goal with test one was to use the atomic clock controller to detect a simulated spoofing attack. By moving the antennas, the location and speed filter should be triggered as the solved longitude, altitude, latitude and speed change. See section 6.2.4.1 for more about the location and speed filter. The result is observable by analyzing the log files produced by the server and clients and the data collected by the measuring setup.

7.4.2 Description

The following is a step by step description of how the test was conducted. The time in the brackets was obtained from a wristwatch. The time is written in the normal format (hours and minutes) as well as number of seconds after 10:48. This is because the data used to draw the graphs started at 10:48 but uses seconds on the x-axis, to ease comparison. It is important to note that neither the resolution nor the accuracy was of any notable concern when the time was noted down. The time was mainly noted to make it easier to find any correlation between the steps taken and patterns found in the log files.

- 10:58 - 600: Moved antenna 1 approximately 15 meters to the south.
- 11:03 - 900: Moved antenna 1 back to original location.
- 11:07 - 1140: Moved antenna 2 approximately 15 meters to the north.
- 11:12 - 1440: Moved antenna 2 back to original location.
- 11:14 - 1560: Waved antenna 1 around horizontally in a half circle motion at an increasing tempo.
- 11:18 - 1800: Waved antenna 2 around horizontally in a half circle motion at an increasing tempo.
- 11:20 - 1920: Covered antenna 1 with aluminium foil.
- 11:25 - 2220: Covered antenna 2 with aluminium foil.
- 11:28 - 2400: Removed foil from antenna 1.
- 11:33 - 2700: Removed foil from antenna 2.

Step one and two were designed to trigger the location and speed filter, especially the check of solved latitude, longitude and altitude but also the speed against known values. Step five and six were also designed to trigger the location and speed filter but more specifically the checking of solved speed. By waving the antenna around while standing still, the solved location should not exceed the configured limits except for speed. Step seven and eight were designed to reveal what would happen during a jamming attack as it was believed that covering the antennas with aluminium foil would block all signals out. A photography showing how the antenna was covered with aluminium foil can be seen in section G.2 in the appendix.

7.4.3 Observations

7.4.3.1 Sensor Server logs

By reviewing the log produced by the server, the following was observed:

- No false positives were reported. The filters were not triggered before the test started.
- The location and speed filter was triggered by sensor one at 10:59:19 and cleared at 11:04:35.

```
    [10/10/16 - 10:59:17] [ ALARM ] Sensor 1 triggered LS filter!
    ...
    [10/10/16 - 11:04:35] [ ALARM ] Sensor 1 cleared LS filter!
```

- The location and speed filter was triggered again at 11:08:27, but this time by sensor one. The alarm was cleared at 11:13:43.

```
    [10/10/16 - 11:08:27] [ ALARM ] Sensor 2 triggered LS filter!
    ...
    [10/10/16 - 11:13:43] [ ALARM ] Sensor 2 cleared LS filter!
```

- Once again, 11:22:03 the location and speed filter was triggered by sensor one and was not cleared until 11:29:21

```
    [10/10/16 - 11:22:03] [ ALARM ] Sensor 1 triggered LS filter!
    ...
    [10/10/16 - 11:29:21] [ ALARM ] Sensor 1 cleared LS filter!
```

- Sensor two triggered the location and speed filter 11:27:31 and cleared at 11:34:16.

```
    [10/10/16 - 11:27:31] [ ALARM ] Sensor 2 triggered LS filter!
    [10/10/16 - 11:34:16] [ ALARM ] Sensor 2 cleared LS filter!
```

- The last three seconds, sensor two triggered and cleared the location and speed filter.

```
    [10/10/16 - 11:34:17] [ ALARM ] Sensor 2 triggered LS filter!
    [10/10/16 - 11:34:20] [ ALARM ] Sensor 2 cleared LS filter!
```

7.4.3.2 GPS data

The figures in this section are plots of the GPS data that was logged by the sensors during test one. Figure 7.4 shows the altitude in meters reported by sensor one. The only thing that stands out in the plot is when the antenna is covered in aluminium foil as shown at around the 2000 second mark. At this point the receiver is no longer able to solve for a position thus generating valid but empty fields in the NMEA data. This creates a deep void in the plot. Because both receivers' antennas were covered in aluminium foil, the plots all share this trait. Figure 7.5 shows the altitude in meters as reported by sensor two. When compared with the altitude plot for sensor one 7.4, sensor one seems to correlate better. It remains a puzzle why sensor two reported to have been moved seven meters down which it clearly was not. When examining figures 7.6 and 7.7 showing the latitude for sensor one and two (in that order), one can clearly see that sensor one was moved to the south and sensor two was moved to the north. One might also notice that sensor two traveled further. The difference in travel is because sensor one's cable is shorter. This is explained in the description of test one. Figures 7.8 and 7.9 shows a plot for the solved longitude for sensor one and two. When examining figure 7.8, it seems as if the antenna were never moved. This is because the building the antenna was placed on is position in a north to south line and the receiver was moved along with the roof. The antenna connected to sensor two was not moved in straight line because the cable got caught in vegetation. The last two figures, figure 7.10 and figure 7.11 show the speed solved by sensor one and two. They both correlate well with when the antennas were moved. However, the antenna connected to sensor two seems to be slightly more sensitive than the antenna connected to sensor one.

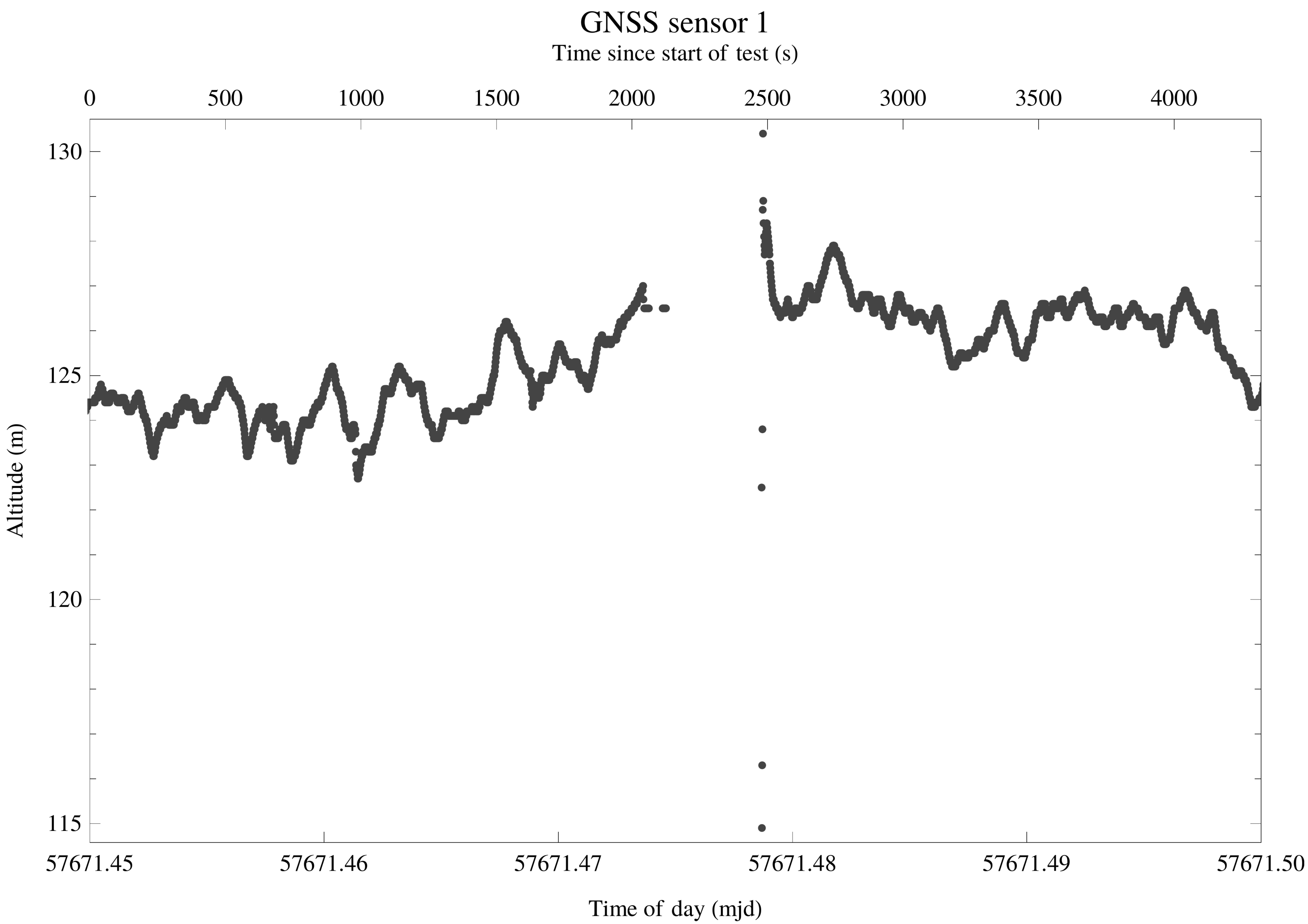


Figure 7.4: The figure shows the solved altitude of the GPS receiver connected to sensor one, during test one

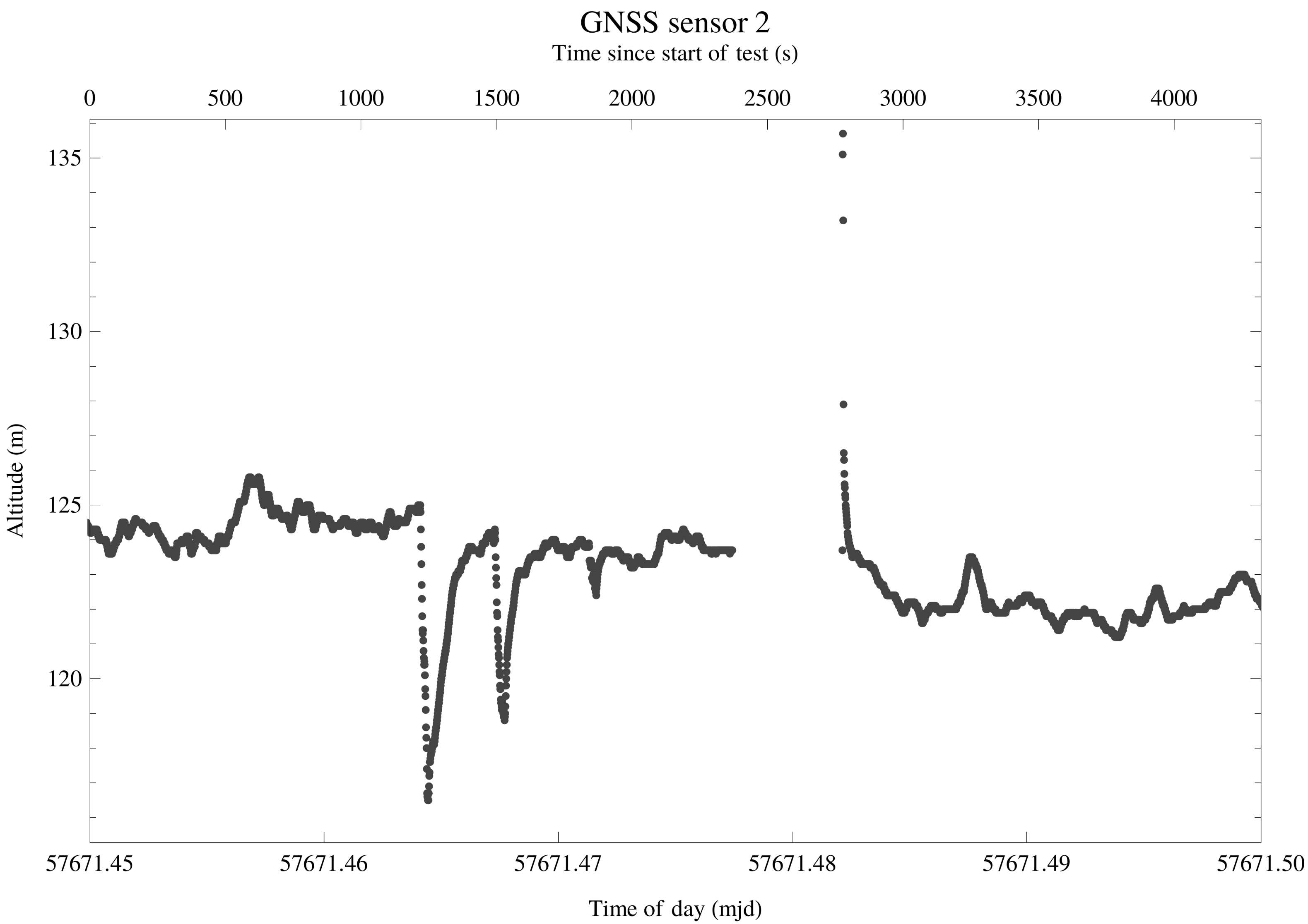


Figure 7.5: The figure shows the solved altitude of the GPS receiver connected to sensor two, during test one

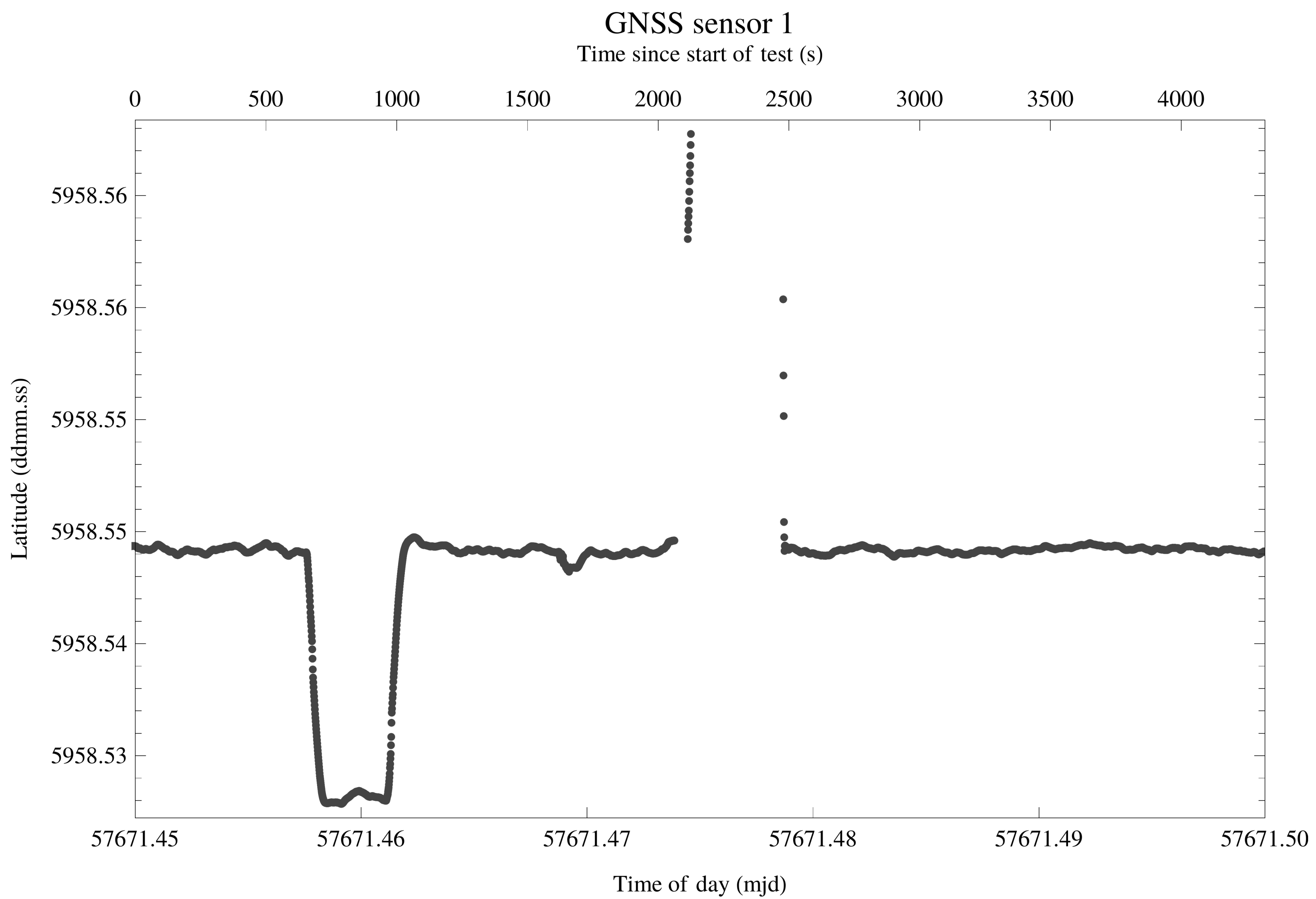


Figure 7.6: The figure shows the solved latitude of the GPS receiver connected to sensor one, during test one

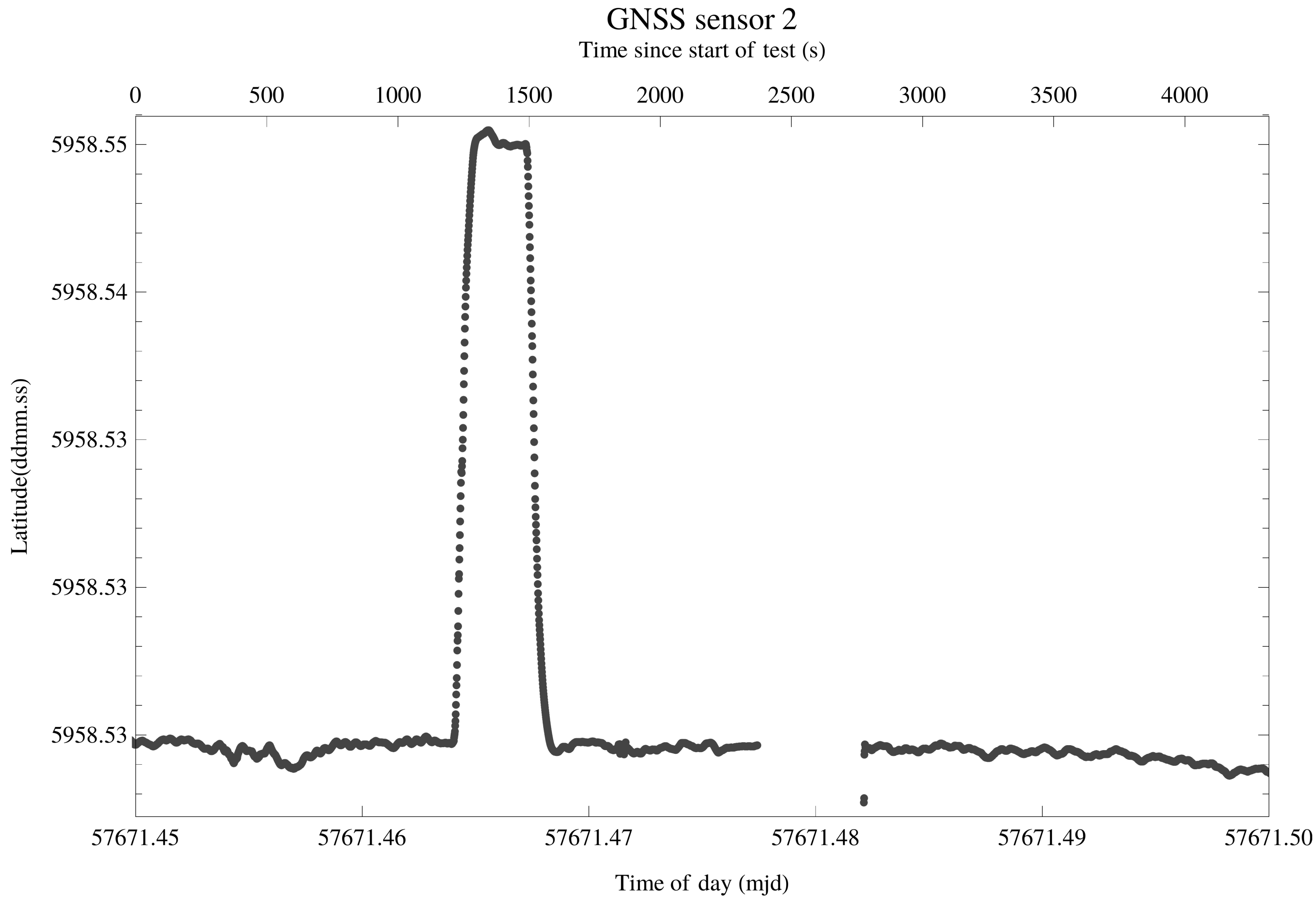


Figure 7.7: The figure shows the solved latitude of the GPS receiver connected to sensor two, during test one

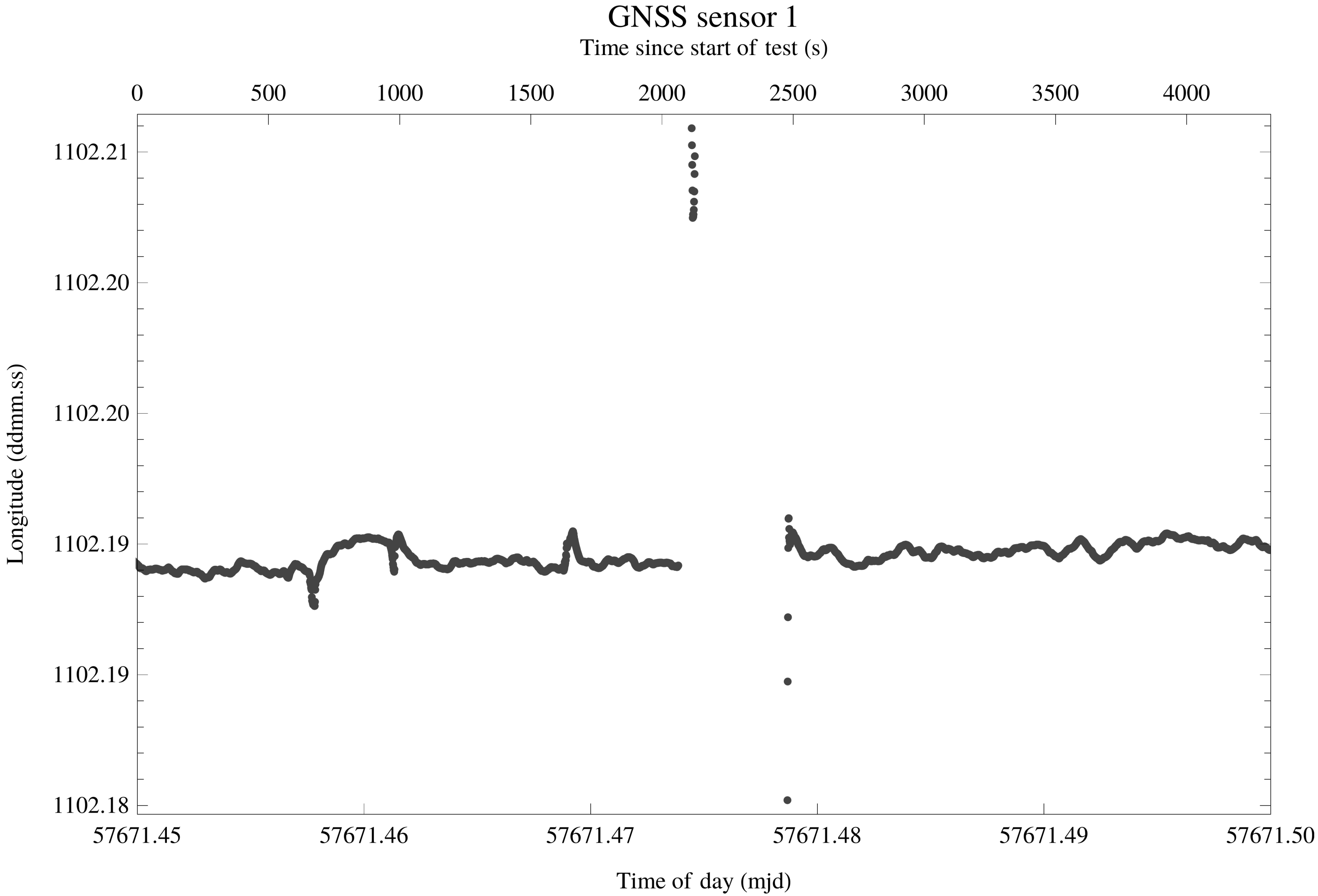


Figure 7.8: The figure shows the solved longitude of the GPS receiver connected to sensor one, during test one

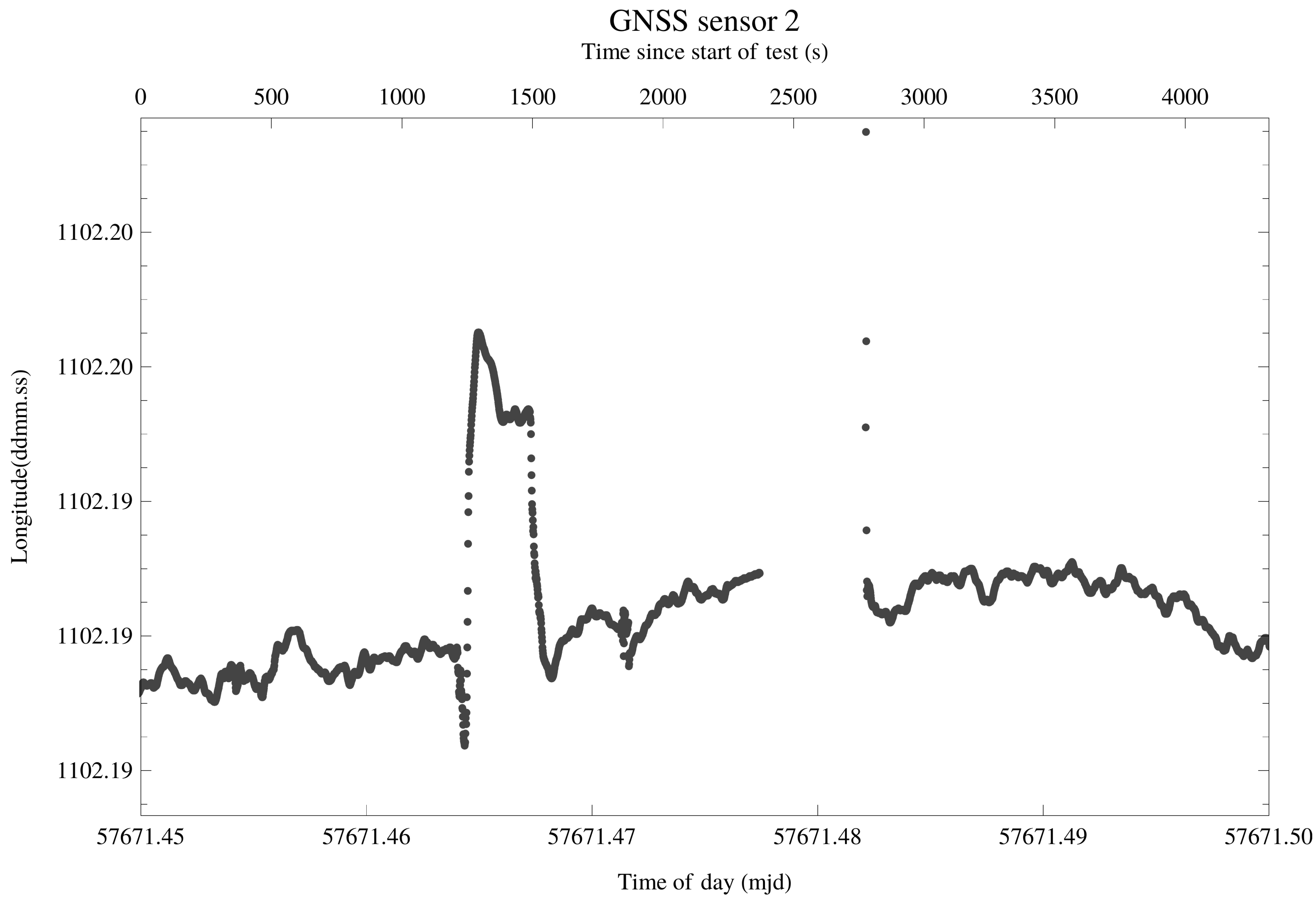


Figure 7.9: The figure shows the solved longitude of the GPS receiver connected to sensor two, during test one

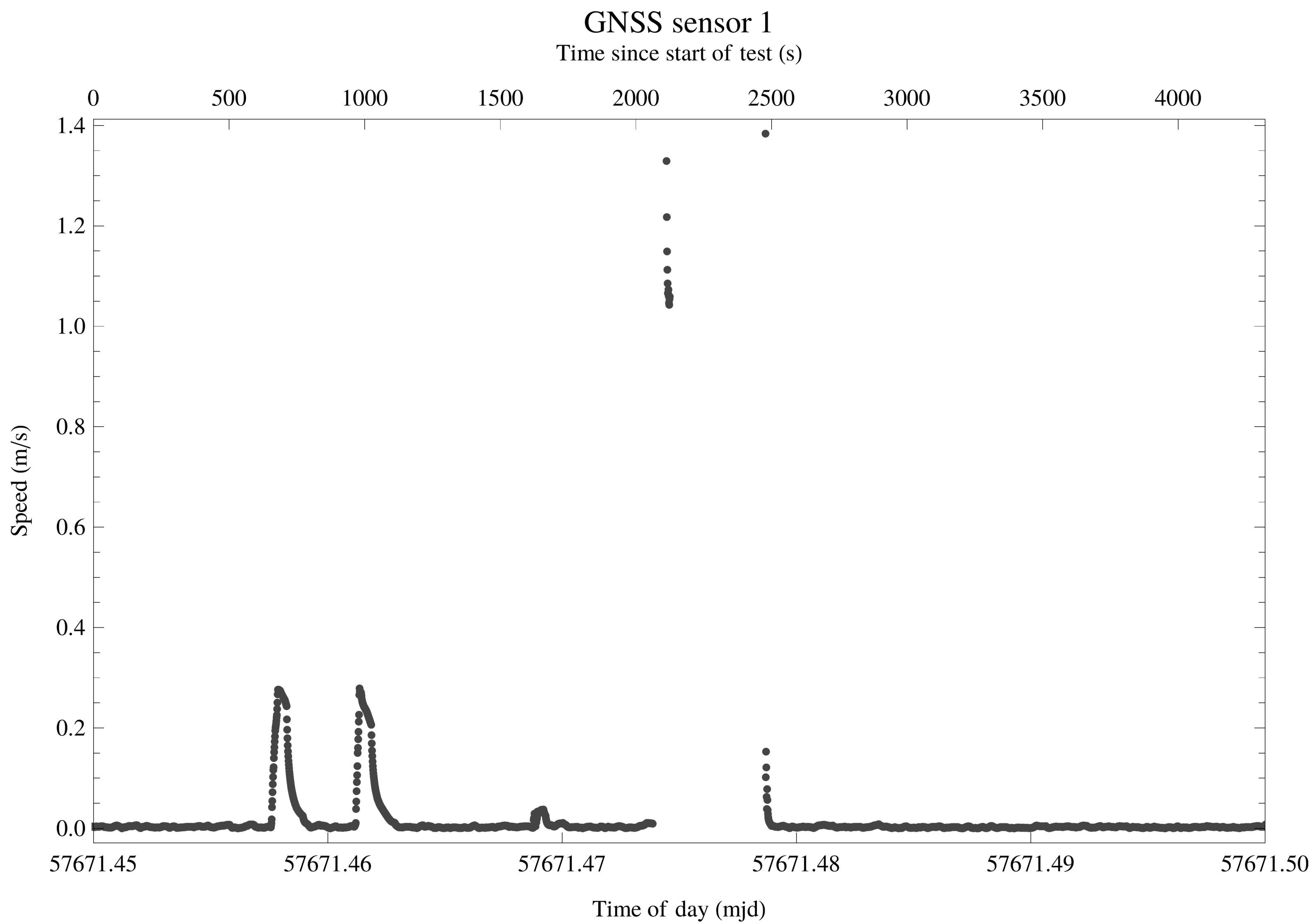


Figure 7.10: The figure shows the solved speed of the GPS receiver connected to sensor one, during test one

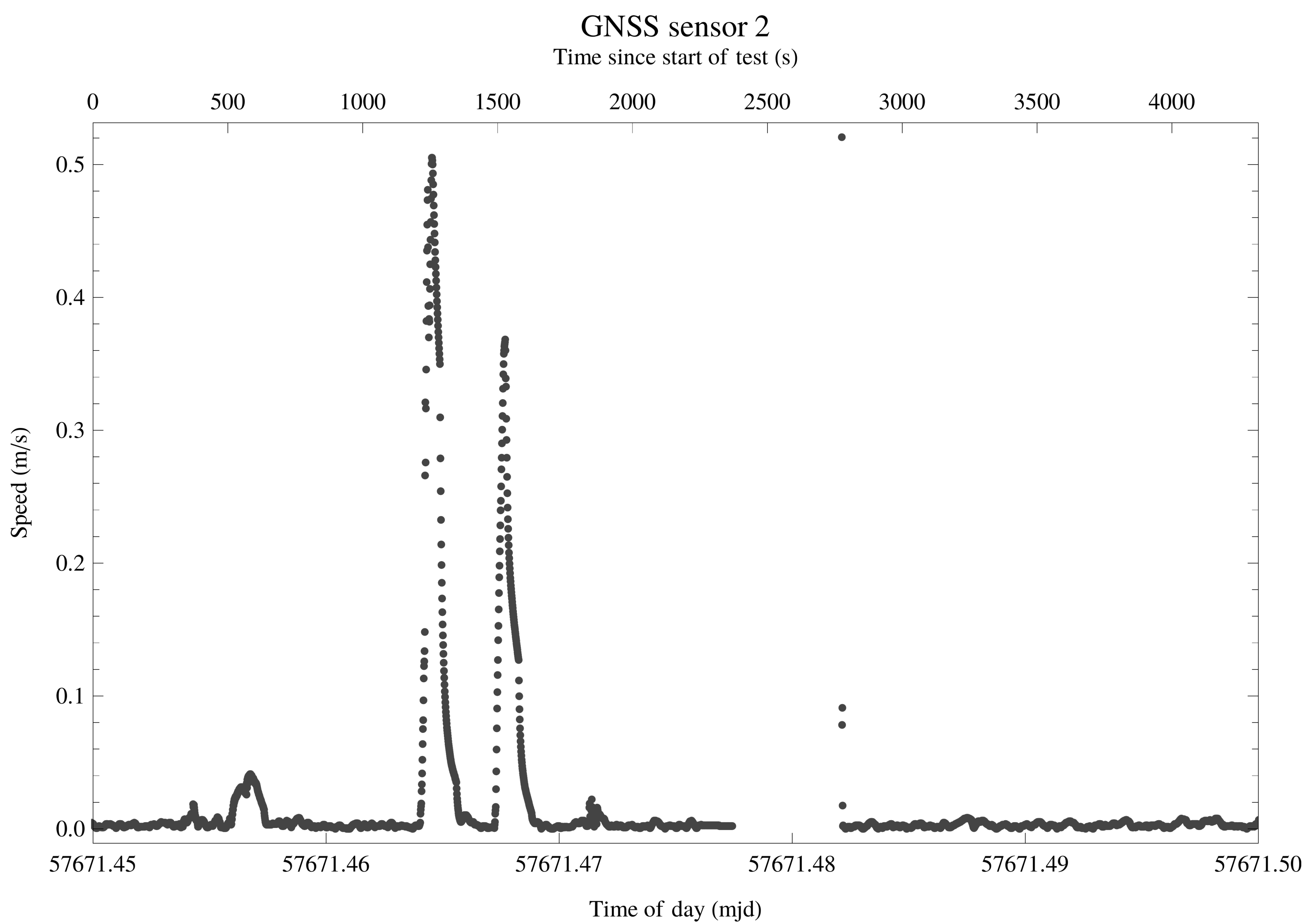


Figure 7.11: The figure shows the solved speed of the GPS receiver connected to sensor two, during test one

7.4.4 Timing measurements

Figure 7.12 shows the relative frequency offset (10^{12}) for the atomic clock. The thick dark plot in the middle is from telemetry data gathered from the atomic clock. Figure 7.13 shows the phase offset in nanoseconds for the atomic clock. The interesting thing to observe in both of these figures, is the jump in frequency and phase offset once the antenna was covered in aluminium foil. There is also a clear correlation between movement of the antenna and the relative frequency offset as seen in figure 7.12.

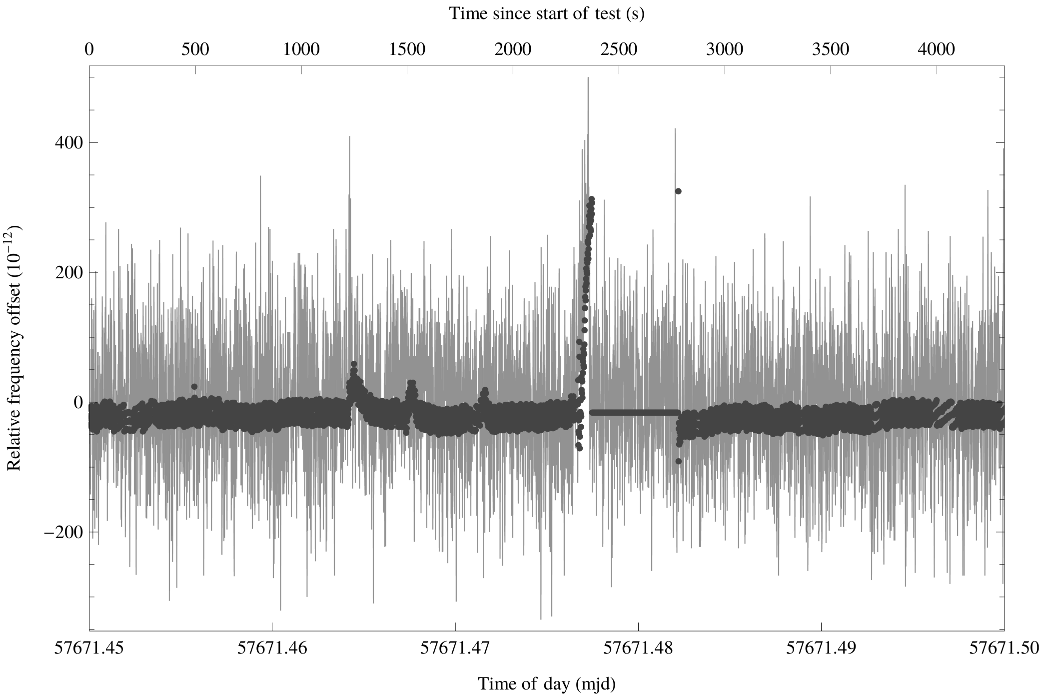


Figure 7.12: The figure shows the relative frequency offset for the atomic clock. The thick dark plot in the middle is from telemetry data gathered from the atomic clock. The lighter plot in the background is data from the CNT-91 frequency counter.

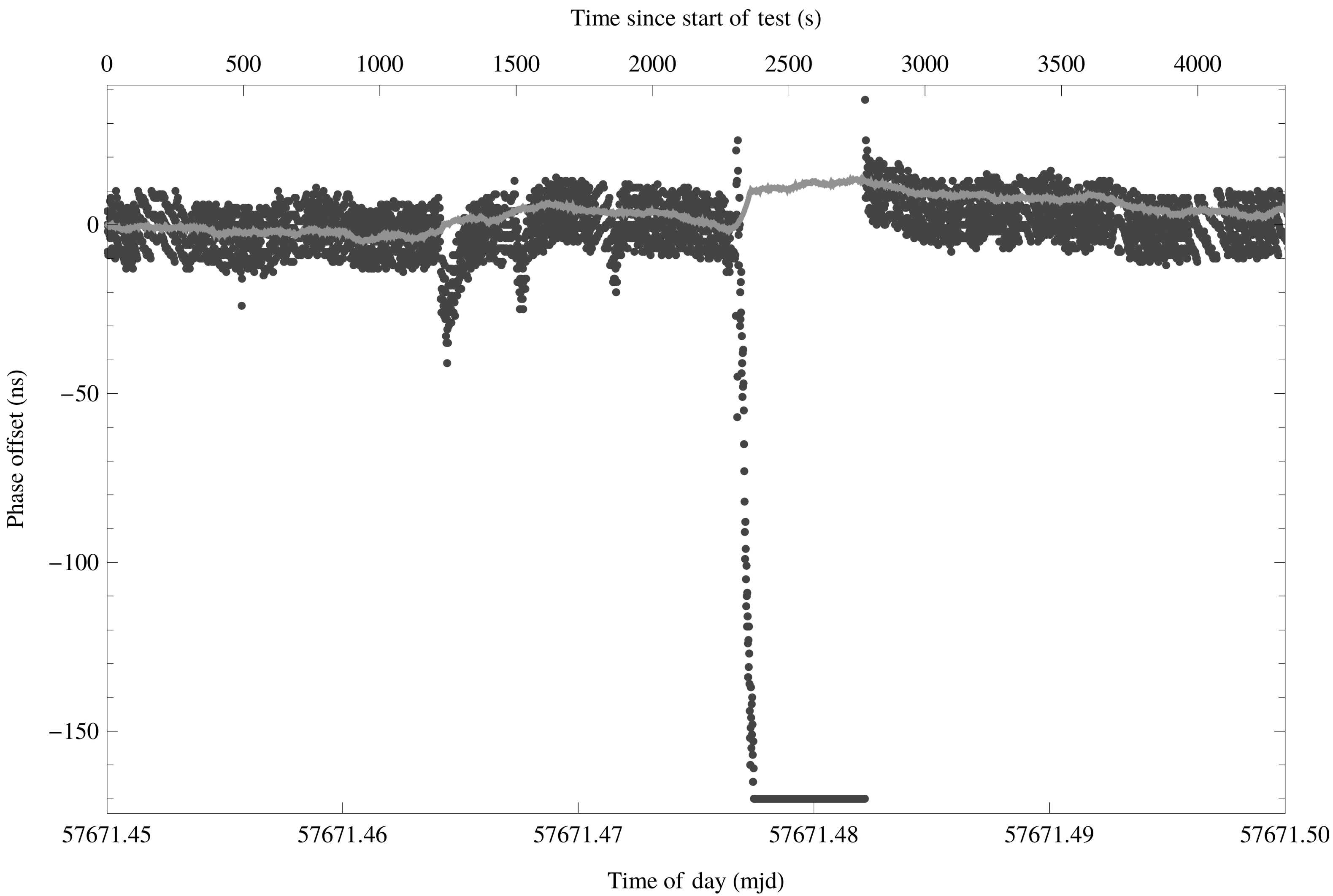


Figure 7.13: The figure shows the phase offset in nanoseconds for the atomic clock. The thick dark plot is from telemetry data gathered from the atomic clock. The lighter thin plot is data from the CNT-91 frequency counter.

7.4.5 Test one results

The observations from test one revealed some surprising results. When the antennas were moved horizontally the altitude solved by sensor two dramatically changed. The movement also resulted in a change in the timing solution. The GPS filter did, however, perform as expected. We discovered that the reason why the step five and six failed to trigger the filter, was that the limit for speed (see section 7.1 for more) was erroneously set too high.

7.5 Test two

7.5.1 Goal of test two

The goal of the second test was to test the clock model based filters. The test should make the atomic clock controller disable the atomic clock's disciplining mode and steer the atomic clock based on the clock model's predictions. We also wanted to test the location and speed filter again to make sure we had fixed the error in the filter configuration from the last test.

7.5.2 Change in setup

The setup in test one is quite similar to test two. There is however some changes. Sensor one is no longer connected to the atomic clock controller because the location and speed filter no longer was the main priority to test.

7.5.3 Test two filter limits

Table 7.2 shows the filter limits used in test two. Note that the speed deviation is set to 1 knot. Table 7.3 shows the limits for the clock model based filters.

Table 7.2: Filter thresholds used during test two

Config value	Sensor 2
Altitude reference	122.427
Longitude reference	1102.1934
Latitude reference	5958.5231
Speed reference	0
Altitude deviation	10
Longitude deviation	0.005
Latitude deviation	0.005
Speed deviation	1

Table 7.3: Clock model filter configuration

Phase limit	50
Steer limit	50
Time constant	10000
Warmup time	2
Prediction limit	200

7.5.4 Description

The description of test two is similar to test one (see section 7.4.2). One difference is that greater care was taken in obtaining an accurate time for the steps.

- 13:12:00 - 125: Started to move antenna two towards north.
- 13:12:45 - 170: Reached destination.
- 13:18:00 - 485: Started the move back to original location.
- 13:19:00 - 545: Reached destination.
- 13:23:00 - 785: Waved the antenna around at an increasing tempo in a half circle motion.
- 13:23:45 - 830: Stopped waving.
- 13:27:00 - 1070: Manually enabled the disciplining of the atomic clock.

7.5.5 Observations

7.5.5.1 Sensor Server logs

By reviewing the log produced by the sensor server the following was observed:

- No false positives, the filters were not triggered before the test started.
- The location and speed filter was triggered by at 13:12:10 and cleared at 13:19:54

```
    [10/24/16 - 13:12:10] [ ALARM ] Sensor 2 triggered LS filter!
    ...
    [10/24/16 - 13:19:54] [ ALARM ] Sensor 2 cleared LS filter!
```

- The location and speed filter was triggered again at 13:23:12 for only a second. It was then triggered again two seconds later and was not cleared until 13:23:43.

```
    [10/24/16 - 13:23:12] [ ALARM ] Sensor 2 triggered LS filter!
    ...
    [10/24/16 - 13:23:13] [ ALARM ] Sensor 2 cleared LS filter!
    ...
    [10/24/16 - 13:23:15] [ ALARM ] Sensor 2 triggered LS filter!
    ...
    [10/24/16 - 13:23:43] [ ALARM ] Sensor 2 cleared LS filter!
```

- More interesting was the fact that the *Frequency correction filter* was triggered.

```
      [10/24/16 - 13:12:42] [ ALARM ] Steer > predicted!
```

7.5.5.2 GPS data

Figure 7.14 shows the plotted GPS data collected during test two. As with the GPS data observed during test one, there is a clear correlation between the plotted data and when the antenna was moved. The strange phenomena as described in test one (7.4.3.2) where the altitude was reported to be way too low, occurred in test two as well.

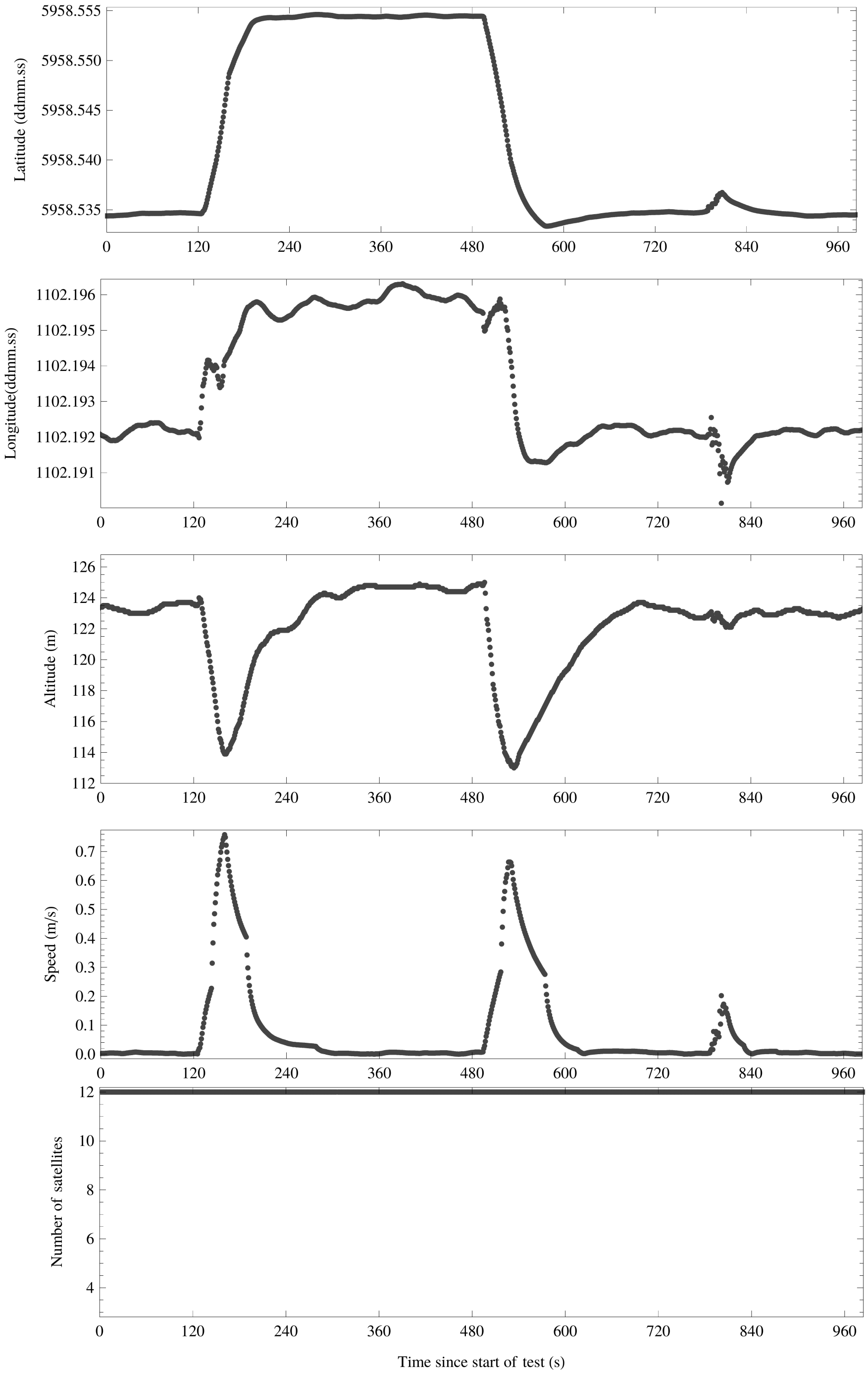


Figure 7.14: The figure shows (from the top) the latitude, longitude, altitude, speed and number of satellites as plotted from the GPS data collected during test two.

7.5.6 Timing measurements

Figure 7.15 shows measurements done during test two as two graphs with two time series each. The thin solid line in the top panel shows phase offset in nanoseconds. It reveals that the atomic clock was *not* steered based on the clock model once the disciplining was disabled. The panel at the bottom of figure 7.15 shows relative frequency offset as measured by the CNT-91 frequency counter. The thin darkly colored line is a plot of the data received as telemetry from the atomic clock. The plot shows that the disciplining of the atomic clock stopped after roughly 140 seconds.

7.5.7 Test two results

As mentioned under subsection 7.5.6, the disciplining of the atomic clock was successfully deactivated when the steer value reported by the atomic clock exceeded the steer value that was predicted by the model. The phase jump filter was not triggered because the phase offset never reached the configured limit. The frequency correction filter, on the other hand, was triggered because the steer value exceeded the configured limit (see table 7.3 for limits). It also shows that the atomic clock sadly was *not* steered using the clock model.

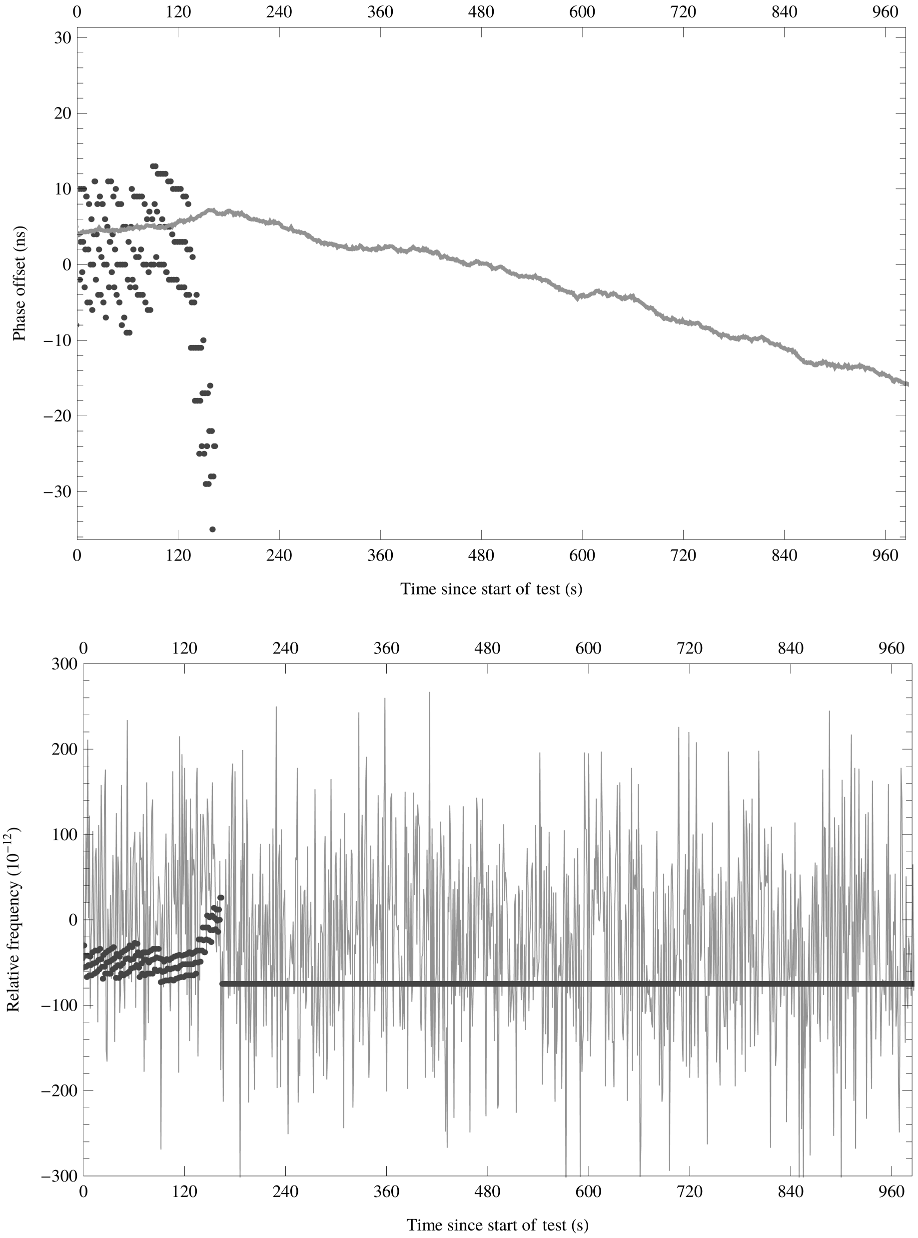


Figure 7.15: Timing measurements and clock telemetry data. The upper panel is phase offset in nanoseconds. The long, sloped light line is a plot of the measurements done by the CNT-91 frequency counter. The dotted dark plot, is also the phase offset but as reported by the atomic clock. The lighter plot in the background of the bottom panel is frequency steering as measured by the CNT-91 frequency counter. The dark plot is frequency offset as reported by the atomic clock.

7.6 Unplanned disturbance

The data presented in this section was gathered while the atomic clock controller was building the clock model for one of the planned tests. The data is interesting because it shows a disturbance from an unknown source. The data shown in the figures are from the 5 October 2016 to 6 October 2016. Figure 7.16 shows that about 10 minutes after midnight on 6 October 2016, the atomic clock entered "holdover mode", indicating that the 1 PPS signal from the GPS receiver was lost. Figure 7.17 shows the GPS receiver's solved altitude, longitude, latitude, speed and number of satellites. By examining the figure, it is obvious that the signal was lost minutes after the 57667 MJD (midnight) mark. The GPS receiver did not achieve consistent lock before approximately 57667.32 MJD (7:45 in the morning). In the meantime, the clock stability was impaired as figure 7.18 and 7.19 clearly show. Figure 7.19 is interesting because it shows how the atomic clock seems to have a delay in its steering algorithm. If the atomic clock controller applied the clock model filters while this data was collected, the phase jump filter would have triggered before the atomic clock would have been able to apply steering.

We have no idea what might be the origin of the disturbance. The following is therefore pure speculation: It is possible that a trucker spent the night parked by the road used a GPS jammer to hide his or her activities from an employer, and that the observed disturbance is a reflection of the jammer's signal. We did not find any traces of the disturbance when examining the logs from sensor one. Sensor one might have been shielded from a reflection because of its position. The time-frame for the disturbance also supports this theory since the disturbance started around midnight and ended at around 7:45, in other words "a good night sleep". This is of course only speculation as we do not have any data supporting it.

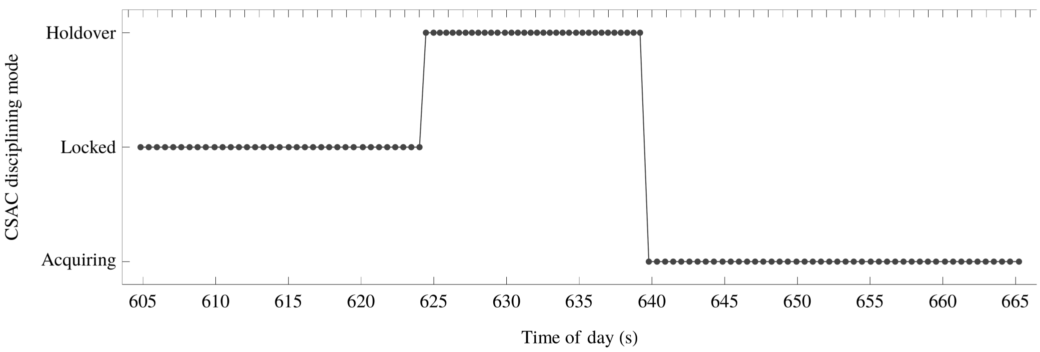


Figure 7.16: The figure shows the disciplined mode as reported by the atomic clock.

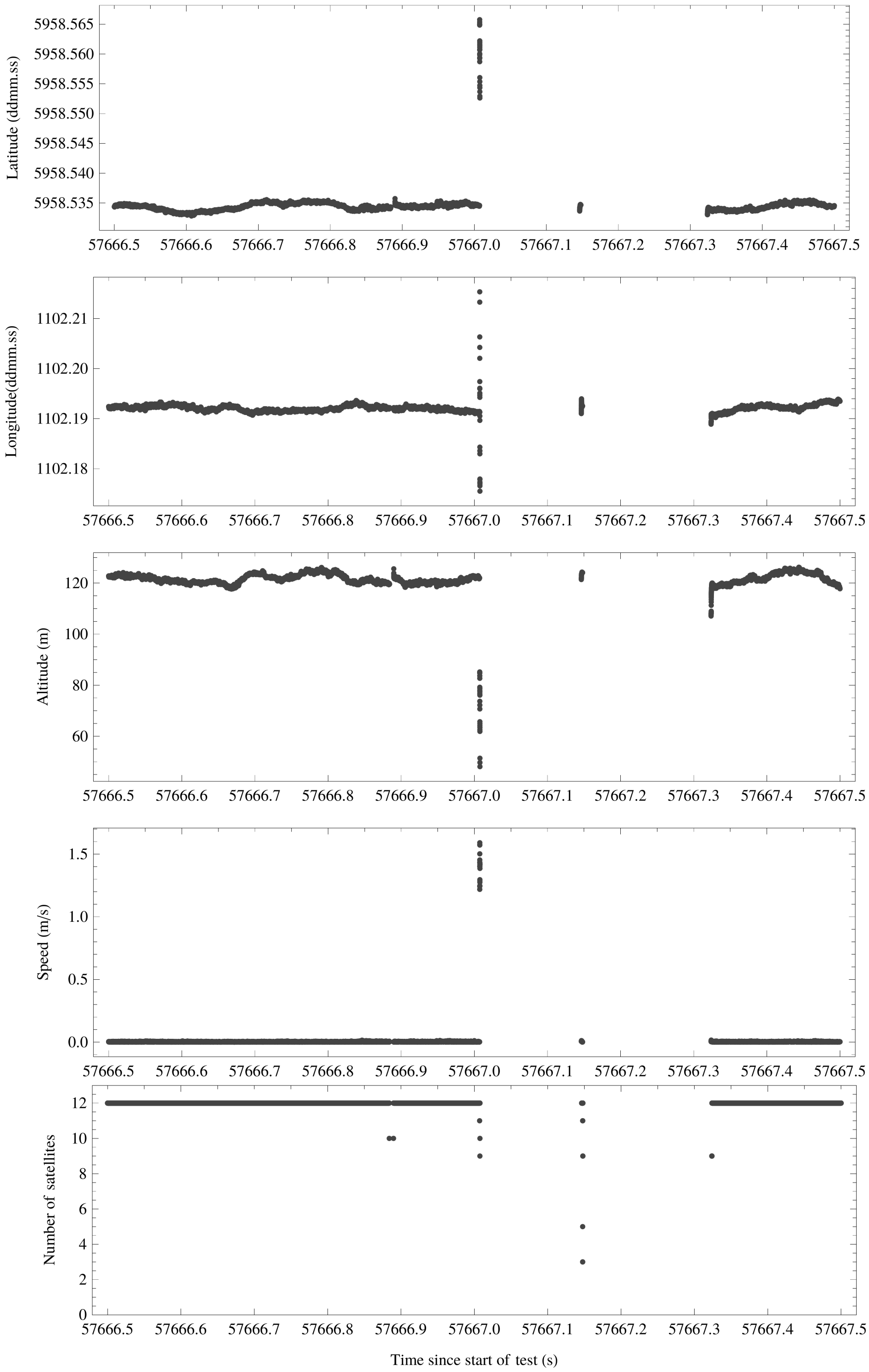


Figure 7.17: Figure shows solved position, speed and number of satellites as reported by the GPS receiver.

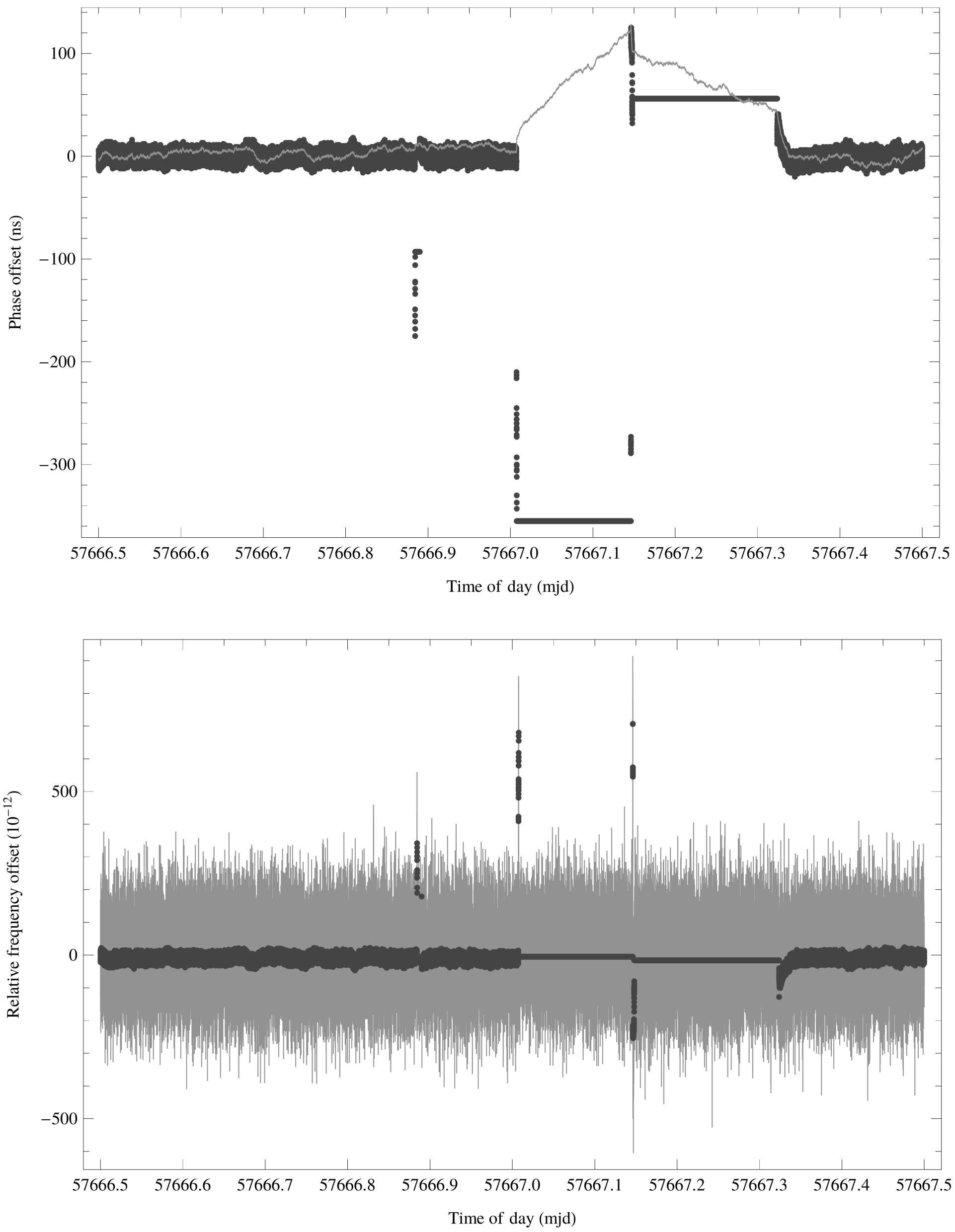


Figure 7.18: Top figure shows phase offset in nanoseconds. The thin lightly colored line is as measured by the CNT-91 frequency counter. The solid darker line is from the telemetry string from the atomic clock.

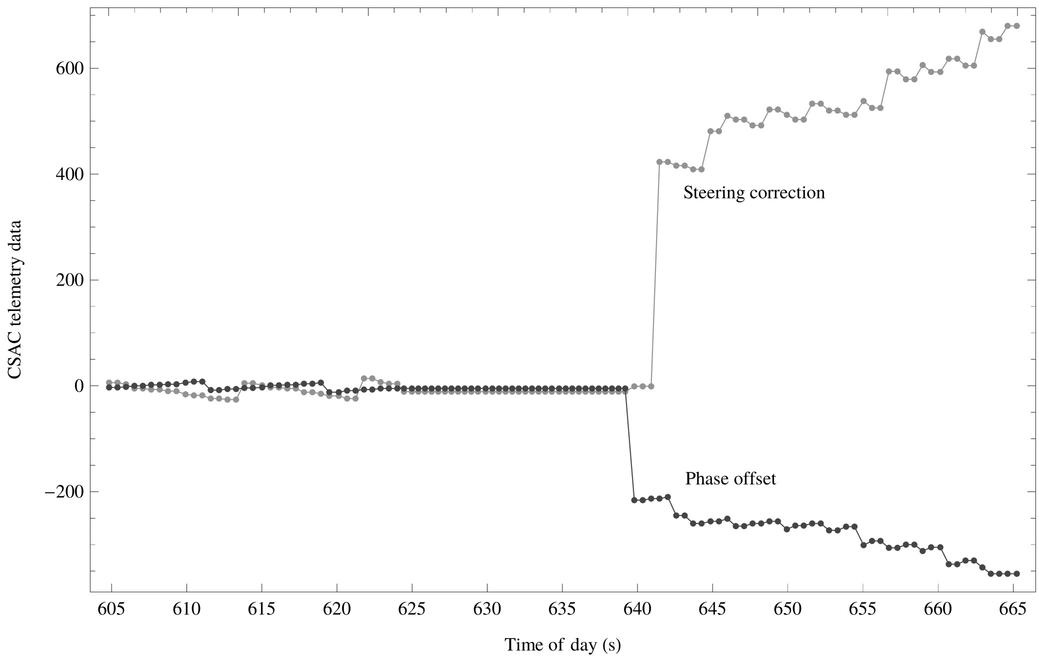


Figure 7.19: Figure shows the phase offset and steering correction as reported in the telemetry string from the atomic clock.

Chapter 8

Discussion

8.1 Test results

The GPS manipulation test as described in chapter 7 demonstrated the ability to quickly identify a GPS disturbance and also demonstrated the ability to protect the atomic clock by disabling the disciplining. The system did, however, fail to steer the atomic clock. When we attempted to reset and debug the system, we encountered a bug in the clock model as well. The bug affected the way the clock model initializes from the configuration file, which meant that the model would need at least 48 hours to rebuild. This is probably just a minor bug but because of the time constraint, we where forced to seize testing.

8.2 Shortcomings in current implementation

This section is used to discuss some of the shortcomings in the current version of the atomic clock controller and sensor server architecture. Some functions were never implemented and others were not finished in time.

8.2.1 Resizing shared memory segments

Ideally the shared memory segments containing the client list should be resizeable and its size should depend on the number of connected clients. This proved difficult and I was not able to implement it. M. Kerrisk explains in his book "The Linux Programming Interface" [27] that most UNIX implementations do not support resizing of a memory map like the shared memory segments used in the sensor server implementation. There is however a non-portable and Linux-specific system call named `mremap()` that can be used on Linux systems for this purpose. Unfortunately the address returned by `mremap()` might be different from the old address to the shared memory segment. This would mean that a pointer inside the shared segment might no longer be valid after a resize operation has been performed. A way to avoid this problem, caused by the remapping would be to use offsets instead of pointers when referring to addresses in the mapped region. While troubleshooting problems I had using `mremap()`, I stumbled upon a bug report in the the Kernel Bug Tracker [28] reported by someone with similar issues as I was having. This indicates that the trouble I had, might have been because of a bug in the Linux kernel. I have yet to confirm this but it did convince me to leave the implementation with its shortcomings rather than potentially wasting my time on something way out of my reach. The waste of memory would never by substantial anyway considering that the size of the `client_table_entry` struct is a modest 4664 bytes.

8.2.2 Atomic clock management

The sensor server should have used a separate process to handle the filters and communication with the atomic clock. This would free the processes handling the connections to clients from dealing with the filters, and make the filter abstraction more complete. This would also make the clock model cleaner. The atomic clock model is already logically separated from the filters associated with it, but because of the way the code is organized the use of the model implies the use of the filter. The atomic clock controller should still communicate with the atomic clock on its own like it does today, retrieving telemetry data, but the aforementioned process could keep track of the atomic clock's discipline status, steering and other functionality, thus creating a more generic way for the system to communicate with the atomic clock. The GPS based filter could greatly benefit from this approach because they today do not do anything but log occurrences where they were triggered.

8.2.3 External MJD calculation

Modified Julian Day (MJD) is a way to express both date and time as a single number. It's convenient when doing calculations with dates, for example: the difference between the MJD for one day and the next, is exactly 1[1]. Modified Julian Date relies heavily on by the clock model (see B for more about the clock model). During testing a python script (see E for more about this

[1]For example, the MJD for 24/10-2016 00:00 is 57685.0 and the MJD for 25/10-2016 00:00 is 57685.0. 12'o clock at the 25/10 would be 57658.5

script) was used to calculate MJD. This script was scheduled to be replaced by a module written in C, but was left in use because of time constraints. The script is called upon by using `popen()` which in turn calls `fork()` to run the script. The clock model is updated every second which means that this script is also invoked every second. This is of course counterintuitive considering that one of our goals for the software was to focus on efficiency.

8.2.4 External Atomic clock communication

For some reason, it proved to be a significant challenge to implement a solution in C to configure, read and write from the atomic clock. The best solution did not even provide a reliable means of communicating with the atomic clock even though communication with the GPS receivers which in theory should have been exactly the same, was no problem. Once time got tight, I made the decision to drop the development of the atomic clock serial communication module written in C, and decided to use `query_csac.py` by invoking with `popen()`.

8.3 Choice of programming language

The atomic clock controller software was originally planned to be written in Java since this was my most fluent programming language. Java is a great language. It is object oriented, it has a garbage collector and a lot of useful libraries. As development commenced it quickly became apparent that some parts of the code would be performance-critical and that portability really was not that important anyway. The platform was already chosen and I could not think of any reason for us to change it. I decided to look at other languages. Because performance was a concern Python was also quickly dismissed as an option. C++ would probably have been the best choice but never having written anything in C before made it sound more exciting and like a nice opportunity to learn something new. During the planning phase of the atomic clock controller development, raspbian-2015-05-07 was the latest build. It came with GCC 4.6.3 which only had experimental support for C11([29]). With C11 no longer considered an option, C99 was the obvious choice given its attractive features like:

- Variable-length arrays.
- Single line comments.
- snprintf() as standard [30].

8.4 Alternative approaches

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

8.4.1 Single computer, many GPS receivers

A single computer is used to run the atomic clock controller software. The atomic clock controller 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 GPS receivers you could connect as the USB specification limits the number possible endpoints to an absolute 127([31, pp. 3]) because of addressing. This does not mean that 127 devices can be connected because a single device might use more than one endpoint. It is also worth mentioning that a USB hub might "reserve" multiple endpoints. Depending on the GPS 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 GPS receiver is distanced from the atomic clock controller, a signal amplifier might be necessary to compensate for the signal attenuation. In some cases where a network is absent, this approach might be only option.

8.4.2 Store in database and analyze

With this approach, the idea of a GPS receiver and Raspberry PI as a single "sensor" unit is the same as with sensor server approach. The difference is that each sensor stores the collected data in a database. The atomic clock controller software monitors the clock directly as with the sensor 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 the sensor server approach, but the atomic clock controller 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 atomic clock controller to perform its tasks.

Chapter 9

Conclusion

The prototype developed and described in this report, demonstrates that a fully operational spoof proof atomic clock controller would resist a spoofing attack mounted with a sophisticated GPS spoofer like the Civil GPS spoofer [3]. We demonstrated the current implementation's ability to detect and in a limited sense mitigate, a simulated spoofing attack, using multi-layered defense mechanisms. The frequency steering filter made possible by the clock model would have detected steering attempts larger than 50×10^{-12} or 0.05 nanoseconds per seconds as shown in section 7.5. Even if a spoofing attack was done carefully and slowly enough not to trigger the clock model based filters, it would not have been able to spoof two different receivers without giving away the attack (see section 2.3.2.2). This would have required multiple spoofers spoofing individual GPS receivers. The spoofers would have to be meticulously tuned in order not to spoof neighboring receivers. This makes spoofing attempts very challenging to execute. Introducing a third receiver, which with sensor server architecture can be done with ease, would make it even harder.

During the GPS manipulation tests as described in chapter 7, the sensor server architecture proved its worth and worked perfectly. The system was stable and performed as designed over longer periods of time (over 4 days). The system logged data reliably and without fault, accepted connections and was responsive at all times tested. There were no memory leaks or segmentation faults and the system facilitated the detection of our GPS manipulation attempt with low latency. The system would probably react even faster if it received data from the atomic clock and GPS receivers at a higher frequency. The communication with the atomic clock also worked flawlessly, providing data for logging and the model as expected.

We have demonstrated the efficiency of creating a detection network using the sensor server architecture running on commodity hardware receiving generic GPS data from commodity receivers. We have demonstrated that it is possible to detect GPS disturbances without using an atomic clock as reference but just by using GPS receivers. The detection network combined with an atomic clock and associated model may provide both detection and mitigation during a spoofing attack.

Appendices

Appendix A

Acknowledgments

A.1 Contributions

Harald Hauglin, Chief engineer at Justervesenet was responsible for the following:

- Describing the clock model algorithm as seen under chapter B.
- The concept of CSAC SMACC (Chip scale atomic clock smart miniature atomic clock controller) that this work is built on.
- Created the graphs used in section 7.4.3.2, 7.4.4, 7.5.5.2, 7.5.6 and 7.6.
- Calculating the preliminary filter reference values, both for clock model based filters and GPS based filters. These were based on analysis of data gathered as described under subsection 7.1. Optimal choice of filter parameters is considered to be outside of the scope of this work.

Mr. Matt Davis is the author of `jdutil.py`[32], a library written for converting dates and time to MJD and back. Permission to use the jdutil.py was granted by Mr. Davis. The correspondence with Mr. Davis is included in the appendix under F.1.

Appendix B

Clock model

Clock modelling and clock filters in the spoof proof clock controller V 2.0 20161027 HHA

B.1 Introduction

The goal of clock modelling is to provide an estimate of two key parameters of the clock "state", the frequency offset and the clock drift, i.e. the rate of change of the frequency offset. The clock model will be used for two purposes: (1) As a reference for the clock frequency correction filter, in order to determine whether the current clock correction, as calculated and applied by the CSAC disciplining algorithm, is consistent with normal behavior of the clock; (2) In the case that a valid external disciplining pulse is lost, the model will be used to calculate the frequency corrections to be applied to the CSAC by the "spoof" proof clock controller. The model described below is a simple way of modelling the clock state, motivated by being easy to implement, more than being optimal for the task.

B.2 Input data for the clock model

Input data for the clock model comes from the CSAC telemetry string, sampled nominally every second. The key data is contained in the field identified by "Steer". This is the CSAC disciplining algorithm's current estimate of the frequency correction (in relative units) that has to be applied to the CSAC microwave synthesizer to correct for the frequency error of the CSAC "physics package". Ref [16]. Sample telemetry string and explanation of the data fields below are from the CSAC User Guide[ref]. Note that the "Steer" value reported by the CSAC in disciplining mode is the frequency correction that is applied to make the CSAC in sync with the external applied reference pulse. If the reference pulse is accurate – such as the 1 PPS pulse output from a properly operating GPS timing chip - then the negative "Steer" value is an estimate of the free-running CSAC frequency offset, i.e. a calibration of the CSAC.

As long as there is a valid and accurate reference pulse, a series of "Steer" values provide the basis for estimating the frequency offset of the CSAC, as well as its rate of change (drift).

One should note that the PPS output from the GPS chip has substantial noise in the short term. In contrast, the CSAC itself is more stable than the GPS reference over timespans up to 10000 s [16]. As a consequence, the steering corrections calculated by the CSAC are also noisy and that noise primarily due to noise in the reference pulse. In order to provide good estimates of the clock state, it is useful to filter sampled steering data to remove the influence of noise. This will be described next. Figure B.1 show an example telemetry string.

Listing B.1: Example of a telemetry string received from the atomic clock

```
0,0x0000,1209CS00909,0x0010
    ,4381,0.86,1.573,17.62,0.996,28.26,-24,---,-1,1,1268126502,586969,1.0
```

Identifier	Description	Notes
Status	Unit Status	*See Note 1*
Alarm	Pending Unit Alarms	*See Note 3*
SN	Unit serial number	*See Note 2*
Mode	Mode of operation	see **6.4.3** for bit definitions.

Figure B.1: Part 1 of table showing telemetry parameters. The table is taken from the SA.45's manual [16]

Contrast	Indication of signal level	Typically ≈ 4000 when locked, and ≈0 when unlocked.
LaserI	Laser current (mA)	
TCXO	Tuning Voltage (V)	0-2.5 VDC tuning range ≈ +/- 10 ppm
HeatP	Physics package Heater Power (mW)	Typical 15mW under NOC
Sig	DC Signal Level (V)	Typical 1.0 V under NOC
Temp	Unit temperature (°C)	Absolute accuracy is +/- 2°C
Steer	Frequency adjust	In pp10^{12}
ATune	Analog tuning voltage input	Reads "---" when analog tuning is disabled
Phase	1PPS_CSAC-1PPS_EXT (ns)	Only present when disciplining enabled
DiscOK	Discipline status (0-2)	0=acquiring, 1=locked, 2=holdover
TOD	Time (seconds)	Starts at 0 upon powerup unless set by command
LockT	Time since lock (seconds)	Starts at 0 upon lock
FWver	Firmware version	

Figure B.2: Part 2 of table showing telemetry parameters. The table is taken from the SA.45's manual [16]

B.3 Smoothing of sampled frequency steering data and estimates of the clock state

The frequency steer filter will be based on smoothed values of current and previous "steer" data from the CSAC telemetry string. The method described here is a very simple approach, primarily chosen for its ease of implementation. It is based on the assumption that "Steer" values are sampled at regular intervals. The design of an optimized approach for clock parameter estimation is way beyond the scope of this work. Calculation of smoothed values. Smoothed values of the clock steering correction are calculated using an exponential filter. Input to the calculation is the current sampled "Steer" value, steer_current, along with its associated timestamp (in mjd), t_current. The exponential filter has a parameter, w, whose inverse is the weight that each new sampled value adds to the existing smooth value:

$$steer\_smooth\_current = \frac{w-1}{w} steer\_smooth\_previous + \frac{1}{w} steer\_current$$

Smoothing is initialized by setting steer_smooth_previous = steer_current . Since the current smoothed value of the steering parameter is a weighted average of all previous samples as well, it effectively "lags" behind in time. The effect of this lag can be calculated by applying the same exponential smoothing to the associated timestamps

$$t\_smooth\_current = \frac{w-1}{w} t\_smooth\_previous + \frac{1}{w} t\_current$$

Again, smoothing is initialized by setting t_smooth_previous = t¿_current. Updated smoothed values t_smooth and steer_smooth will be computed every time the telemetry string is read (about once per second). Representative data for sampled and smoothed "steer" values are shown in fig XXX. Daily updated clock model Smoothed values will be used to estimate the frequency drift. A simple approach that "works" is to use values at the start of every day and compare to the values from the previous day. Again, this is not an optimized approach. At the start of every new day set the following:

$$t\_smooth\_today = t\_smooth\_currentsteer\_smooth\_today = steer\_smooth\_current$$

and label previous day's values t_smooth_yesterday and steer_smooth_yesterday. The drift of the steering parameter can be estimated as:

$$steer\_drift = \frac{(steer\_smooth\_today – steer_smooth\_yesterday)}{(t\_smooth\_today – t\_smooth\_yesterday)}$$

The predicted steering parameter for a given point in time t can now be computed as a simple linear relationship

$$steer\_predicted(t) = steer\_smooth\_today + (t – t\_smooth\_today) * steer\_drift$$

The output of this model is the predicted CSAC steering correction for a given point in time t.

B.4 Phase jump filter – fast timing filter

The "fast" timing filter will check the value of the "Phase" CSAC telemetry phase field. The CSAC disciplining algorithm steers the clock so that the generated PPS output is in sync with the reference PPS input. The reference value for the phase offset filter is therefore 0. Limits for acceptable deviations from the reference value has to be based on normal noise in the phase data. A representative series of phase data is shown in figure B.3.

The filter parameter is a constant phase_limit will be read from a configuration file. Flag for invalid timing will be raised when:

$$abs(phase\_current) > phase\_limit$$

Tentatively phase_limit is set to 50 ns.

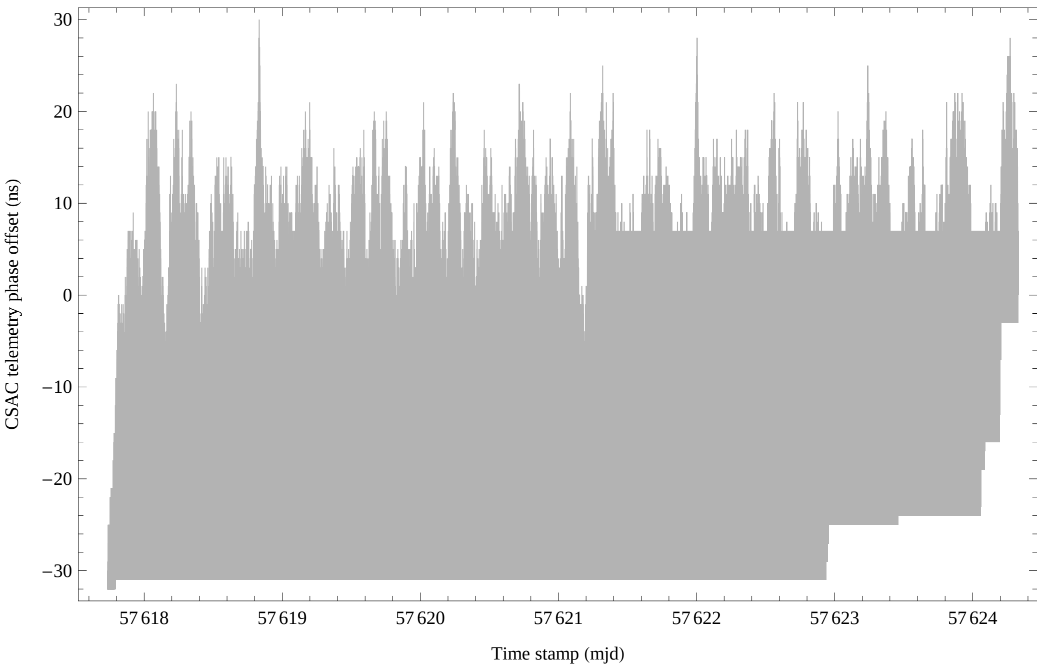


Figure B.3: Figure shows phase offset in nanoseconds as measured by the CSAC

B.5 Frequency correction filter

The frequency correction filter will check the value of the CSAC telemetry "steer" field and compare it to the steering correction predicted by the clock model. Limits for acceptable deviations from the predicted value has to be based on normal noise in the frequency steering data. A representative series of sampled steering data along with model predictions is shown in figure B.4. Filtering of the current frequency steering can now be implemented as follows:

$$Abs(steer\_current – steer\_predicted(t\_current)) > steer\_limit$$

The parameter steer_limit will be read from a configuration file. The limit is tentatively set to 50 based on the data below. Note that this is a relative frequency correction in units of (10^{12}), corresponding to rate of change of 0.05 ns/s in the timing output of the clock.

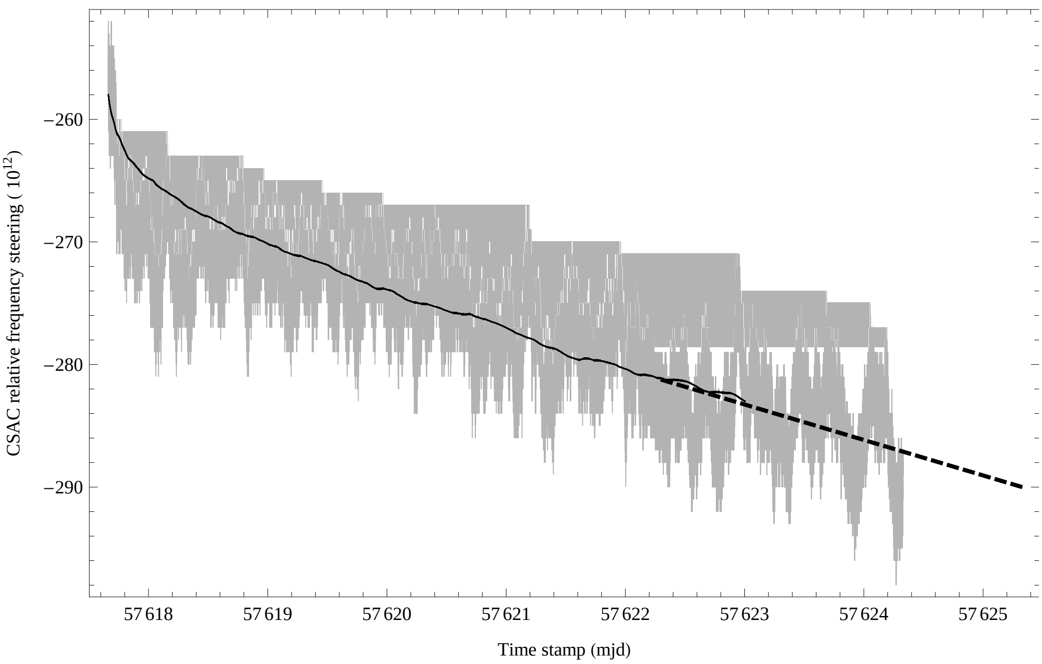


Figure B.4: Sampled clock steering corrections (gray), smoothed values (black solid line) and predicted clock steering values (dashed). Sampled data at 1 s intervals have been smoothed using an exponential filter with filter parameter w = 100000.

Appendix C

Data acquisition

C.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()
```

C.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'

```

```
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()
```

C.3 GPS Logger source code

```
'''
:Author: Aril Schultzen
:Email: aschultzen@gmail.com
'''
# This script attempts to connect to the
# Sensor Server at <ip> : "port" and
# IDs itself as <id>. It will then
# poll the time solved by the GNSS receiver
# connected to Sensor<id> until
# terminated.

```

```
import socket
import sys
import time

ip = "10.1.0.46"
port = 10001
id = 1

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, msg:
    print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
    sys.exit();
try:
    remote_ip = socket.gethostbyname( ip )

except socket.gaierror:
    print 'Could not resolve hostname'
    sys.exit()

s.connect((remote_ip , port))
s.sendall(b'IDENTIFY -10')
recv_buff = s.recv(1024)

while(1):
    s.sendall(b'PRINTTIME' + str(id))
    time.sleep(0.1)
    recv_buff = s.recv(1024)
    recv_buff = recv_buff.strip('>\n')
    print("Sensor " + str(id) + " GNSS solved time: " + recv_buff)
    time.sleep(0.9)
```

Appendix D

Sensor server software

D.1 Client

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];
```

```

    /* 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);
        }
```

```
    }
    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;
}
```

sensor_client.h

```
#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 */
```

client_config.ini

```
serial_interface: /dev/ttyS0
client_id: 1
log_nmea: 1
log_file_name: log_sensor
connection_attempts_max: 10
```

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
```

```
        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()
```

D.2 Server

sensor_server.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"
```

```
#define WAITING_FOR_CONNECTIONS "Waiting for connections...\n"
#define CON_ACCEPTED "Connection accepted\n"
#define CLIENT_DISCONNECTED "Client [%d] at [%s] 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;

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

/* Pointer to client list map */
struct client_table_entry **client_list_map;

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

/* Pointer to shared CSAC model data */
struct csac_model_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();
static void handle_sig(int signum);
static void initialize_config(struct config_map_entry *conf_map,
                              struct server_config *s_conf);
static int start_server(int port_number);
static int usage(char *argv[]);
static void setup_session(int session_fd, struct client_table_entry *new_client);
static int release_mem_piece(struct client_table_entry* release_me);

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

```
        return 0;
    }
    return 1;
}

/* 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* cli;
    struct client_table_entry* temp;
    int found = 0;

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

/* Removes a client with the given PID */
static void remove_client_by_pid(pid_t pid)
```

```
{
    struct client_table_entry* cli;
    struct client_table_entry* temp_remove;

    sem_wait(&(s_synch->client_list_sem));
    list_for_each_entry_safe(cli, temp_remove,&client_list->list,
                             list) {
        if(cli->pid == pid) {
            /* Decrementing sensor count */
            if(cli->client_id > 0) {
                s_data->number_of_sensors--;
            }
            t_print(CLIENT_DISCONNECTED, cli->client_id ,cli->ip);
            list_del(&cli->list);
            release_mem_piece(cli);
        }
    }
    /* Decrementing total client count */
    s_data->number_of_clients--;
    sem_post(&(s_synch->client_list_sem));
}

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

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

static int release_mem_piece(struct client_table_entry* release_me)
{
    int i;
    for(i = 1; i < s_conf->max_clients; i++){
        if(client_list_map[i] == NULL){
            client_list_map[i] = release_me;
            return 1;
        }
        i++;
    }
    return 0;
}

static struct client_table_entry* get_mem_piece()
{
    int i;
    for(i = 1; i < s_conf->max_clients; i++){
        if(client_list_map[i] != NULL){
            struct client_table_entry *tmp = client_list_map[i];
            client_list_map[i] = NULL;
            return tmp;
        }
```

```
        i++;
    }
    return NULL;
}

/* 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_sem));
    s_data->number_of_clients++;
    struct client_table_entry* tmp;
    tmp = get_mem_piece();
    list_add_tail( &(tmp->list), &(ptr->list) );
    sem_post(&(s_synch->client_list_sem));
    return tmp;
}

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

        if(pid > 0) {
            remove_client_by_pid(pid);
        }
    }
}

/* 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());
    s_synch->done = 1;
}

/* 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;
}

/* 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.ls_f.moved = 0;
    new_client->fs.ls_f.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(!s_synch->done) {
        if(!respond(new_client)) {
            break;
```

```c
        }
    }
}

/*
* Main loop for the server.
* Forks everytime a client connects and calls setup_session()
*/
static int 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 + 1) * sizeof(struct client_table_entry)),
                           PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);
        if(client_list == MAP_FAILED){
            t_print("Failed to allocate memory for the client list!\n");
        }
    } else {
        t_print(ERROR_CONFIG_LOAD_FAILED);
        exit(0);
    }

    client_list_map = malloc((s_conf->max_clients + 1)
        * sizeof(struct client_table_entry*));
    int i;

    /* Skip the first entry for some reason */
    for(i = 1; i < s_conf->max_clients; i++){
        client_list_map[i] = client_list + i;
    }

    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);
    if(s_data == MAP_FAILED){
            t_print("Failed to allocate memory for the server data!\n");
    }

    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);

    if(s_synch == MAP_FAILED){
            t_print("Failed to allocate memory for the semaphores! \n");
    }

    sem_init(&(s_synch->ready_sem), 1, 1);
    sem_init(&(s_synch->client_list_sem), 1, 1);
    sem_init(&(s_synch->csac_sem), 1, 1);

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

    if(cfd == MAP_FAILED){
            t_print("Failed to allocate memory for the CSAC filter data! \n");
    }

    if( &(s_synch->ready_sem) == SEM_FAILED
            || &(s_synch->client_list_sem) == SEM_FAILED) {
        t_print(ERROR_SEMAPHORE_CREATION_FAILED);
        sem_close(&(s_synch->ready_sem));
        sem_close(&(s_synch->client_list_sem));
        sem_close(&(s_synch->csac_sem));
        exit(1);
    }


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

    /* Waiting for filter to start */
    sleep(1);
    if(s_synch->done)
        return 1;

    /* 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);
    }

    /* 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 (!s_synch->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);
                setup_session(session_fd, new_client);
                close(session_fd);
                if(new_client->marked_for_kick) {
                    t_print(CLIENT_KICKED, getpid());
```

```
                }
                _exit(0);
            } else {
                t_print(CON_ACCEPTED);
                close(session_fd);
            }
        }
    }

    /* Destroying semaphores */
    sem_destroy(&(s_synch->csac_sem));
    sem_destroy(&(s_synch->ready_sem));
    sem_destroy(&(s_synch->client_list_sem));

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

    /* Closing server FD */
    close(server_sockfd);
    t_print(SERVER_STOPPED);
    return 1;
}

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);
}
```

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
#define CLIENT_TIMEOUT 5
#define MONITOR_TIMEOUT 1000
#define UNIDENTIFIED_TIMEOUT 10

#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_model_data *cfd;
```

```

/** @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);

int set_timeout(struct client_table_entry *target,
                        struct timeval h_timeout);

#endif /* !SENSOR_SERVER_H */
```

config.ini

```
humanly_readable_dumpdata: 1
max_clients: 100
warm_up: 24000
csac_serial_interface: /dev/ttyUSB0
logging: 1
log_path: server_log.txt
csac_logging: 1
csac_log_path: csac_log.txt
```

sensor_server_common.h

```
/**
 * @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"

/* 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 lsf_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 lsf {
    struct lsf_data lsf_d;
    int moved;
    int was_moved;
    struct disturbed_values dv;
};

struct filters {
    struct lsf ls_f;
```

```
};

/*
* CLIENT TABLE STRUCT
*
* 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 timeout;             /* Timeout in seconds if not active */
    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_sem;
    sem_t csac_sem;
    sem_t client_list_sem;
    volatile sig_atomic_t done;
};

/*
* 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 */
```

session.c

```
#include "session.h"

/* 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_LSD_LOAD_FAILED "ERROR:Failed to load LS data from file\n"
#define ERROR_CHECKSUM_FAILED "ERROR: Checksum failed!\n"
#define ERROR_ILLEGAL_NMEA "ERROR: Received illegal/corrupt NMEA data\n"

static int nmea_ready();
static int 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 parse_input(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* c_iter;
    struct client_table_entry* temp;
    int ready = 0;

    /* iterating through the list of clients */
    list_for_each_entry_safe(c_iter, temp, &client_list->list, list) {
        /* Is it a SENSOR?*/
        if(c_iter->client_type == SENSOR) {
            /* Is it ready?*/
            if(c_iter->ready){
                ready++;
            }
        }
    }
    /* if everyone is ready, clear ready flag and carry on */
    if(ready == s_data->number_of_sensors) {
        list_for_each_entry_safe(c_iter, temp, &client_list->list, list) {
            c_iter->ready = 0;
        }
        return 1;
    } else {
        return 0;
    }
}

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

    /* Extracting latitude */
```

```
    if(substring_extractor(LATITUDE_START,LATITUDE_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_rmc, strlen(cte->nmea.raw_rmc)))
    {
        cte->nmea.lat_current = atof(buffer);
    } else {
        return 0;
    }


    /* Extracting longitude */
    if(substring_extractor(LONGITUDE_START,LONGITUDE_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_rmc, strlen(cte->nmea.raw_rmc)))
    {
        cte->nmea.lon_current = atof(buffer);
    } else {
        return 0;
    }

    /* Extracting altitude */
    if(substring_extractor(ALTITUDE_START,ALTITUDE_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_gga, strlen(cte->nmea.raw_gga)))
    {
        cte->nmea.alt_current = atof(buffer);
    } else {
        return 0;
    }

    /* Extracting speed */
    if(substring_extractor(SPEED_START,SPEED_START + 1,',',buffer, buffsize,
                        cte->nmea.raw_rmc, strlen(cte->nmea.raw_rmc)))
    {
        cte->nmea.speed_current = atof(buffer);
    } else {
        return 0;
    }

    return 1;
}

/* 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)
```

```c
{
    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);
}

/*
* Parses input from clients. Return value indicates status.
* Uses the command_code struct to convey parameter and command code.
*
* 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_LOADKRLDATA */
    else if(strstr((char*)incoming, PROTOCOL_LOADKRLDATA ) == (incoming)) {
        int length = (strlen(incoming) - strlen(PROTOCOL_LOADKRLDATA) );
        memcpy(cte->cm.parameter,
               (incoming)+(strlen(PROTOCOL_LOADKRLDATA)*(sizeof(char))), length);
        cte->cm.code = CODE_LOADKRLDATA;
    }

    /* PRINT_LOADKRLDATA_SHORT */
```

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

    /* 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 */
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) {
            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;
                    s_write(&(cte->transmission), "ID in use!\n", 11);
                    if(cte->cm.id_parameter < 0){
                        return 1;
                    }
                    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_sem));
                s_data->number_of_sensors++;
                sem_post(&(s_synch->client_list_sem));
            }
            /* 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_lsf_data(cte)) {
                    t_print("Loaded filter data for client %d\n", cte->client_id);
                } else {
                    t_print("Failed to load LS filter data for Sensor %d\n", cte->client_id);
                    s_write(&(cte->transmission),ERROR_LSD_LOAD_FAILED,
                            sizeof(ERROR_LSD_LOAD_FAILED));
                    return 0;
                }
            }

            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);

            if(rmc_start == NULL || gga_start == NULL){
                s_write(&(cte->transmission), ERROR_ILLEGAL_NMEA, strlen(ERROR_ILLEGAL_NMEA));
                return 1;
            }

            memcpy(cte->nmea.raw_rmc, rmc_start, gga_start - rmc_start);
            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) {
                s_write(&(cte->transmission), PROTOCOL_OK, strlen(PROTOCOL_OK));
                cte->timestamp = time(NULL);
                cte->nmea.checksum_passed = 1;

                if(!extract_nmea_data(cte)){
                    return 1;
                }

                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_sem));

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

                /* If everyone is ready, process data */
                if(ready) {
                    apply_filters();
                }
                /* Releasing ready-lock */
                sem_post(&(s_synch->ready_sem));
            } else {
                cte->nmea.checksum_passed = 0;
                t_print("RMC and GGA received from %d , checksum failed!\n", cte->client_id);
                s_write(&(cte->transmission), ERROR_CHECKSUM_FAILED, strlen(ERROR_CHECKSUM_FAILED));
            }
        }

        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_LOADKRLDATA) {
            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_lsf_data(candidate)) {
                    s_write(&(cte->transmission), PROTOCOL_OK, sizeof(PROTOCOL_OK));
                } else {
                    s_write(&(cte->transmission),ERROR_LSD_LOAD_FAILED,
                            sizeof(ERROR_LSD_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) {
            print_cfd(cte, cte->cm.id_parameter);
        }

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

session.h

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

int respond(struct client_table_entry *cte);

#endif /* !SESSION_H */
```

actions.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 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 KRL_FILENAME "ls_data_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 KRL_DATA_ENTRIES 8

/* HELP */
#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"\
" LOADLSFDATA  | LLSFD | 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_sem));
    sem_wait(&(s_synch->ready_sem));
    client->marked_for_kick = 1;
    sem_post(&(s_synch->ready_sem));
    sem_post(&(s_synch->client_list_sem));
}

/* 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\n",
                                    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;
    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)
{
    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.ls_f.lsf_d.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_sem));

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

    /* Releasing lock */
    sem_post(&(s_synch->csac_sem));
    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 ls filter data into the client struct.
* Re-using the config loader.
* This whole function needs some work! Magic numbers beware.
*/
int load_lsf_data(struct client_table_entry* target)
{
    /* Request lock */
    sem_wait(&(s_synch->client_list_sem));
    sem_wait(&(s_synch->ready_sem));
    struct config_map_entry conf_map[KRL_DATA_ENTRIES];

    int filename_length = strlen(KRL_FILENAME) + 10;
    char filename[filename_length];
    memset(filename,'\0' ,filename_length);
    strcpy(filename, KRL_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.ls_f.lsf_d.alt_ref;

    conf_map[1].entry_name = LON_REF;
    conf_map[1].modifier = FORMAT_DOUBLE;
    conf_map[1].destination = &target->fs.ls_f.lsf_d.lon_ref;

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

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

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

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

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

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

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

    /* releasing lock */
```

```
    sem_post(&(s_synch->ready_sem));
    sem_post(&(s_synch->client_list_sem));
    return load_config_status;
}
```

actions.h

```
/**
 * @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
#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
 *
```

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

/** @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);

/** @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_lsf_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 */
```

utils.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);
```

```
}

/*
* 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;
    int file_len;
    char *input_buffer;

    int status = 0;

    config_file=fopen(path, "r");
    if(!config_file) {
        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;
            }
        }
        else{
            return 0;
        }
        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;
        }

        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);
}

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;
}
```

utils.h

```
/**
 * @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
 *    @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
 *
```

```c
 *     @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 */
```

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);
}
```

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 */
```

csac_filter.c

```
#include "csac_filter.h"

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

/* CONFIG CONSTANTS */
#define CONFIG_CFD_PATH "cfd_state_path: "
#define CONFIG_INIT_FROM_FILE "init_cfd_from_file: "
#define CONFIG_TELEMETRY_LOG "telemetry_log: "
#define CONFIG_LOG_PREDICTION "log_predict: "
#define CONFIG_LOG_PRED_PATH "pred_log_path: "
#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 14

#define ALARM_FAST_TIMING_FILTER " [ ALARM ] Phase > Limit\n"
#define ALARM_STEER_TO_BIG " [ ALARM ] Steer > limit!\n"
#define ALARM_FREQ_COR_FILTER " [ ALARM ] Steer > predicted!\n"

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

static double get_mjdf()
{
    double mjd_today = 0;
    const int BUFFER_LEN = 100;
    char buffer[BUFFER_LEN];
    memset(buffer, '\0', BUFFER_LEN);
    if(!get_today_mjd(buffer)) {
        t_print("get_mjdf(): Failed to calculate current MJD\n");
        return 0;
    } else {
        if(sscanf(buffer, "%lf", &mjd_today) == EOF) {
           t_print("get_mjdf(): Failed to cast MJD to float\n");
           return 0;
        }
    }
    return mjd_today;
}

static int load_telemetry(struct csac_model_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;
        }
    }

    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 = get_mjdf();
    if(!mjd_today){
        return 0;
    }

    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++;
    }

    // Initializing 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_model_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_model_data *cfd)
{
    if ( abs(cfd->steer_current -
             cfd->steer_prediction) >
            cfd->cf_conf.steer_limit) {
        return 1;
    }
    return 0;
}

static void update_model(struct csac_model_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_model_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_model(struct csac_model_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_model(struct csac_model_data
```

```
                       *cfd, char *telemetry)
{
    /* Load new telemetry into the filter */
    if(!load_telemetry(cfd, telemetry) ) {
        fprintf(stderr,"Telemetry failed to load\n");
        return 0;
    }

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

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

        /* Update prediction */
        update_model(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);
        }
    }
    return 1;
}

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

    conf_map[1].entry_name = CONFIG_INIT_FROM_FILE;
    conf_map[1].modifier = FORMAT_INT;
    conf_map[1].destination = &cf_conf->init_cfd_from_file;

    conf_map[2].entry_name = CONFIG_INIT_SSC;
    conf_map[2].modifier = FORMAT_DOUBLE;
    conf_map[2].destination = &cf_conf->init_cfd_ssc;

    conf_map[3].entry_name = CONFIG_INIT_SST;
    conf_map[3].modifier = FORMAT_DOUBLE;
    conf_map[3].destination = &cf_conf->init_cfd_sst;

    conf_map[4].entry_name = CONFIG_INIT_SSP;
```

```
    conf_map[4].modifier = FORMAT_DOUBLE;
    conf_map[4].destination = &cf_conf->init_cfd_ssp;

    conf_map[5].entry_name = CONFIG_PHASE_LIMIT;
    conf_map[5].modifier = FORMAT_DOUBLE;
    conf_map[5].destination = &cf_conf->phase_limit;

    conf_map[6].entry_name = CONFIG_STEER_LIMIT;
    conf_map[6].modifier = FORMAT_DOUBLE;
    conf_map[6].destination = &cf_conf->steer_limit;

    conf_map[7].entry_name = CONFIG_TIME_CONSTANT;
    conf_map[7].modifier = FORMAT_DOUBLE;
    conf_map[7].destination = &cf_conf->time_constant;

    conf_map[8].entry_name = CONFIG_WARMUP_DAYS;
    conf_map[8].modifier = FORMAT_INT;
    conf_map[8].destination = &cf_conf->warmup_days;

    conf_map[9].entry_name = CONFIG_INIT_SSY;
    conf_map[9].modifier = FORMAT_DOUBLE;
    conf_map[9].destination = &cf_conf->init_cfd_ssy;

    conf_map[10].entry_name = CONFIG_PRED_LIMIT;
    conf_map[10].modifier = FORMAT_DOUBLE;
    conf_map[10].destination = &cf_conf->pred_limit;

    conf_map[11].entry_name = CONFIG_TELEMETRY_LOG;
    conf_map[11].modifier = FORMAT_STRING;
    conf_map[11].destination = &cf_conf->telemetry_log_path;

    conf_map[12].entry_name = CONFIG_LOG_PREDICTION;
    conf_map[12].modifier = FORMAT_INT;
    conf_map[12].destination = &cf_conf->pred_logging;

    conf_map[13].entry_name = CONFIG_LOG_PRED_PATH;
    conf_map[13].modifier = FORMAT_STRING;
    conf_map[13].destination = &cf_conf->pred_log_path;
}

void steer_csac(int prediction)
{
    /* 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, "%d",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);


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

```c

    /* Adjusting frequency according to the models prediction */
    run_command(steer_com_string, program_buf);

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

    char log_buf[200];
    memset(log_buf, '\0', 200);
    strcat(log_buf, " Steer: ");
    strcat(log_buf, pred_string);
    strcat(log_buf, ", response: ");
    strcat(log_buf, program_buf);

    /* Logging steer value */
    log_to_file(s_conf->log_path, log_buf, 2);
}

void disable_csac_disc()
{
    /* Allocating buffer for run_program() */
    char program_buf[200];
    memset(program_buf, '\0', 200);

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

    /* Disabling disciplining */
    run_command("python query_csac.py Md",
                program_buf);

    fprintf(stderr,"Disabling CSAC disciplining: %s \n", program_buf);

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

void enable_csac_disc()
{
    /* Allocating buffer for run_program() */
    char program_buf[200];
    memset(program_buf, '\0', 200);

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

    /* Disabling disciplining */
    run_command("python query_csac.py MD",
                program_buf);

    fprintf(stderr,"Enabling CSAC disciplining: %s \n", program_buf);

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

int check_filters(struct csac_model_data *cmd)
{
    if(freq_cor_filter(cmd)){
        log_to_file(s_conf->log_path, ALARM_FREQ_COR_FILTER, 2);
        return 1;
    }
```

```
    /* If current steer is bigger than the predicted limit */
    if( abs(cfd->steer_current) > cfd->cf_conf.pred_limit ){
        log_to_file(s_conf->log_path, ALARM_STEER_TO_BIG, 2);
        return 1;
    }

    if(fast_timing_filter(cfd->phase_current, cfd->cf_conf.phase_limit)){
        log_to_file(s_conf->log_path, ALARM_FAST_TIMING_FILTER, 2);
        return 1;
    }

    return 0;
}

int start_csac_model(struct csac_model_data
                      *cfd)
{
    int raised_alarm = 0;
    int csac_disc = 1;

    /* Allocating buffer for run_program() */
    char program_buf[200];
    memset(program_buf, '\0', 200);
    int model_init = 0;

    /* csac_filter config */
    struct config_map_entry
        conf_map[CSAC_FILTER_CONFIG_ENTRIES];

    /* Initialize config map */
    initialize_config(conf_map, &cfd->cf_conf);

    /* Load the config */
    if(!load_config(conf_map, CSAC_FILTER_CONFIG_PATH,
                CSAC_FILTER_CONFIG_ENTRIES)){
        t_print("CSAC model/filter: Failed to load config\n");
        s_synch->done = 1;
        return -1;
    }

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

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

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

        /* Initialize model if not already initialized */
        if(!model_init) {
            model_init = init_csac_model(cfd, program_buf);
        }

        /* checking alarm */
        if(cfd->days_passed >= cfd->cf_conf.warmup_days){
            raised_alarm = check_filters(cfd);
        }
```

```
        /* If the alarm is raised */
        if(raised_alarm){
            if(csac_disc){
                disable_csac_disc();
                csac_disc = 0;
            }

            /* Get mjd to update filter */
            double mjd_today = get_mjdf();

            /* Calculating MJD */
            cfd->t_current = mjd_today;

            /* Calc steer predict */
            int steer_pred = (int)get_steer_predict(cfd);
            steer_pred = steer_pred * 1000;

            /* Steering CSAC */
            steer_csac(steer_pred);
        }

        /* If the alarm is not raised */
        if(!raised_alarm){
            if(!csac_disc){
                enable_csac_disc();
                csac_disc = 1;
            }
            update_csac_model(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_state_path);

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

csac_filter.h

```
/**
 * @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>
```

```c
#include <stdarg.h>
#include <errno.h>
#include <unistd.h>
#include "utils.h"
#include "serial.h"

#include "sensor_server.h"

struct csac_map_config {
    int pred_logging;
    char cfd_state_path[PATH_LENGTH_MAX];
    char telemetry_log_path[PATH_LENGTH_MAX];
    char pred_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_model_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;

```

```c
    /* fast timing filter status */
    int ftf_status;

    /* Frequency correction filter status */
    int fqf_status;

    /* Config */
    struct csac_map_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_model(struct csac_model_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_model(struct csac_model_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_model_data *cfd);

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

#endif /* !CSAC_FILTER_H */
```

cfilter_config.ini

```
cfd_state_path: cfd_state.txt
telemetry_log: telemetry.txt
pred_log_path: pred.txt
log_predict: 0
init_cfd_from_file: 0
init_cfd_steer_smooth_current: -49.134799
init_cfd_steer_smooth_today: -43.508808
init_cfd_steer_smooth_previous: -49.135013
init_cfd_steer_smooth_yesterday: -43.091711
phase_limit: 50
steer_limit: 50
time_constant: 10000
warmup_days: 0
```

```
pred_limit: 200
```

get_telemetry.py

```python
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()
```

filters.c

```c
#include "filters.h"

#define ALARM_RDF "[ ALARM ] Sensor %d triggered LS filter!\n"
#define ALARM_RDF_RETURNED "[ ALARM ] Sensor %d cleared LS filter!\n"

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

/** @brief Checks for any "moving" SENSORS
*
* Checks solved position against known position.
* Known position loaded from the config file.
* @return Void
*/
static void ls_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
 */
static void raise_alarm(void);
```

```c

static int log_alarm(int client_id, char *alarm)
{
    /* allocating memory for string */;
    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 ls_filter */
            if(iterator->fs.ls_f.moved == 1) {
                iterator->fs.ls_f.was_moved = 1;
                iterator->fs.ls_f.moved = 0;
                if(s_conf->logging) {
                    log_alarm(iterator->client_id, ALARM_RDF);
                }
            } else {
                if(iterator->fs.ls_f.was_moved) {
                    iterator->fs.ls_f.was_moved = 0;
                    if(s_conf->logging) {
                        log_alarm(iterator->client_id, ALARM_RDF_RETURNED);
                    }
                }
            }
        }
    }
}

void ls_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.ls_f.lsf_d.lat_ref +
                iterator->fs.ls_f.lsf_d.lat_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.lat_disturbed = HIGH;
        } else if(iterator->nmea.lat_current < iterator->fs.ls_f.lsf_d.lat_ref -
                  iterator->fs.ls_f.lsf_d.lat_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.lat_disturbed = LOW;
        } else {
```

```
            iterator->fs.ls_f.dv.lat_disturbed = SAFE;
        }

        if(iterator->nmea.alt_current > iterator->fs.ls_f.lsf_d.alt_ref +
                iterator->fs.ls_f.lsf_d.alt_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.alt_disturbed = HIGH;
        } else if(iterator->nmea.alt_current < iterator->fs.ls_f.lsf_d.alt_ref -
                  iterator->fs.ls_f.lsf_d.alt_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.alt_disturbed = LOW;
        } else {
            iterator->fs.ls_f.dv.alt_disturbed = SAFE;
        }

        if(iterator->nmea.lon_current > iterator->fs.ls_f.lsf_d.lon_ref +
                iterator->fs.ls_f.lsf_d.lon_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.lon_disturbed = HIGH;
        } else if(iterator->nmea.lon_current < iterator->fs.ls_f.lsf_d.lon_ref -
                  iterator->fs.ls_f.lsf_d.lon_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.lon_disturbed = LOW;
        } else {
            iterator->fs.ls_f.dv.lon_disturbed = SAFE;
        }

        if(iterator->nmea.speed_current > iterator->fs.ls_f.lsf_d.speed_ref +
                iterator->fs.ls_f.lsf_d.speed_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.speed_disturbed = HIGH;
        } else if(iterator->nmea.speed_current < iterator->fs.ls_f.lsf_d.speed_ref -
                  iterator->fs.ls_f.lsf_d.speed_dev) {
            iterator->fs.ls_f.moved = 1;
            iterator->fs.ls_f.dv.speed_disturbed = LOW;
        } else {
            iterator->fs.ls_f.dv.speed_disturbed = SAFE;
        }
    }
}

void apply_filters()
{
    ls_filter();
    raise_alarm();
}
```

filters.h

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

```c

void apply_filters();

#endif /* !ANALYZER_H */
```

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);
}
```

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);
```

```c

#endif /* !NET_H */
```

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;
}
```

serial.h

```c
/*
## 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>
#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 */
```

colors.h

```
#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 */
```

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 */
```

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

```

```c
#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;

    /* 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 */
```

list.h

```c
/**
 * @author kazutomo@mcs.anl.gov
 * @file list.h
 * @brief Linked list implementation from linux kernel source code.
 *
 * This code was lifted from http://www.mcs.anl.gov/~kazutomo/list/.
 * I stumbled upon when writing when writing a Linux clone autumn 15',
 * and tested it in this project. It was planned to be replaced by something
 * smaller.
 * Kazutomo's description:
 *
 * I grub it from linux kernel source code and fix it for user space
```

```
 * program. Of course, this is a GPL licensed header file.
 *
 * Here is a recipe to cook list.h for user space program
 *
 * 1. copy list.h from linux/include/list.h
 * 2. remove
 *     - #ifdef __KERNE__ and its #endif
 *     - all #include line
 *     - prefetch() and rcu related functions
 * 3. add macro offsetof() and container_of
 *
 * - kazutomo@mcs.anl.gov
 */

#ifndef _LINUX_LIST_H
#define _LINUX_LIST_H

/**
 * @name from other kernel headers
 */
/*@{*/

/**
 * Get offset of a member
 */
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

/**
 * Casts a member of a structure out to the containing structure
 * @param ptr        the pointer to the member.
 * @param type       the type of the container struct this is embedded in.
 * @param member     the name of the member within the struct.
 *
 */
#define container_of(ptr, type, member) __extension__({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
/*@}*/


/*
 * These are non-NULL pointers that will result in page faults
 * under normal circumstances, used to verify that nobody uses
 * non-initialized list entries.
 */
#define LIST_POISON1  ((void *) 0x00100100)
#define LIST_POISON2  ((void *) 0x00200200)

/**
 * Simple doubly linked list implementation.
 *
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */
struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

```c

#define LIST_HEAD(name) \
	struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
	(ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_add(struct list_head *new,
			      struct list_head *prev,
			      struct list_head *next)
{
	next->prev = new;
	new->next = next;
	new->prev = prev;
	prev->next = new;
}

/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static inline void list_add(struct list_head *new, struct list_head *head)
{
	__list_add(new, head, head->next);
}

/**
 * list_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
	__list_add(new, head->prev, head);
}

/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
	next->prev = prev;
	prev->next = next;
}
```

```c

/**
 * list_del - deletes entry from list.
 * @entry: the element to delete from the list.
 * Note: list_empty on entry does not return true after this, the entry is
 * in an undefined state.
 */
static inline void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    //entry->next = LIST_POISON1;
    //entry->prev = LIST_POISON2;
}



/**
 * list_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
static inline void list_del_init(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    INIT_LIST_HEAD(entry);
}

/**
 * list_move - delete from one list and add as another's head
 * @list: the entry to move
 * @head: the head that will precede our entry
 */
static inline void list_move(struct list_head *list, struct list_head *head)
{
    __list_del(list->prev, list->next);
    list_add(list, head);
}

/**
 * list_move_tail - delete from one list and add as another's tail
 * @list: the entry to move
 * @head: the head that will follow our entry
 */
static inline void list_move_tail(struct list_head *list,
                                  struct list_head *head)
{
    __list_del(list->prev, list->next);
    list_add_tail(list, head);
}

/**
 * list_empty - tests whether a list is empty
 * @head: the list to test.
 */
static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}

static inline void __list_splice(struct list_head *list,
                                 struct list_head *head)
{
    struct list_head *first = list->next;
```

```
    struct list_head *last = list->prev;
    struct list_head *at = head->next;

    first->prev = head;
    head->next = first;

    last->next = at;
    at->prev = last;
}

/**
 * list_splice - join two lists
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
static inline void list_splice(struct list_head *list, struct list_head *head)
{
    if (!list_empty(list))
        __list_splice(list, head);
}

/**
 * list_splice_init - join two lists and reinitialise the emptied list.
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 *
 * The list at @list is reinitialised
 */
static inline void list_splice_init(struct list_head *list,
                                    struct list_head *head)
{
    if (!list_empty(list)) {
        __list_splice(list, head);
        INIT_LIST_HEAD(list);
    }
}

/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:    the type of the struct this is embedded in.
 * @member:    the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

/**
 * list_for_each    -    iterate over a list
 * @pos:    the &struct list_head to use as a loop counter.
 * @head:    the head for your list.
 */

#define list_for_each(pos, head) \
  for (pos = (head)->next; pos != (head);    \
       pos = pos->next)

/**
 * __list_for_each    -    iterate over a list
 * @pos:    the &struct list_head to use as a loop counter.
 * @head:    the head for your list.
 *
 * This variant differs from list_for_each() in that it's the
```

```
 * simplest possible list iteration code, no prefetching is done.
 * Use this for code that knows the list to be very short (empty
 * or 1 entry) most of the time.
 */
#define __list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

/**
 * list_for_each_prev    -    iterate over a list backwards
 * @pos:    the &struct list_head to use as a loop counter.
 * @head:    the head for your list.
 */
#define list_for_each_prev(pos, head) \
    for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
            pos = pos->prev)

/**
 * list_for_each_safe    -    iterate over a list safe against removal of list entry
 * @pos:    the &struct list_head to use as a loop counter.
 * @n:        another &struct list_head to use as temporary storage
 * @head:    the head for your list.
 */
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); \
        pos = n, n = pos->next)

/**
 * list_for_each_entry    -    iterate over list of given type
 * @pos:    the type * to use as a loop counter.
 * @head:    the head for your list.
 * @member:    the name of the list_struct within the struct.
 */
#define list_for_each_entry(pos, head, member)                \
    for (pos = list_entry((head)->next, typeof(*pos), member);    \
         &pos->member != (head);                     \
         pos = list_entry(pos->member.next, typeof(*pos), member))

/**
 * list_for_each_entry_reverse - iterate backwards over list of given type.
 * @pos:    the type * to use as a loop counter.
 * @head:    the head for your list.
 * @member:    the name of the list_struct within the struct.
 */
#define list_for_each_entry_reverse(pos, head, member)            \
    for (pos = list_entry((head)->prev, typeof(*pos), member);    \
         &pos->member != (head);     \
         pos = list_entry(pos->member.prev, typeof(*pos), member))

/**
 * list_prepare_entry - prepare a pos entry for use as a start point in
 *            list_for_each_entry_continue
 * @pos:    the type * to use as a start point
 * @head:    the head of the list
 * @member:    the name of the list_struct within the struct.
 */
#define list_prepare_entry(pos, head, member) \
    ((pos) ? : list_entry(head, typeof(*pos), member))

/**
 * list_for_each_entry_continue -    iterate over list of given type
 *            continuing after existing point
 * @pos:    the type * to use as a loop counter.
```

```
 * @head:    the head for your list.
 * @member:    the name of the list_struct within the struct.
 */
#define list_for_each_entry_continue(pos, head, member)         \
    for (pos = list_entry(pos->member.next, typeof(*pos), member);    \
         &pos->member != (head);    \
         pos = list_entry(pos->member.next, typeof(*pos), member))

/**
 * list_for_each_entry_safe - iterate over list of given type safe against removal of list entry
 * @pos:    the type * to use as a loop counter.
 * @n:        another type * to use as temporary storage
 * @head:    the head for your list.
 * @member:    the name of the list_struct within the struct.
 */
#define list_for_each_entry_safe(pos, n, head, member)            \
    for (pos = list_entry((head)->next, typeof(*pos), member),    \
        n = list_entry(pos->member.next, typeof(*pos), member);    \
         &pos->member != (head);                    \
         pos = n, n = list_entry(n->member.next, typeof(*n), member))

/**
 * list_for_each_entry_safe_continue -    iterate over list of given type
 *            continuing after existing point safe against removal of list entry
 * @pos:    the type * to use as a loop counter.
 * @n:        another type * to use as temporary storage
 * @head:    the head for your list.
 * @member:    the name of the list_struct within the struct.
 */
#define list_for_each_entry_safe_continue(pos, n, head, member)         \
    for (pos = list_entry(pos->member.next, typeof(*pos), member),         \
        n = list_entry(pos->member.next, typeof(*pos), member);        \
         &pos->member != (head);                        \
         pos = n, n = list_entry(n->member.next, typeof(*n), member))

/**
 * list_for_each_entry_safe_reverse - iterate backwards over list of given type safe against
 *                        removal of list entry
 * @pos:    the type * to use as a loop counter.
 * @n:        another type * to use as temporary storage
 * @head:    the head for your list.
 * @member:    the name of the list_struct within the struct.
 */
#define list_for_each_entry_safe_reverse(pos, n, head, member)        \
    for (pos = list_entry((head)->prev, typeof(*pos), member),    \
        n = list_entry(pos->member.prev, typeof(*pos), member);    \
         &pos->member != (head);                    \
         pos = n, n = list_entry(n->member.prev, typeof(*n), member))




/*
 * Double linked lists with a single pointer list head.
 * Mostly useful for hash tables where the two pointer list head is
 * too wasteful.
 * You lose the ability to access the tail in O(1).
 */

struct hlist_head {
    struct hlist_node *first;
};
```

```
struct hlist_node {
    struct hlist_node *next, **pprev;
};

#define HLIST_HEAD_INIT { .first = NULL }
#define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
#define INIT_HLIST_NODE(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)

static inline int hlist_unhashed(const struct hlist_node *h)
{
    return !h->pprev;
}

static inline int hlist_empty(const struct hlist_head *h)
{
    return !h->first;
}

static inline void __hlist_del(struct hlist_node *n)
{
    struct hlist_node *next = n->next;
    struct hlist_node **pprev = n->pprev;
    *pprev = next;
    if (next)
        next->pprev = pprev;
}

static inline void hlist_del(struct hlist_node *n)
{
    __hlist_del(n);
    n->next = LIST_POISON1;
    n->pprev = LIST_POISON2;
}


static inline void hlist_del_init(struct hlist_node *n)
{
    if (n->pprev)  {
        __hlist_del(n);
        INIT_HLIST_NODE(n);
    }
}

static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first)
        first->pprev = &n->next;
    h->first = n;
    n->pprev = &h->first;
}



/* next must be != NULL */
static inline void hlist_add_before(struct hlist_node *n,
                                    struct hlist_node *next)
{
    n->pprev = next->pprev;
```

```
    n->next = next;
    next->pprev = &n->next;
    *(n->pprev) = n;
}

static inline void hlist_add_after(struct hlist_node *n,
                                   struct hlist_node *next)
{
    next->next = n->next;
    n->next = next;
    next->pprev = &n->next;

    if(next->next)
        next->next->pprev  = &next->next;
}



#define hlist_entry(ptr, type, member) container_of(ptr,type,member)

#define hlist_for_each(pos, head) \
    for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
         pos = pos->next)

#define hlist_for_each_safe(pos, n, head) \
    for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
         pos = n)

/**
 * hlist_for_each_entry    - iterate over list of given type
 * @tpos:    the type * to use as a loop counter.
 * @pos:    the &struct hlist_node to use as a loop counter.
 * @head:    the head for your list.
 * @member:    the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry(tpos, pos, head, member)             \
    for (pos = (head)->first;                          \
         pos && ({ prefetch(pos->next); 1;}) &&             \
        ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
         pos = pos->next)

/**
 * hlist_for_each_entry_continue - iterate over a hlist continuing after existing point
 * @tpos:    the type * to use as a loop counter.
 * @pos:    the &struct hlist_node to use as a loop counter.
 * @member:    the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry_continue(tpos, pos, member)         \
    for (pos = (pos)->next;                          \
         pos && ({ prefetch(pos->next); 1;}) &&             \
        ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
         pos = pos->next)

/**
 * hlist_for_each_entry_from - iterate over a hlist continuing from existing point
 * @tpos:    the type * to use as a loop counter.
 * @pos:    the &struct hlist_node to use as a loop counter.
 * @member:    the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry_from(tpos, pos, member)             \
    for (; pos && ({ prefetch(pos->next); 1;}) &&             \
        ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
```

```
        pos = pos->next)

/**
 * hlist_for_each_entry_safe - iterate over list of given type safe against removal of list entry
 * @tpos:    the type * to use as a loop counter.
 * @pos:    the &struct hlist_node to use as a loop counter.
 * @n:        another &struct hlist_node to use as temporary storage
 * @head:    the head for your list.
 * @member:    the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry_safe(tpos, pos, n, head, member)          \
    for (pos = (head)->first;                          \
         pos && ({ n = pos->next; 1; }) &&                 \
        ({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
         pos = n)


#endif
```

protocol.h

```
#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_LOADKRLDATA "LOADLSFDATA"
#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_LOADKRLDATA_SHORT "LSFD"
#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_LOADKRLDATA    20
#define CODE_PRINTCFD       21

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

#endif /* !PROTOCOL_H */
```

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 -g -std=gnu99 -pedantic

cpu := $(shell uname -m)

ifeq (£(cpu),armv7l)
    CFLAGS = -Wall -Wextra -c -std=gnu99 -pedantic -g -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
```

Appendix E

Scripts

CSAC query source code

```
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
        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()
```

MJD calculator

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

import datetime
import jdutil
from dateutil import parser

today = datetime.datetime.utcnow()
print(jdutil.jd_to_mjd(jdutil.datetime_to_jd(today)))
```

Script example

```
'''
:Author: Aril Schultzen
:Email: aschultzen@gmail.com
'''
# This script attempts to connect to the
# Sensor Server at <ip> : "port" and
# IDs itself as <id>. It will then
# poll the time solved by the GNSS receiver
# connected to Sensor<id> until
# terminated.

import socket
import sys
import time

ip = "10.1.0.46"
port = 10001
id = 1

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

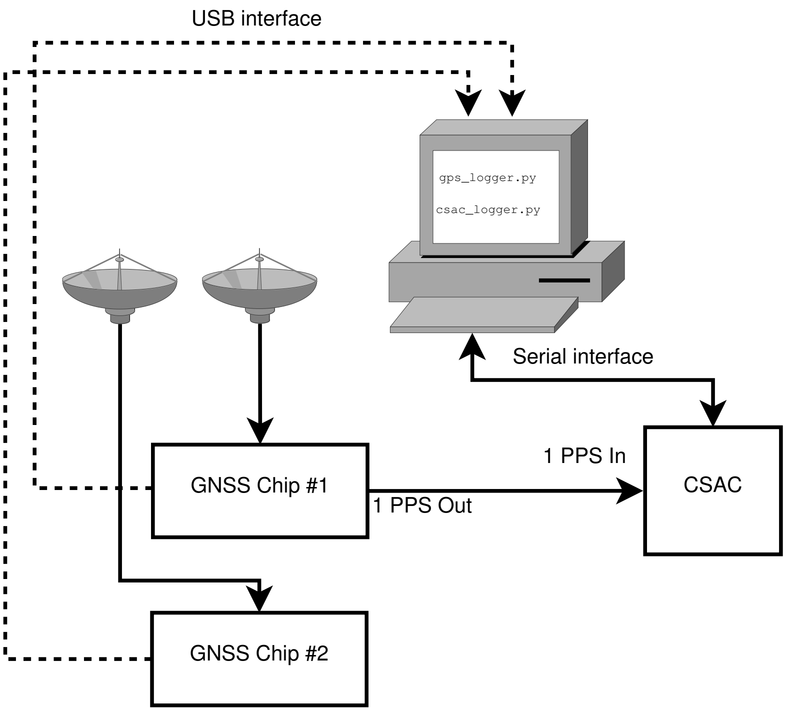
```
except socket.error, msg:
    print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
    sys.exit();
try:
    remote_ip = socket.gethostbyname( ip )

except socket.gaierror:
    print 'Could not resolve hostname'
    sys.exit()

s.connect((remote_ip , port))
s.sendall(b'IDENTIFY -10')
recv_buff = s.recv(1024)

while(1):
    s.sendall(b'PRINTTIME' + str(id))
    time.sleep(0.1)
    recv_buff = s.recv(1024)
    recv_buff = recv_buff.strip('>\n')
    print("Sensor " + str(id) + " GNSS solved time: " + recv_buff)
    time.sleep(0.9)
```

E.1 Logger setup schematic



Appendix F

E-mails

F.1 Correspondence with Mr. Davis

Hi Aril,

This would be fine, but you may want to take a look at the Astropy library
and see if their time package would meet your needs. It's certain to be
more robust and well tested. But if you'd like to use my module, please do.
http://docs.astropy.org/en/stable/time/index.html

Best,
Matt Davis

On Sun, Oct 23, 2016 at 2:25 PM Aril Schultzen <aschultzen@gmail.com> wrote:

> Hi!
>
> I am currently writing my master thesis in compsci and I wanted to ask you
> if it was OK if I used your library for converting dates to/from JD and MJD
> (https://gist.github.com/jiffyclub/1294443) in my implementation? It
> will be used to convert time to MJD for a model and also for stamping logs.
> Your work will of course be acknowledge as your own.
>
> Kind regards
>
> Aril Schultzen
>

Appendix G

Figures

Figure G.1: Photograph of the system used to measure the 10 MHz output from the atomic clock. Not in the picture is the source of the 10 MHz reference.

Figure G.2: Antenna covered in aluminium foil to simulate a jamming attack.

Complete Bibliography

[1] 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.

[2] Yilu Liu et al. "State Estimation and Voltage Security Monitoring Using Synchronized Phasor Measurement". In: (2001).

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

[4] Todd E. Humphreys Daniel P. Shepard and Aaron A. Fansler. "Going Up Against Time". In: *GPS World* (2012).

[5] Navigation National Coordination Office for Space-Based Positioning and Timing. *Timing*. `http://www.gps.gov/applications/timing/`. Accessed: 16-04-2015.

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

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

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

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

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

[11] 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.

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

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

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

[15] National Marine Electronics Association. *NMEA 0183 standard*. http://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp. Accessed: 23-10-2016.

[16] 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.

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

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

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

[20] 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.

[21] u-blox. UBX-13003221 - R11. Accessed: 18-10-2016. URL: https://www.u-blox.com/sites/default/files/products/documents/u-blox8-M8_ReceiverDescrProtSpec_(UBX-13003221)_Public.pdf?utm_source=en/images/downloads/Product_Docs/u-bloxM8_ReceiverDescriptionProtocolSpec_(UBX-13003221)_Public.pdf.

[22] 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.

[23] The GNOME Project. Accessed: 02-10-2015. URL: https://wiki.gnome.org/Home3.

[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.

[26] Norgeskart.no. Accessed: 30-10-2016. URL: http://www.norgeskart.no/?sok=Fetveien%2099,%20skedsmo#15/278837/6655264/+hits.

[27] M. Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010. ISBN: 9781593272913. URL: https://books.google.no/books?id=Ps2SH727eCIC.

[28] *The Kernel Bug Tracker*. Accessed: 02-03-2016. URL: https://bugzilla.kernel.org/show_bug.cgi?id=8691.

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

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

[31] 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.

[32] Matt Davis. *Functions for converting dates to/from JD and MJD*. Accessed: 02-09-2016. URL: https://gist.github.com/jiffyclub/1294443.