



ÉCOLE NATIONALE
DES SCIENCES
GÉOGRAPHIQUES



Kartverket

Internship report

Cycle: 3rd year of engineering, GTSI specialisation

Developing a graphical user interface for the real-time ionosphere scintillation monitor



Clément Drouadaine

September 2015

Not confidential Confidential IGN Confidential industry Until...

ÉCOLE NATIONALE DES SCIENCES GÉOGRAPHIQUES
6-8 Avenue Blaise Pascal - Cité Descartes - 77420 Champs-sur-Marne
Phone +33 1 64 15 31 00 - Fax +33 1 64 15 31 07

Jury

President of jury:

Pierre-Yves HARDOUIN

Host Company:

Kartverket, Norwegian Mapping Authority

Internship training supervisors:

Oddgeir KRISTIANSEN, NMA
Emmanuel BARDIÈRE, ENSG
Vincent DE OLIVEIRA, ENSG

Educational person in charge of the engineering course:

Serge BOTTON, IGN/ENSG/DE/DPTS

School's professional training counsellor:

Patricia PARISI, IGN/ENSG/DE/DSHEI

Internship from 05/05/2015 to 25/09/2015

Web diffusion: Internet Intranet ENSG

Document situation:

Internship report presented in the end of the 3rd year of engineer cycle

Number of pages: 39 pages including 5 annexes

Host system: L^AT_EX

*It's still magic even if you know
how it's done.*

A Hat Full of Sky, TERRY PRATCHETT

Acknowledgements

I would like to thank my internship supervisor in the Norwegian Mapping Authority (NMA), Rune Hanssen, for all the help and support he provided me all along the internship, and his availability whenever I needed him.

Thanks also to the whole of the geodesy department team for their welcome and especially Yngvild Linnea Andalsvik for her help, and Geir Arne Hjelle for the non work-related times we had.

Last but not least, I would like to give a big thanks to Zuheir Altamimi to whom I resorted once more to find me this internship. Like last year, I contacted him after unfruitful researches on my own, and he managed to find me an internship where I wanted, in the field I wanted, and within 24 hours. Once more, thank you very much.

Résumé

Le Moniteur de Scintillation Ionosphérique en Temps Réel (RTIS) est un logiciel développé par l'Autorité Norvégienne de Cartographie (NMA) pour mesurer et quantifier la scintillation ionosphérique, dont un des effets est de diminuer la qualité des observations GNSS. Il est déployé dans douze stations d'observation, répartis dans toute la Norvège et la Mer de Norvège.

La but de ce stage était de concevoir et de développer une interface graphique ergonomique qui pourrait être utilisée par la NMA pour surveiller toutes les stations du réseau RTIS en temps réel et modifier ou remplacer leurs fichiers de configuration.

Mots clés : interface, web, surveillance.

Abstract

The Real-Time Ionospheric Scintillation Monitor (RTIS) is a software developed by the Norwegian Mapping Authority (NMA) to measure and quantify the ionospheric scintillation, whose main effects are to lessen the quality of GNSS observations. It is deployed in twelve observation stations, spread all across Norway and the Norwegian Sea.

The goal of this internship was to conceive and develop an ergonomic Graphical User Interface (GUI) that could be used in the NMA to monitor all the stations of the RTIS network in real-time and to edit or replace their configuration files.

Key words: interface, web, monitoring.

Contents

Glossary and useful acronyms	7
Introduction	9
1 Definition of the objectives	11
1.1 What RTIS is: a brief overview	11
1.1.1 Space weather and scintillations	11
1.1.2 RTIS receivers	12
1.1.3 Operational environment	12
1.2 What exists to monitor it: description of the existing	14
1.3 What needs to be done: goals and expected functionalities	15
1.3.1 Visualising data	15
1.3.2 Modifying the RTIS behaviour	16
1.3.3 The problem of the bandwidth	16
1.3.4 Miscellaneous	17
2 The proposed solution	19
2.1 General considerations	19
2.2 Architecture	19
2.2.1 Global overview	19
2.2.1.1 A brief overview of the RTIS software	19
2.2.1.2 The general idea behind the architecture	20
2.2.2 In-depth descriptions of the modules	21
2.2.2.1 Command senders	21
2.2.2.2 Command reader & data sender	23
2.2.2.3 Data reader	24
2.2.2.4 A previous model	24
2.3 The GUI	26
2.3.1 General presentation	26
2.3.2 Receiving data	27
2.3.3 Sending & saving configuration files	29
Conclusion	33
A Example of a RTIS configuration file	37

List of Figures

1.1	Auroral oval (Credit: NASA)	11
1.2	Frequency of scintillations (P. Kintner, 2009)	12
1.3	Map of NMA scintillation receivers	13
1.4	Monitoring table as seen with rtis_mon	14
1.5	Statement Of Work, as present in the internship agreement	15
1.6	Use-case diagram of the GUI	16
2.1	General organisation of the solution, working document	20
2.2	Detailed organisation of the modules, working document	22
2.3	Explanation of the algorithm, transposed to Python pseudocode	25
2.4	Screenshot of the interface, data tab	26
2.5	Responsive design: minimised menu on the left, deployed menu on the right	27
2.6	The error messages panel	28
2.7	An example of a minimised panel	28
2.8	Screenshot of the config tab	29
2.9	Activity diagram of saving a file F in the database.	30
2.10	Screenshot of filling in the form for a configuration file, halfway through: successfully filled input fields are circled in green, errors are filled in red with a little explanation text.	31

Glossary and useful acronyms

CPU Central Processing Unit

DE Direction de l'Enseignement = Education Department

DPTS Département du Positionnement Terrestre et Spatial = Terrestrial & Spatial Positioning Department

DSHEI Département des Sciences Humaines, des relations Entreprises et des relations Internationales = Social Sciences, Industry relations and International relations Department

ENSG École Nationale des Sciences Géographiques = National School of Geographic Sciences

ERR ERror messages

FTP File Transfer Protocol

GPS Global Positioning System

GLONASS GLObal NAVigation Satellite System

GNSS Global Navigation Satellite System

GTSI Géomatique et Technologie des Systèmes d'Information = Geomatics and Technology of the Information Systems

GUI Graphical User Interface

IGN Institut Géographique National = National Geographical Institute

IP Internet Protocol

MON MONitoring table

NMA Norwegian Mapping Authority, the English name of Kartverket

OS Operating System

PHP PHP: Hypertext Preprocessor (recursive acronym)

PROC list of active PROCesses

RTIS Real-Time Ionosphere Scintillation

RTIM Real Time Ionosphere Monitor

SATREF SATellittbasert REFeransesystem = Satellite-based Reference System

SSH Secure SHell

TCP Transmission Control Protocol

UDP User Datagram Protocol

UML Unified Modelling Language

USB Universal Serial Bus

Introduction

DEFINITION OF THE OBJECTIVES

1.1 What RTIS is: a brief overview

1.1.1 Space weather and scintillations

The performance of GNSS systems can be severely affected by space weather. The main effects occur when the signals pass through the enhanced electron densities in the ionosphere. One of the main effects is a delay of the signal. This can to a certain degree be mitigated by dual frequency positioning. During high solar activity, the ionosphere can be highly variable and the formed irregularities generate rapid fluctuations in the amplitude and phase of signals: this is also known as scintillations. The signal is scattered in random directions while propagating through the ionosphere causing it to interfere with itself. Amplitude scintillation may cause deep signal fades that can interfere with a user's ability to receive GNSS signals. Phase scintillations can lead to increased phase noise, cycle slips, and even loss of lock if the phase fluctuations are too rapid for the receiver to track.

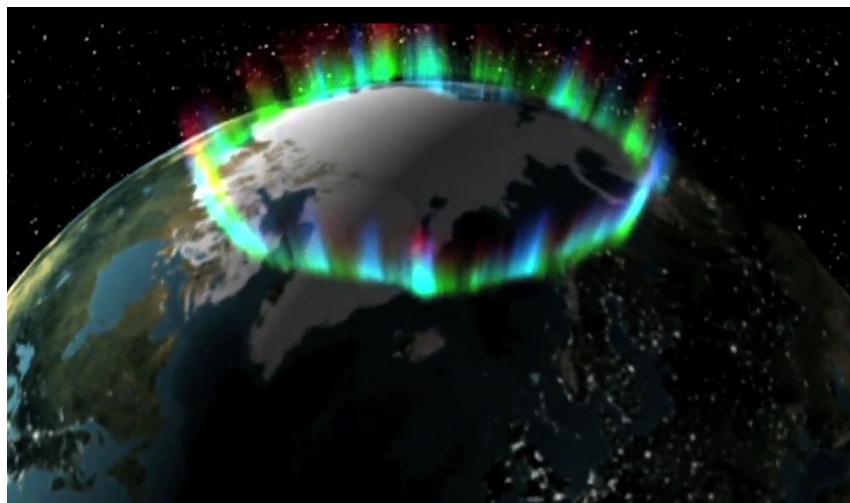


Figure 1.1: Auroral oval (Credit: NASA)

Equatorial latitudes and high latitude-regions are most severely affected by scintillations. Amplitude scintillations occur most commonly at equatorial latitudes, in particular the regions 15 degrees north and south of equator, while phase scintillations are more common at high latitudes. At high latitudes, scintillations are associated with processes in the polar cap and auroral oval. While the low latitude scintillations have clear seasonal and daily variations, the high latitude scintillations are more directly dependent on solar storms and the ionospheric activity that follows them.

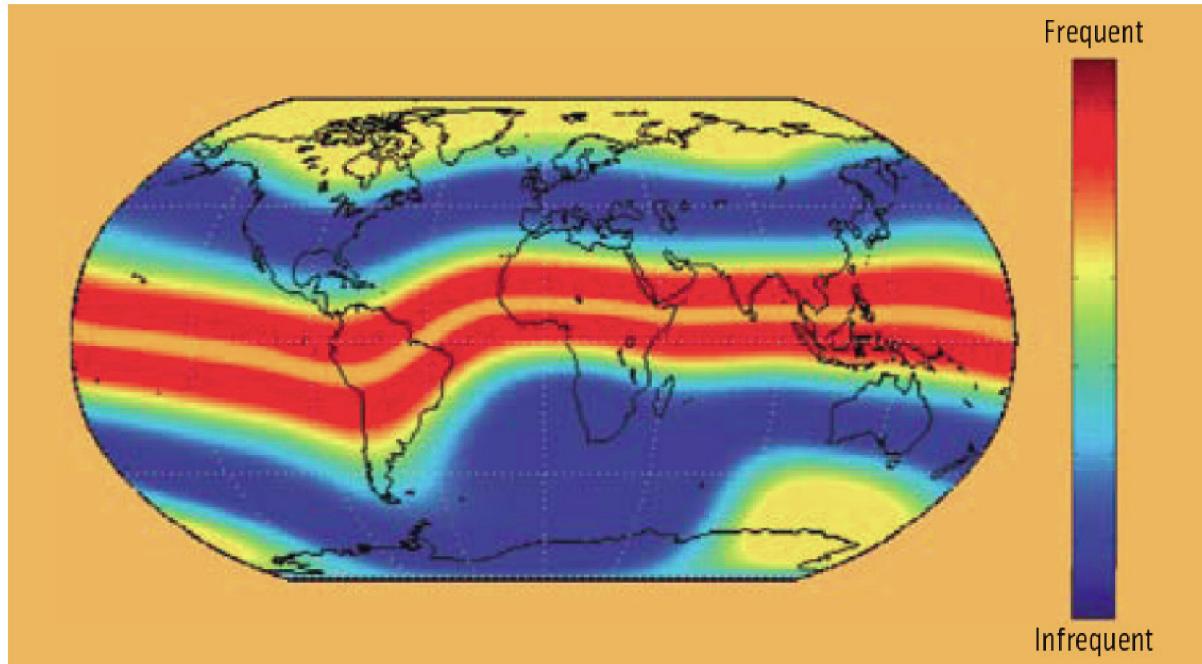


FIGURE 1 Scintillation map showing the frequency of disturbances at solar maximum. Scintillation is most intense and most frequent in two bands surrounding the magnetic equator, up to 100 days per year. At poleward latitudes, it is less frequent and it is least frequent at mid-latitude, a few to ten days per year.

Figure 1.2: Frequency of scintillations (P. Kintner, 2009)

The scintillations are often quantified by the scintillation indices S_4 and σ_φ for amplitude and phase scintillations respectively.

1.1.2 RTIS receivers

The Real-Time Ionospheric Scintillation Monitor (RTIS) at the NMA is capable of detecting the occurrence of both amplitude and phase scintillation by providing numerical data and maps of the S_4 and σ_φ indexes. Currently, the NMA operates 11 Septentrio PolaRxS scintillation receivers, deployed in Norway, in Iceland and in Faroe Islands. Additionally, one receiver is located at Hønefoss (NMA headquarters) for testing purposes. The receivers support multi-constellation and multi-frequency measurements, so that scintillation indexes are determined for both GPS and GLONASS data. Raw data measurements necessary for determining the scintillation indexes, such as signal phase, Doppler values, signal intensity and signal-to-noise ratio are carried out using a sampling rate of 100 Hz.

1.1.3 Operational environment

The software is running on Lenovo PCs with Linux Lubuntu 32 bit environments. In addition, an external USB hard drive (of at least 2TB) is connected to the PC. The external hard drive is used for the raw data transport to NMA.

The RTIS software stores the raw data from the receiver and calculates the scintillation indices S_4 and σ_φ in real time. The software tool performs the following main tasks:

-
- Calculate the scintillation indices S_4 and σ_φ indices in one minute resolution.
 - Produce a SATREF ¹ message with 1 Hz observation data and send it to Hønefoss.
 - Produce raw data files necessary for the determination of scintillation parameters.
 - Produce output files containing the scintillation indices.
 - Produce and send SATREF messages containing the scintillation indices and send them to RTIM ².

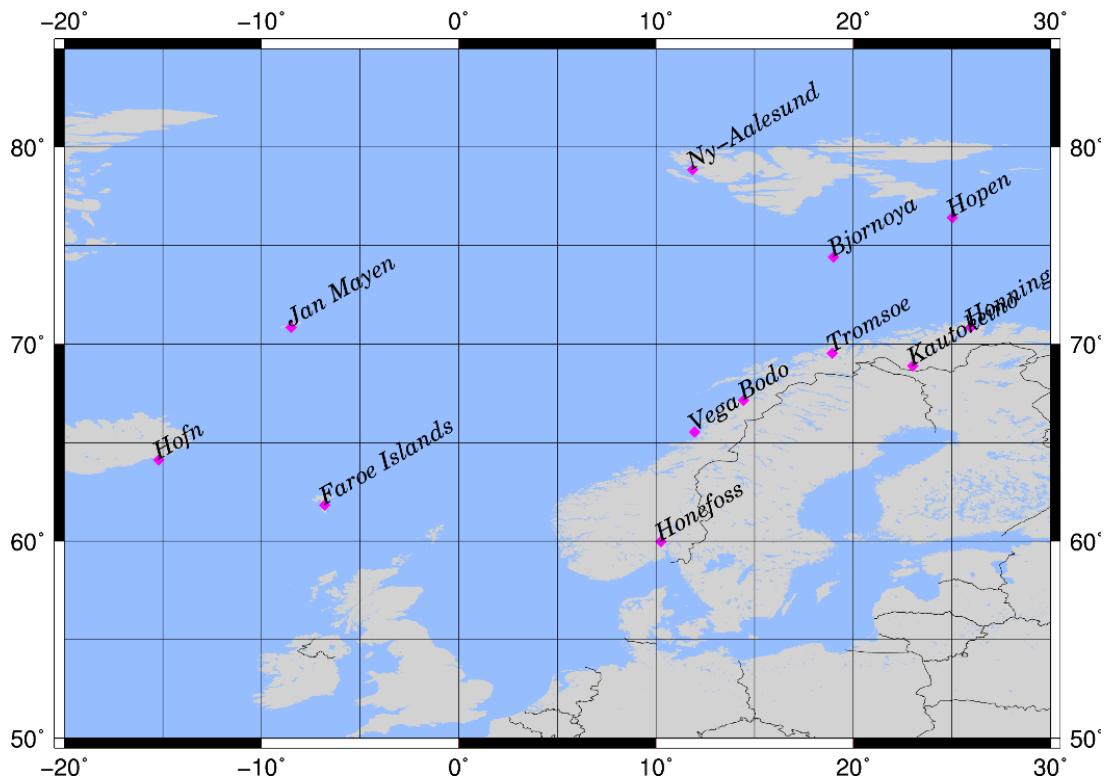
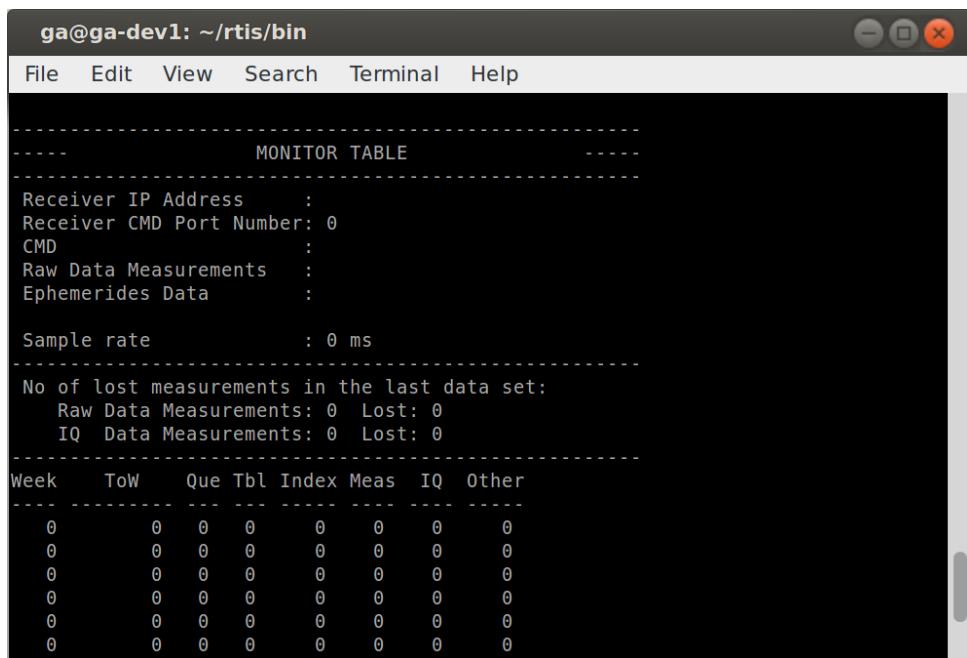


Figure 1.3: Map of NMA scintillation receivers

-
1. These messages are a NMA standard, can convey any kind of information, and aim to be as small as possible by converting all the values it contains in binary format. The NMA has developed high-level C libraries to write and read these messages.
 2. RTIM is the NMA server centralising all the RTIS data, located in Hønefoss.

1.2 What exists to monitor it: description of the existing

As we have seen, RTIS data transfers have been fully implemented, and do not need any further work. However, before my arrival at Karverket, there was only one way of monitoring a distant RTIS station: connecting to it via SSH, and manually launch a monitoring program, called *rtis_mon*, on the console. Depending on the parameter it is given, this displays a table containing either information about the running RTIS processes (option *-proc*) or different values relevant to monitoring the performances of RTIS (option *-mon*). This *rtis_mon* program will then run indefinitely until it is stopped, and update its values approximately once every second.



```
ga@ga-dev1: ~/rtis/bin
File Edit View Search Terminal Help

-----
          MONITOR TABLE
-----

Receiver IP Address      :
Receiver CMD Port Number: 0
CMD                      :
Raw Data Measurements    :
Ephemerides Data         :

Sample rate              : 0 ms

No of lost measurements in the last data set:
  Raw Data Measurements: 0 Lost: 0
  IQ Data Measurements: 0 Lost: 0

Week   ToW   Que Tbl Index Meas   IQ   Other
-----
0      0     0   0     0     0     0     0
0      0     0   0     0     0     0     0
0      0     0   0     0     0     0     0
0      0     0   0     0     0     0     0
0      0     0   0     0     0     0     0
0      0     0   0     0     0     0     0
```

Figure 1.4: Monitoring table as seen with *rtis_mon*

Although this method works, a few problems arise.

- Only one type of information can be displayed at once. It is obviously not possible to display several tables at the same time on a terminal, and opening several terminals and launching several instances of *rtis_mon* is not very ergonomic.
- The process of connecting to the distant RTIS station via SSH is not intuitive, and requires the user to have a list of the IP addresses of all the remote stations, to look through it to find the one he wants, and then to enter the command in a terminal. This does not allow to quickly browse through all stations or to easily switch between two of them.
- To modify the configuration file, the editing can be made locally and transferred by FTP to the remote site or remotely, using an editor on the remote computer. While manageable, this method is not optimal.
- The update frequency cannot be chosen by the user. Though it is not a big issue, it can be annoying for the list of active processes for example, which is not going to change a lot, and certainly not every second.

1.3 What needs to be done: goals and expected functionalities

The making of a new Graphical User Interface (GUI) obviously aims at solving the aforementioned problems. It should be a web application, preferably a light client that requires no installation whatsoever on the user side. The new GUI should allow the user to quickly see the status of every station, to connect to any of them, and to see any of the monitoring table (corresponding to the `-mon` option in `rtis_mon`), the list of active processes (`-proc`), or the error messages³ returned by the software, to reboot the whole RTIS software, or to change its configuration file. It combines both the status information obtained from the alarm system and the monitor data.

In the following, we will refer to the monitoring table as MON, the list of active processes as PROC, and the error messages as ERR.

The work entrusted to the trainee consists in:

Developing a web based GUI for the RTIS monitor. That will be installed in NMA main office in Hnefoss and must be able to connect to one or multiple (simultaneously) RTIS computers. The GUI will have to be used to configure and update the configuration file, to monitor processes and optionally download files. The monitoring can be based on values that already exist in the RTIS as "shared memory" tables or files and may include an overview of the processes currently running, error messages that are generated, and printing special monitor parameters. All this is dynamic information and information should be displayed in the GUI in near real time.

In addition, the web application should be made in PHP or in Java.

Figure 1.5: Statement Of Work, as present in the internship agreement

1.3.1 Visualising data

It should be possible to see any combination of MON, PROC and ERR simultaneously, and to have independent user-defined refresh frequency on each of them. For example, one should be able to choose to see both MON, refreshed every 2 seconds, PROC, refreshed every 3 minutes, and no error message. The "real-time" factor is not very important, as the data will not be subject to high-frequency changes. A delay of a few seconds is acceptable.

Displaying ERR should work differently than for PROC and MON: there will be no refresh frequency for it, and messages will always be displayed as soon as they are sent by the RTIS software. It should be possible, however, to filter them by their severity. Indeed, each error message within the RTIS software has a severity level. Ranked by order of magnitude, the possible levels are: 1. Fatal 2. Error 3. Warning 4. Info. There are also two special unranked levels, Debug and Notice. The user should be able to

3. The *error message* denomination encompasses all of the status messages returned by the applications within the RTIS system. The denomination can be misleading, since this also includes success messages.

select a severity level between 0 and 4, and to receive all error messages with a severity lower or equal to the selected level. Choosing the level 2, corresponding to Error for example, should make visible every message with a severity of Error or Fatal. The user should additionally be able to choose whether he wants any of Debug or Notice messages, regardless of the "main" severity filter he chose.

1.3.2 Modifying the RTIS behaviour

The GUI should not only be able to observe what the RTIS software does, but also to act on it in some ways: it should be possible to change the RTIS configuration file, and to reboot the whole RTIS software.

There are no specification on how modifying the configuration file should be handled. However, a new feature to add is a history of the previous configuration files. It should be possible to retrieve a previous configuration file, and to revert back to it. It should be noted that a change in the configuration file will not take effect until the RTIS is rebooted. It is thus possible to edit the configuration file several times in a row without actually changing anything, just to store the configuration files in the history for later. Again, there is no specification on how the storage should be done.

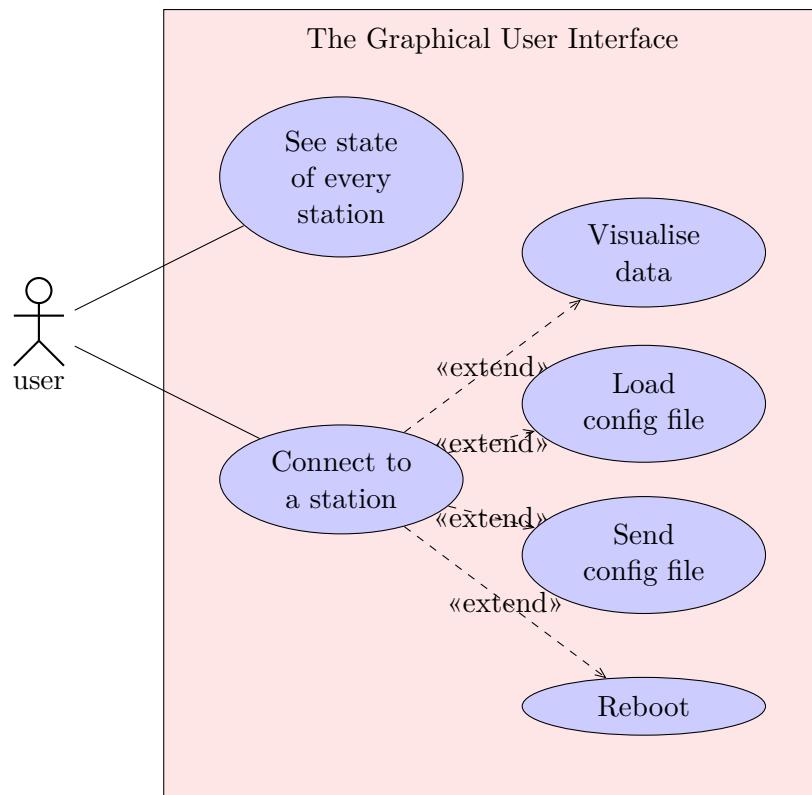


Figure 1.6: Use-case diagram of the GUI

1.3.3 The problem of the bandwidth

The stations of the RTIS network are scattered all across continental Norway, some are in Svalbard and there is even one in Iceland. All of these stations are located close to

inhabited locations and benefit from good and reliable connections to the internet.

But some stations have been placed in extremely remote locations, like Bjørnøya⁴ (*Bear Island* in Norwegian), which is not linked to any other place by submarine internet cables, and whose entire internet traffic transits through satellite connections. Due to the costs of bandwidth using these communications, the available bandwidth is very low, and due to the island being more than 74° north, even the satellite connection is unreliable and can be subject to unpredictable interruptions.

Moreover, there already is data transiting through this connection: the actual observation data that is sent to the calculation centre, RTIM. One of the requirements for the new RTIS GUI is thus to minimise the amount of data sent or received, to be sure that every message could be received on time, and would not conflict with the observation data for the bandwidth access. The number of exchanges and/or the size of messages exchanged between the GUI and the stations should be as small as possible.

It was also demanded that the remote RTIS station would somehow detect if the connection with the GUI had been broken, and would then stop sending data through a broken pipe. The user should still be able to cleanly disconnect from the station to stop receiving data from it, but in the event of a breakage of a connection, the station should also be able to detect it by itself and not indefinitely continue sending data while waiting for a disconnection signal from the GUI.

1.3.4 Miscellaneous

The website should be able to be visited on any kind of support, with any kind of OS: Linux and Windows computers, Android and iOS phones... This has technical implications and implies to check compatibility when choosing of technologies, and also influence the design of the web page, which should be responsive and adapt to mobile devices with a smaller screen.

About the languages to code the website in, the choice is limited to PHP or Java. Although some other technologies might be tempting, like Node.js for the real-time display, all the maintenance of my work is going to be done by the team behind RTIS, and they logically chose to limit the technologies involved to ones they were familiar with.

It is not required to have any login system, nor to be able to deal with simultaneous users on the GUI. It is only meant for an internal use, with restricted access. Anyone able to access the website is able to do all the interface allows to do. I am not aware of the inner workings of the internal network in NMA, and am only expected to have the product working on my development server. They will deal with the deployment.

4. see Fig. 1.2

THE PROPOSED SOLUTION

2.1 General considerations

The solution I have come up with is a light client: since there are no calculations nor any resource-consuming operation that is going to be needed, everything will be done on the web server without any risk of overloading it.

The website is coded in PHP, because it is a language I and the RTIS team are familiar with. Moreover, while environments like J2EE could be useful for bigger projects, this relatively simple interface does not require such a complex technology. As the RTIS program is written in C and includes some low level libraries for communications, all of the code behind the interface used to communicate with the RTIS system will be written in C as well.

Also, while the GUI can ask for the RTIS software to send data, it cannot directly fetch it: some modules have to be added directly in the RTIS software, in the distant stations, to be able to communicate with them.

2.2 Architecture

2.2.1 Global overview

The general architecture of my solution is presented in figure 2.1.

In white, we have a schematization of the existing RTIS software, which will be detailed, and in yellow are a simplified version of the modules and programs I should develop.

2.2.1.1 A brief overview of the RTIS software

First, let us go through the existing RTIS software to better understand how I will interact with it. Right on top, we have the GNSS receiver, whose role is obviously to collect data. This data is stored into one of the two data buffers, while the other one is being processed by the RTIS software. Every minute, the buffers are swapped. Every buffer contains one minute of observations, which implies that RTIS can take up to one minute to process one minute of data. If it takes longer, the buffers are swapped anyway, the remaining data is erased by new observations, and is thus lost. This can happen in period of strong ionospheric activity, hence the need for the modules I add to take as few resources as possible.

Just underneath the GNSS receiver, we have the Controller. It is responsible for managing all the modules that do the processing, ensuring that they are all up and running, and relaunching them if necessary. To reboot the whole RTIS system, one only needs to

reboot the Controller.

We then have a lot of independent processes that process data, store their intermediate or final results in a shared memory, and then send the results to a control center in NMA. We are only interested in some parts of the shared memory that we want to read. We do not need to know any more about the inner workings of the software or its data.

2.2.1.2 The general idea behind the architecture

We will here refer to the web server that hosts the GUI as "the server" and to the distant RTIS installation as "the client". The architecture is pretty straightforward: the

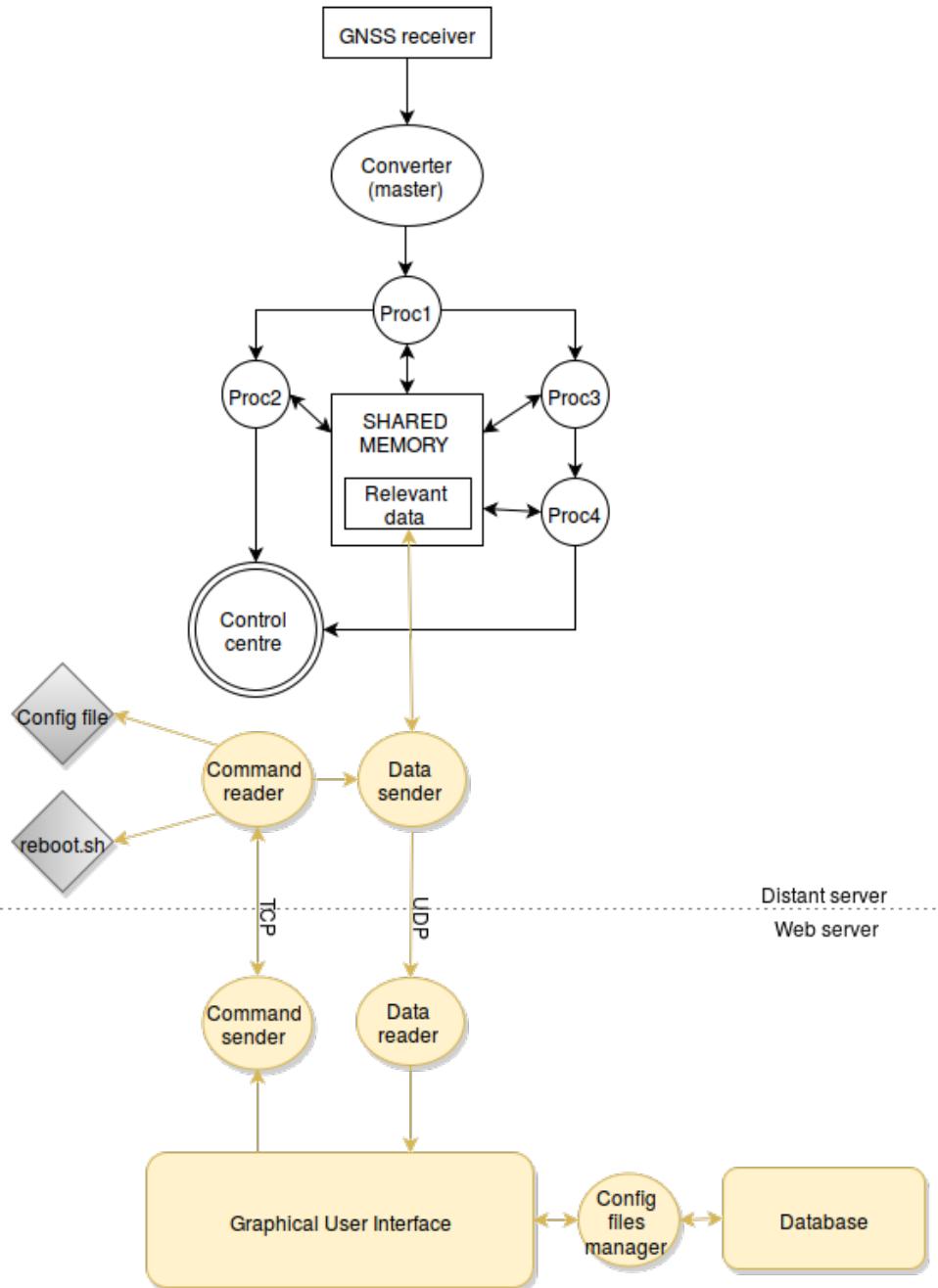


Figure 2.1: General organisation of the solution, working document

GUI can send commands to the client via the *Command Sender*, which are then read on the client by a dedicated *Command Reader*. Depending on the message, the *Command Sender* will either ask for data to be sent by the *Data Sender*, update the configuration file on the client, or launch a bash script to reboot the system. The *Data Sender* reads data from the RTIS shared memory according to what the *Command Reader* demands, and sends it back to the GUI, which then reads and displays it.

There are two types of connections that link the server to the client: the server sends its request with the TCP protocol, while the client sends its data using UDP. The reason for that is that the TCP protocol offers a very reliable connection: whenever a message is sent through a TCP stream, it is almost certain that it will be received on the other side. The receiver has to send a confirmation of the reception of the message, and if the sender does not receive the confirmation, it will send the message again. As we want to be sure that our commands have been sent, this is the solution to use here. The UDP protocol, on the other hand, does not perform any check, nor needs the receiver to actually be connected. It just sends data and hopes that it will be received on the other side. Although obviously less reliable than TCP, it is also much faster for real-time applications: if the message has not been received, there is no point in trying to re-send it as it is already obsolete, and we need to send new messages with the new data instead. What matters here is sending information quickly, and if some of it is lost, it is not a major problem.

About the history of configuration files, now. To limit the use of bandwidth, whenever the user wants to view the history of the configuration files for a station, or even the current configuration file, it is not efficient to fetch it from the client: even though these files only weigh 2 kB, this would result in unneeded bandwidth usage. A database has thus to be created on the server, where configuration files are duplicated whenever they are sent to the client. This way, we are ensured to always have all of the configuration files handy without any need for costly data transfer.

2.2.2 In-depth descriptions of the modules

First of all, all communications between client and server are made via SATREF messages. It is an efficient and easy-to-implement way of communication.

2.2.2.1 Command senders

Commands senders are a combination of PHP and C scripts. Their role is, as their name suggests, to convert user input from the GUI into a command, and to send it to the client. There is one command sender per type of command that can be sent: fetch ERR, fetch MON, fetch PROC, send configuration file, reboot. The command reader will make the difference between all of these types of messages based on their type, an integer placed into the header message. Thus, most of the messages going from the server to the client do not even need a body: they will just consist of a header whose only meaningful information will be the message type, the only exception to that being for configuration file updating. If the user has entered in the interface that he wants to see MON, refreshed every 7 seconds, the PHP script will just call, every 7 seconds, the C script whose job is just to create and send one specific message. And every 7 seconds, a message of type "MON" will be sent to the client to notify it that it needs to immediately send MON data.

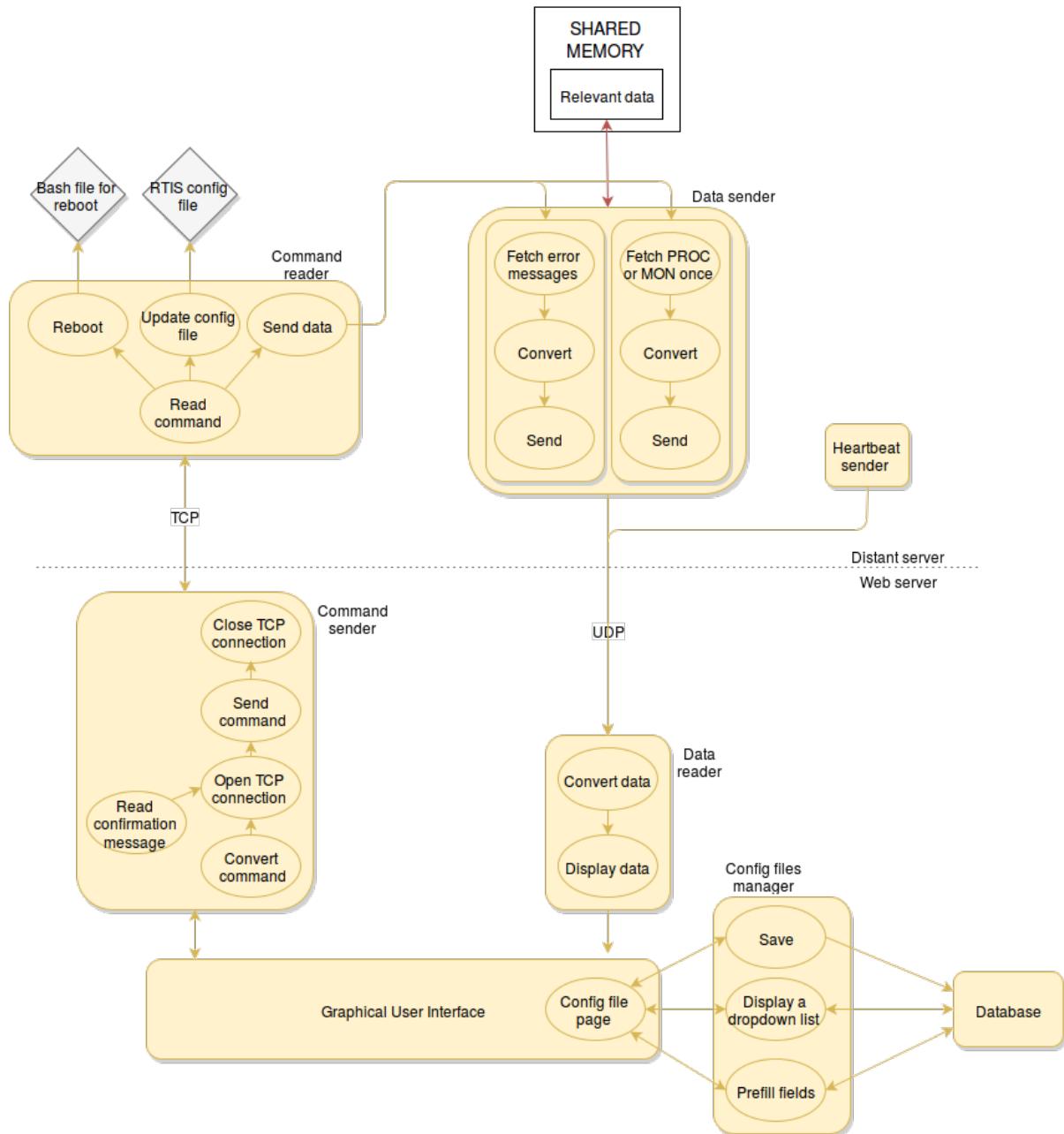


Figure 2.2: Detailed organisation of the modules, working document

It works the same way with PROC.

The "Reboot" command consists of a single message, exactly like for MON & PROC, though sent only once.

For ERR, however, one cannot simply choose the frequency of the updates: if ERR is enabled, messages will be sent as soon as they are emitted by the client. The technique of "sending a message per update" used for MON and PROC cannot be used anymore. Here, only one message will be sent every two minutes, regardless of what happens to the connection. When received, the *Data Sender* will send messages for the next two minutes. While not optimal, this method, like the one used for PROC & MON, fulfil the requirements: the messages sent from the server are extremely small in size, and if the

server-client connection is broken, the *Data Sender* will stop sending messages.

Finally, when sending a configuration file, the whole file is parsed, all of its values are extracted before being put, one after the other, into the SATREF message. The size of the message is greatly reduced compared to the file's one: only the values are taken into account, while the file has a lot of text that does not change from one file to the other and is thus useless to send. An example of a configuration file is given on Annexe A.

All of these messages need to be acknowledged by the receiver, who should send an empty SATREF message back at the sender. Only then would the command sender know that its command has been received and obeyed to. Should this confirmation message not be received, the command will be sent again, until a certain period of time is elapsed. If no answer is received by then, an error notification is sent to the GUI, that will display it. This whole process of confirming the messages is arguably redundant with the very idea of choosing a TCP connection, but we do not only want to be sure that the message has been received, we also need to be sure that it has been processed.

2.2.2.2 Command reader & data sender

The *Command Reader* consists of only one program, constantly running and listening for messages. The way things are done here are that the TCP server that listens to messages is set in "blocked" mode. Concretely, "non-blocked" would have meant having an infinite loop running at full speed and checking at every iteration for messages to read, thus asking a lot from the CPU, while "blocked" just puts the program to sleep and wakes it up whenever a message comes, which is a lot more efficient.

Once a message is received, the *Command Reader* adjusts its behaviour depending on the nature of the message, indicated by the "message type" contained in the header. If the message is asking for PROC or MON, it will read it from the shared memory, once, convert it into a SATREF message, send it to the *Data Reader* (via UDP) and send a confirmation to the *Command Sender* (via TCP), before putting itself to sleep once more and waiting for new messages. A "reboot" message will result in a system call to launch a bash file on the client, that will do the rebooting. As with any kind of received message, a confirmation is then sent to the command sender. For configuration files, the contents of the message are converted to C variables, a configuration file is written based on these variables, and is moved to erase the old file.

The tricky part here was for the error messages. Indeed, and unlike the other messages, what these commands ask for is not a "once only" behaviour. A single command asking for ERR means fetching it for the next 2 minutes. If it was left in the same flow of execution as the command reader, that would interrupt its normal behaviour and prevent it from reading any other command from the GUI while the errors are fetched. Moreover, the method for storing and reading the error messages within the RTIS software is a bit special. They are stored in a ringbuffer, which means that if the messages are not read quickly enough, they will get erased by new ones. To read that, a function exists within the RTIS libraries: once more, it is set to a "blocked" mode, which means that it will read new messages when they come, remove them from the ringbuffer, and go back to sleep immediately after.

To deal with the problem of the infinite loop of waiting for error messages within the infinite loop of waiting for GUI commands, I have resorted to a fork of the *Command Sender* program. The general idea is detailed in figure 2.3.

2.2.2.3 Data reader

The *Data Reader* are, like the *Command Senders*, making a link between the PHP-based GUI and the C-written RTIS software. As such, they are composed of both C and PHP codes. The C codes are just here to receive the messages using the SATREF library. When messages are received, they are either written into files that will be read by PHP or directly returned to the PHP programs. Each type of message then go through a different processing before being inserted and displayed in the GUI. Notably, it is in here that the error messages are selected based on their severity: all of the error messages from the client are received, while we only want to display the messages with certain severities.

2.2.2.4 A previous model

The solution proposed here is the result of numerous iterations, and there is one of them I would like to present here, as it has major differences to the selected one, and has not been obsoleted until a few months in the internship.

The idea behind it was to have as few messages sent by the GUI to the distant station as possible. And, for fetching data, this has a lot of implications; messages would only be sent whenever something was changed, and while no command was received by the *Command Reader*, it would continue to send data the same way it was. The commands sent by the GUI related to fetching data would then include a type of data and the frequency it should be fetched at (for PROC & MON) or the minimum severity level (for ERR). For ERR, this way of doing things allows to filter messages directly at the source, which is more bandwidth-efficient.

However, and while this method now seems better than the currently selected one, a few problems arose from it.

- This would have requested three *Data Senders*, running all the time, on the client. It is not very resource-friendly.
- It would have requested that when receiving commands to change the way data is fetched, the *Command Reader* notifies the relevant *Data Sender*, which should modify its inner parameters while it is still running. This would have called for much more serious and complex coding.
- The main problem would be detecting if the connection is broken: this aspect has not been mentioned yet, but it is crucial. It would have needed yet another module whose sole purpose would be to monitor the state of the TCP connection, and check that it is still operational. From what I have read, I have understood that it could only be done by sharing handshakes with the server: all the benefits of this model, the low number of exchanges, have just disappeared. This would, once more, have required complex, resource-consuming code, and would have resulted in no benefit when compared to the simpler (even maybe simplist) method that has been described earlier.

```

hasForked = false
#Beginning of the infinite message-reading loop
while(True)
    message = readMessages(blocked=true)

    #If the message asks for error messages
    if(message.Type == errType)
        #If the fork has not been done yet
        if(!hasForked)
            hasForked = true

            #Creation of a pipe, to communicate
            #between the two processes that
            #will result of the fork
            pipe = pipe()

            #Creates a duplicate of this
            #very program
            pid = fork()

            #If this is the child program
            #(child and parents are the
            #same, but we still need to
            #tell them appart)
            if(pid == 0)

                #The child enters this
                #infinite loop, and will
                #never leave it
                while(true)
                    date = now()

                    #reads messages for two minutes
                    while(now()-date < 2 minutes)
                        read & send error messages

                    #Pipe communications are in
                    #"blocked" mode. The child goes
                    #to sleep and waits for further
                    #instructions.
                    read(pipe,blocked=true)

            #If this is the parent
            else
                #awakens the child
                write(pipe, 1)

```

Figure 2.3: Explanation of the algorithm, transposed to Python pseudocode

2.3 The GUI

2.3.1 General presentation

The GUI had been designed using an existing template based on the Bootstrap framework. This front-end framework already implements a lot of useful features, like boxes, drop-down menus, grids... It allows developers to quickly build nice-looking web pages. This section aims at showcasing the design of the GUI, to explain the choices made regarding ergonomics, and to be a sort of "user documentation" when the previous section was more of a "programmer documentation".

First, let's have a look at the general design of the web page¹.

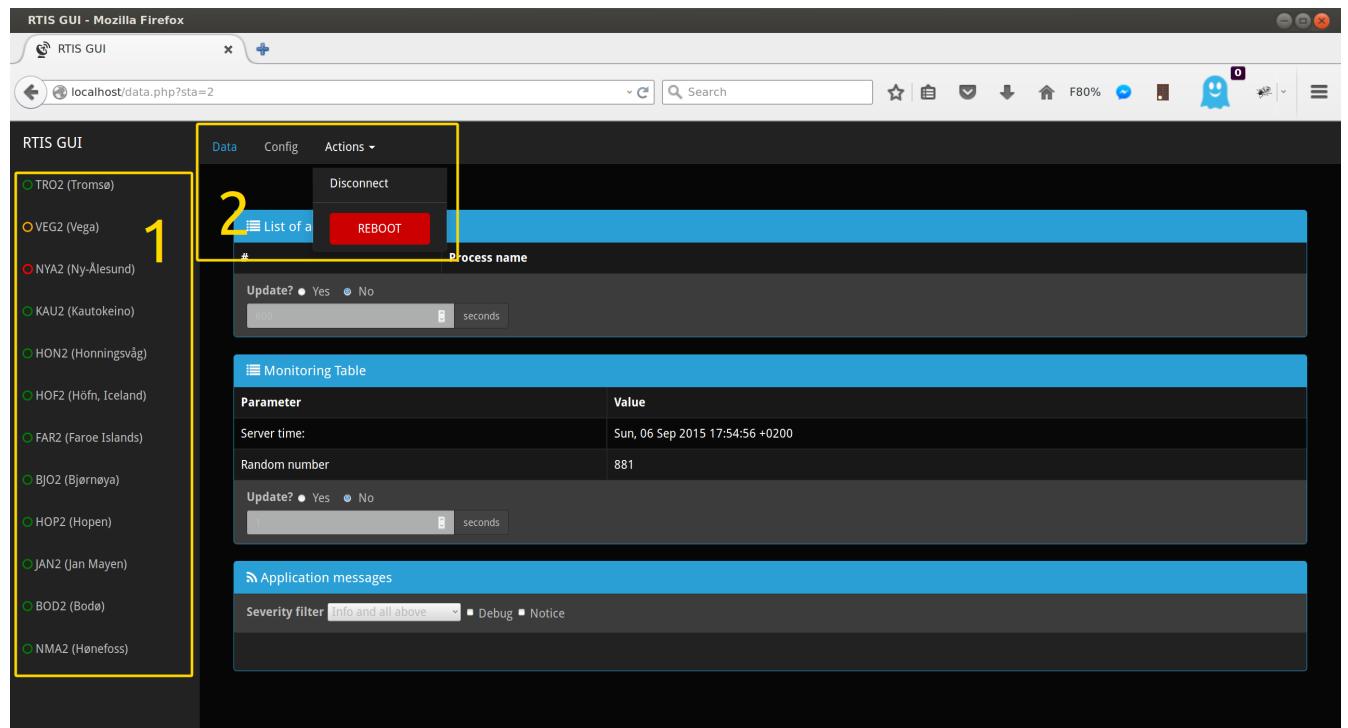


Figure 2.4: Screenshot of the interface, data tab

The two circled menus are present at all time when navigating the interface. The menu on the left, numbered 1, allows the user to choose the station he wants to monitor. It also indicate the state of every station thanks to the colour of the circles: green circles for healthy stations which send heartbeats with a "healthy" code, orange for non-healthy stations whose heartbeats indicate failure, and red in case of a total absence of heartbeat. The second menu, numbered 2, on top, is quite self-explanatory; once the user has chosen a station to connect to, it takes him to either the *Data* page, to visualise monitoring data, or to the *Config* page, for everything related to configuration files. The *Disconnect* option on the *Action* drop-down menu takes the user to the blank index page, and the *Reboot* option leaves the user where he is, in order for him to see the data beginning to display

1. This report has been written before the end of the internship. The data tab has not yet been linked to the C part, hence the random numbers and messages displayed in the different tables that are just here to give an idea of the final look of the GUI.

again when the reboot of the remote station is complete.

For mobile devices, the menu 1 with the list of stations is minimised in the top right corner of the screen, and deploys on click.

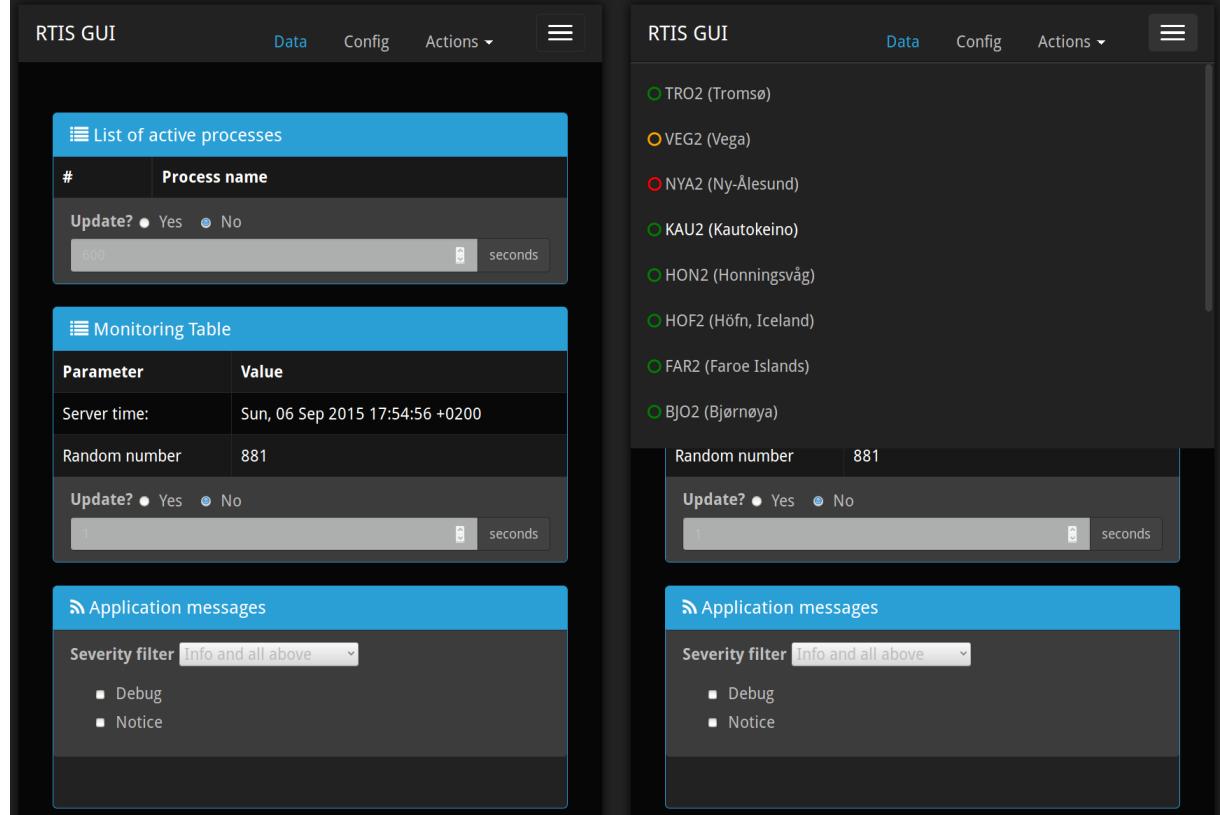


Figure 2.5: Responsive design: minimised menu on the left, deployed menu on the right

2.3.2 Receiving data

When monitoring a station, the first thing to do is to make sure it is correctly functioning. The data tab is therefore the first tab the user is taken to when connecting to a station. This page can display the list of running RTIS processes (PROC), the monitoring table (MON) and/or the error messages from the RTIS software (ERR).

Each of this type of information has its own panel, and can be updated separately from the others. For MON and PROC, it is possible to choose whether to update or not, and if yes, at which frequency, just under the display. The choice of colours for the page has been made so that the different areas (title, information, settings) are easily discernible, in an effort to be as intuitive as possible. For ERR, it is possible to choose from a drop-down list a minimum severity level under which the messages will not be displayed. A "no message" option is also available. Since they are different than the regular options, the "Debug" and "Notice" filters have their own check-box, and can be chosen regardless of the "regular" severity filter.

When the data tab is opened, it has been decided that PROC would receive only one update and be set to "no update", that MON would be set to one update per second, and

ERR would display all error messages except the special ones (Debug and Notice). The reason for not updating PROC by default is that it should not change: if everything is running like it should, there is no reason that a process would crash while we are monitoring the station. And if it is believed that it could happen, there is always the option to set the settings to "Update" manually.

The ERR panel will only display the ten most recent messages, and erases older messages as new ones arrive. This amount can easily be changed in the code, but given the frequency of the messages, it has been agreed that it should be enough. Each message is displayed with a colour and an icon depending on its severity. Please note that the example that follows has been done with dummy error messages composed of only random strings.

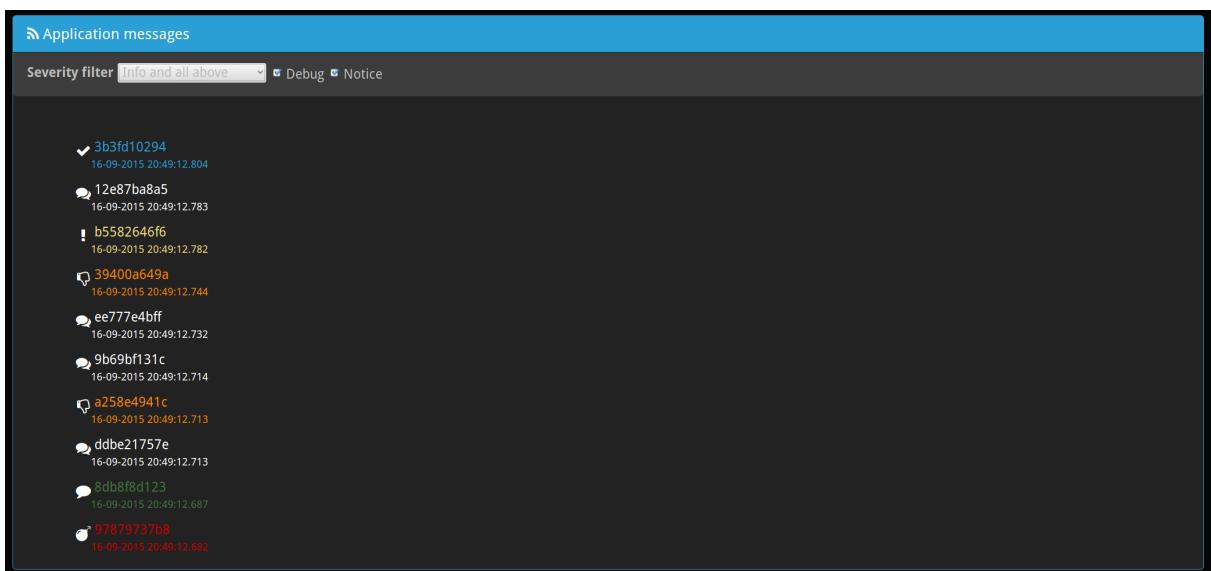


Figure 2.6: The error messages panel

As we can see here, the messages are displayed in a way that their content is on top, with the time the error arose just underneath, in an attempt to be easily readable and understandable.

Each of these panels can be minimised by clicking on the blue title bar, and makes it possible to hide informations. It is for example possible to minimise the middle panel, so as to see both the top and the bottom one simultaneously, which would not be possible otherwise.

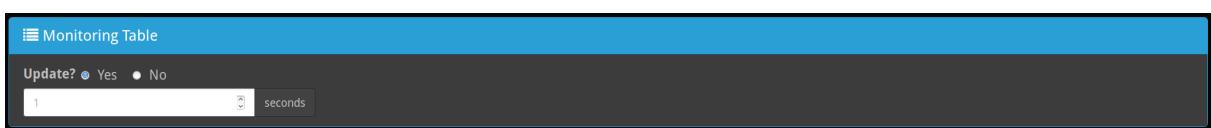


Figure 2.7: An example of a minimised panel

2.3.3 Sending & saving configuration files

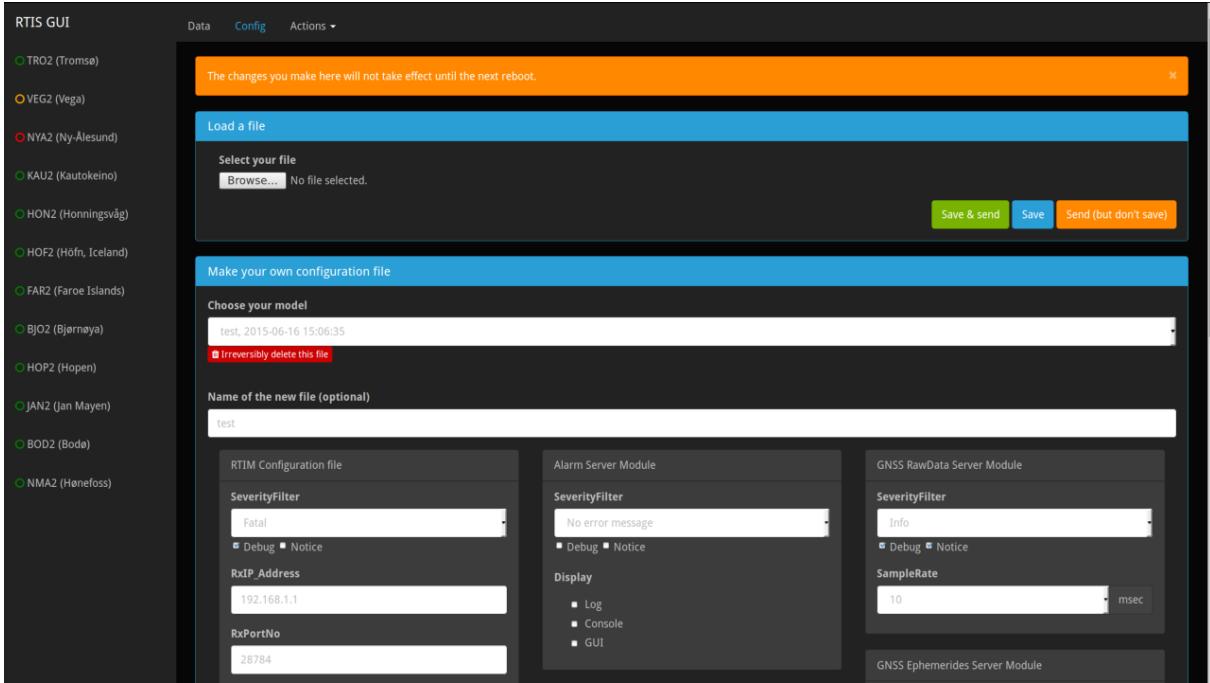


Figure 2.8: Screenshot of the config tab

The config tab is here to allow the user to send its own configuration files to the remote RTIS station. I chose to allow the file to be input either by a direct file upload or by a form that has to be manually completed. Indeed, it is possible that the user wants to send the same file for several stations, and filling the same (huge) form other and other again would be extremely counterproductive and show no advantage over the FTP deposit model that was the norm before the GUI was created. On the other hand, it is possible that the user only wants to make quick changes on a small number of parameters, and in this case opening the file in an external text editor, modifying it and uploading it would not be optimal. Moreover, the very idea of having to upload files from the device does not go well with the requirement of the web page being mobile-device friendly.

All the configuration files that are entered through this tool are stored into a database on the web server. This database only consists of one table, the configuration files, will not contain a lot of data, and is only ever asked to do simple read or write operations, without any join. I thus chose to use a basic mySQL relational database.

The idea is that whenever the user sends a configuration file to a station, this file has to be saved into the database, in order to be able to be reloaded later. When saving a file, it is possible to give it a name. While purely aesthetic, the name could make it easier to find it again later, instead of choosing a file based solely on its modification date. To transfer data from the GUI to the database, the user has to choose between three options:

- sending the file to the station, and not saving the file in the database. This is not recommended, as the GUI will only display files from the database and thus will not know what file is active: we would have no idea of what is currently running on the distant station;

- saving the configuration file in the database and sending the configuration file to the station, which is the option that should be used in most cases;
- just saving the file in the database without sending it, to continue working on it later.

The latter two options will call the “Save” function of the configuration file module. When saving a file in the database, a new field is added to it: the "active" attribute. It is used to know which file is the current file in the remote station.

The process of saving a file in the database is explained in a UML activity diagram, figure 2.9.

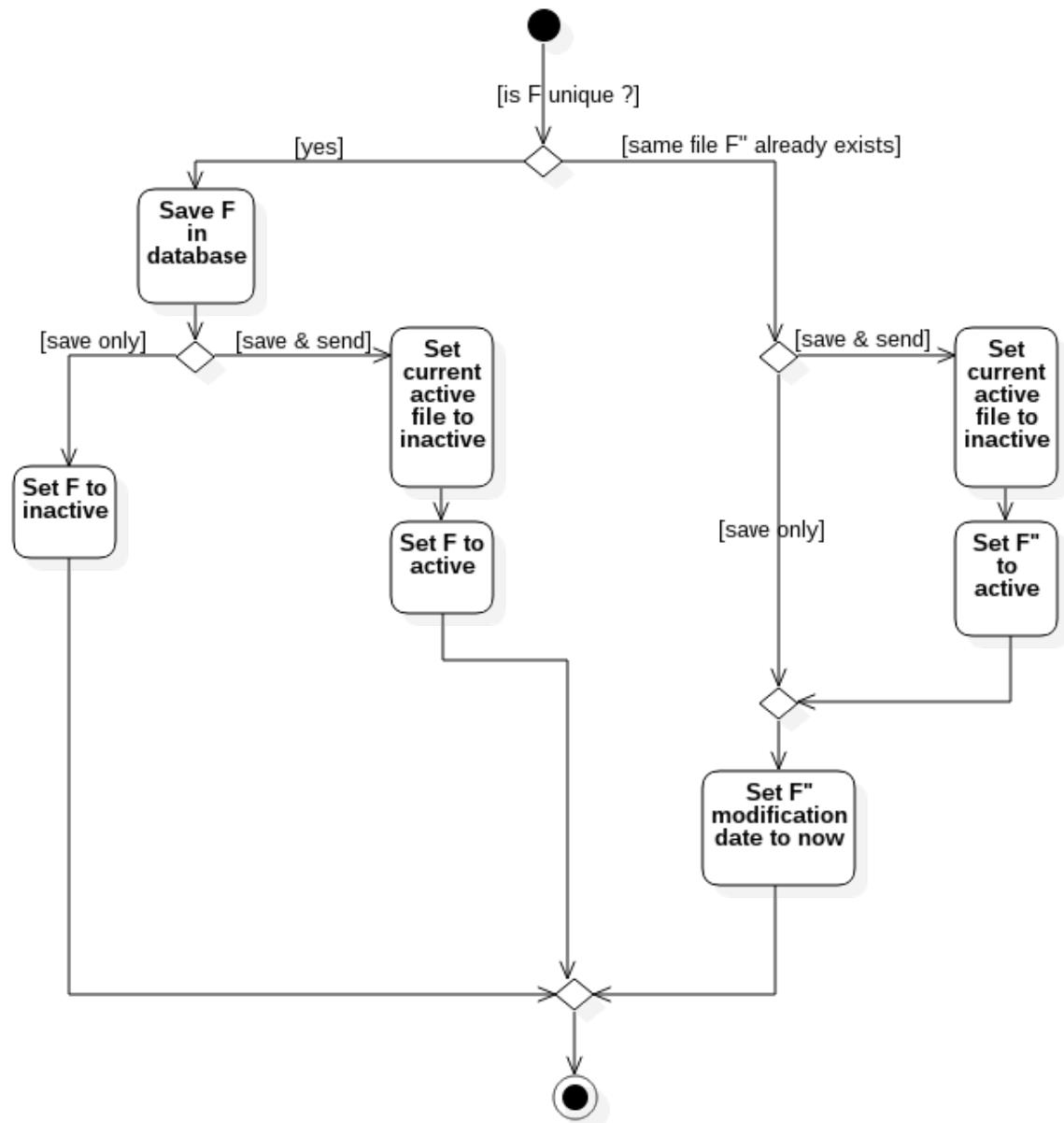


Figure 2.9: Activity diagram of saving a file F in the database.

Before saving a configuration file however, be it by direct file input or through the use of the form, all of the values it contains are checked, for malicious input, first, but

also to see if the values entered correspond to what is expected: is the port number an integer? Is it lower than 65536? If any of the value is incorrect, a message is printed on the interface explaining what the problem is.

When entering it through the form, the input fields are accepted or rejected as they are filled, with an explicit colour code, so that the user can see what is wrong in case of error, or have a confirmation that what he typed is acceptable.

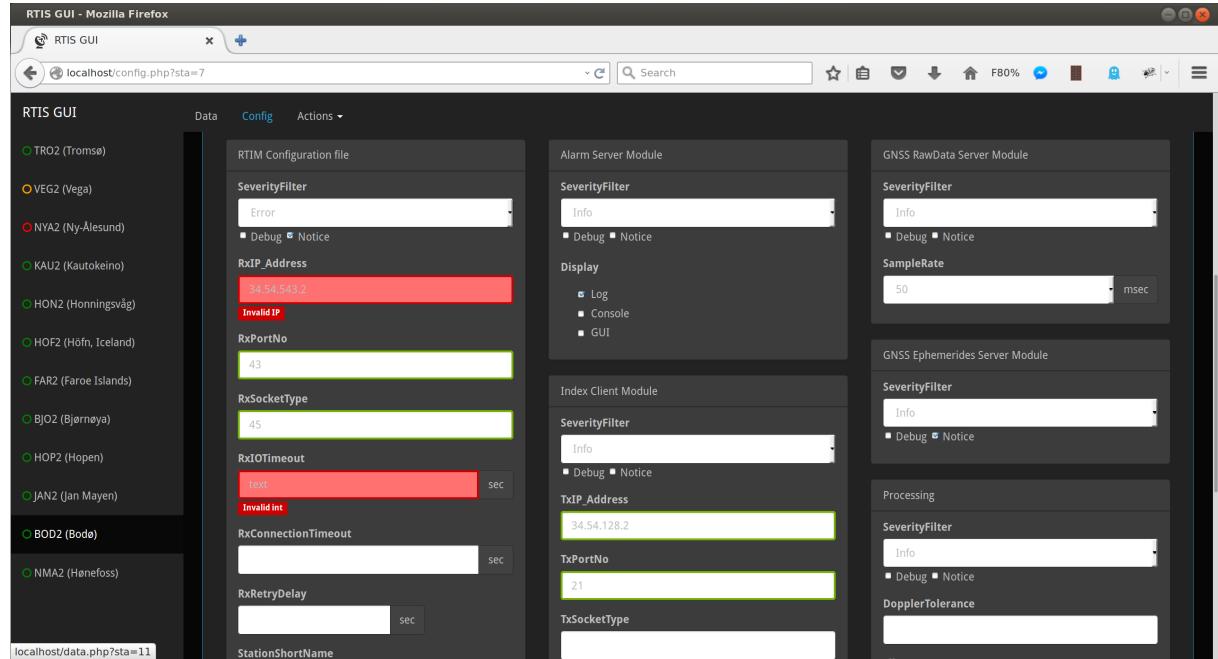


Figure 2.10: Screenshot of filling in the form for a configuration file, halfway through: successfully filled input fields are circled in green, errors are filled in red with a little explanation text.

For communication the other way around, the GUI calls the function “Display a drop-down list” whenever the user goes to the configuration file modification page. This function queries to the database to get the list of names of the configuration files for the current station, and formats it so that it can directly be included in the HTML code as a drop-down list. Once the drop-down list is made, the user can choose one of its elements, that is to say one of the previous configuration files. Choosing an element triggers the “Prefill fields” function, which queries the database for all of the information concerning the selected configuration file, and prefills the GUI fields accordingly. Initially, the fields are filled with the values from the current active configuration file on the RTIS installation, which is the only “active” file in the database for the current station. If no active file is found, then the latest modified station is taken.

It is also possible to remove a file from the database from the GUI by selecting it in the drop-down list and clicking on the “Delete” button. This calls the “Delete” function. For obvious reasons, the active file cannot be deleted.

Conclusion

Annexes

A Example of a RTIS configuration file	37
--	----

EXAMPLE OF A RTIS CONFIGURATION FILE

```
; ****
; RTIM Configuration file
; Version 0.0.1
; Date updated: 2010-11-12
; Severity filter :
;          0(0x00) = No error messages
;          1(0x01) = F
;          3(0x03) = F+E
;          7(0x07) = F+E+W
;         15(0xF) = F+E+W+I
;         31(0x1F) = All
;
;          Add 128(0x80) to the above = D
; ****
[MAIN]
SeverityFilter=159
StationId=100
RxIP_Address=192.168.1.1
RxPortNo=28784
RxSocketType=5
RxIOTimeout=1;
RxConnectionTimeout=30
RxRetryDelay=9
StationShortName=TEST
ReceiverPosition=2102940.5,876.32,5958192.0
;
; ****
; GNSS Alarm Server Module
; ****
;
[GNSS_ALARM_SRV]
SeverityFilter=159
Log=on
Console=off
Gui=on
;
; ****
; GNSS Receiver Command Server Module
```

```

; ****
;
[GNSS_RXCMD_SRV]
SeverityFilter=159
;
; ****
; GNSS RawData Server Module
; The sample rate is in msec and must be
; one of the following values:
; [10, 20, 40, 50, 100, 200, 500, 1000]
; ****
;

[GNSS_RAWDATA_SRV]
SeverityFilter=159
SampleRate=10;
;
;
; ****
; GNSS Ephemerides Server Module
; ****
;

[GNSS_EPH_SRV]
SeverityFilter=159
;
; ****
; Index Client Module
; ****
;

[INDEX_CLIENT]
SeverityFilter=159
TxIP_Address=159.255.31.33
TxPortNo=29999
TxSocketType=5
TxIOTimeout=1;
TxConnectionTimeout=30
TxRetryDelay=30
;
; ****
; Processing
; ****
;

[PROCESSING]
SeverityFilter=159
DopplerTolerance=10.0
FilterFreq=.3
;
; ****
; Output
;
```

```
; ****
;
[OUTPUT]
SeverityFilter=159
RootDirectory=/rtis/rawData/
```