MASTER'S THESIS

Generic Data Acquisition and Instrument control System (GDAIS)

Degree: Telecommunication Engineering

Author: Pau Haro Negre

Advisors: Prof. Adriano José Camps Carmona
Dr. Xavier Bosch i Lluis

Year: 2011

Abstract

Remote sensing instrument development usually includes a software interface to control the instrument and acquire data. Although there are a similarities among softwares, it is hardly ever reused, since it is not designed with reusability in mind. The goal of this project is to develop a multi-platform software system to control and acquire data from one or more instruments in a generic and adaptable way. Thus, in future instruments, it can be used directly or with some minor modifications. The main feature of this system, named Generic Data Acquisition and Instrument control System (GDAIS), consists in adapting to a wide variety of instruments with a simple configuration text file for each one. Furthermore, controlling multiple instruments in parallel and co-register their acquired data, having remote access to the data and being able to monitor the system status are key points of the design.

To satisfy these system requirements, a modular architecture design has been developed. The system is divided in small parts, each responsible of an specific functionality. The main module, named GDAIS-core, communicates independently with each connected instrument and saves the received data. Acquired data is saved in the Hierarchical Data Format (HDF5) binary format, designed specially for remote sensing scientific data, which is directly compatible with the commonly used network Common Data Form v4 (netCDF-4) format. The other main module of the system is named GDAIS-control. Its job is to control and monitor GDAIS-core. In order to make this module accessible from anywhere, its user interface is implemented as a web page. Apart from these two main modules, two desktop applications are provided to help with the configuration of the system. The first one is used to create an instrument text descriptor, which defines its interaction, connection and parser. The second one is used to define text descriptor of a set of instruments that the system will be controlling.

Due to its modular design, the system is very flexible and it allows to significantly change the implementation of some subsystem without requiring any modification to the other parts. It can be used in a wide range of applications, from controlling a single instrument to acquiring data from a network of several complex instruments and saving them all together. Furthermore, it can be operated as a file data converter, reading from a raw capture or text file and parsing it to store it in the more optimized and well-organized HDF5 format.

Resum

El desenvolupament de qualsevol instrument de teledetecció acostuma a incloure una interfície software per controlar l'instrument i adquirir dades. Tot i que aquesta part software sol ser molt similar cada cop, no acostuma a ser reutilitzada, ja que no es dissenya tenint-ho en compte. L'objectiu d'aquest projecte és desenvolupar un sistema software multi-plataforma per controlar i adquirir dades d'un o més instruments de forma genèrica i adaptable, de manera que pugui ser utilitzat directament o amb alguna lleugera modificació. La principal característica del sistema, anomenat Sistema Genèric d'Adquisició de Dades i Control d'Instruments, consisteix en la capacitat d'adaptar-se a molts tipus diferents d'instruments amb un simple fitxer de configuració per a cada un. A més a més, altres punts importants del disseny són la possibilitat de controlar múltiples instruments en paral·lel, desant alhora la informació rebuda de cada un; permetre l'accés remot a les dades capturades; i proporcionar una interfície de monitorització de l'estat del sistema.

Per tal que el sistema compleixi amb tots aquests requeriments, s'ha dissenyat una arquitectura modular. El sistema està dividit en diversos blocs, cada un responsable d'una funcionalitat específica. El bloc principal, anomenat GDAIS-core, es comunica independentment amb cada instrument connectat i guarda les dades que rep. Les dades adquirides són desades en el format binari HDF5, dissenyat especialment per a dades científiques de teledetecció, i que és directament compatible amb un altre format molt utilitzat, el netCDF-4. L'altre bloc principal del sistema es diu GDAIS-control. Aquest s'encarrega de controlar i monitoritzar el bloc GDAIS-core. Per tal de fer accessible aquesta interfície des de qualsevol lloc, s'ha implementat com una aplicació web. A més d'aquests dos mòduls principals, també s'han creat dues aplicacions gràfiques d'escriptori per ajudar amb la configuració del sistema. La primera permet crear un fitxer de text amb la descripció d'un instrument i la segona serveix per crear un fitxer amb la descripció d'una combinació d'instruments a controlar conjuntament.

Gràcies al seu disseny modular, el sistema és molt flexible i permet fer modificacions importants a un dels subsistemes sense haver de fer cap canvi a les altres parts. Hi ha moltes aplicacions possibles per aquest sistema, des de controlar un sol instrument a adquirir dades d'una xarxa d'instruments i guardar-ho tot en un sol fitxer. També es pot utilitzar com un convertidor de fitxers, partint d'un fitxer de text o binari d'una captura, per obtenir la mateixa informació en un fitxer en el format HDF5, més optimitzat i ben organitzat.

Resumen

El desarrollo de cualquier nuevo instrumento de teledetección suele incluir una interfaz software para controlar el instrumento y adquirir datos. Aunque esta parte software es muy similar cada vez, no suele ser reutilitzada ya que no se diseña teniendo en cuenta esta idea. El objetivo de este proyecto es desarrollar un sistema software multi-plataforma para controlar y adquirir datos de uno o más instrumentos de forma genérica y adaptable, para así poder ser usado directamente o con alguna ligera modificación. La principal característica de este sistema, llamado Sistema Genérico de Adquisición de Datos y Control de Instrumentos, consiste en la capacidad de adaptarse a diferentes tipos de instrumentos con sólo un fichero de configuración para cada uno. Además, otros elementos importantes del diseño incluyen la posibilidad de controlar múltiples instrumentos en paralelo, guardando a la vez la información recibida de cada uno; permitir el acceso remoto a los datos capturados; y proporcionar una interfaz de monitorización del estado del sistema.

Para que el sistema cumpla con todos estos requisitos, se ha diseñado una arquitectura modular. El sistema está dividido en múltiples bloques, cada uno responsable de una funcionalidad específica. El bloque principal, llamado GDAIS-core, se comunica independientemente con cada instrumento conectado y guarda los datos que recibe. Estos datos son guardados en el formato binario HDF5, diseñado especialmente para datos científicos de teledetección, y que es directamente compatible con otro formato muy usado, el netCDF-4. El otro bloque principal del sistema se llama GDAIS-control. Este se encarga de controlar y monitorizar el bloque GDAIS-core. Para facilitar el acceso a esta interfaz de control desde cualquier sitio, ha sido implementado como una aplicación web. Además de estos dos módulos principales, también se han creado dos aplicaciones gráficas de escritorio para ayudar con la configuración del sistema. La primera permite crear un fichero de texto con la descripción del instrumento y la segunda sirve para crear un fichero con la descripción de una combinación de instrumentos a controlar conjuntamente.

Gracias a su diseño modular, el sistema es muy flexible y permite modificaciones importantes a cualquiera de sus sistemas sin tener que cambiar nada de las otras partes. Hay muchas aplicaciones posibles para este sistema, desde controlar un solo instrumento hasta adquirir datos de una red de instrumentos y guardarlo todo en un solo fichero. También se puede usar como conversor de ficheros, partiendo de un fichero de texto o binario de una captura, para obtener la misma información en un fichero con el formato HDF5, más optimizado y organizado.

Acknowledgements

I am very grateful to Adriano Camps for giving me the opportunity of doing this final degree project at the Remote Sensing Laboratory (RSLab), as well as all the people who have collaborated by providing information and their time for this project. Also, I would like to thank them for lending me their instruments for developing and testing the system.

Specially, I would like to acknowledge Xavier Bosch, who has provided me advice and enthusiasm throughout the whole project and has believed in my ideas for the design of the system. Without his support and trust, the result would not have been so successful.

I would also like to thank all the friends from this university who have accompanied me all over these years, and the ones I have met these last few months while working on this project.

Finally, a very special thanks to my sister and my parents, for their unconditional support and encouragement.

Contents

Acronyms

AJAX	Asynchronous JavaScript and XML.
API	application programming interface.
ARIEL	Airborne RadIomEter at L band.
blob	binary large object.
BSD	Berkeley Software Distribution.
CDF	Common Data Format.
CLI	command-line interface.
CSS	Cascading Style Sheets.
DR	Dickie Radiometer.
DSP	Digital Signal Processor.
DTD	Document Type Definition.
ESA	European Space Agency.
FITS	Flexible Image Transport System.
FSF	Free Software Foundation.
GDAIS	Generic Data Acquisition and Instrument control System.
GNSS	Global Navigation Satellite System.
GNU GPL	GNU General Public License.
GPIB	General Purpose Interface Bus.
GPL	General Public License.
GPS	Global Positioning System.
griPAU	Reflectometer Instrument for PAU.
GUI	Graphical User Interface.
HDF	Hierarchical Data Format.
HP-IB	Hewlett-Packard Interface Bus.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
ICML	Instrument Control Markup Language.

IEEE	Institute of Electrical and Electronic Engineers.
IMS	Inertial Measurement System.
IMU	Inertial Measurement Unit.
IP	Internet Protocol.
IPT	Interference Pattern Technique.
IRC	Interoperable Remote Component.
JSON	JavaScript Object Notation.
MVC	Model-view-controller.
NASA	National Aeronautics and Space Administration.
NCSA	National Center for Supercomputing Applications.
netCDF	network Common Data Form.
NTP	Network Time Protocol.
OOP	Object-oriented programming.
OS	Operating System.
OSI	Open Source Initiative.
PC	Personal Computer.
PCI	Peripheral Component Interconnect.
PRSG	Passive Remote Sensing Group.
PSF	Python Software Foundation.
pythics	Python Instrument Control System.
RDBMS	Relational Database Management System.
RFC	Request for Comments.
RS-232	Recommended Standard 232.
RSLab	Remote Sensing Laboratory.
SQL	Structured Query Language.
SSH	Secure Shell.
TCP	Transmission Control Protocol.
TPR	Total Power Radiometer.
TSIP	Trimble Standard Interface Protocol.
UART	Universal Asynchronous Receiver/Transmitter.
UI	User Interface.
UPC	Universitat Politècnica de Catalunya.
URL	Uniform Resource Locator.
USB	Universal Serial Bus.

VISA	Virtual Instrument Software Architecture.
WSGI	Web Server Gateway Interface.
XHTML	eXtensible HyperText Markup Language.
XML	eXtensible Markup Language.

Chapter 1

Introduction

1.1 Motivation of this project

Since the year 2000, the Universitat Politècnica de Catalunya (UPC) Passive Remote Sensing Group (PRSG) has designed and built a series of instruments to be used in remote sensing experimental campaigns. For each instrument developed at the RSLab, a software system has been implemented to control it and acquire data from it. Until now, even though there was a clear similarity among these systems, each project developed a new solution from the ground, not relying on the already implemented systems. This was due to the fact that the reusability of the solution was not part of the requirements of the project, so this was not considered in the design and implementation.

Since many instruments have been developed in the recent years, the idea of creating a reusable control and acquisition software has been gaining strength with each new project. Finally, a first effort in this direction was made in a project by Maruan Mussaif in 2007 [1], which is described in the next section.

Despite in [1] there was an initial plan of reusing it and the quality of the final system, this solution was very coupled with the instrument it controlled, so it was not flexible enough for the system to be used in any further equipment development. For this reason, some time later, the decision was made to start a new project, the current one, centered in just providing a reusable and adaptable system. This project does not target any specific instrument in its development, instead it uses the knowledge base of all the previous systems as the ground from which to create this new generic solution.

1.2 Previous work on this field

The first attempt to solve this problem consisted in the development of an instrument control system for the PAU-RAD radiometer [2]. This project was named "Design and implementation of a graphical user interface and controller for a radiometer with digital beamforming" by Maruan Mussaif Pradas [1].

This work was conducted as part of the development of the PAU system, in the PAU-RAD project subgroup. Its goal was to define the application layer and the user interface for this equipment, so that it could be controlled and generated data could be acquired. Another objective of this development was to provide access to this system from any Internet connected computer through a website. Also, it was planned that, once the development for PAU-RAD equipment was completed, the system would be extended to the other PAU project areas to help with the study and processing of the provided information.

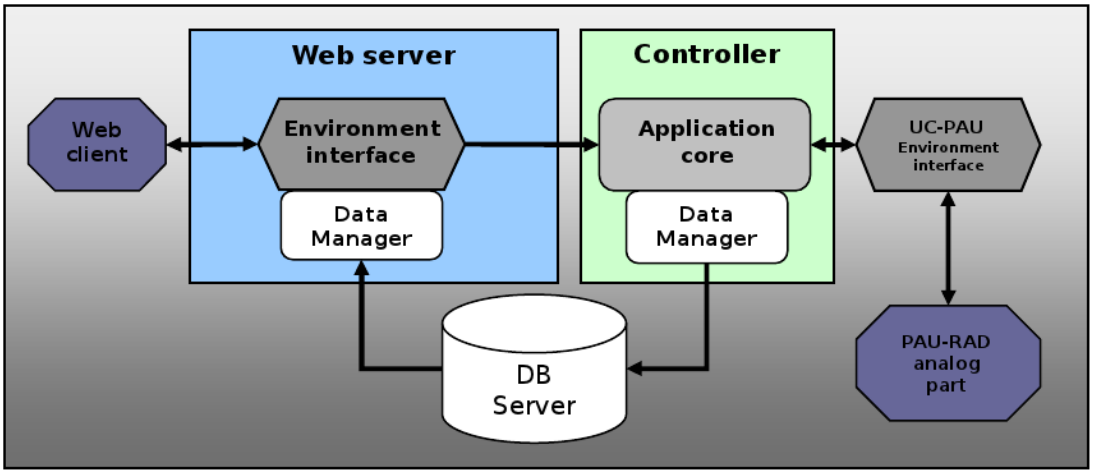


Figure 1.1: System architecture based on reactive programming [1]

In this project, an advanced architecture was developed based on the reactive systems theory. Figure 1.1 shows this architecture, with the controller block, that interacts with the instrument through an Ethernet connection using Transmission Control Protocol (TCP) and manages the acquired data, the database server, which stores the acquired data and application settings, and the web server, that provides the user interface to interact with the system and download acquired data.

To implement this architecture, a set of different technologies were used. These were chosen based on several criteria: platform independence, free software, low implementation cost and wide community support. For this reasons, the chosen web server was Apache, using PHP[1] programming language to implement the web application, and MySQL[2] Re-

[1]PHP: Hypertext Preprocessor: http://www.php.net/
[2]MySQL open source database: http://www.mysql.com/

lational Database Management System (RDBMS) for the data persistence. The controller block, in charge of the communication with the instrument and acquired data storage, was implemented in Java[3] programming language, which is platform independent, based on Object-oriented programming (OOP) and free software. Moreover, the web application design was based on Model-view-controller (MVC) design pattern and the advanced Asynchronous JavaScript and XML (AJAX)[4] programming methodology was used in the web client implementation.

Figure 1.2: Equipment control panel view [1]

Figure 1.2 shows the final user interface to control the system. As it can be observed, the TCP connection settings could be defined and, under this settings, a command could be selected to send to the instrument. The reply to the commands was shown at the bottom, in the output part. On the right, a console to monitor the equipment status was included. The website section from which acquired data could be downloaded and deleted is displayed Figure 1.3. Some details for each generated file were provided.

Figure 1.3: Web interface of acquired data download section [1]

[3]Java programming language: http://www.java.com/
[4]http://en.wikipedia.org/wiki/Ajax_(programming)

The final software system developed in this project successfully satisfied the initial requirements. It was well-designed, reliable and accessible from Internet through any web browser. However, the resulting solution was very focused on PAU-RAD instrument control, making it difficult to reuse it in subsequent projects.

For this reason, in the present project a new approach was considered, while retaining the original idea, but designing a system as universal as possible.

1.3 Project goals

The main objective of this project is to provide a generic and reusable system for remote sensing instruments control and data acquisition. For this reason, code reuse, modularity and flexibility are essential requirements.

Another objective is that the system can be adapted to the most usual instrument communication systems and protocols, providing a simple interface to setup the control and acquisition, not requiring any programming modifications by the user.

Moreover, since most RSLab experiments include multiple instruments, this project should aim to adapt to this complex systems, being able to control them and join all the acquired data from each instrument into a single file.

For the solution to be used in as many environments as possible, it should be portable to multiple platforms, and the generated acquisition data files should follow a standard format that can be easily accessed from the most common engineering applications, like MATLAB.

Finally, from the previous experience at RSLab in deploying experiments in remote and unaccessible locations, another project goal is to provide a reliable solution that can operate alone anywhere, and that can be controlled and monitored even in very limited communication conditions.

1.4 Report structure

This report is divided in five chapters, including this first introduction chapter and the final conclusions and future work lines. The content chapters follow the chronological order of the project development, from the analysis of the problem to the final testing and validation of the designed system.

Chapter two analyzes in detail instrument control and data acquisition systems, starting from the most general concepts and then centering on the remote sensing instruments developed at the RSLab. At the end of the chapter some available products are analyzed as possible solutions to implement the system.

Chapter three describes the development of the whole GDAIS system. After a brief introduction on the development model used, each step of the process is explained: problem requirements analysis, solution architecture definition, software technology selection and, finally, design and implementation.

Chapter four presents two experiments with different equipments that were used to test the implemented system. For each instrument a short description is provided, followed by the experiment set up and the results and conclusions that were extracted from it.

Chapter 2

Problem analysis

This chapter is devoted to the analysis of the problem being solved. This analysis is initially based on generic concepts of instrument control and data acquisition software. Afterwards, this concepts are refined through the review of three previous software developments in RSLab. Finally, once the problem has been precisely defined, some existing solutions that may be adequate for this problem are evaluated.

2.1 Instrument control and data acquisition

Instrument[1] control and data acquisition is an essential part in any experimental test in a research. Instruments involved in this tests may be of quite different types. From the most simple ones, which include only a single sensor and provide data continuously after they are plugged-in; to the most advanced ones, which may require multiple configuration options and command sequences to acquire data from them. Therefore, different approaches have been considered along the time to adapt to each possible application.

Nowadays, almost all data acquisition systems are implemented as software programs in Personal Computer (PC) systems outside of the instrument's core hardware. This permits a higher level of functionality to be integrated into the system, such as processing the acquired data before storing it or joining data acquired from multiple instruments to generate a single combined output file.

Nonetheless, there are certain situations where a hardware implemented system that records data as it is acquired is necessary. For example, when the data rate provided

[1] In the context of this project, an instrument refers to any device with one or more sensors that can be connected to a computer to acquire data from it.

by the sensors is so high that it requires some fast processing before recording the information to a memory. This memory can then be shared with an external computer through a simple software program that retrieves the contents of this shared memory.

Focusing on PC driven systems, in the following sections the concepts of instrument control and data acquisition are defined and analysed.

2.1.1 Instrument control

Instrument control consists in communicating a computer with a device formed by one or more sensors. The computer, through a software program, sends command messages to the device, which executes the events assigned to each received message. These events may be related with changes in instrument parameters, data acquisition from device sensors or status information reports.

In order to communicate the instrument with the computer, a connection between them is required. This connection can be wired, using Ethernet, Universal Serial Bus (USB), RS-232, Peripheral Component Interconnect (PCI), ...; or wireless, using Infrared, Bluetooth, Wireless USB, WiFi, XBee, ... There are also some buses defined specifically for instrument control applications, such as Hewlett-Packard Interface Bus (HP-IB), which evolved to the General Purpose Interface Bus (GPIB) standard bus published by the Institute of Electrical and Electronic Engineers (IEEE) [3].

Once a connection is available, a protocol is used to enable the communication between the instrument and the computer on top of the physical connection. Some instruments use existing protocols like Hypertext Transfer Protocol (HTTP) or Virtual Instrument Software Architecture (VISA). Whereas other instruments use a proprietary protocol designed by the developer of the device, usually with the help of another low-level protocol, like TCP, which provides the basic functionality.

Finally, an Operating System (OS) device driver for the connection being used and a library with the implementation of the protocol are necessary for the software system to be able to communicate with the instrument. The OS driver is commonly provided with the instrument or by the manufacturer of the connection hardware of the instrument. Regarding the protocol library, it may be also provided with the device or, if it is an existing well-known protocol, there may already exists a library available to use. However, for some instruments with a proprietary protocol, or in case of self-developed instruments, there is no existing library, but only the protocol specification is available, requiring the protocol to be programmed within the control software.

Despite the previous analysis of instrument control refers to communicating a single device with a computer, computers usually have multiple connection interfaces. Therefore, control systems can benefit from this fact, by connecting to multiple instruments at the same time. This allows the computer to coordinate the operation of all connected instruments together. For example, in case of having two instruments which provide different measurements and the experiment requires that both measurements are acquired at the same time, the control system can send the acquire command to both instruments simultaneously. As a result of this, recorded data will not need to be synchronized afterwards.

2.1.2 Data acquisition

Data acquisition, in its most general form, refers to the process of digitizing and storing data from any sensor connected to a test system. In the context of this project, data acquisition is the process of reading values and measurements from a single or multiple sensors and storing them in a computer.

Obviously, data acquisition is closely linked with instrument control. In this case, there is also a necessary communication between the instrument and the computer. Moreover, usually the computer has to send a control command to the instrument to start acquiring data.

Acquisition modes

There are different acquisition modes, depending on the instrument design. The most simple one consists in just powering the device on and it starts sending data as it is acquired. This approach is enough for some applications, but most times more advanced features, like changing instrument parameters or checking sensors status, are required. In these occasions, there is a specific control command that triggers the acquisition. Then, the device may just wait for a single measurement to be acquired or continue acquiring data until it receives a command to stop. In the first working mode, if multiple measurements are desired, the computer system has to continue asking periodically for more data, either after a defined wait time or just after receiving the data. In the second acquisition mode, the software system has to record acquired data continuously and send the stop command if it needs to change a parameter on the instrument.

Besides these usual modes, some devices do not send measurement data as soon as it is acquired. This is due to the connection being slower than the acquisition speed or the instrument requiring some internal processing of measurements before data is ready for

the computer. Some of these instruments have two commands to deal with this extra waiting time: a command to check whether the measurement is ready to be read and a command to read the measurement. Yet other instruments just wait until all data is ready to send it to the computer. Thus, the computer has to wait the required time without sending any more commands to the device.

Managing data

Once the data acquired by the sensor arrive to the computer it is usually processed before storing it. The most simple transformation consists in converting the binary data stream to a more meaningful representation, identifying the structure of the information and its values. For example, as seen in Figure 2.1, an instrument may send a sequence of 14 bytes for some measurements: the first byte is used to detect the start of the measurement information; the second byte identifies the measurement type and the last 12 bytes are 3 integer values of 4 bytes each. When this binary sequence is received, the 3 relevant values can be extracted and stored. Thanks to this processing, the stored data size is reduced, because only relevant information is saved, removing auxiliary structure bytes. Furthermore, other processing may be applied to acquired data, like an algorithm to check data consistency using a checksum or even to perform some error correction.

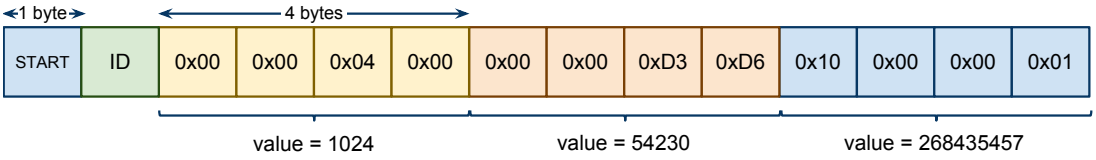


Figure 2.1: Example binary sequence received from a sensor.

Finally, data is ready to be saved to a file in a storage device. There are multiple formats for storing information. The most simple approach consists in saving data in their binary original format, which will require some decoding process when reading these data to use them. Another method is to save the information, once parsed, in a text file. This way, it is ready to be read without any transformation, except for the conversion from text to numeric format. However, the resulting file is much larger than necessary, for example, when representing a 4 byte number, each digit will use a byte when stored as a character, so it may take up to 10 bytes. Moreover, these files cannot be easily shared with other teams, as some parsing is required to convert the contents of the file to meaningful values that can be processed and the method to convert the text to values is specific for each file.

To overcome these limitations, a structured binary file format can be used. This may be

a general purpose database like MySQL or a more convenient and scientific oriented data format like Common Data Format (CDF), network Common Data Form (netCDF) [4], Hierarchical Data Format (HDF), Flexible Image Transport System (FITS) [5], ... All these formats are specifically designed for storing and manipulating multi-dimensional scientific data sets. Furthermore, as these are standard formats, any of them can be shared with external teams and they will be able to read and use it using the libraries available for the chosen format.

2.2 Common aspects of remote sensing instruments

Focusing on the context of remote sensing instruments development in RSLab, the common features for an instrument control and data acquisition software system are:

Joining commercial and homemade sensors

Most instruments developed at RSLab are made from a mix of self-manufactured sensors and commercial products. For this reason, software provided with external sensors can hardly ever be reused, at least the most high-level part. Therefore, the control software for these instruments has to implement the communication with commercial sensors besides the homemade devices, which normally use a simpler protocol.

Connecting with multiple instruments at the same time

In some applications it is useful to join multiple instruments and acquire data in parallel from all of them. Most times, despite being all instruments together, data from each instrument is stored separately and joined afterwards, when acquired data is analyzed. A better solution may be to store data from all instruments at the same time to a shared file. However, this approach requires acquisition software to use parallel programming techniques to serialize write operations, as writing to the same file from multiple processes is really dangerous and data loss is probable to occur if not properly controlled.

Adapting to different operation modes

Provided the heterogeneous instrument designs developed to adapt to each application being tested, operation of these instruments often differs from one to another. For example, if an instrument has to make some measurements from an airplane, it is essential to acquire data at the maximum possible rate in order to make the most of the flight. Furthermore, as seen in subsection 2.1.2, there exist multiple acquisition methods, which are tied to how sensors interfaces are designed.

Real-time monitoring and data preview

Another feature commonly found in these systems is being able to monitor the status of the device and preview acquired data while it is operating. This is mainly used during the development phase, when the designer needs to check if the instrument is working as expected. Normally, a graphical interface is added to the acquisition system for this purpose.

No specialized acquisition hardware in PC required

Despite in some other data acquisition applications it is common to use specialized hardware for the acquisition process, it is not usual in remote sensing instruments. Instead, instrument takes care to convert measurement information into digital data and send it through a common connection to the PC. From previous instruments analysis, this is generally a serial connection, over Recommended Standard 232 (RS-232) or USB, or TCP over Ethernet.

Digital input signals

Acquired data, when it arrives to the PC, is already in digital format. Normally it is a sequence of bytes following a defined packet structure. This data needs to be parsed and, for each packet, its fields need to be converted to meaningful values that can be used afterwards. Data fields type is commonly integer or floating point, with single or double precision in the generally used representation defined by the IEEE 754-2008 standard. There are sometimes other fields used for configuration and information, which use each bit to represent an option. Also some fields may be used to check data integrity and perform error correction.

Remote control and access to data

Remote sensing experiments are always done outdoors. Sometimes, it may be necessary to deploy an experiment in a location far away without connectivity or even easy access, such as on top of a mountain. Therefore, they are designed to be as much autonomous as possible, being able to run for a long period of time without requiring operation from the outside. However, if a minimum connectivity is achieved, software control system usually includes some remote access method to check system status and retrieve acquired data. This allows to check that everything is working as expected without requiring to go where the experiment is deployed.

Cost-efficient designs

Since funding and time are limited, a design criterion is that if a problem can be solved by two different methods, the faster and less expensive one is chosen. Therefore, when developing the software system to control and acquire data from an instrument, it is common to program everything from the ground, even if there

exists some commercial software which implements most of the necessary features. This results in perfectly fitted solutions, even though they are more prone to errors and unexpected problems.

2.3 Previous instruments software analysis

In order to complete the definition of the problem, this section consists of an analysis of the software which controls and acquires data from three instruments previously developed in RSLab. A comparison between all these instruments can be found in Table 2.1.

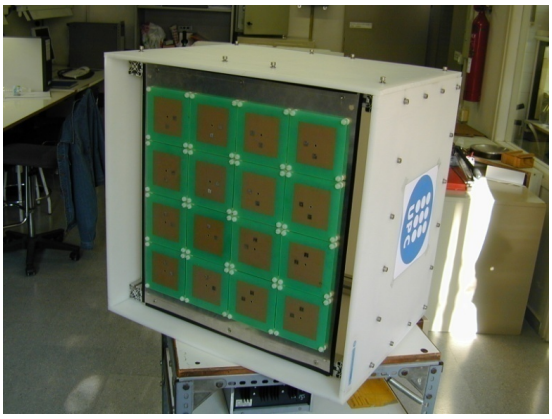
Table 2.1: Instrument comparison

	PAU-RAD	**PAU-SA**	**gripPAU**
Instrument connection	TCP over Ethernet	RS-232	USB
Behaviour	state machine with continuous output	single measurement	continuous measurements
Initialization	TCP connection	none	firmware download
Setup	at any time	before each measurement	no settings
Watchdog signal	yes	no	no
Instrument → *Controller*	state info + data stream	data type + data stream	data stream
Instrument ← *Controller*	command + parameters (if any)	command (returned by the instrument)	

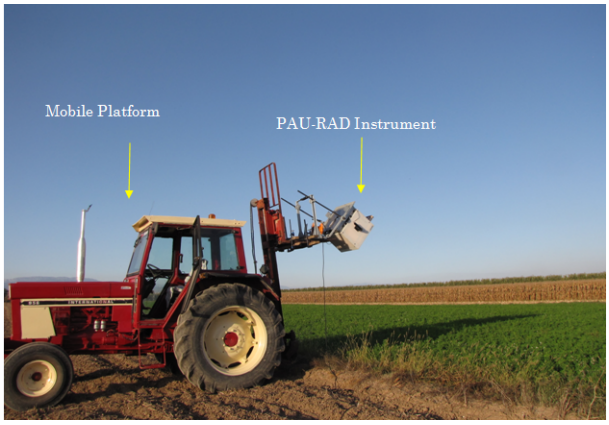
2.3.1 PAU-RAD

The first analyzed instrument is a polarimetric radiometer with digital beamforming called PAU-RAD [1] (see Figure 2.2). It connects to the computer that controls it through an Ethernet connection. It uses TCP as the connection protocol, as it is implemented on the hardware of the device.

Internally this instrument works as a state machine, switching between 5 states: init, normal, open loop, close loop and debug. In each state the output format changes to



(a) Front view of the instrument



(b) Measuring an alfalfa field, in the 2010 Palau d'Anglesola field campaign

Figure 2.2: PAU-RAD instrument [2]

provide the data associated with the current state. Through the TCP connection, the control software can send several parameters (integration time, incidence angle, ...) and working modes (to change the current state) to the instrument at any time.

The low-level communication protocol implemented over the TCP layer is different for each direction. Data packets sent from the instrument to the control system start with the information of the state of the instrument (1 or 2 bytes), followed by the data associated with the state (variable length: 13 or 17 blocks of 32 bits, or 19 blocks of 16 bits). In the other way, the system sends commands to the instrument starting with the desired command identifier (4 bits), followed by the required data for the command (28 bits of data or empty if the command has no parameters).

Eventually, as the instrument sends data bytes continuously, it is not able to monitor whether the connection is still alive. To solve this issue, a watchdog signal is used, which consist in the control system sending periodically a command to the instrument to prove it is still there listening for new data and that the data flow has to continue. If the instrument does not receive this periodic signal for some time, it stops sending more data and returns automatically to the *init* state.

2.3.2 PAU-SA

This other instrument is a synthetic aperture radiometer called PAU-SA [6] (see Figure 2.3). It is connected with the control system through a serial port connection.

Every time the computer software needs to acquire new data it has to follow a specific algorithm, which consists in a sequence of commands being sent to the instrument. Firstly,

(a) View of the whole instrument with one arm opened

(b) PAU-SA mounted and deployed in its mobile unit

Figure 2.3: PAU-SA instrument [7]

the control system has to configure the switches and parameters of the measurement. Secondly, it has to send a command to trigger the begin measurement event and, afterwards, start surveying the instrument until it replies positively, which means that the measurement has completed. Finally, the acquisition software asks the instrument to copy the new data to a temporary memory which can be accessed from the control system through the serial connection. When the instrument completes the measurement phase the next measurement can be requested following the same structure. This can be done before the acquisition system has received all the data from the previous measurement, as it has been copied to the shared memory and the internal memory can be overwritten without problems.

The low-level communication protocol between the control software and the device consists of commands with a length of 2 bytes. As a measure to prevent communication errors, when the instrument receives a command it sends it back to the control system in order to confirm that it was received correctly. For the command used to check whether the measurement has completed, the instrument returns a boolean value indicating which is the status of the measurement. In reply to the command which asks for measurement data, the instrument sends a header (8 bits) continued by the measurement settings (2 bits) and the data type (2 bits). Finally, a blob of binary data from the measurement is sent.

2.3.3 griPAU

The third instrument analyzed is a Global Navigation Satellite System (GNSS) signals reflectometer called Reflectometer Instrument for PAU (griPAU) [8] (see Figure 2.4). It is connected to the computer system through a USB connection, using a driver in the operating system that emulates a serial port. This simplifies its usage, as nowadays serial connection is supported by any library and programming language.

(a) External view of the processing unit

(b) griPAU deployed during the ALBATROSS 2009 field experiment

Figure 2.4: griPAU instrument [9]

Before starting to measure, the firmware has to be loaded into the instrument. Once it is loaded, the device starts sending measurement data continuously to the acquisition system. The control software is expected to capture data for a certain configured time and then it shutdowns the instrument. This device has no configuration commands.

2.4 Existing solutions

Once the problem has been completely defined in the previous sections, this section consists in an analysis of existing software solutions that may be used to implement data acquisition and instrument control software for the devices developed at RSLab.

2.4.1 MATLAB

As one of the most widely used tools in engineering projects, MATLAB (matrix laboratory) provides a numerical computing environment and a set of extension modules, named toolboxes. It is developed by MathWorks and allows matrix manipulations, plotting of

functions and data, implementation of algorithms, creation of user interfaces and interfacing with other programs, even if they are written in another language. An example instrument control Graphical User Interface (GUI) application is shown in Figure 2.5.

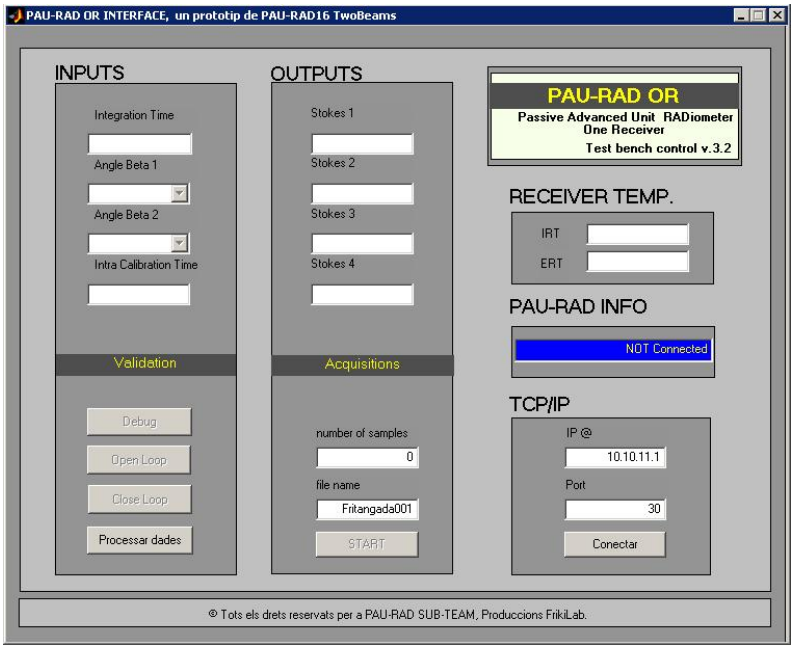


Figure 2.5: MATLAB instrument control GUI

Even though it does not exist a specific toolbox to fulfill all the required features, there are some toolboxes that used together may ease the development of a software solution. These toolboxes include:

Data Acquisition Toolbox

Provides a set of tools for digital I/O from external hardware devices, allowing to configure a connection, read data into MATLAB and send out data.

Instrument Control Toolbox

Implements the communication with instruments directly from MATLAB via instruments drivers, such as IVI and VXIplug&play, and commonly used communication protocols, such as GPIB, VISA, TCP/IP, and UDP; over many connection types, such as RS-232 or Ethernet.

Graphical user interface development environment (GUIDE)

Provides a set of tools for creating GUIs. These tools simplify the process of laying out and programming GUIs. Layout Editor is a tool that eases the process of creating a layout and adding GUI components — such as axes, panels, buttons, text fields, sliders, ...– into the layout area. Once the layout is completed, GUIDE automatically generates MATLAB code with functions to control how the designed GUI operates. An example GUI developed using these tools is shown in Figure 2.5.

MATLAB is a proprietary product of MathWorks, so its is subject to a restrictive license. Although MATLAB Builder can be used to generate library files from MATLAB functions which can be used with other application development environments — such as .NET and Java —, any software developed in MATLAB which needs to be modified will still be tied to MATLAB language.

Adequacy to the project

After analyzing MATLAB as a possible solution to implement acquisition and control software, it was concluded that it was not the most adequate solution for the purpose of this project. The main reason for this decision was that, despite MATLAB is a great tool for prototyping, it is not very reliable for long-running applications as this is not what it was designed for. Furthermore, MATLAB does not provide all the necessary features directly, but some programming would be required to define a useful base for future instruments software development.

Moreover, in order to implement an autonomous system which did not require the full MATLAB environment to be running at the same time as the application, it would be necessary to generate a library with MATLAB code and develop a main application environment in a general programming language. Finally, MATLAB proprietary licensing and its cost might limit future usage of this software base, as it forces any project that may use the existing code to use MATLAB and acquire a license for it.

2.4.2 LabVIEW

LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) is a platform and development environment for a visual programming language from National Instruments. This simplifies implementing systems to control and acquire data from any sensor or bus — USB, PCI Express, PXI, WiFi, ...Automatic measurements can be done for several devices, data can be analyzed in parallel with acquisition, and then custom reports can be created. See Figure 2.6 for an example of a simple acquisition application which acquires data and writes measurements to a file after applying an algorithm to them. As it can be observed, it is graphically programmed, without requiring any line of code to be written.

Except for the Base Development System license which only runs in Microsoft Windows, LabVIEW runs in multiple platforms including Microsoft Windows, various versions of UNIX, Linux, and Mac OS X. It is a proprietary product of National Instruments and,

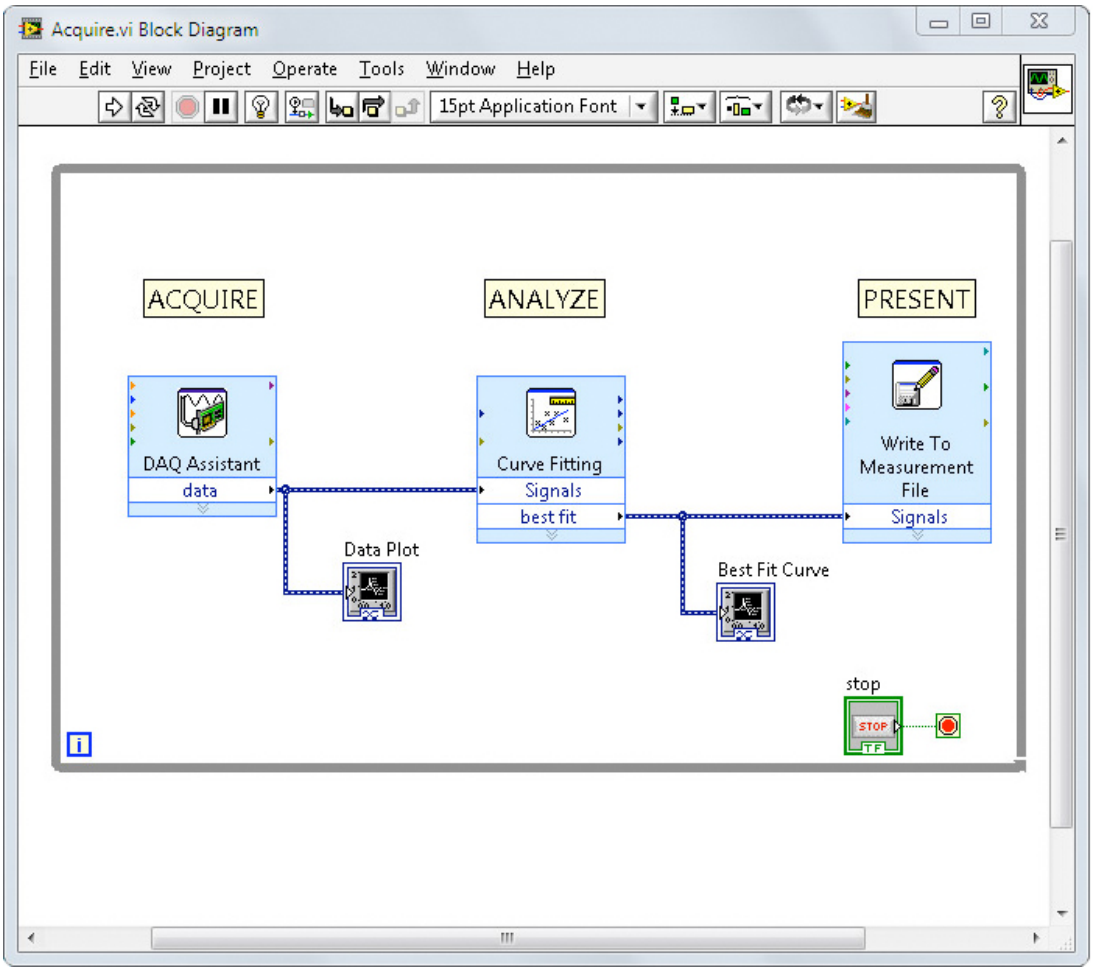


Figure 2.6: LabVIEW example for acquiring data to a file.

unlike common engineering programming languages such as C or FORTRAN, LabVIEW graphical programming language is not managed or specified by a third party standards committee such as ANSI, IEEE, ISO, ...

Stand-alone applications can be built with LabVIEW using the Application Builder component that is included within the Professional Development System, the most expensive environment offered by National Instruments. Purchasing a cheaper environment, only a separate add-on is provided for this feature.

Adequacy to the project

Unlike MATLAB, LabVIEW does provide most of the required features for instrument control and data acquisition bundled with the development environment, requiring just a minimal graphical programming to implement a full system. However, other MATLAB limitations also apply to LabVIEW. Only the most expensive Professional Development System version provides platform independent execution of code and is able to create stand-alone applications that accomplishes the autonomy required for instrument control software. Furthermore, LabVIEW license is also restrictive and, as with MATLAB, developed applications can only be extended and reused inside the same environment, as it uses a non-standard programming language. In conclusion, it was decided that LabVIEW was not an adequate solution either.

2.4.3 Interoperable Remote Component (IRC) Architecture

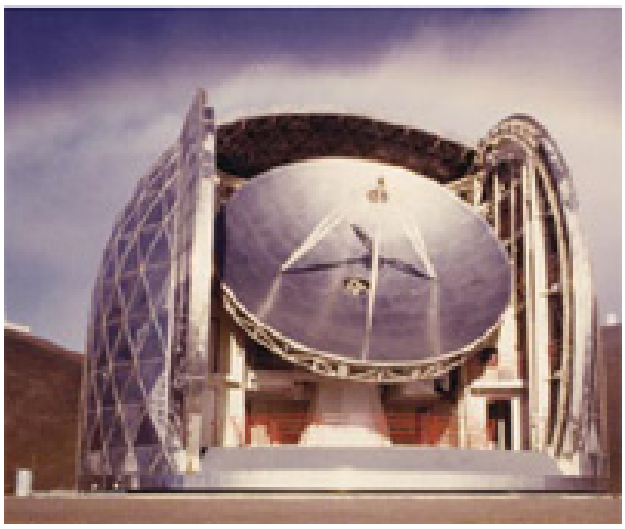
National Aeronautics and Space Administration (NASA) Goddard Space Flight Center, led by the Advanced Architectures and Automation branch (Code 588), and Century Computing developed an extensible application framework for instrument command and control, known as Interoperable Remote Component (IRC).

The IRC architecture is a flexible, platform-independent application framework that is well suited for the distributed control and monitoring of remote devices and sensors. The architecture is designed to be simple enough to use and maintain as well as flexible enough to be useful in a wide variety of applications and domains.

The architecture emphasizes the capability to configure itself for a specific application based on eXtensible Markup Language (XML) descriptions. There are descriptions to tell the framework which application components to plug in, what the GUI should look like, what devices to connect to and how to communicate with them, what algorithms to include in the application, and what interface to present to other distributed peers.

To enable a dynamic discovery and configuration capability for a collection of devices, each IRC instance can advertise and publish a description of itself on a virtual network. This simple capability of dynamically publishing and subscribing to interfaces enables a very flexible, self-adapting architecture for monitoring and control of complex instruments in diverse environments.

(a) The rover, developed by Carnegie-Mellon University



(b) Calltech Submillimeter Observatory

Figure 2.7: Instruments that use IRC software. Credit: NASA GSFC

IRC was originally designed as a low-cost control system for all types of remotely operated instruments and platforms. For example, IRC is used in the Submillimeter High Angular Resolution Camera on the ground-based Caltech Submillimeter Observatory, see figure 2.7b, as well as in autonomously operated ocean-faring research platforms developed by NASA and the National Oceanic and Atmospheric Administration. Through continuous improvements, the technology now allows all types of software systems to operate with one another, broadening its potential use across multiple industries.

The most important features of IRC are:

Platform independence Implemented in Java, IRC can be used on Windows, Mac OS, Solaris, several variants of Linux operating systems, and embedded environments.

High performance IRC can process data within the framework at some hundreds of megabytes per second without special hardware.

Flexibility Processes can run on a single computer or on multiple heterogeneous computers as well as remotely over the Internet.

Configurability Software solutions are easily developed, enhanced, and reused for different devices, instruments, and domains, thus saving time and development costs.

The code for the IRC architecture is available for download via NASA Goddard Space Flight Center's Open Source software site. The software is released under the terms and conditions of the NASA Open Source Agreement (NOSA) Version 1.1 or later [10].

This is an OSI-approved software license, however the Free Software Foundation (FSF) raises issue with the following clause:

> G. Each Contributor represents that its Modification is believed to be Contributor's original creation and does not violate any existing agreements, regulations, statutes or rules, and further that Contributor has sufficient rights to grant the rights conveyed by this Agreement.

The FSF states that "free software development depends on combining code from third parties" [11], but as this license does not permit this because it requires that changes are of "Contributor's original creation", the NASA license is not a free software license.

NASA's Instrument Control Markup Language (ICML)

As explained in the previous section, IRC can be configured for a specific application based on eXtensible Markup Language (XML) descriptions, which are well suited to describing hierarchical, structured information. This XML description format was named ICML [12] and a custom Document Type Definition (DTD) for it was implemented.

ICML can be used to describe control capabilities, data streams, message formats, pipeline algorithms, and communications mechanisms, as well as for online documentation and the association of housekeeping metadata within acquired images.

An illustrative example of this description format in included in Listing 2.1. This example describes two Commands, `HeatSwitch` and `HeatSwitchCurrent`, both of which accept exactly one argument. Note that the "`current`" argument of `HeatSwitchCurrent` command has `Valids` (constraints) defined. This results in a GUI component which guarantees input only within the constrained range.

```
<Control id="HAWC.ADR">
  <Command id="HeatSwitch">
    <Argument id="state" type="boolean" required="true"/>
    <!-- other Arguments would go here -->
  </Command>
  <Command id="HeatSwitchCurrent">
    <Argument id="current" type="float" required="true">
      <Valids>
        <Range low="0.0" high="200.0" units="amps" />
      </Valids>
```

```
    </Argument>
  </Command>
  <!-- other Commands would go here -->
</Control>
```

Listing 2.1: ADR subsystem ICML example

Adequacy to the project

This solution was initially considered adequate for this project, as it accomplished most of the required features and had been designed as a solution to the same problem: acquiring data and controlling instruments. Therefore, an attempt was made to implement a simple instrument control system using the source code available at NASA's website.

Even though a very simple system was chosen to begin with, it proved to be very difficult to implement. Two factors determined this result: firstly, despite of the quality of the available source code being excellent, there was no documentation provided with it, only a few examples of previously developed instruments; secondly, the amount of source code was enormous and, without documentation, it was nearly impossible to deduce which was the expected methodology to use the available features in the implementation of the system.

Finally, the decision was made neither to use this software, as it was too difficult to use without documentation. This was an important downside, as one of the most important requirements of this project is the ease to reuse the resulting solution for future projects development. Moreover, the unclear terms of IRC's license, which was announced as being open source but uncredited by the FSF [11], reinforced this decision.

2.4.4 Python Instrument Control System (pythics)

Pythics is a multiprocess Python framework for creating applications, developed by the D'Urso research group at the University of Pittsburgh in the Department of Physics and Astronomy[2]. Its developers describe it as "an application for running Python code intended to be used for simple interfaces to laboratory instrument or numerical simulations. It features a simple system for making GUIs, useful controls including plotting, clean separation between GUI and application code, and multithreading and multiprocessing so running backend code does not interfere with the functionality of the GUI."

[2]Homepage of the group with information of this project can be found at http://www.nanomaterials.phyast.pitt.edu/

Pythics aim is to provide all the functionality for creating a GUI for an instrument control system, so that all effort can be concentrated on the program development. Despite of its name, no instrument control features are included in the system, except for the user interface. Therefore, a data acquisition and instrument application will require from other tools to accomplish all the necessary features.

As it can be seen in Listing 2.2, the file format which describes the GUI layout in pythics is a XML-compliant HyperText Markup Language (HTML) format, similar to a subset of eXtensible HyperText Markup Language (XHTML). Tag elements — text, controls, ... — within the XML file can be controlled with the help of Cascading Style Sheets (CSS) which specifies element *properties* and element *attributes*. Using this format, a full GUI can be created without having to program any line of code, just describing the interface like a web page is enough for pythics to create a complete GUI with advanced features for instrument control.

```
<html>
<head>
  <title>Hello World</title>
  <style type='text/css'>
    <!-- Style Sheet (CSS) goes here -->
  </style>
</head>

<body>
<h1>Hello World</h1>

<object classid='Button' width='200'>
    <param name='label' value='Run'/>
    <param name='action' value='hello_world.run'/>
</object><br/>

<object classid='TextBox' id='result' width='200'>
</object><br/>

<object classid='ScriptLoader' width='100%'>
    <param name='filename' value='hello_world'/>
</object>

</body>
</html>
```

Listing 2.2: Pythics GUI description

In the previous Listing 2.2 example, the `object` element with `classid='Button'` contains a `param` element which defines that when the described button is clicked the action

`hello_world.run` should be run. In Listing 2.3 an example code for this action is provided, which just inserts the text "Hello, world!" to the `object` with `classid='TextBox'` that is also part of the described GUI.

```
1 def run(result, **kwargs):
2     result.value = "Hello, world!"
```

Listing 2.3: Pythics user program written in python

This example demonstrates the simplicity of pythics framework. Furthermore, the code of this project is open source and it relies in other open source libraries for GUIs programming. Pythics license is GPL v3, which is a free software license that allows its code to be reused and extended provided that derived versions are also licenced under a free license.

Adequacy to the project

In the previous description of pythics it is clear that it has an important downside: it just provides the GUI part of instrument control. Therefore, to develop a full instrument control and data acquisition system, nearly all the functionality would have to be programmed from the ground.

Nonetheless, provided its unrestrictive license, pythics would be clearly a useful addition when developing an instrument control system.

2.5 Conclusion

After analyzing the problem in detail and reviewing the most interesting software solutions already available, it was concluded that no existing system was adequate enough for all the requirements of this project. Therefore, it was decided to build a new solution from the ground that provided a complete framework for future development of instrument control and data acquisition software for instruments developed at RSLab. The next chapter concentrates in the design and implementation of this system.

Chapter 3

GDAIS development

This chapter is devoted to the development of a software system that provides a generic base functionality for data acquisition and instrument control. Firstly, the software development model which will be used is defined. Secondly, from the description of the problem in the previous chapter, the requirements of the development and the expected functionalities are specified. Based on this requirements and functional specification, an architecture for the system is proposed and described. Finally, the full software system is designed following the architecture definition and choosing the most appropriate technologies to implement it.

3.1 Software development model

In order to be useful, this project requires a well-thought and clear design that can be easily understood by future instrument developers that wish to take advantage of it. Therefore, this system will be designed following a formal software development process that assures its quality and future reusability.

A software development process can be described generally as a sequence of activities that go from the initial definition of the problem to the final deployment of the implemented software [13]. Specifically, this process can be divided in the following steps:

Requirements analysis

This is the initial phase of any software development. It encompasses those tasks that help in defining the needs or conditions that the product will have to fulfill, taking account of all the possible uses and applications that are expected from it.

Functional specification

In software development, functional specification consists in describing the requested behaviour of a system that complies all the previously detailed requirements, without defining the final internal implementation of the proposed system. Its aim is to provide a precise idea of the problem to be solved so that a solution can be efficiently designed and the cost of design alternatives can be estimated.

Software architecture

The architecture of a software system is the set of structures, and the relations among them, that describe the implementation of the system. It is also referred as the high-level design, in contrast with the low-level component and algorithm implementation of the design phase. There are many well-known common architectural styles and patterns such as the client-server model, the service-oriented architecture or the event-driven architecture pattern.

Design

Software design consists in solving the problem and planning for a software solution. This is the first platform-specific phase, as it takes into account the technology environment that will be used in the final implementation. As design problems repeat frequently between projects, some templates or patterns that describe solutions to common problems can be used when identified in order to speed up the software development process.

Implementation

This phase consists in programming the designed solution in a specific programming language, and with the help of some existing libraries and external tools, to generate a final program that can be run in a computer.

Testing

Testing phase is the process of validating and verifying that the developed product meets the defined requirements and works as expected. Software testing can be done in any of the steps following the initial requirements phase.

Deployment

This final phase of the development consists of all the activities that make the system completely functional and available for use. These activities include releasing the software, installing, adapting, updating, uninstalling and finally retiring.

Maintenance

Software maintenance is the modification of an existing product after being released to correct faults and to improve performance or other attributes. It mainly consist

in fixing bugs that are detected during the normal usage of the system.

From a classical point of view, all these steps can be seen as a linear sequence in time, which is also known as the waterfall model [14]. See Figure 3.1.

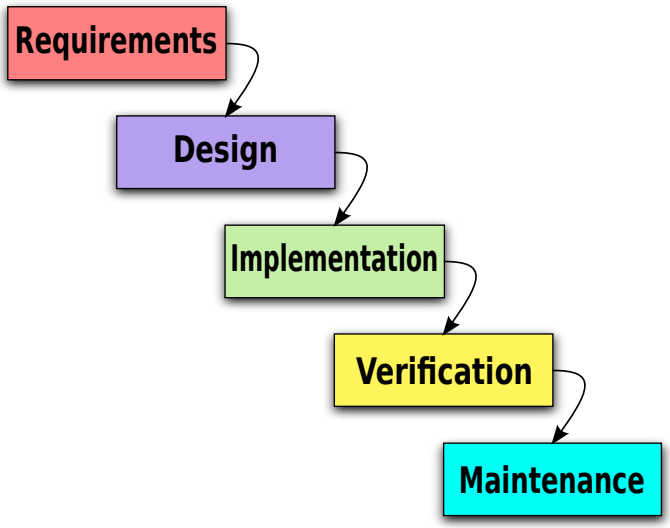


Figure 3.1: Waterfall model. Development advances from the top to the bottom, like a waterfall. Credits: Paul Smith

However, as it is nearly impossible to complete each phase independently from the following ones, some newer methodologies have been defined based on a more realistic point of view. Specifically for this project, a spiral development methodology [15] has been chosen (see Figure 3.2) which consists in combining design and prototyping techniques in a set of iterative incremental stages, each including all waterfall model phases, and going from short iterations at the beginning to longer ones as the development advances. As this is not a very large project, it was considered that two iterations, with a prototype evaluation at the end of the first iteration, was enough to ensure the quality of the final result.

3.2 Requirements analysis

The aim of this section is to completely and accurately describe the needs that GDAIS software will cover and the scope of this solution, based on an analysis of the problem. The features that will be included in the system and the ones that will not be included are detailed, clearly defining the scope of the development.

System requirements analysis can be divided in functional and non-functional requirements. On the one hand, the functional requirements describe the functionalities expected from the system and its behavior, the plan for implementing these requirements is de-

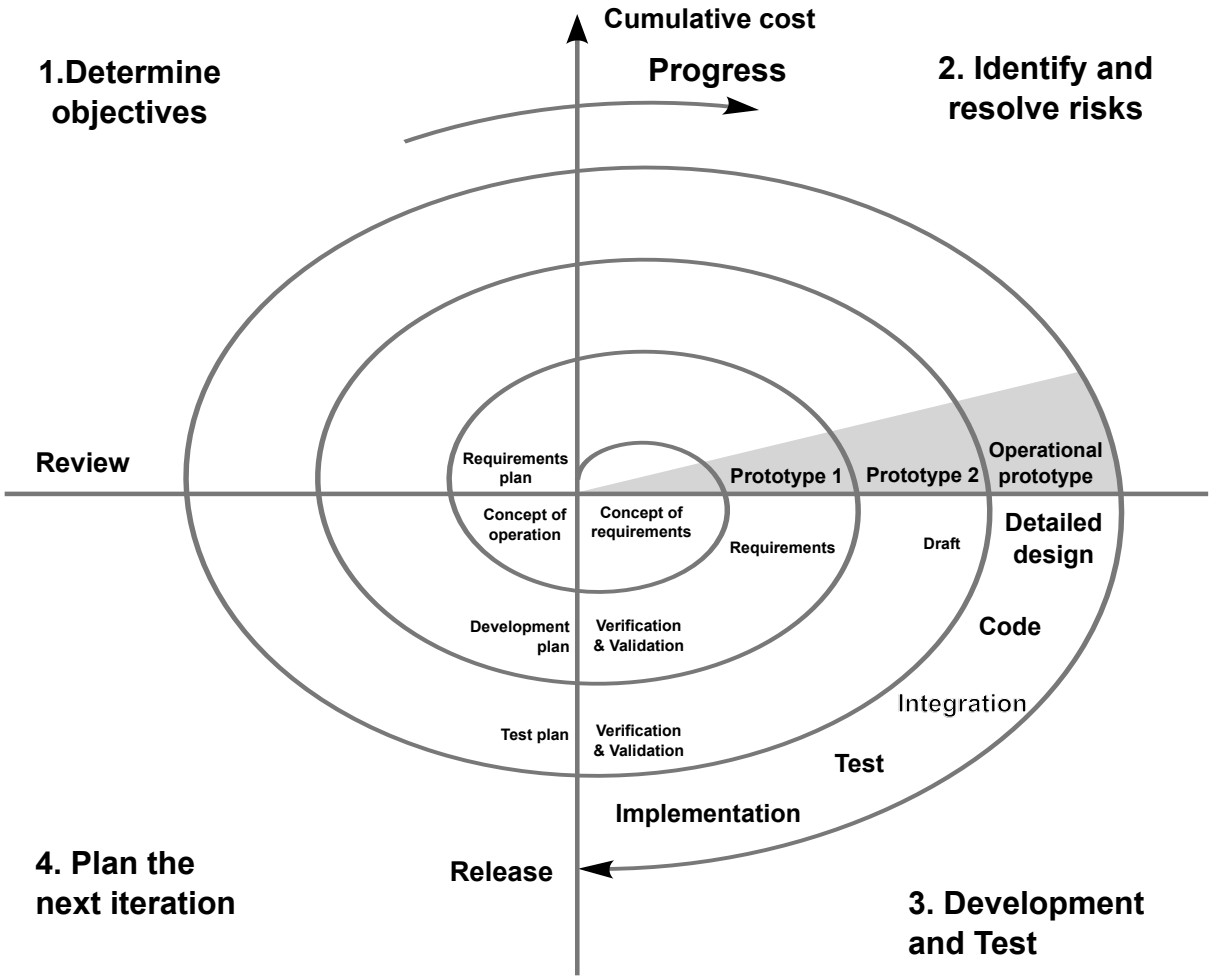


Figure 3.2: Spiral model (Boehm, 1986).

tailed in the system design. On the other hand, non-functional requirements define the user-level requirements that are not directly related to functionality and that are used to judge the operation of the system rather than an specific behavior. Some examples of these requirement are security, usability, maintainability, scalability, ... The plan for implementing non-functional requirements is detailed in the system architecture.

The following subsections describe both types of requirements for this project.

3.2.1 Functional requirements

Functional requirements consist in the description of the high-level functionality of the system and its behaviour.

As previously introduced, section 1.3, the main objective of GDAIS is to control instruments and acquire data from them, being able to adapt to different situations. To better contextualize the system, Figure 3.3 provides a diagram of the connections and relations between the system and its environment.

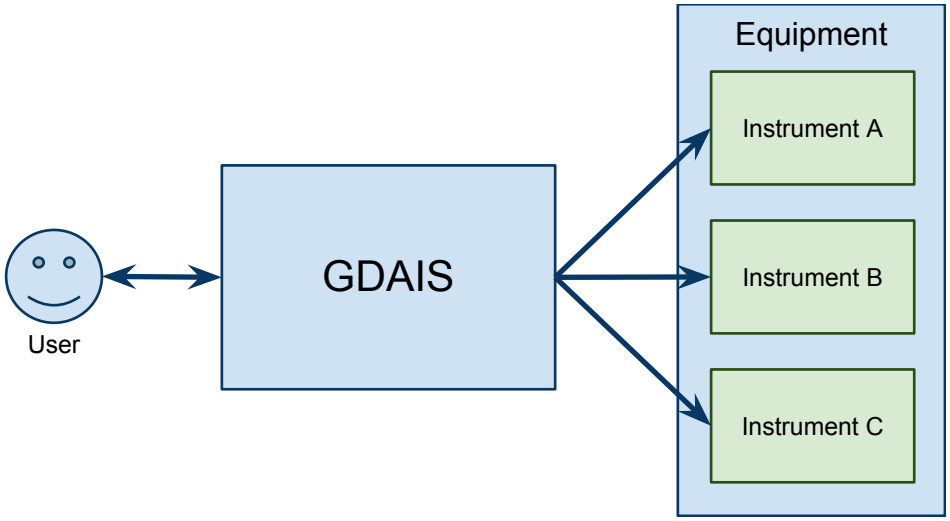


Figure 3.3: Generic context diagram of the system

In this diagram the concepts of instrument and equipment are represented. In the context of this project, an instrument refers to a single device that can be controlled from a computer and data acquired from it, sometimes it is also called a sensor. When data for an experiment has to be acquired from multiple instruments, the set of all these instruments is referred as an equipment.

The main functionality of GDAIS system is controlling one or more instruments in an equipment and acquiring data for them. The usual workflow for a single instrument consists of creating a connection with it, sending initialization commands if the instrument

requires an initial setup, and finally starting the acquisition of measurement data. As seen in section 2.1.2, the acquisition phase depends on the operation mode of the device: some instruments have a command to start a measurement and then acquired data is returned to the system, and some others continuously send measurements. Whenever new measurement data is received, GDAIS has to parse it, which consists in identifying the start and the end of a packet[1] and then converting the binary data inside this packet to meaningful values — integer, floating point, ...— that can be stored in a file.

When multiple instruments are used in the same experiment, the process is quite similar. For each instrument the system has to proceed as before independently from the other devices, except once the data is ready to store to a file. Then, the measurements from all instruments in the equipment should be saved together in a single file. This way, when an acquisition is finished, everything is stored in a single file that can be used for later processing the information received from all instruments of the equipment together.

In addition to the main functionality, some other features that complement the main one should be provided by GDAIS. As the system has to adapt to many different instrument types, it will need a detailed description of how to interact with a specific device and also with a set of instruments, an equipment. To help the user in creating these descriptions, a GUI for creating, modifying and saving instrument and equipment description files should be provided with the system. Moreover, some interface has to be provided to control the system, monitor it and retrieve acquired data files. This interface should be accessible both from the computer where GDAIS is running and remotely through Internet.

After this general description of the system and in order to completely describe the functionalities and behaviour of the system, in the following list the functional requirements of the system are detailed:

Interface to create instrument description

The system has to provide a GUI to create description files of the instruments which will be used by the system to connect, interact and acquire from them. This description has to include the connection type and parameters, transmission and reception packets structure and the fields included in each packet.

Interface to create equipment description

In addition to instrument descriptions, the system has to provide a GUI to create description files of equipments. For each equipment the description has to include all the instruments it includes and, for each instrument, its initial configuration

[1] A packet is a structure that contains a message and consists of an identifier and a set of values, which are referred as fields. Packets can be converted to, and from, a binary sequence that can travel through any connection.

sequence, the acquisition mode and the command sequences while acquisition is running, depending on the operation mode.

Multiple instruments at the same time

As an equipment may consist of one or more instruments, the system has to be able to interact with multiple connected instruments in parallel and store the acquired data from all of them together in an organized way.

Perform initial configuration of devices

Some devices need to receive a sequence of commands before they start operating in a convenient mode to acquire data from them. For these devices, the system has to be able to send these commands before it begins the acquisition mode.

Start and stop data acquisition

The system has to provide an interface for the user to start and stop the acquisition process for an equipment.

Send commands to device

The system has to be able to send messages to each instrument. These messages include configuration commands, operation mode selection commands and commands to request specific data packets to be sent from the device.

Convert values to binary data

Whenever a command has to be sent to the device, it has to be converted into its binary form, generating a sequence of bits with the packet structure associated with the command and the values for each field. To transform values to bit sequences the description of the instrument is also used.

Interact with the device

The system has to be able to connect to a device through a data connection to send and receive data. This includes setting up a connection, converting packets to binary data sequences which can be transported through the connection, and the reverse process, converting connection data to a binary sequence which can be processed and data packets extracted from it.

Convert binary data to values

Whenever information is received from the device, it needs to be converted to meaningful structured values. For each measurement the instrument sends a packet to the system that has to be detected, identified and parsed. Parsing a packet consists in, once its start and end have been correctly found, joining bits of data into blocks of bits that can be converted to values — integer, floating point, characters, ... — and assigning a name to each field from the description of the instrument.

Store acquired data of each execution in a file

Once data has been acquired and binary information converted to meaningful values, each measurement packet is stored in a structured binary file. For each acquisition execution a new file is created with all the information acquired from the instruments which form the equipment, classified by instrument and packet type.

Acquire data in different modes

The system has to implement different acquisition modes to cope with as many devices as possible.

Provide remote access to acquired data

The system has to provide an interface to remotely retrieve files with data from previous acquisition sessions for each equipment.

Report the status of the system

The system has to report any information that may be of interest for the user and log it to be reviewed afterwards if needed. Status messages will be logged in 4 different levels — error, warning, info and debug — to facilitate the identification of the most important ones (error level) while also providing exhaustive information (debug level).

Interface to monitor system status

The system has to provide an interface which shows to the user the status of the system, whether it is running or stopped, and any messages that may provide valuable information to the user on what is happening on the device. This interface has to be accessible from the computer where the systems is executed as well as from remote computers.

Fail gracefully on errors

In case of error, the system has to report a detailed description of the problem to the logging system and try to recover from the error. If the error is unrecoverable, for example when connection with the instrument can not be established, acquisition session should be finished. However, remote monitoring and control interface should always remain available and, in case of serious problem, restart with a default setup that is known to work.

3.2.2 Non-functional requirements

In contrast with functional requirements, non-functional requirements describe the user-level requirements that are not directly related to functionality. For this project, the

following requirements have been considered:

Reliability

As the aim of this system is to acquire data from instruments that may be in unreachable or poorly connected locations, reliability is an essential requirement for this project. Furthermore, due to this software providing remote control and monitoring of the instrument, it should be reachable in any conditions and should be able to recover from errors, because in case of loosing remote connectivity the control of the device would be lost.

Usability

To ease the interaction with the system, all implemented functionality will be accessible through a GUI. As this software is clearly not intended for computer programmers, this GUI has to be as simple and intuitive as possible, not expecting that the user knows how it has been programed or that a specific interaction flow will be followed. Moreover, reference documentation will be provided for the application to help the user with complex or advanced features.

Reusability

As in the first word of GDAIS — generic — the system shall be reusable to adapt, with little effort, to each possible instrument that may be used. Furthermore, the developed solution shall be simple to extend with new functionalities and easy to adapt to new situations.

Maintainability

Once the application has been developed, in case errors are found, they shall be easily repairable. Programming style should be plain and clear, and code well-documented so that anyone can modify and solve an error without relying on the programmer who wrote the original source code.

Portability

For the system to be as generic as possible and reused in different projects, it shall be flexible on the working environment requirements. Specifically, the system shall work in different hardware architectures and the main operating systems including Microsoft Windows, GNU/Linux and Mac OS X, without requiring many changes. Thus, it should rely on multi-platform programming languages and libraries for its implementation.

Compatibility

As a generic solution, the system shall adapt to multiple different instruments, through several communication interfaces and protocols. Moreover, acquired data

shall be stored using a standard file format that can be read from most relevant scientific and engineering applications, assuring that, in the future, data will remain compatible with new versions of the programs.

Interface

User interface will be provided in different forms to adapt to each situation the system may have to cope with. On one hand, to configure and test the system while developing the instrument a GUI will be useful for a faster and simpler usage. On the other hand, when the instrument is installed in an outside location with poor connectivity it may be more useful a command-line interface (CLI) that can be accessed through Telnet or Secure Shell (SSH), which are network protocols that have lower bandwidth requirements than the required ones for a full GUI.

Legal

As the system is intended as a base for future instrument control and data acquisition applications, its license should allow source code to be reused and modified. However, this licensing should ensure that original developers are credited for their work.

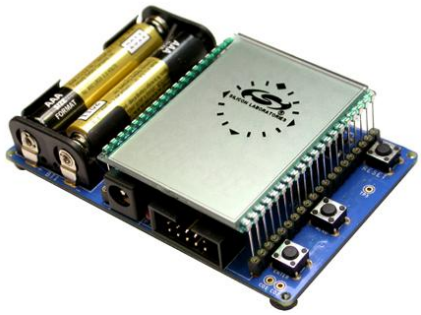
3.3 Prototype

As detailed in section 3.1, a spiral development methodology has been chosen for this project, producing a prototype after the first iteration. Therefore, this section consists in the development of a prototype of the system, based on the requirements previously analyzed.

In order to reduce the complexity of the prototype, this first development only implements the main functionality of GDAIS for two simple instruments that do not have an initialization phase. Consequently, only two different operation modes are considered in this version.

3.3.1 Tested instruments

Next, a description of each of the instruments used to test the prototype system follows. Both can be seen in Figure 3.4.



(a) Silicon Laboratories F350 compass

(b) Trimble - Lassen SK II

Figure 3.4: Instruments tested in the prototype iteration

Table 3.1: F350 compass serial port settings

Setting	Value
Baud Rate	57600
Data Bits	8
Parity	N
Flow Control	disabled

Silicon Laboratories F350-Compass-RD™

F350-Compass-RD (Figure 3.4a) is a digital compass that also provides temperature and inclination information. This device can be connected to a computer through USB and provides a virtual serial interface over this connection using a CP2101 USB to Universal Asynchronous Receiver/Transmitter (UART) bridge controller. The settings for this serial connection can be found in Table 3.1.

The communication protocol with the device is quite simple: whenever a measurement is desired, a 1 byte command is send to the device and it replies a sequence of 9 bytes with orientation, temperature and inclination measured values.

Trimble - Lassen SK II GPS

Lassen SK II GPS (Figure 3.4b) is a parallel tracking Global Positioning System (GPS) receiver designed to operate with the L1 frequency, Standard Position Service, Coarse Acquisition code. This device provides a serial port that can be connected to a computer to interact with it. The settings for this serial connection can be found in Table 3.2.

In contrast with the first instrument, apart from being able to provide some measurements in reply to a command, this device also sends measurements periodically at a constant

Table 3.2: Lassen SK II GPS serial port settings

setting	value
Baud Rate	9600
Data Bits	8
Parity	N
Flow Control	disabled

rate. The prototype will be able to receive and store these periodic measurements, and also send a command to request satellite tracking status information and save the reply.

3.3.2 Architecture

As a simplified version of the system, this prototype will be implemented as a single application providing all the functionality and also the GUI for the user to test the application. The workflow, represented in Figure 3.5, will also be simple: the user provides a description of the instrument in a text file, then the system starts acquiring data based on the information on the provided file and stores received measurements to a file that is saved in the `data` folder.

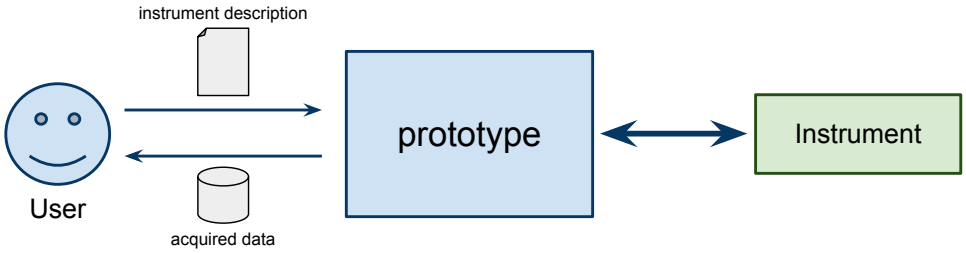


Figure 3.5: Operation workflow of the prototype

Internally, the prototype has been divided into 3 modules:

Instrument settings This module interprets the contents of the text file and creates a representation of it in the memory of the program.

Data Acquisition This module implements the process to acquire measurement data from the instrument, sending commands if needed.

GUI This module interfaces with the user to obtain configuration information and shows system messages.

The second module, *Data Acquisition*, is the most complex one, thus it has been subdivided in 3 submodules, one for each part of the acquisition process:

Connection

This submodule implements the connection to the instrument, a serial connection in this simplified version, based on the parameters provided in the settings file. Once started, it listens continuously the connection until enough data is received, at least the binary length of a measurement packet, and then transmits this binary data to the parser. This submodule is also in charge of sending the periodic command to request new measurements if it is required by the instrument, defined in its description.

Parser

This submodule, using the information in the instrument description, identifies the packet fields in the binary data provided by the connection and extracts its values. Then, it converts these values to meaningful information and stores them in a memory structure, which contains the name and value of each field of the packet, to finally transfer it to the recorder module.

Data Recorder

Before acquisition starts, this submodule creates a data file with an structure prepared to store the acquired measured data in an organized way. Then, when acquisition starts, for each measurement it receives the structure created by the parser with the data and saves all the contained values to the file that was initially created.

With all these subdivisions, the final prototype architecture is represented in Figure 3.6. As it can be observed in this figure, apart from the advantage of reducing implementation complexity, the segmentation of the system also clearly separates its functionalities into 3 levels: user interface, application logic and data management.

3.3.3 Design

Once the architecture of the system has been defined, each module and submodule has to be designed. As most of these designs are reused in the final version of the system, just an overview of them is provided and further details are explained in section 3.6.

To implement the prototype, Python programming language has been chosen. Python is an object oriented high-level language with a flexible syntax and many libraries are available to be used with it. Some of these libraries are used in this prototype, the most relevant one of them being Qt, which is a cross-platform application and GUI framework that provides many features in a high quality and highly readable application programming interface (API). As occurs with the design details, technology choices are also further detailed in section 3.5.

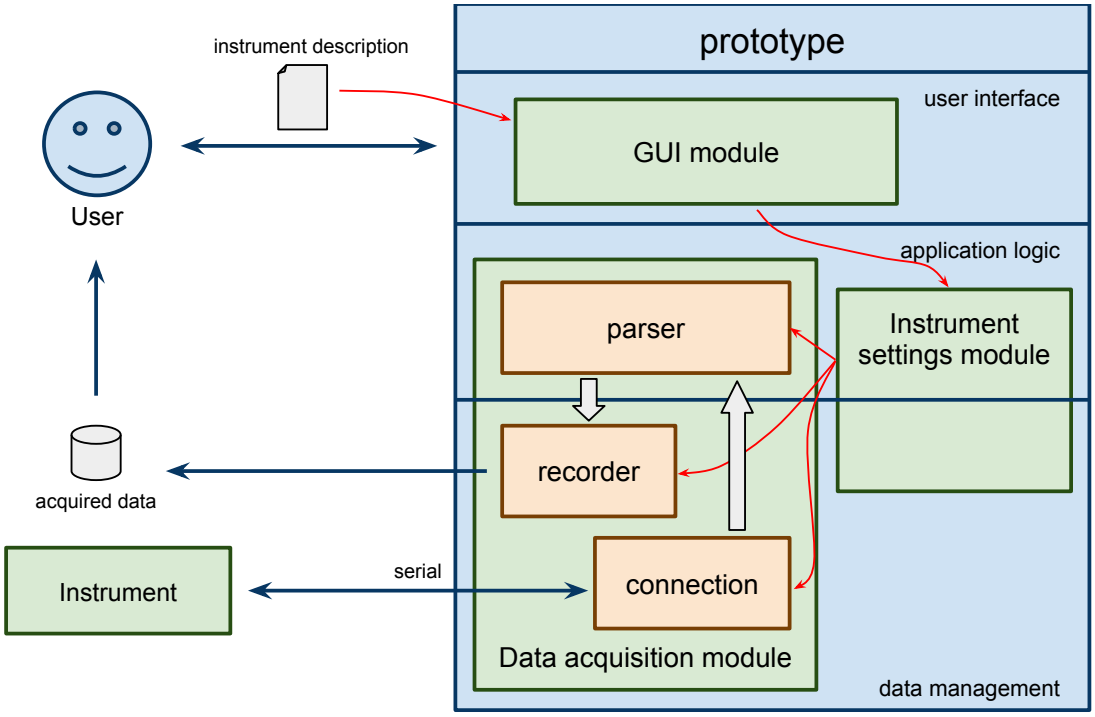


Figure 3.6: Prototype architecture diagram

Instrument settings module

This module is responsible of parsing the instrument settings file and providing a representation of it to the other modules. To ease the parsing of the text file, the JavaScript Object Notation (JSON) standard format was used to write the description of the instrument. As an example, the contents of the description of F-350 compass instrument is provided in Listing 3.1.

```
{
  "name": "Compass F350", "short_name": "compass_f350",
  "connection": { "type":"serial", "port":"/dev/ttyUSB0",
    "baudrate":57600, "data_bits":8, "parity":"N", "stop_bits":1 },
  "byte_order": "big-endian", "start_bytes": [], "end_bytes": [],
  "rx_packets": { "0": {
    "name": "Compass Data", "short_name": "compass_data",
    "short_fields": ["direction_degrees","direction_minutes",
      "temperature","inclination_x","inclination_y","status",
      "checksum"],
    "fields": ["Direction Degrees","Direction Minutes","Temperature",
      "Inclination X","Inclination Y","Status","Checksum"],
    "types":["uint16","uint8","uint16","uint8","uint8","uint8","uint8"],
    "parse_string": "!hBhBBBB" } },
  "tx_packets": { "17": {"name": "Request Compass Measure"} }
```

```
16 }
```

Listing 3.1: F-350 compass description file

This listing includes all the information of the instrument: its name, the serial connection settings (`connection`), the format of the packets to parse (`byte_order, start_bytes, end_bytes`) and its structure (`rx_packets, tx_packets`). For each packet its name, and fields name and types are included.

Python language itself is able to parse JSON text format and convert it to a structure. This module implementation just transforms this structure to an object of `Instrument` class that will be available globally to the other modules.

Data acquisition module

Provided that the submodules of data acquisition module are always executed sequentially for each measurement, a pipeline design has been chosen in its implementation. To achieve this behaviour each submodule has been implemented as a class that inherits from `QThread` Qt class, which implements all the functionalities needed to run class instances in independent threads. They communicate using a feature of Qt called *signals and slots*, which provides a high-level method to interchange messages between threads without having to consider any of the possible problems that may occur in multi-thread programming. See section 3.5.

The connection submodule uses Python `serial` module to implement the communication with the parameters stored in `Instrument` instance. Also, to periodically send a command to request a measurement, it defines a timer using `QTimer` class. When a new packet is received it *emits a signal*[2] with the received binary data and continues listening for more data.

The parser submodule, which runs in another thread, has a *slot*[3] defined for receiving binary data from the connection. This *slot* receives the generated *signal* with the binary information and extracts the values from it. This is implemented with the help of a Python module called `struct`, which is specially designed for extracting numeric values from binary sequences using a parsing string, which in this case is stored in the `parse_string` field of the instrument description, see Listing 3.1. Once it has the value for each field, it *emits a new signal* with the values.

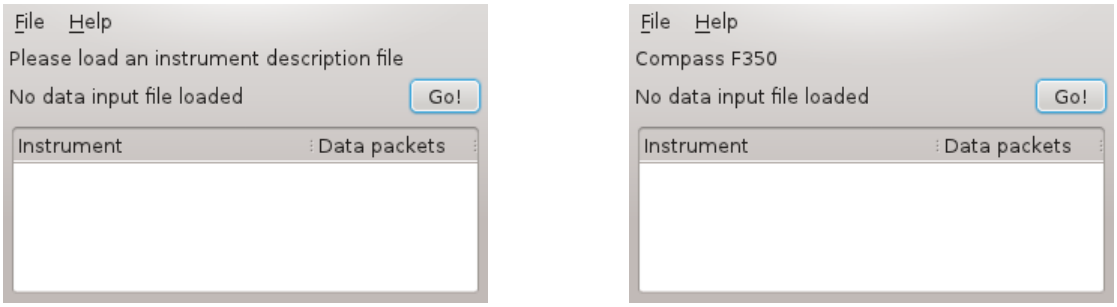
[2] A *signal* is an event that a `QObject` instance may create, *emit*, and that some other `QObject` instances can monitor, *connecting* this *signal* to one of their *slots*.

[3] A *slot* is a method of a `QObject` instance, the parser `QThread` in this case, that can receive *signals* from other `QObject` instances, the connection `QThread` for example.

Finally, the recorder thread receives the information in a defined *slot* and saves it in a file using `tables` Python module, which is able to create a binary file in HDF5 format. This file has been previously initialized with the information of the packets contained in the description of the instrument.

GUI module

The user interface module of the prototype is designed to be simple and provide access to its features and acquired data details. In Figure 3.7 the designed GUI is shown.



(a) Initial view (b) F-350 compass description loaded

Figure 3.7: Prototype user interface

Once an instrument description file has been loaded (Figure 3.7b), when the "Go!" button is pressed the system begins acquiring data and displays each measurement received, as in Figure 3.8.

To achieve this behaviour the code that implements the GUI is responsible for creating all the other modules instances and starting the threads for each submodule of the data acquisition module. Moreover, it also uses the *signals and slots* system to receive notifications of new received measurements from the parser and display its values.

3.4 Software architecture

Based on the analysis of the requirements and functionalities expected from GDAIS and the experience gained during the development of the prototype iteration, this section explains the proposed final architecture of the system.

From requirements analysis phase (section 3.2) there are clearly some functionalities that can be joined into independent groups, defining the main blocks of the system as follows:

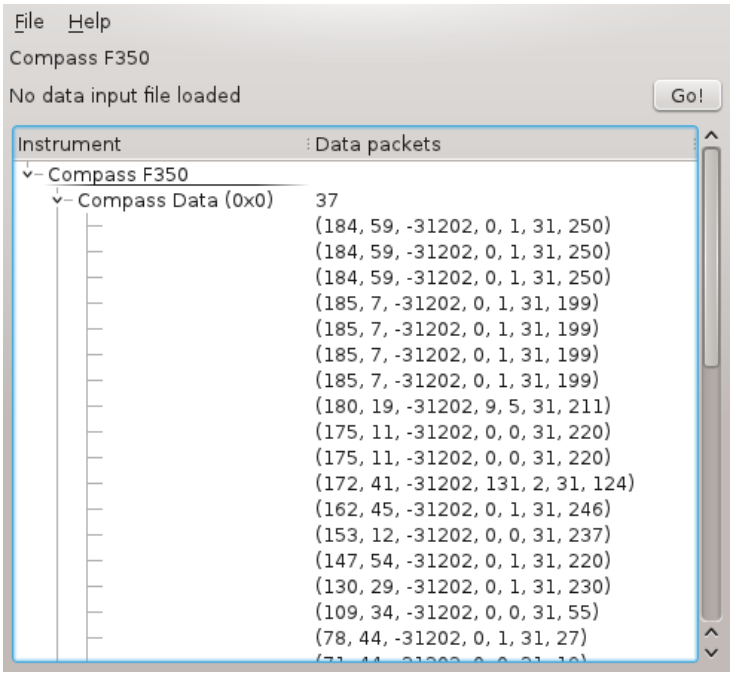


Figure 3.8: Prototype user interface while acquiring data

GDAIS-core

Provides the main functionality of the system: instrument control and data acquisition. To fulfill the requirements of being reliable and autonomous, this block should not have any interface available to the user and should be able to run independently. However, in most situations some control and monitoring of this block will be desirable, therefore a communication interface with GDAIS-control block is also included.

GDAIS-control

This block is aimed to provide the user interface functionalities not included in the main GDAIS-core block. In order to satisfy the requirements of remote control and access to acquired data, this user interface should be accessible both from the computer where GDAIS-core is running and from any other computer through a network connection.

Instrument editor

As stated in requirements section, a GUI for the user to create a description of each instrument involved in an experiment has to be included. This interface has to include all the properties that can be defined for an instrument in an intuitive and organized way, and generate a text file that will be parser afterwards by GDAIS-core to know how to interact with the described instrument.

Equipment editor

The same applies to the GUI for generating a description of an equipment, the set of instruments included in the experiment. This block is able to load instrument descriptions generated by the instrument editor, thus preventing the inconsistencies between instrument and equipment descriptions. Even if just one instrument has to be controlled by GDAIS-core it has to be included in an equipment description, as it defines the operation mode of the instrument in the experiment.

Having defined these blocks, the usual workflow for the whole system is represented in Figure 3.9. It should be noted that the full workflow is hardly ever followed. Instead, once an instrument or equipment has been described, its description file can be reused anytime. Moreover, GDAIS-core block is designed to work independently from GDAIS-control, without any user interaction. In this non-interactive mode, GDAIS-core can be started with an equipment description provided as a parameter and it will generate a file with all the acquired data that can be retrieved from a folder when the acquisition is completed.

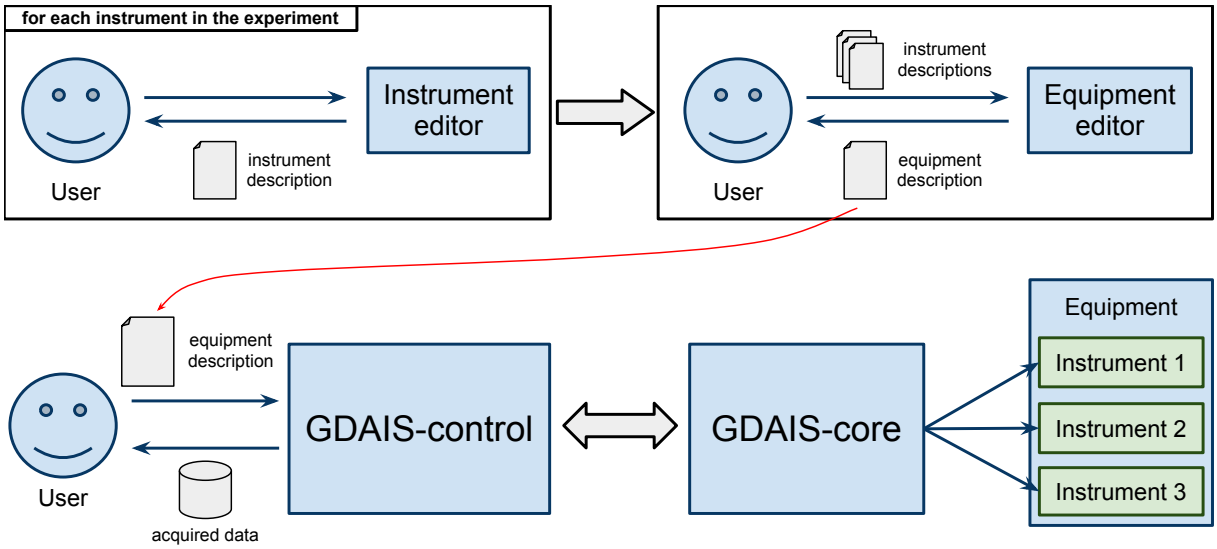


Figure 3.9: GDAIS system workflow

In the next subsections the architecture of each block is detailed and explained.

3.4.1 GDAIS-core

This block extends the functionalities implemented in the prototype phase, as it is responsible for the communication with all the instruments and recording the received measurements from all of them to a single file. As these functionalities are clearly separable, a module has been planned for the interaction with each instrument and another module

for recording the received measurements. The detailed architecture defined for this block, which is represented in Figure 3.10, has the following structure:

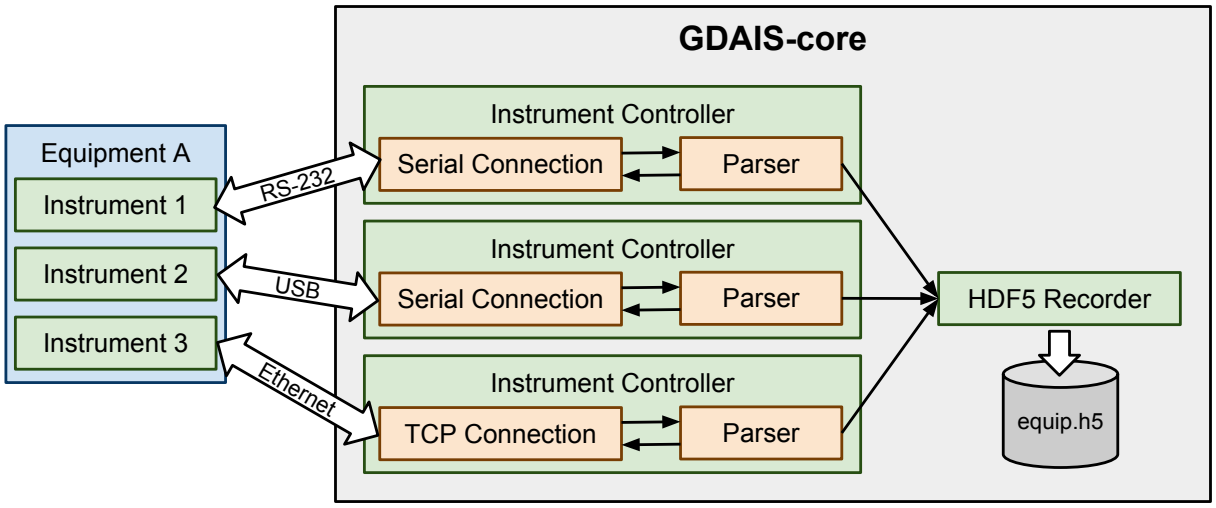


Figure 3.10: GDAIS-core architecture

Equipment description

This module interprets the contents of the equipment description text file that GDAIS-core receives as input and creates a representation of it in the memory of the program.

Instrument controller

For each instrument described in the equipment file, a copy of this module is created and associated with it. Then, each of these copies is responsible for communicating with the linked instrument and providing the information of the measurements received from it. As in the *Data acquisition* module of the prototype, a similar subdivision has been considered in this module:

Instrument description

As in the prototype, this submodule parses the instrument description text file and creates a representation of it, available to the other submodules.

Connection

This submodule is a generalization of the submodule developed in the prototype, as now it has to be able to adapt to different connection types and other operation modes. Moreover, to generalize the format of the commands sent to the instrument, the transmitting capabilities are reduce to sending any binary sequence that is provided by another submodule.

Parser

This module includes the functionality of processing the received binary data

already available in the prototype, and also adds the responsibility of creating binary sequence to send commands to the instrument.

Data recorder

As the final system should be able to work with multiple instruments together, this module has to join the data received from each of them and write it to a single file for the equipment. Therefore, before data acquisition begins, a new file is created with an structure representing each instrument and the possible measurements types that it may receive.

In addition to the main functionalities of this block, some other modules are needed to provide the communication interface with the GDAIS-control block.

Control server

This module provides an interface that listens for control commands from an external application like GDAIS-control. For example, the command to stop acquisition or change the verbosity level of the notifications.

Notifier

This module is responsible for transmitting any event that occurs in the system to an external application. The verbosity level, and thus the number of notifications has to be controllable, as in some high-demanding applications having many notifications may slowdown the whole system.

3.4.2 GDAIS-control

This block provides a remote user interface for GDAIS-core. It should be able to start an acquisition, show notification messages during acquisition phase, stop the acquisition and provide access to created data files.

To satisfy the requirement of both local and remote access, and also the portability non-functional requirement, this block is implemented as a web application. Therefore, its architecture is based on the client-server architectural pattern, which is represented in Figure 3.11.

This pattern has some clear advantages:

- Any computer or device with a web browser can access the GUI of the application.
- Access can be both local or from any network location connected with the application.

- GUI has to be implemented only once in a single programming language and environment, thus any modification or bug fixes are easier to apply.
- As the whole application is provided by the server, when an update is available the user does not have to download the new version, which does happen in desktop applications.

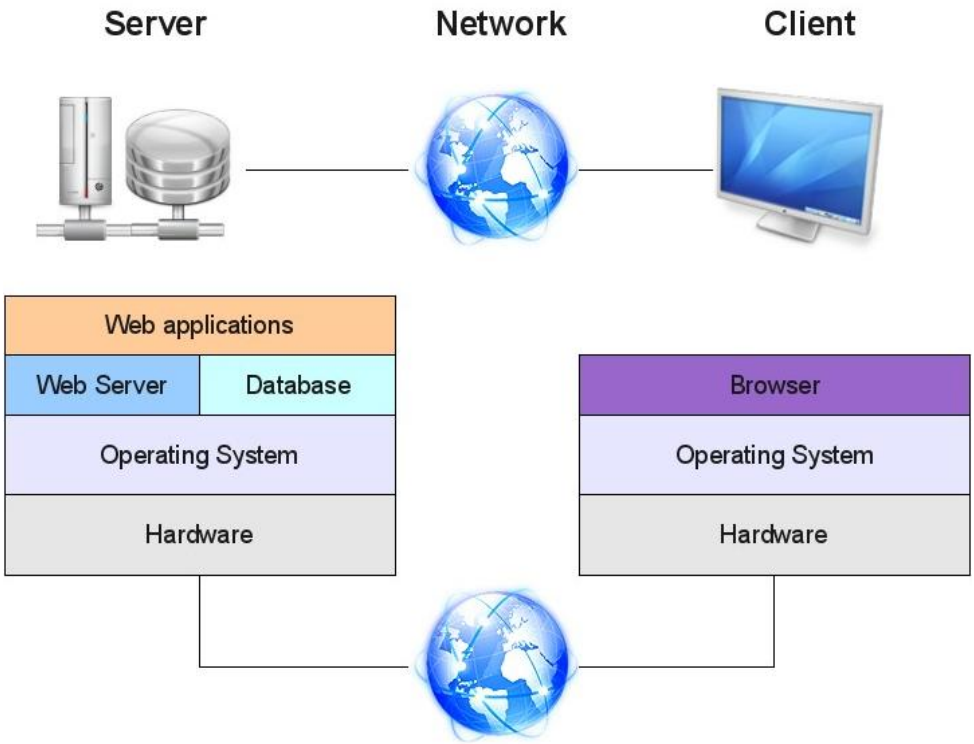


Figure 3.11: Client-Server architecture

3.4.3 Instrument and equipment editors

These two blocks are very similar, both of them provide a GUI to create and edit text file descriptions of instruments or equipments. In order to prevent code duplication, equipment description and instrument description modules from GDAIS-core block are reused in these blocks. Thanks to this, its implementation is reduced to just implementing a GUI module. This interface has to provide access to all the values and properties defined in the configuration text files in a user-friendly manner.

In contrast with GDAIS-control block, as these blocks do not require access to GDAIS-core, a usual desktop application architecture has been chosen to implement them. The internal architecture of instrument editor block is represented in Figure 3.12 and the equipment editor architecture is represented in Figure 3.13. In this last figure, it can be observed that equipment settings module uses the instrument settings module to obtain

instrument details from the instrument description files provided by the user when a new instrument is to be added to the equipment. Thanks to this, the equipment descriptions are always consistent with the description of the instruments it contains.

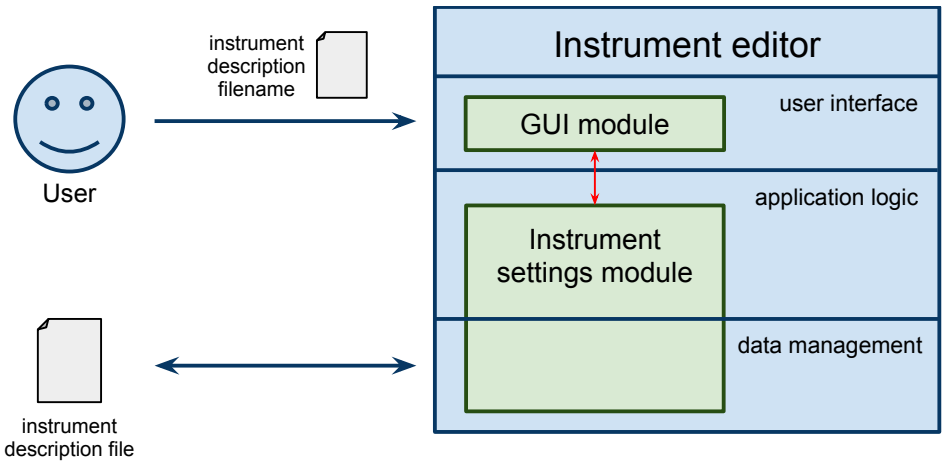


Figure 3.12: Instrument editor architecture

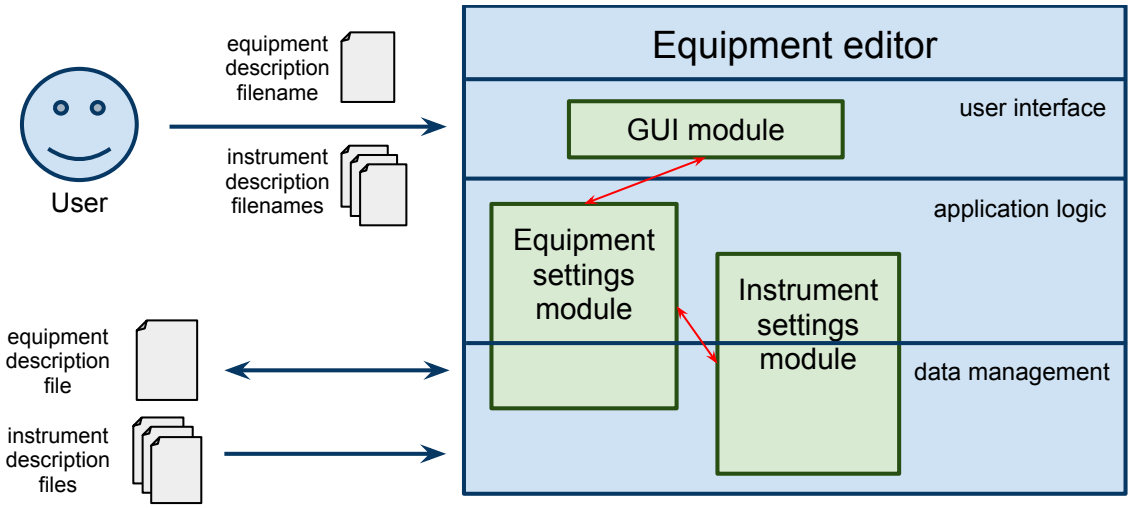


Figure 3.13: Equipment editor architecture

3.5 Technologies

Once the architecture of the whole software system and each block has been defined, and before starting with the design details for each block, an analysis of the technologies used to develop the application is provided, as the final design depends on the requirements and features of these technologies.

Based on the requirements of the project and the architecture chosen for each of the blocks that conform GDAIS system, the following technologies have been selected to implement it:

Python

Python is an interpreted, general-purpose high-level programming language whose design philosophy emphasizes code readability. As stated in the official website of Python programming language,

> Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs.

Some of the reasons for choosing Python programming language are:

Python is free to use...

...even for commercial products, because of its Open Source Initiative (OSI)-approved open source license, the Python Software Foundation (PSF) license agreement, that allows distribution of modified versions without making the changes open. Moreover, this license is GPL-compatible, which means that Python can be combined with other software that is released under the General Public License (GPL).

Python is a high-level language

Python syntax is very clear and readable. It provides intuitive object orientation, natural expression of procedural code and full modularity, supporting hierarchical packages. Error handling is based on exceptions and very high level dynamic data types like strings, lists, queues, and dictionaries are provided. Furthermore, the language also includes some advanced features such as meta-classes, duck typing and decorators.

Python is productive both for beginners and experts alike

Python has a short learning curve and most people can do real and useful work with it in a day of learning. Its clean syntax and interactive nature facilitate this. This makes this language accessible to engineers, scientists, and others who consider programming a necessary evil. In the context of this project, this is a clear advantage, as the final users will be instrument developers which prefer spending time working on the devices, not tracking down bugs and learning a new language.

Python code runs on any platform

Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines. Therefore, Python code can be run virtually in any system available without having to modify a single line of code. This is nearly always true, as some Python libraries rely on C libraries, which are platform-dependent, for some low-level interaction or to achieve higher efficiency and faster execution times.

Moreover, provided the portability of the C implementation of Python, CPython, code can also run in embedded environments, like Embedded Linux. This can be achieved using cross-compiling tools like OpenEmbedded, Scratchbox, Buildroot or Crosstool. Also, some Linux distributions which have an embedded version, like Debian GNU/Linux[4], already provide Python compiled packages to run on these platforms. In the context of GDAIS project this is very interesting since, once implemented, the system will be able to run in an embedded system, being able to integrate with the equipment itself, the control and data acquisition, and storage.

Python comes with "batteries included"

This is the usual phrase used by Python enthusiasts to describe its large and comprehensive standard library, which provides everything from asynchronous processing to zip files. GDAIS project makes extended used of this standard library to reduce the external dependencies of the resulting implementation. Some of the features implemented thanks to these libraries are binary to numeric conversions, numeric data processing, serial connections, file access, regular expressions, error and debug messages logging, program arguments parsing, ...

Python has many libraries available

In addition to the standard library included with the language itself, lots of third party modules are available which provide support for virtually every possible task. Extension modules are usually implemented in either Python, C or C++. In this project, some of these externals modules have been used, for example PyTables and PyQt. Moreover, in case of finding something Python cannot do, or if low-level performance advantage is required, extension modules can be written in C or C++ or existing code can be wrapped with the help of some libraries.

Python is interactive

Most used programming languages, like C or Java, need to compile the code before any change can be executed. In Python, as it is an interpreted language, this is not needed, and this feature permits to use this language as a command line interface, sequentially introducing and executing code lines. This is specially useful when

[4]Embedded Debian Project http://www.emdebian.org

working with scientific data sets, testing network connections and interacting with small parts of the application while it is being developed.

All these reasons have determined the decision to base the whole GDAIS system on Python programming language.

JSON

When deciding on the implementation of instrument and equipment description files, some available representations were considered. It was initially decided to used a text-based format to allow the user to modify its contents without having to rely on any other tool.

After analyzing some possibilities, there were two technologies chosen for detailed evaluation:

XML

This a set of rules for encoding documents in a machine-readable text form. It has a gratis open standard specification, and emphasizes simplicity, generality, and usability over the Internet. It was originally designed to represent documents, but is widely used for the representation of arbitrary data structures. XML itself is not a language, it requires a schema definition, which details its final format. For this reason, to read from XML files a parser has to be implemented. Moreover, documents encoded in this format are usually bigger than others, as it adds a lot of text to define the structure of the document. An example XML document can be found in Listing 2.1.

JSON

This is a lightweight text-based open standard designed for human-readable data interchange and described in RFC 4627. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship with JavaScript, it is language independent, with parsers available for most languages. Python is one of these languages, as it includes in its standard library a JSON module with parsing and generation capabilities for this format. In comparison with XML, this format is more focused on representing structured data, rather than narrative documents. Furthermore, it provides a much low-overhead format, reason that makes it adequate for serializing and transmitting structured data over a network connection. A JSON format example file can be found in Listing 3.1, in the prototype design section.

Finally, the decision was made to use JSON format to represent instrument and equipment descriptions, as its features suited perfectly the requirements of this representation, and

it was clearly better than XML for this application. Moreover, as Python was used as implementation language, the creation and parsing of configuration files was simplified to a single method call of the Python JSON module.

HDF5 and PyTables

As GDAIS system final aim is to provide acquired data to the user in a binary structured format, this format had to be perfectly chosen to assure the success of the project. For this reason many formats were considered before making a decision. The requirements for this format to be useful in this application were:

Structured format so that retrieving stored information was simplified.

Hierarchical organization to represent instruments inside an equipment and packets inside the instrument.

Scientific data oriented as acquired data will be arrays of numeric values, resulting from measurements.

Compression available as sometimes acquisition output can be large.

Python friendly to easily integrate into the implementation.

Compatible with engineering and scientific tools as the generated files are to be used for data analysis and results extraction without requiring format conversions.

Based on these requirements, non-scientific binary formats, as databases, and text-formats were discarded. As previously explained in subsection 2.1.2, some scientific data oriented formats exist. These are specifically designed for storing and manipulating multi-dimensional scientific data sets, and they are based on public standards, which allows anyone to access the contained data and results in most used scientific applications providing interfaces to read and write them.

At the same time that scientific formats were evaluated, research on Python binary data storage libraries was performed. This resulted in finding a Python package specially designed for managing hierarchical datasets and to efficiently cope with extremely large amounts of data, PyTables. Coincidentally, this packages were built on top of HDF5 storage format, which was between the ones previously chosen for its scientific-oriented binary data representation and for fulfilling all the previously detailed requirements.

Provided this discovery, it was decided to analyse in detail HDF5 binary storage format and PyTables Python packages, as the solution to implement data acquisition persistence in GDAIS application. Next follows this detailed analysis of both technologies.

HDF5

Hierarchical Data Format (HDF) refers to a set of formats and libraries, originally developed at the National Center for Supercomputing Applications (NCSA), and designed to store and organized large amounts of numerical data. Specifically, HDF5 format is the latest version of this format available and provides a simplified file structure based on datasets, which are multidimensional arrays of a homogeneous type, and groups, which are container structures which can hold datasets and other groups. These structures perfectly fit the requirements of GDAIS system, as groups can be used to hierarchically represent equipments, instruments and packet types, and datasets can be used to represent the acquired packets of each type, with its values organized by field name and with the right type (integer, float, double,...).

HDF5 is the adequate solution for GDAIS data storage for the following reasons:

- Nearly any data object can be represented with is associated metadata.
- Datasets can be organized hierarchically.
- The format is completeley portable, with no limits on the contents dimensions.
- Very flexible and well tested in scientific environments.
- Designed to efficiently manage very large datasets.
- Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection are available.
- Technical excellence.[5]

Provided all these advantages, it was decided to use HDF5 format as the data acquisition storage format for GDAIS. Moreover, the latest version of another well-known and widely used format, NetCDF-4, is based on HDF5, and PyTables generated files can be read by NetCDF-4 viewers and libraries. Finally, another important factor for this decision was that recent releases of MATLAB software use HDF5 as its primary storage format[6]. Therefore, this format can be easily imported and manipulated from this important engineering tool.

PyTables

PyTables is a Python package for managing hierarchical datasets efficiently. It is integrated with NumPy package, which allows Python programs to efficiently deal with large datasets in-memory, providing containers for both homogeneous and

[5] R&D 100 Award http://www.hdfgroup.com/HDF5/RD100-2002/

[6] Version 7.3 MAT-files http://www.mathworks.com/help/techdoc/import_export/braidzi-1.html#braid3s

heterogeneous data and optimized operations to apply on these data. As previously stated, on-disk data representation is provided by HDF5 library.

PyTables design goals are the following ones:

- Allow to structure data in a hierarchical form, following HDF5 hierarchical groups structure.
- Easy to use, implementing an naming scheme that allows convenient access to the data.
- Most I/O operation should only be limited by the underlying I/O subsystem, be it disk or memory.
- Enable the end user to save and deal with large datasets in an efficient way.

As PyTables design goals completely covered the requirements for this system, it was chosen as the data storage library. Moreover, PyTables is *Open Source* software, licensed under the liberal Berkeley Software Distribution (BSD) license, allowing to freely adapt the library to specific needs and include it in any software, even if it is commercial.

PyQt

PyQt is a set of Python bindings of Nokia's cross-platform application and User Interface (UI) framework Qt. Like Qt, PyQt is free software available under both the GNU General Public License (GNU GPL) and a commercial license. It is developed by Riverbank Computing and supports Linux, Unix flavours, Mac OS X and Microsoft Windows.

As a binding of Qt toolkit, all of its features are inherited from it and available using Python syntax and programming idioms. These features include a GUI toolkit, thread management and network support. Projects using Qt include Autodesk Maya, Google Earth, KDE, Skype, VLC media player and it is used by the European Space Agency (ESA), Siemens, Volvo, Samsung, Phillips and Panasonic. Therefore, it is a well-tested and flexible platform for developing applications like GDAIS.

Moreover, Qt is officially released on many platforms: Linux/X11, Mac OS X, Microsoft Windows, Embedded Linux, Windows CE/Mobile, Symbian and Maemo. Provided its open source availability, some non-official ports to other platforms have also appeared: Qt-iPhone, Qt for webOS, Qt for Amazon Kindle DX and Necessitas (Qt for Android).

Its official support for Embedded Linux and mobile platforms is specially interesting in the context of this project, as this would allow to easily port GDAIS to any of these platforms,

which can be used in low-consumption and cheaper devices that may be integrated directly with an equipment to control it and acquire data. However, this only refers to Qt, and GDAIS does not use this toolkit directly, instead it is used through its Python bindings. Luckily, PyQt can also run on the Embedded Linux platform[7]. So using GDAIS in an embedded system, integrated with the equipment, should be feasible.

To ease the development of GUIs, Qt includes Qt Designer, a graphical user interface designer. Thanks to PyQt, Python code can be generated from Qt Designer UI description files. In Figure 3.14, a snapshot of this tool editing Instrument editor block of GDAIS is provided.

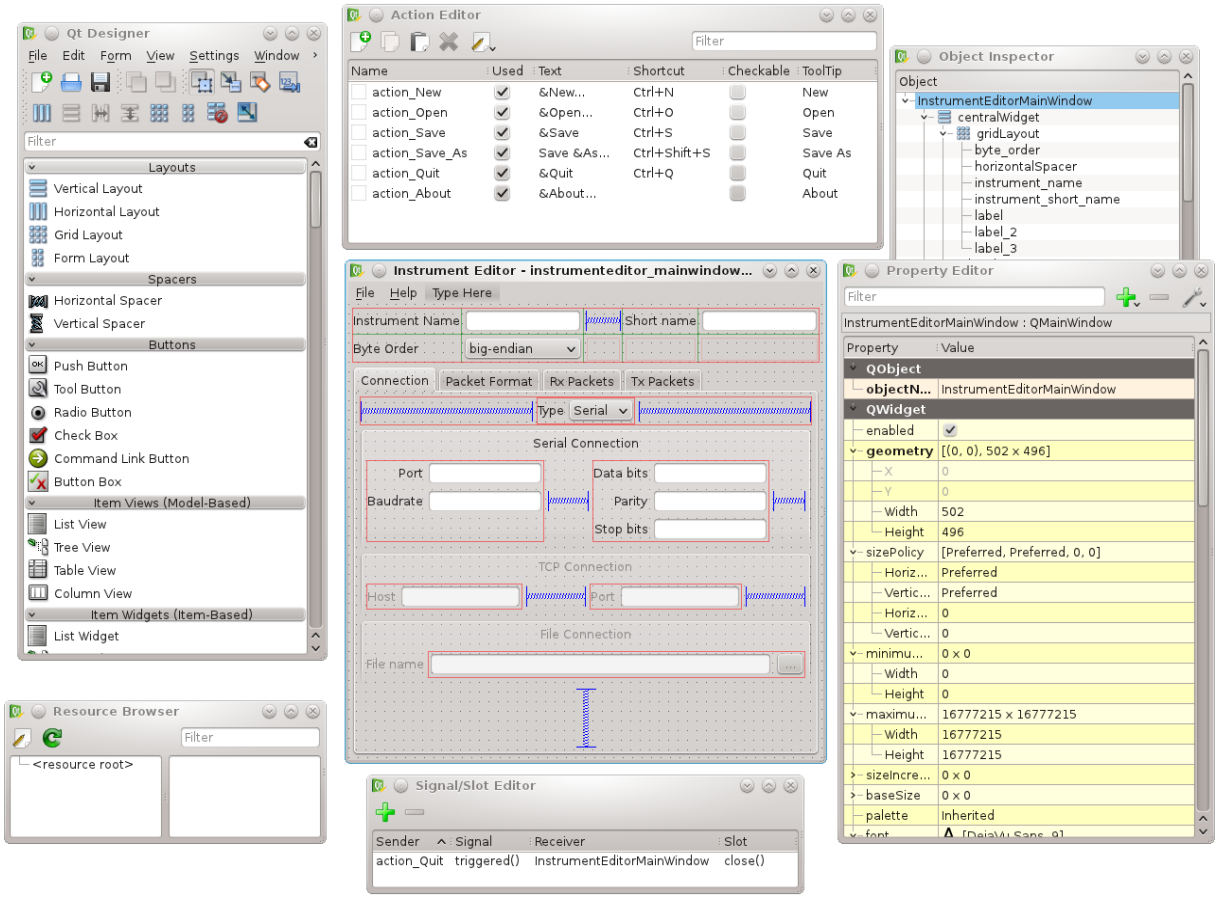


Figure 3.14: Qt Designer showing Instrument editor block

Qt, and thus PyQt, features are organized in a set of modules. GDAIS project makes extensive use of these Qt features. The most relevant Python modules of PyQt used in the system implementation are:

QtCore

This module contains the core non-GUI classes of the framework, including the

[7] PyQt4 for Embedded Linux https://bitbucket.org/dboddie/pyqt4-for-embedded-linux/overview

event loop and Qt signal and slot mechanism. The main features used in GDAIS from this module are the following ones:

QApplication and QCoreApplication

These two classes provide application control flow and main settings functionalities. They are designed as a base for any Qt application.

QCoreApplication provides an event loop for console non-GUI applications, where all events form the operating system (e.g., timer and network events) and other sources are processed and dispatched. It also handles the application's initialization and finalization, as well as system-wide and application-wide settings. This class is used as the base for GDAIS-core application.

QApplication inherits from QCoreApplication to add GUI application managing functionalities. Therefore, it depends on the QtGui module and also integrates to the main event loop all events from the window system to be processed and dispatched. This class is used as the base for Instrument and Equipment editor GUIs.

signals and slots

This feature makes it easy to implement the observer programming pattern in Qt applications. It is a central feature of Qt and probably the part that differs most from the features provided by other frameworks. As previously introduced in prototype design (subsection 3.3.3), a *signal is emitted* when a particular *event* occurs and a *slot* can be *connected* to this *signal* to take some action in the object where the *slot* is defined.

For example, in Figure 3.14, the Signal/Slot Editor window (at the bottom) shows that the *signal* `triggered()` from the `action_Quit` menu action is *connected* with the `close()` *slot* of `InstrumentEditorMainWindow` object, which represents the window that contains all the GUI. When the user selects the `Quit` menu action, the `triggered()` *signal is emitted* and, as this *signal* has been previously *connected* to `InstrumentEditorMainWindow` `close()` *slot*, the *slot* is executed, which makes the application exit.

Thanks to Qt Designer, this can be achieved without writing a single line of code, as each widget has some signals and slots defined by default, and they can be graphically connected.

This notification system is not limited to GUI objects. In fact, it can be used with any object of QObject class or that inherits from it. Thanks to this, two main GDAIS features are implemented using this mechanism: communication

between threads, when implemented in a class that inherits from QThread class, and event-driven networking applications using classes from QtNetwork module.

QThread

QThread Qt class provides a platform-independent implementation of threads. A QThread represents a separate thread of control within the application; it shares data with all the other threads within the process, but executes independently in the way that a separate program does on a multitasking operation system.

Implementing a new thread just requires defining a class that inherits from QThread class and implementing its `run()` method. Then, when a new thread is to be created, the previously defined class has to be instantiated, and the `start()` method has to be called to begin the execution of the new thread. When the thread has been set up, it calls the previously defined `run()` method. Execution of the thread ends when it returns from this method.

Each QThread can have its own event loop, which can be started calling `exec_()` from `run()` method. Having an event loop in a thread makes it possible to connect signals from other thread to slots in this thread, using a mechanism called queued connections, which allows communication between threads without having to cope with the usual synchronization problems of thread programming. This feature is extensively used by GDAIS, as when it was tested during prototype iteration (subsection 3.3.3) in the *Data acquisition* module implementation it proved to be very useful.

QtGui

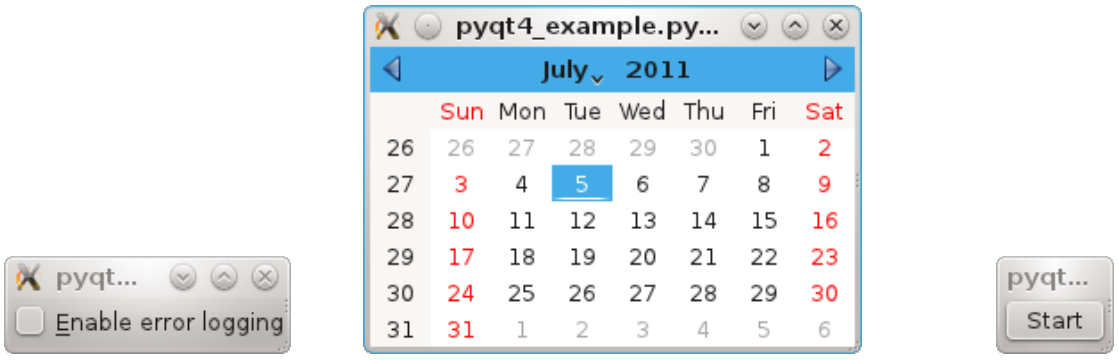
This module contains most of the GUI classes, which define the widgets, or components, that can be used to build the GUI. These provide the features which are used to implement the GUI of Instrument and Equipment editor blocks of GDAIS. Thanks to Qt Designer tool, most GUI elements and functionality do not need to be programmed in code, instead they are visually defined, and Qt, making extensive usage of *signals and slots* mechanism, takes care of bringing them to life.

Listing 3.2 shows an example code writen in PyQt that creates three Qt widgets which can be seen in Figure 3.15.

```
1 import sys
2 from PyQt4.QtGui import *
3
4 app = QApplication(sys.argv)
```

```
5
6  check_box = QCheckBox("&Enable error logging")
7  check_box.show()
8
9  push_button = QPushButton("Start")
10 push_button.show()
11
12 calendar = QCalendarWidget()
13 calendar.show()
14
15 sys.exit(app.exec_())
```

Listing 3.2: Simple PyQt application that shows three basic widgets



(a) QCheckBox widget (b) QCalendarWidget widget (c) QPushButton widget

Figure 3.15: Qt example widgets generated by PyQt code in Listing 3.2

QtNetwork

This module contains classes for writing UDP and TCP clients and servers. Developing of networked applications is made easy thanks to network events being integrated with the event loop provided by QApplication or QCoreApplication classes. All GDAIS-core connectivity features have been implemented using this module. The most relevant classes used are:

QNetworkAccessManager

QNetworkAccessManager class allows a Qt application to send network requests and receive replies. For the whole application a single instance of this class is enough, as it just makes requests and returns a QNetworkReply instance that can be used to monitor and obtain the reply data.

Using this class in GDAIS simplifies the implementation of the Notifier module of GDAIS-core block, as in a single line, just providing an Uniform Resource Locator (URL), a request is made and when the reply is ready a signal is emitted and reply data can be retrieved.

QTcpServer

This class provides the opposite functionality to the previous one, it implements a TCP server that can accept incoming TCP connections from remote clients. The `newConnection()` signal is emitted whenever a client connects to the server. Then the connection can be accepted and it returns a QTcpSocket instance in `QAbstractSocket::ConnectedState` that can be used for communicating with the client.

This class is also used in GDAIS-core block to implement the ControlServer module, listening for remote commands on a TCP port.

Pyramid web framework

As previously explained in subsection 3.4.2, the chosen implementation architecture for GDAIS-control block is a web application. Nowadays there are many different technologies available for developing web applications, so many options have been evaluated. The requirements for the chosen technology were:

- Compatibility with the other GDAIS blocks
- Web request managing and storage features
- Reliability, as it may be the only interface with the system
- Low overhead in order to not interfere with GDAIS-core block

Based on these requirements some option were discarded. It was decided to search for Python based solutions, assuring a better compatibility with the other blocks and simplifying the maintenance of the whole GDAIS to a single programming language.

Once this restriction was set, two technologies already used by GDAIS system were considered:

Bare Python

Python standard library, as previous said, provides nearly everything that a programmer may need. Thus, it also features some packages for web programming. In fact, a standard API was developed to facilitate this Python applications, Web Server Gateway Interface (WSGI), and it is often used when implementing web applications, e.g. via `mod_wsgi` for the Apache web server.

WSGI defines a simple and universal interface between web servers and web applications or frameworks for the Python programming language. For example, in

Listing 3.3 a simple application written in Python which just returns "Hello World" is provided.

```
1 def app(environ, start_response):
2     start_response('200 OK', [('Content-Type', 'text/plain')])
3     yield 'Hello World\n'
```

Listing 3.3: A WSGI-compatible "Hello World" application written in Python

Even though this solution is widely used, implementing a whole web interface for GDAIS-core nearly from the ground with Python modules was discarded, as better solutions are available that provide many of the required functionalities.

PyQt

As PyQt had already been chosen as a technology to implement GDAIS system and, as previously describe, it offers networking capabilities to implement client and server applications based on the QTcpServer class, it was considered to also be used in the implementation of GDAIS-core block. However, the same problem as in bare Python solution occurs, PyQt just provides a foundation of what is required; implementing a whole web application on top of it would have been a huge project by itself and better solutions were available.

Having discarded all the already available technologies in GDAIS system implementation, it was decided to incorporate a new technology to the system that provided most of the required features for implementing a web application. Such technology is usually known as a web framework.

As programming language had been previously limited to Python, solutions available in this language were searched. Fortunately, many high-quality and widely-used Python web application frameworks like CherryPy, Django, Pylons, Pyramid, TurboGears, web2py, Flask and Zope support developers in the design and maintenance of complex applications.

After analyzing each of these frameworks, some were discarded as they were larger, thus more demanding, than what GDAIS-control required (Django, TurboGears and Zope); others were discarded for the opposite reason, they did not provide enough features or relied in many external libraries to implement its functionalities and would have required a lot of programming to provide a solution (web2py and Flask).

Finally, the remaining three frameworks were more deeply analyzed and CherryPy was chosen as it was the most simple one of them. An initial version of GDAIS-control was implemented with it to check if it was the right framework to use. However, it was found to be too simple to implement some of the features expected from GDAIS-control. The main reason was that CherryPy did not provide a session mechanism, which is used to

store user information between web requests, integrated in the framework. Even though this could have been implemented on top of this framework, the decision was made to try the other two selected solutions.

After a deeper analysis of Pylons and Pyramid web frameworks, it was discovered that they were quite related. Actually, since January 2011, the Pylons web framework is in a maintenance status, not being further enhanced. This is because of the Pylons Project Organization having decided that the future of Pylon-style web application development was Pyramid. Due to this fact, Pyramid was chosen to implement GDAIS-control, and it later proved to be the perfect match for this task.

Pyramid is a general, small, fast, down-to-earth, open source Python web development framework. As a framework, its primary job is to make it easier for a developer to create an arbitrary web application. Pyramid 1.0 was released on January 30, 2011. However, this does not mean it is a new technology, as it is just the result of the Pylons framework developers having merged efforts with another exiting framework developed between June, 2008 and November of 2010, known as `repoze.bfg`, under the Pylons Project Organization. Therefore, years of experience from two important Python projects support this web application framework.

As most current web frameworks, Pyramid is platform-independent and is usually included in the MVC web frameworks family. This design pattern isolates the application logic fo the user from the UI (input and presentation), as it can be observed in Figure 3.16.

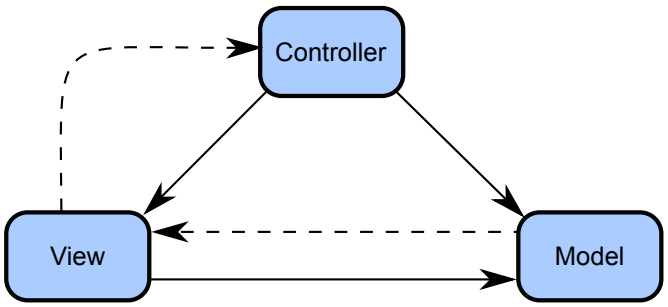


Figure 3.16: Model-view-controller design pattern representation

For data persistence it can integrate with different databases, in this project an Structured Query Language (SQL) database is used via SQLAlchemy. Pyramid also helps in defining routes to map requests to objects in the system, as most other frameworks. Finally, to generate the views output multiple template engines are available; for GDAIS-control block Mako template engine was selected.

According to Pylons project website[8], Pyramid is developed using the following tenets:

- Simplicity
- Minimalism
- Documentation
- Speed
- Reliability
- Openness

All these principles were coherent with the requirements of the project, therefore Pyramid was finally chosen as the development base for the web application of GDAIS system, GDAIS-control block.

3.6 Design

Once the architecture and the most appropriate technologies to implement each block of the system have been described, in this section the design of the final GDAIS system implementation is explained.

Following the initial workflow previously explained in section 3.4, the design of each block and its modules is detailed. Firstly the GUI blocks Instrument editor and Equipment editor are described. Secondly GDAIS-core design is analyzed. Finally, GDAIS-control block and its connection with the main block are explained.

Before entering to the design section details, an overview the Unified Modeling Language (UML) is provided. This graphical language is used to represent the object-oriented design of each module.

3.6.1 UML

UML is a standard modeling language in the field of object-oriented software engineering [13]. This language includes a set of graphic notation techniques to create visual models of object-oriented software systems.

From all the possible models that this language can represent, only the data modeling is used in this section. For this model type, UML defines a set of rules to graphically represent in a diagram the entities and the relationships between them, which is called a class diagram.

[8] http://pylonsproject.org/projects/pyramid/about

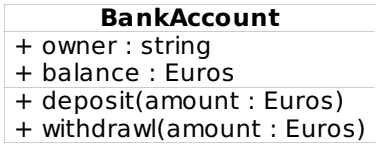


Figure 3.17: UML representation of a class

The main building block for class diagrams is the object-oriented concept of *class*. As it can be observed in Figure 3.17, this is represented with a rectangle divided in three sections: the one at the top contains the class name, the middle one contains the class attributes, and the one at the bottom contains the class methods.

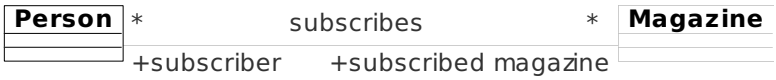


Figure 3.18: UML representation of an association

Once the classes are represented, relations between them can be added. This relations can be between instances of classes, which is known as an *association*, see Figure 3.18. An special type of association is an *aggregation*, which is an association with a "has a" relationship, see Figure 3.19. As it can be observed in both diagrams, at each end of the relationship a number or * is included, to represent the *multiplicity*, which is the amount of instances of the class it connects that may be linked at the same time. The * means that any number of instances is possible.

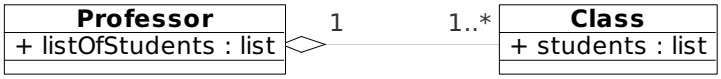


Figure 3.19: UML representation of an aggregation

Another type of relation can be defined at class level. This is known as the *generalization* concept of OOP, which defines a class as a subclass of another, defining an "is a" relationship. Figure 3.20 contains an example of this relation type.

Finally, a more general relationship can be defined between two classes, the *dependency*, to represent that one class depends on another because it uses it at some point of time. An example of this relationship type is displayed in Figure 3.21.

3.6.2 Instrument editor

Instrument editor block is implemented as an independent application using the Qt class `QApplication`. Its GUI has been created using Qt Designer tool, as it can be observed in Figure 3.14, and the in-memory representation of the instrument is implemented in

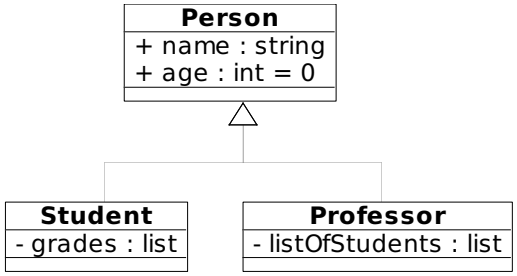


Figure 3.20: UML representation of a generalization

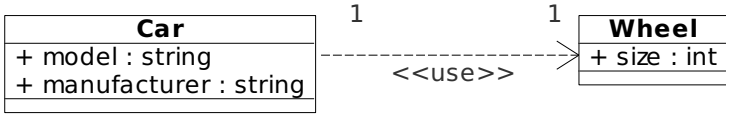


Figure 3.21: UML representation of a dependency

a separate class shared with GDAIS-core main application. The following two sections describe this instrument representation class and the GUI created for this module.

Instrument description

This module is implemented as a set of Python classes that store a representation of an instrument and are able to read and write instrument description files in JSON to/from disk. Figure 3.22 shows a diagram of all the classes defined in this module.

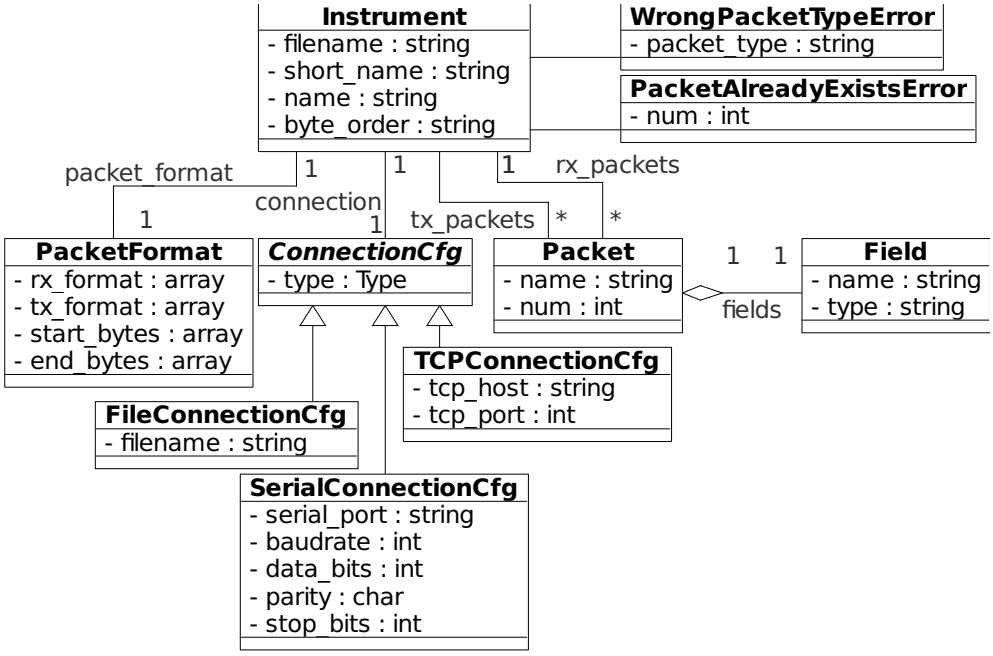


Figure 3.22: Instrument description class model

As it can be observed in the class diagram, an instrument contains a set of properties, the

connection settings and packet format. Also it includes the description of all the packets the instrument can receive (`tx_packets`) and send (`rx_packets`) from/to the system with the fields that each packet contains, if any.

Regarding the `ConnectionCfg` class, it can be observed that 3 types of connection are included using the inheritance mechanism. This is useful for extending the system with new connection types, as it just requires creating a new class that also inherits from `ConnectionCfg` and adding the specific attributes for the new connection.

In order to load an instrument description from a file and write it back, each of these classes includes two methods that help in the process. The first one is used to set the attributes of the instance from the representation extracted from the JSON file; and the other one implements the reverse process, it creates a representation of the instance that can be converted to JSON and written to the file.

Using these two methods of each class and the Python `json` module, `Instrument` class is able to load a description of an instrument from a given filename and also create or update a file.

Instrument editor GUI

In order to manage all the options that can be included into an instrument description file, a GUI has been designed and implemented.

In Figure 3.23 the view of the interface that is initially shown to the user is provided. This figure shows the main settings for the instrument (name, short name and byte order) and also the connection settings tab. Depending on which connection type is selected, the related settings are enabled and the other ones disabled.

Figure 3.24 shows the packet format tab that provides the interface to define the structure of the packets GDAIS will interchange with the instrument in each direction. As it can be observed, both for received and transmitted packets, its structure can be defined using 4 different blocks: start mark bytes, packet number, packet fields, and end mark bytes. There is no need to include all these blocks, just the ones that conform with the message structure defined by the instrument. If in packet format settings start or end marks are defined, the lists at the bottom provide an interface to insert the values that define these marks.

Finally in the last two tabs, in Figure 3.25, the structure of the packets that will be received from and transmitted to the instrument is defined.

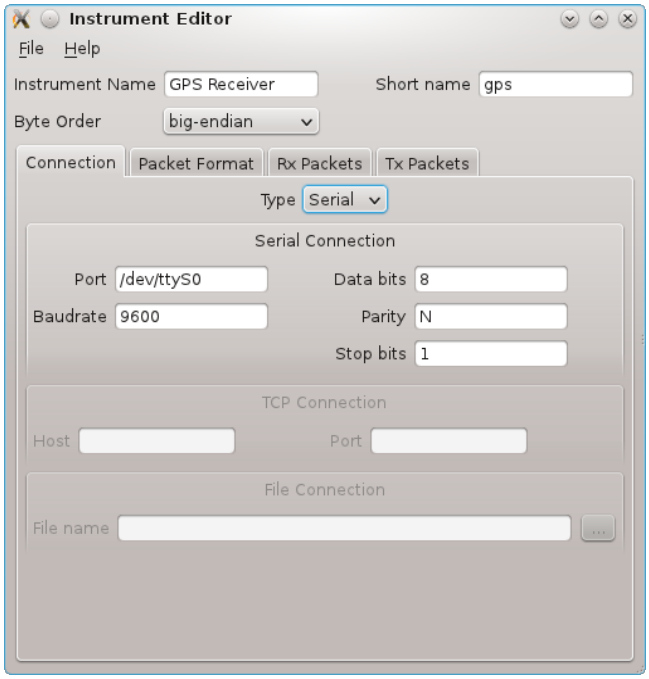


Figure 3.23: Snapshot of instrument editor initial view and connection tab

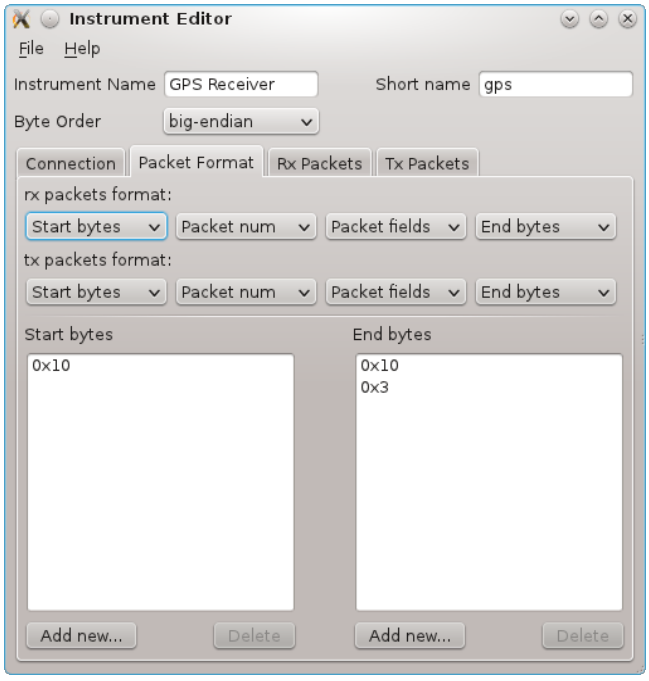
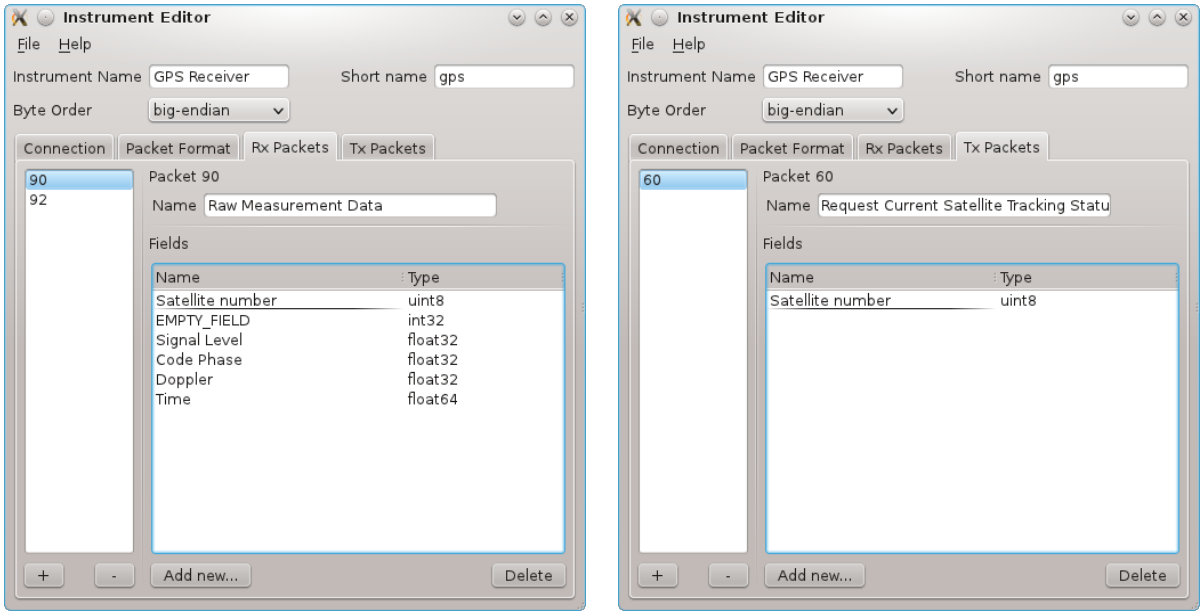


Figure 3.24: snapshot of instrument editor packet format tab

3.6.3 Equipment editor

Like Instrument editor, Equipment editor block is implemented as an independent application using Qt `QApplication` class. Figure 3.26 shows its development using Qt Designer tool. Next, the classes implementing the representation of an equipment and the GUI to



(a) Tab to configure packets received from the instrument structure

(b) Tab to configure packets transmitted to the instrument structure

Figure 3.25: Instrument editor showing received and transmitted packets structure tabs

manage its description files are explained.

Equipment description

As the Instrument description module, this module is implemented as a set of Python classes that store a representation of an equipment and are able to read and write equipment description files in JSON to/from disk. Figure 3.27 shows a diagram of all the classes defined in this module.

In the Equipment description class diagram it can be observed that an `Equipment` has three attributes: `name` and `short_name`, that are used to represent the equipment in the system and to identify it in the UI, and the `filename`, which is used when an equipment is loaded from a description file to save it back after making some modifications.

An `Equipment` may have associated one or more `InstrumentConfig` instances, which represent an `Instrument` and its settings in the defined equipment configuration, the initial and operation sequences of commands. The `Instrument` class that appears in this diagram is, in fact, the same that in the `Instrument` description module, but its relations have not been included as they can already be observed in the previous diagram, see Figure 3.22.

An `InstrumentConfig` instance may have some `InitCommands`, which form a sequence of

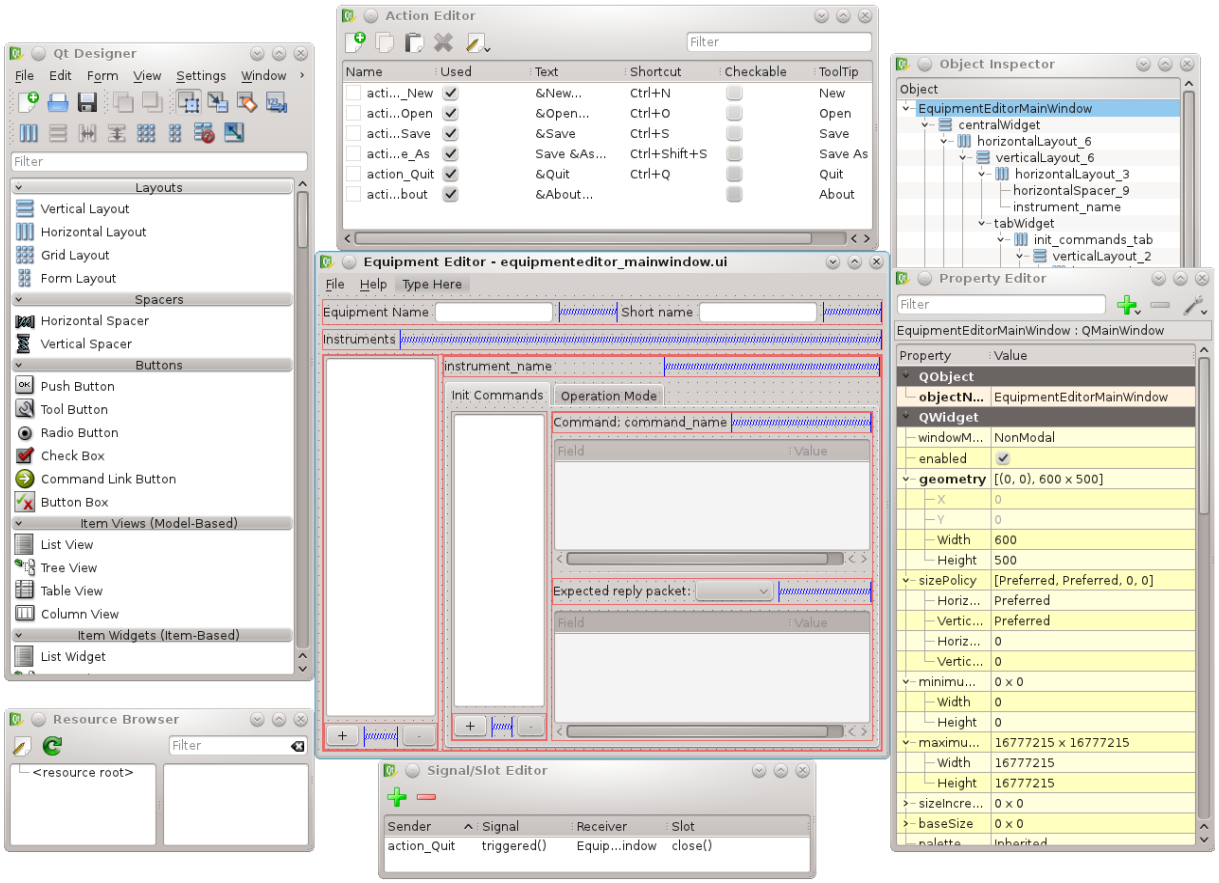


Figure 3.26: Qt Designer showing Equipment editor block

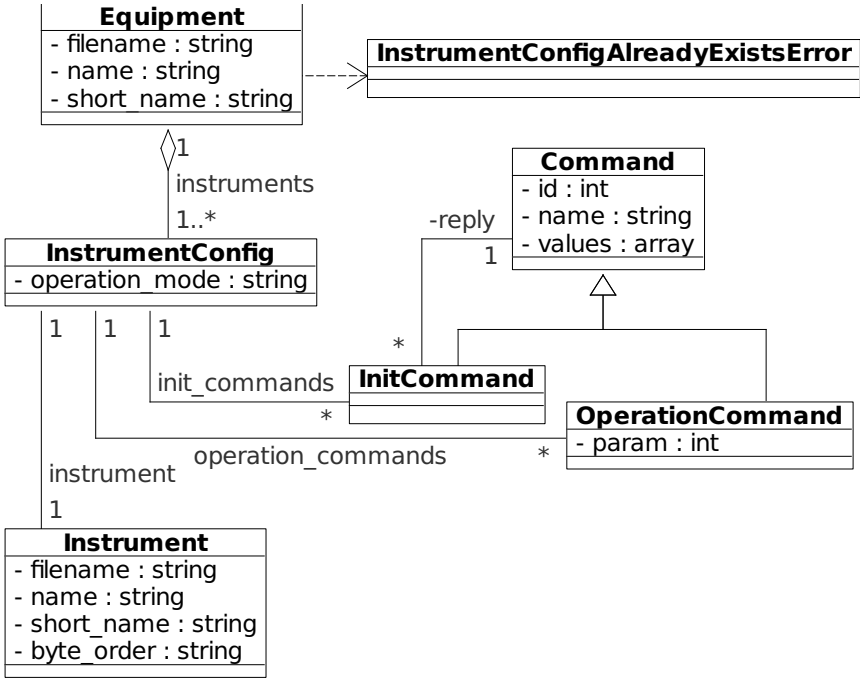


Figure 3.27: Equipment description class model

commands that will be sent as soon as the connection with the instrument is available, before starting the acquisition in the system. For each of these commands an expected reply is associated, which will be used to check that the values returned by the instrument are the expected ones, if any.

Finally, depending on the operation mode of the instrument defined in the equipment configuration, it may also have some `OperationCommands` added. The `param` attribute of this class has a different meaning in each operation mode. There are 3 possible operation modes defined:

Periodic Commands

One or more commands can be defined in this mode and a period of time associated with each of them. Then, each command is executed periodically at the defined period. This mode is useful when some measurement is desired repeatedly every some periodic time. Furthermore, if no command is defined, the system will simply listen for any measurement it receives from the instrument.

Time Sequences

A sequence of commands can be defined and, for each command, a fixed time to wait before sending the next command. When the sequence is completed it is restarted from the beginning, continuing until the acquisition is finished. This mode is useful when some measurements are desired periodically but they need to be requested in an specific order.

Blocking Sequences

This mode allows the user to define a sequence of commands that will be sent sequentially to the instrument when the reply from the previous command is received. If some measurement is desired multiple consecutive times, a parameter is provided to define this value.

The mechanism for loading and writing back description files in JSON format is the same that in Instrument description. Each class has two methods: one to initialize itself from the data in the description file, and the other one to produce a description that can be converted to JSON format. With the help of Python `json` module and these methods, `Equipment` class reads and writes equipment description files.

Equipment editor GUI

This section describes GUI for the Equipment editor application. It has been designed to be consistent with the Instrument editor, and it provides some aids for the user to match

correctly the information of the instruments created previously in the other editor, with the settings defined in the equipment with this editor.

Figure 3.28 shows the initial view of the Equipment Editor with two instrument loaded, which are the ones that were explained and used for testing in prototype iteration, see subsection 3.3.1. In the snapshot, the selected tab shows the Init Commands configuration for the first instrument, with a configured command which takes no parameters, and the expected reply packet with the value for each field.

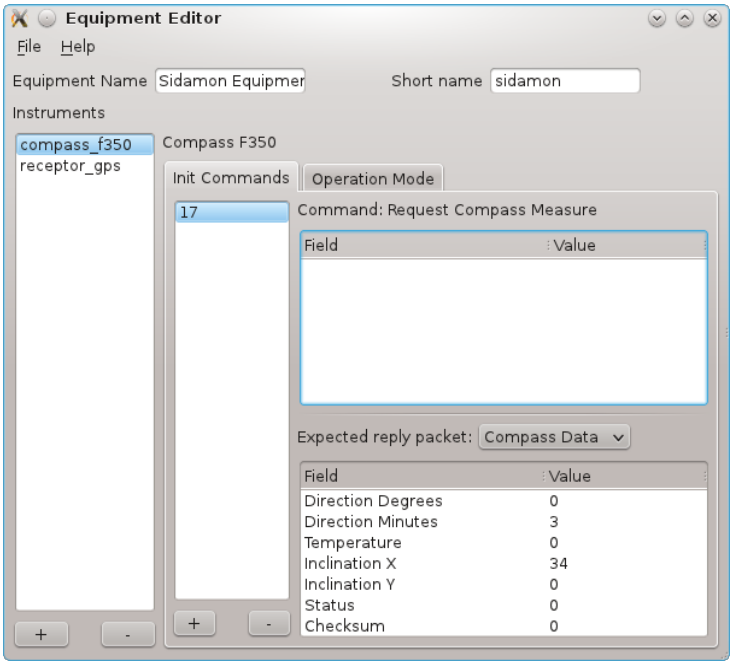


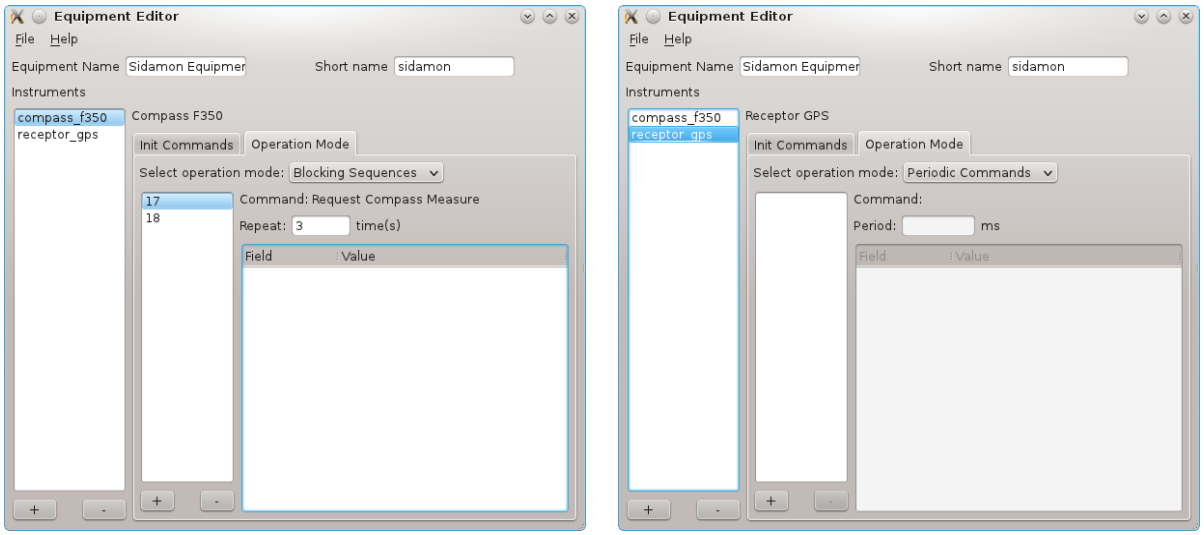
Figure 3.28: Snapshot of equipment editor initial view with the init commands tab

In this same figure, it can be observed that instruments list has two buttons at the bottom to add (+) and remove (-) instruments. If the user chooses to add a new instrument, a new dialog is shown for the user to provide an instrument description file. Therefore, the user does not have to introduce any information of the instrument, as it is loaded directly from the selected file.

Moreover, in the column on the right of the instruments list, the commands list, which also has add and remove buttons, a similar aid is provided. When the user clicks the add button, a dialog is shown with a list of the packets defined in the instrument description, and the user just has to select one of them. Once selected, the packet is added to the list and, if it has any fields, they are shown on the top right table, the one which is empty in the figure, for the user to provide the values for each field.

The Operation Mode tab is displayed in Figure 3.29 for each of the instruments. As it can be observed, the F350 compass instrument, Figure 3.29a, has the Blocking Sequences

operation mode configured with to commands to execute. The selected command, which was previously defined in the instrument description is *Request Compass Measure* and it is set to be repeated 3 times before sending the next command. The other instrument, Figure 3.29b, has the Periodic Commands mode set but no command has been added. Therefore, the equipment will just listen for measurements from the instrument without sending any request.



(a) Tab to configure packets received from the instrument structure

(b) Tab to configure packets transmitted to the instrument structure

Figure 3.29: Instrument editor showing received and transmitted packets structure tabs

When changing operation mode, as in each mode the settings have different meaning, they have to be reset to the default, an empty list of commands. To prevent the user from accidentally changing the mode and thus loosing all the settings that may have been configured, a confirmation dialog has been added when change operation mode that alerts the user about this risk. Figure 3.30 shows this pop-up window.

3.6.4 GDAIS-core

GDAIS-core is the main application of the whole GDAIS system. It is based on the Qt `QCoreApplication` class, which provides the main environment for the non-GUI application on top of which all the modules of the system will run. Figure 3.31 shows a diagram of the most relevant classes used from the main class `GDAIS`. Some of the related modules are further detailed in the next subsections.

`GDAIS` class controls the startup and shutdown of the whole application. It receives an equipment file through the command line and creates all the structures for each connected instrument as defined in the equipment description to prepare for the acquisition start. It

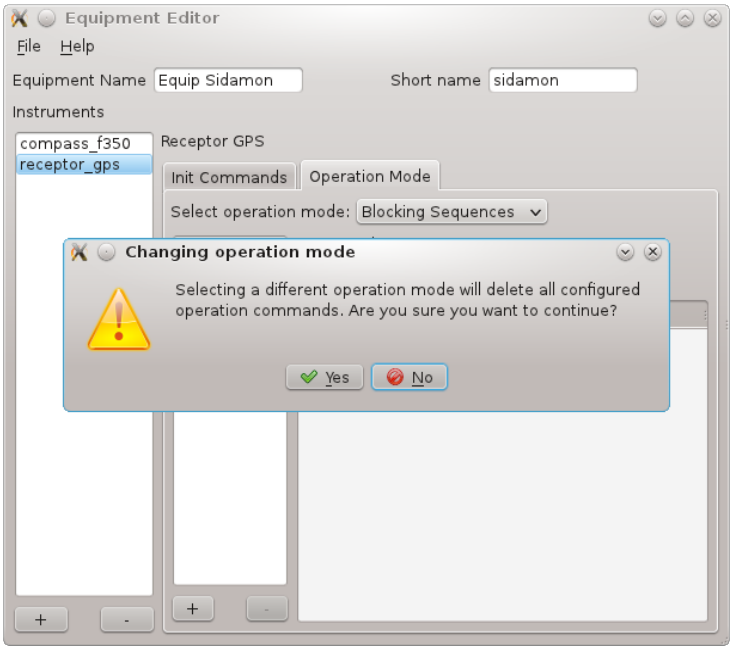


Figure 3.30: Snapshot of equipment editor alerting the user that changing operation mode will reset all configured options

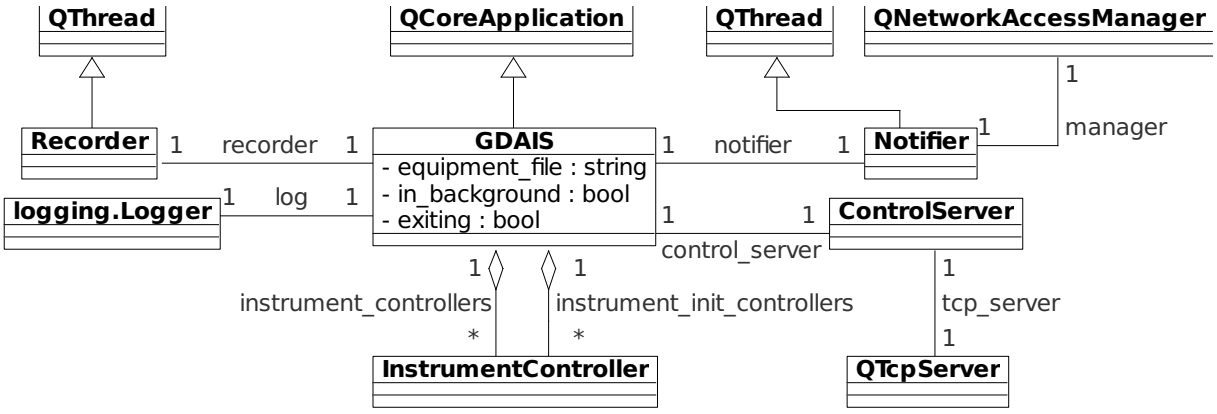


Figure 3.31: GDAIS-core main class model

also creates the communication interfaces to send notifications of the status of the system and to listen for control commands. When the command to stop the acquisition and quit the application is received by the `ControlServer`, `GDAIS` class stops all running threads finishes the execution of the application.

The following subsections describe in detail the design of each of the modules which conform GDAIS-core block, as it is represented in Figure 3.31.

Instrument controller

For each instrument in the equipment which GDAIS is controlling, an instance of this class is created. It is responsible for implementing the control and the acquisition from the instrument. To provide a flexible and extensible implementation of the controlling interface, being able to adapt to different operation modes, the mechanism of inheritance is extensively used in the design of this module, as it can be observed in Figure 3.32.

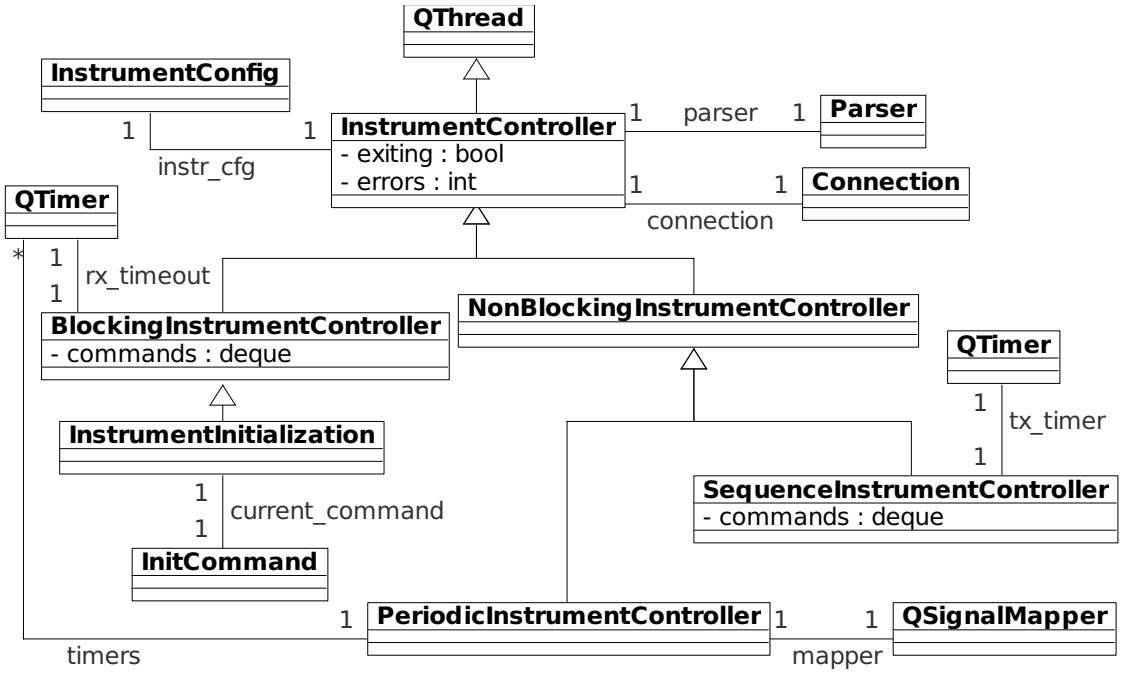


Figure 3.32: InstrumentController class hierarchy model

The most generic `InstrumentController` class implements the functionalities that are required to control any instrument, whichever operation mode it uses. These include managing the `Connection` with the instrument and the `Parser` for the binary data sent and received.

To implement these generic features, the Qt *signals and slots* mechanism is used. Whenever a new measurement is recognized by the `Parser`, it *emits a signal* with the received

packet information. This signal is connected to the `on_new_packet_parsed` *slot* of the `InstrumentController` class. This *slot emits a new signal* that is connected to the main `GDAIS` instance which redirects it to the `Recorder` instance that logs the acquired measurements from all the instruments in the equipment.

The procedure to send a new command to the instrument is also similar. Whenever a command is to be sent, the `new_command` *signal is emitted* with the command information in an instance of `Command` class, previously described in Equipment editor design (section 3.6.3). This *signal is connected to an slot* of the `Parser` instance that is responsible for sending the command to the instrument.

The first inheritance level from `InstrumentController` divides the operation modes that, for each new command, have to wait for the previous measurement reply to be received, `BlockingInstrumentController`, and the ones that do not have this requirement, `NonBlockingInstrumentController`.

The `BlockingInstrumentController` class is used to implement the *Blocking Sequences* operation mode, previously defined in Equipment Editor design (section 3.6.3). It uses a circular queue to control what command has to be sent next and a timer, implemented using Qt `QTimer` class, to define a timeout wait time to prevent a system lock if the instrument does not reply a measurement or the reply is lost.

If the instrument has any initialization commands defined in the equipment description, the `InstrumentInitialization` class is used to setup the instrument before the acquisition starts. This class adds to the `BlockingInstrumentController` class the functions to check that the reply to each initialization command is the expected one, and inform the main `GDAIS` class if any error occurs during the initialization. As if there was an error, the system should not start and the user should be informed.

Regarding the other operation mode, the `NonBlockingInstrumentController` class is not designed to be used directly, instead it is used to join the classes that do not require each command to wait for the previous measurement to be received. This includes two of the previously described operation modes (section 3.6.3): *Periodic Commands* and *Time Sequences*.

For the *Periodic Commands* operation mode, the `PeriodicInstrumentController` class provides a set of timers, one for each command to be sent periodically. As the timers are implemented using `QTimer` class, they use the *signals and slots* Qt feature. In order to ease the management of these multiple *signals* from a single *slot*, the `QSignalMapper` class is used. This class allows multiple *signals* of the same type to be connected to a single *slot*, but with the feature of being able to identify from which object came each

signal. Thanks to this, the *slot* knows which command has to be sent when it receives a *signal*, as it knows which command timer it came from.

Finally,`SequenceInstrumentController` class implements the last operation mode, *Time Sequences*. As the `NonBlockingInstrument` class, this class stores a circular queue of commands and the duration associated with each command. To control when a new command has to be sent, it uses a `QTimer` instance that is set to shot only once after the duration defined on the command passes. For each command sent this procedure is repeated.

Connection

Analogous to the implementation of the connection settings `ConnectionCfg` class in the Instrument editor, the connection with the instrument is also implemented using inheritance, as displayed in Figure 3.33.

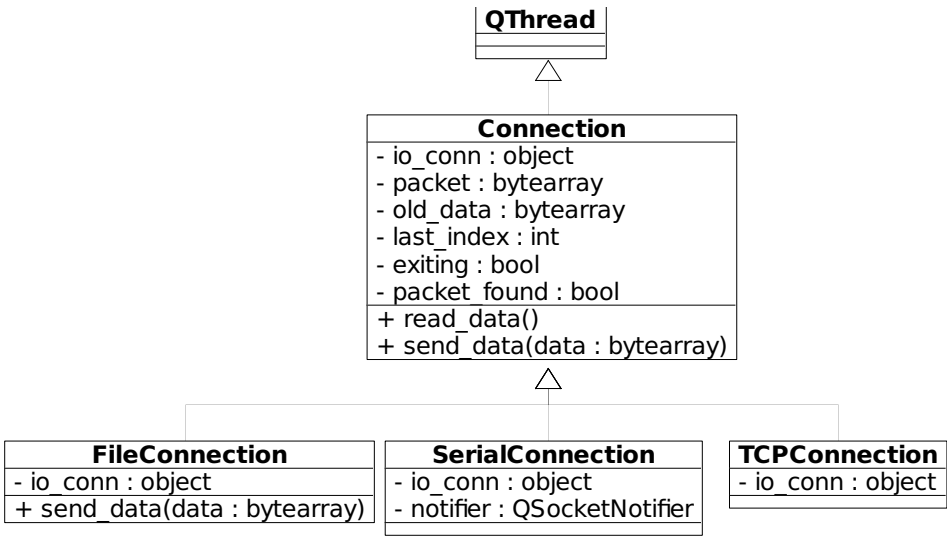


Figure 3.33: Connection class hierarchy model

In this case, the template design pattern is used to provide, from the generic abstract class `Connection.read_data` method, the analysis of any sequence of bytes to identify measurement packets that can be parsed by the `Parser` instance. Therefore, the specific classes that implement a connection type, just need to provide the binary information to this method from the generic class, reducing the code duplication.

The mechanism to send data to an instrument is implemented similarly. The `Connection.send_data` method creates the binary sequence to be sent to the instrument, and each specific connection is responsible for sending this binary data to the instrument through the connection type used. A special case is the `FileConnection` class, which reads data from a file, but is not able to send commands, as this is

not a real connection with an instrument. Therefore, this class reimplements the `send_data()` to discard any command that is received and inform that an error has occurred.

In the diagram it can also be observed that `SerialConnection` class has a `notifier` attribute. This attribute contains an instance of `QSocketNotifier` that is used to monitor the serial port for new data ready to be read. This instance *emits a signal* when there is new data on the port, and this signal is connected to the `read_data` *slot* which processes the binary data, as previously explained.

Parser

The implementation of the `Parser` class is quite simple, it can be observed in Figure 3.34. It uses the Python `struct` package to convert the binary stream received from the connection to a set of values, following the settings in the `PacketFormat` from the `Instrument` description class.

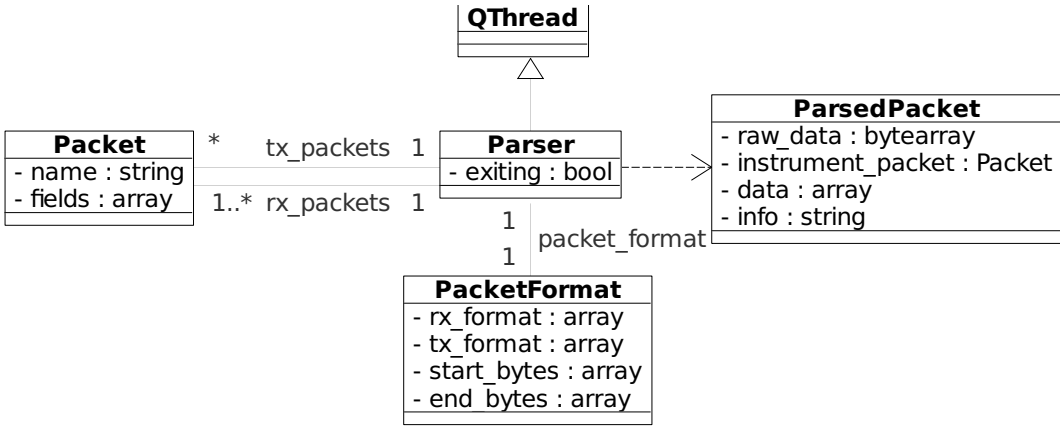


Figure 3.34: Parser class model

When a new `Packet` is received from the `Connection` instance and parsed, `Parser` instance *emits a signal* which is received by the `InstrumentController`. The information of the parsed packet is attached to the *signal* in an instance of `ParsedPacket` class.

Data recorder

Figure 3.35 shows the representation of `Recorder` class and its relations with the other classes. As it can be observed, PyTables Python package, `tables`, and its classes are used to implement the HDF5 file management.

Before starting a `Recorder` thread, a new `tables.File` instance is created with a new filename that contains the `Equipment` short name and the date and time when the ac-

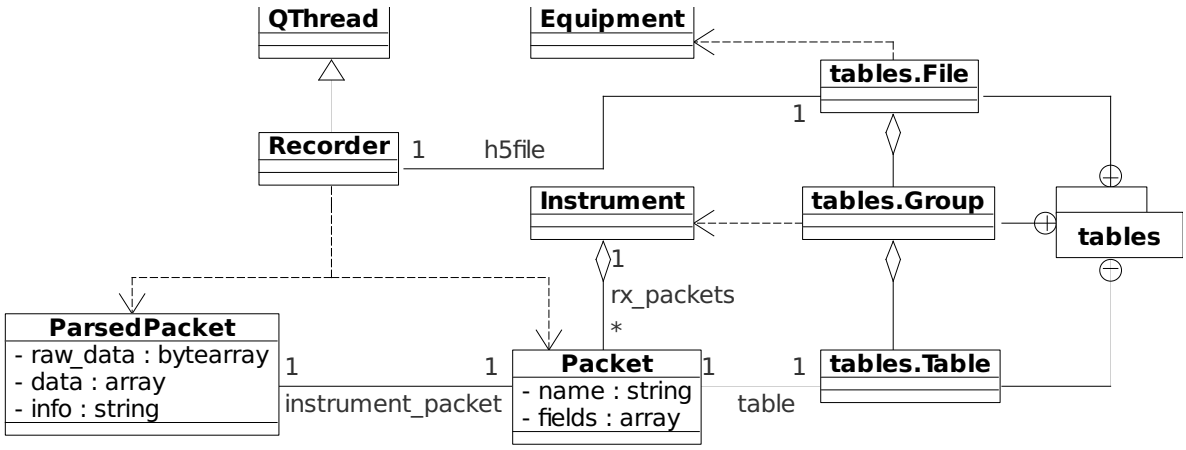


Figure 3.35: Data Recorder class model

quisition has started, in order to identify it afterwards. For each `Instrument` in the `Equipment` description, a new `tables.Group` is created and added to the previously created `tables.File`. Each `tables.Group` will contain a `tables.Table` for each packet type that can be received from the `Instrument` (`Instrument.rx_packets`). Finally, thanks to Python flexibility, each `table.Table` instance is added to the `Packet` instance it references, in a new `table` attribute. Using this mechanism, no new structure is created to store the `tables.Table` instances for each `Instrument`, instead, they are directly available from each `Packet` instance.

When the acquisition starts, for each new packet parsed, the `Parser` *emits a signal* that is *connected to* the `on_new_packet` *slot* of the `Recorder` instance. If the packet was correctly parsed — it contains some values in the `data` attribute — the new data is appended to the table of the packet, adding a time stamp to it. Here, the previous association of the `tables.Table` with each `Packet` instance is used to reduce the time it would require to find every time the `tables.Table` instance associated to the `Packet`.

In the design of this class, inheritance has not been considered, as a single binary data storage format was planned. Moreover, the implementation of this class is very dependent on the package used to implement the binary file management, thus most of the code would have to be redesigned to adapt to other file formats. However, the interface of the class could be reused, as it just requires an slot which can receive `ParsedPacket` instances associated with the *signals* received from each `Parser`.

Communication interface

Control server

Control server class design is displayed in Figure 3.36. It uses Qt `QTcpServer` class

to create a TCP server instance that listens for connections on port 12345.

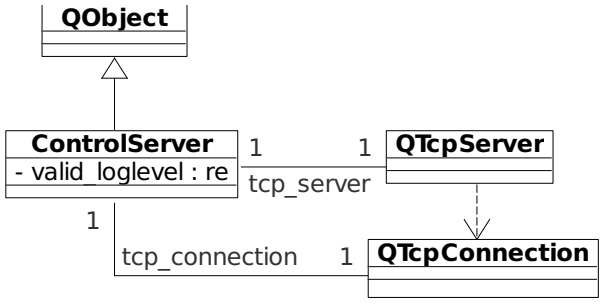


Figure 3.36: Control server class model

Currently two commands are implemented in the control server:

quit

This command informs the system that it should finish the acquisition and exit, shutting down all the connections to the instruments.

set_log_level

This command can be used to change the verbosity level of the logs that are notified to the GDAIS-control block. This can be used to see debugging information from the web interface while testing the system and then being able to disable all information messages when the system is used in an experiment, reducing the performance drop that notifying every event supposes.

To select a log level, a numeric parameter has to be provided. It is defined following the classification used by `logging` Python package. The accepted values are displayed in Table 3.3.

Table 3.3: Log levels

Level	Numeric value
Critical	50
Error	40
Warning	30
Info	20
Debug	10

Notifier

Figure 3.37 shows `Notifier` class design and its relation with the Qt classes used to implement the functionality it provides.

When the `Notifier` instance is created, a `QNetworkAccessManager` is instantiated, which will be used to manage all the requests sent by the `Notifier`. To send any

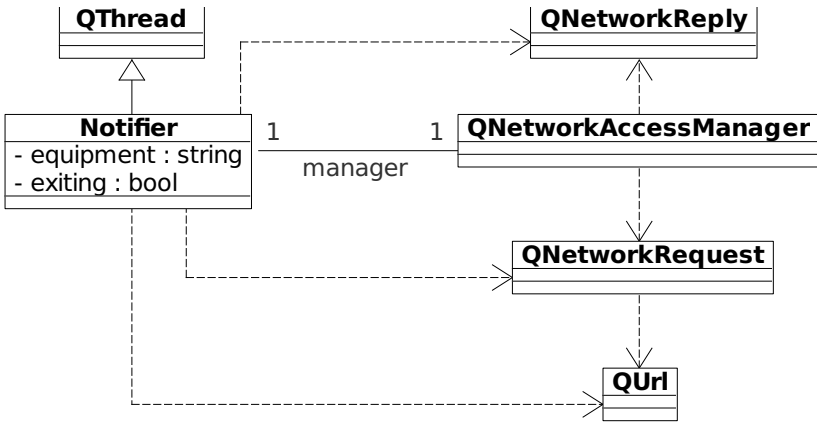


Figure 3.37: Notifier class model

notification to the GDAIS-control block, the `notify` *slot* is provided, which receives an string as a parameter describing the event that has occurred. Using this event name, an URL is generated in a `QUrl` instance and the request is sent using a `QNetworkRequest`. The reply is received as an instance of `QNetworkReply`, which is just used to log if there has been an error in the request.

3.6.5 GDAIS-control

As explained in section 3.5, the implementation of this block is based on Pyramid web application framework. This framework sports a modified version of the MVC pattern which is said to better fit the web. In a Pyramid application, there is a *resource tree*, which represents the site structure, and *views*, which are responsible for presenting the data stored in the resource tree and a user-defined "domain model". However, there is no facility provided in Pyramid which maps to the usual concept of a "controller" or "model".

The description of this block will follow this structure. First the *resources* — *routes* and *models* — available in the implemented web application are explained and, afterwards, the design of the implementation of the *view* associated with each *route* is provided.

Resources

Routes

Table 3.4 shows all the available routes that can be requested, the view associated with each route and a short description of what it does.

Models

To represent equipments and log messages two models have been implemented in

Table 3.4: GDAIS-control web routes

Route	View	Description
/	list	shows a list of the available equipments and its status
/start/{equip}	start	starts GDAIS-core for the given equipment
/notify_start/{equip}	notify_start	registers the acquisition start of the given equipment
/stop/{equip}	stop	sends the command to stop the given equipment acquisition
/notify_quit/{equip}	notify_quit	registers the acquisition end of the given equipment
/view/{equip}	view	shows the information of the give equipment
/view_log/{equip}	view_log	shows a list of the equipments available and its status
/set_log_level/{equip}/{level}	set_log_level	sends the command to set the log level of the given equipment
/log/{equip}	log	saves a log message from GDAIS-core for the given equipment
/download/{equip}/{date}/{time}	download	returns the acquired data for the given equipment on the specified date and time
/delete/{equip}/{date}/{time}	delete	deletes the acquired data of the given equipment on the specified date and time

GDAIS-control. For each of them, a brief description of the attributes is provided.

- `EquipmentModel`: Represents an equipment that can be controlled by GDAIS-control sending control commands to GDAIS-core. It has the following attributes:
  - `name`: name of the equipment
  - `desc`: a longer description of the equipment
  - `running`: whether the equipment is currently acquiring
  - `log_lvl`: log level set for the equipment log messages
- `LogModel`: Represents a log message received from GDAIS-core. It has the following attributes:
  - `name`: module from which the log message comes
  - `levelno`: log level number
  - `levelname`: log level name
  - `msg`: actual message contents
  - `date`: date and time when the message was logged
  - `exc_text`: name of the exception if it has occurred
  - `exc_info`: information of the exception
  - `equip_id`: relation with `EquipmentModel`, stores which equipment the log message came from

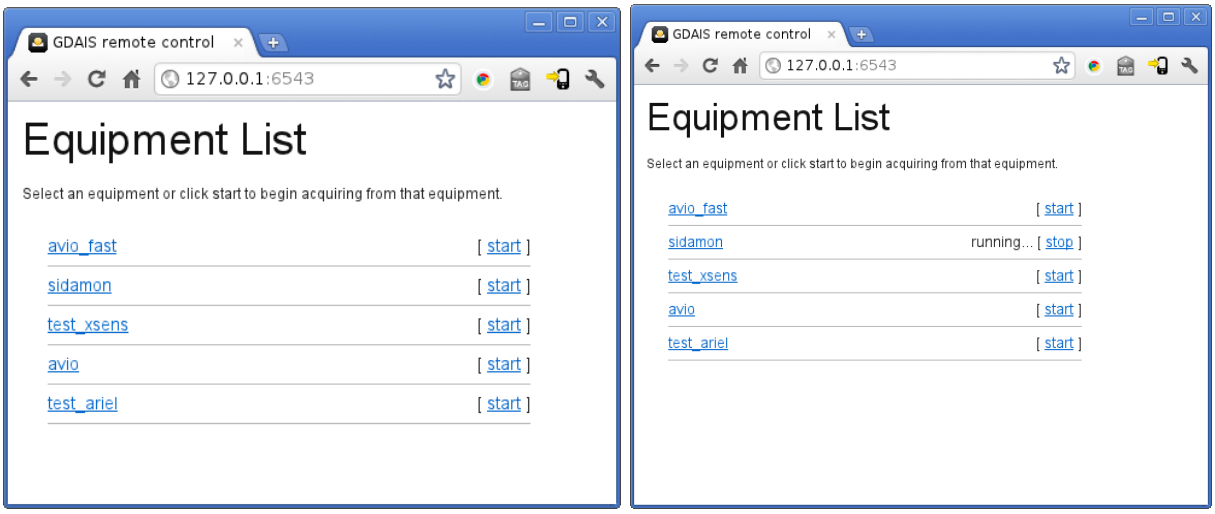
Models persistence is implemented using the `SQLAlchemy` Python package, which is a relational database toolkit. The specific database to store the information is SQLite, which has the advantage of not having to install a database server to the system which is running GDAIS-core. Instead, the whole database is stored in a single file.

Views

This section describes each of the views that can be accessed through the previously detailed routes.

list

This view shows a list of the equipments that GDAIS-core can control, the ones that have a description file created. Each equipment name has a link to the equipment detailed view and, on the right, each equipment can be started, or stopped if it is running. Figure 3.38a shows the list when no equipment is running, and Figure 3.38b shows the list when an equipment is acquiring data.



(a) All equipments stopped (b) *sidamon* equipment running

Figure 3.38: GDAIS-core equipment list

start

When the *start* link is clicked in the equipment list or in the equipment details page, GDAIS-core process is started with the selected equipment description, passed as an argument. As Figure 3.39 shows, when the equipment is started the equipment list page is shown to the user with a message indicating that the equipment was correctly started. If any error occurs, it is shown instead of this message.

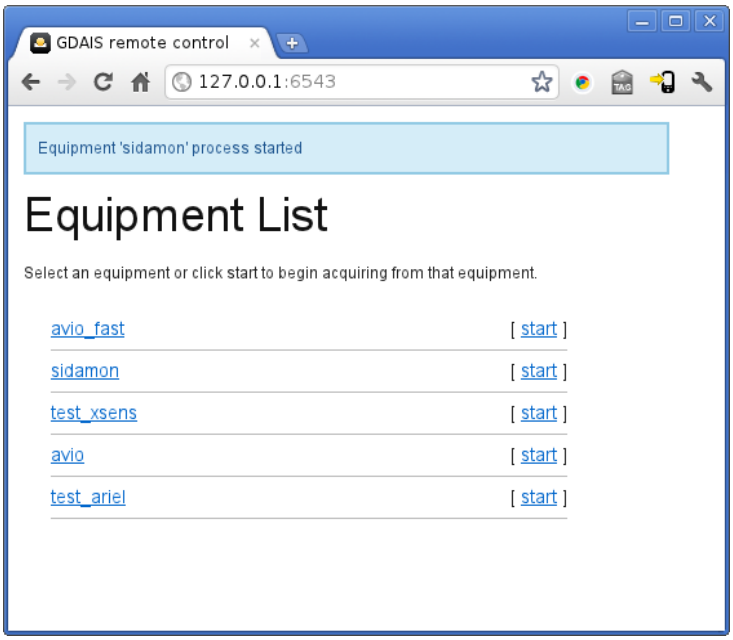


Figure 3.39: Equipment list displaying the message that an equipment has been started

notify_start

This view is not intended to be accessed from a web browser, instead it provides an

interface for GDAIS-core `Notifier` class to inform that an equipment has started. When accessed it returns a reply with information on the management of the request, in JSON format, instead of the usual HTML. For example, the returned data if GDAIS-core notified that *sidamon* equipment had been started would be:

```
{"status": "OK", "info": "sidamon start notified"}
```

view

When an equipment link in the initial list is clicked a new page with the details of the equipment is shown. The displayed information depends on the status of the equipment. On one hand, if it is stopped, the page shows the information like in Figure 3.40.

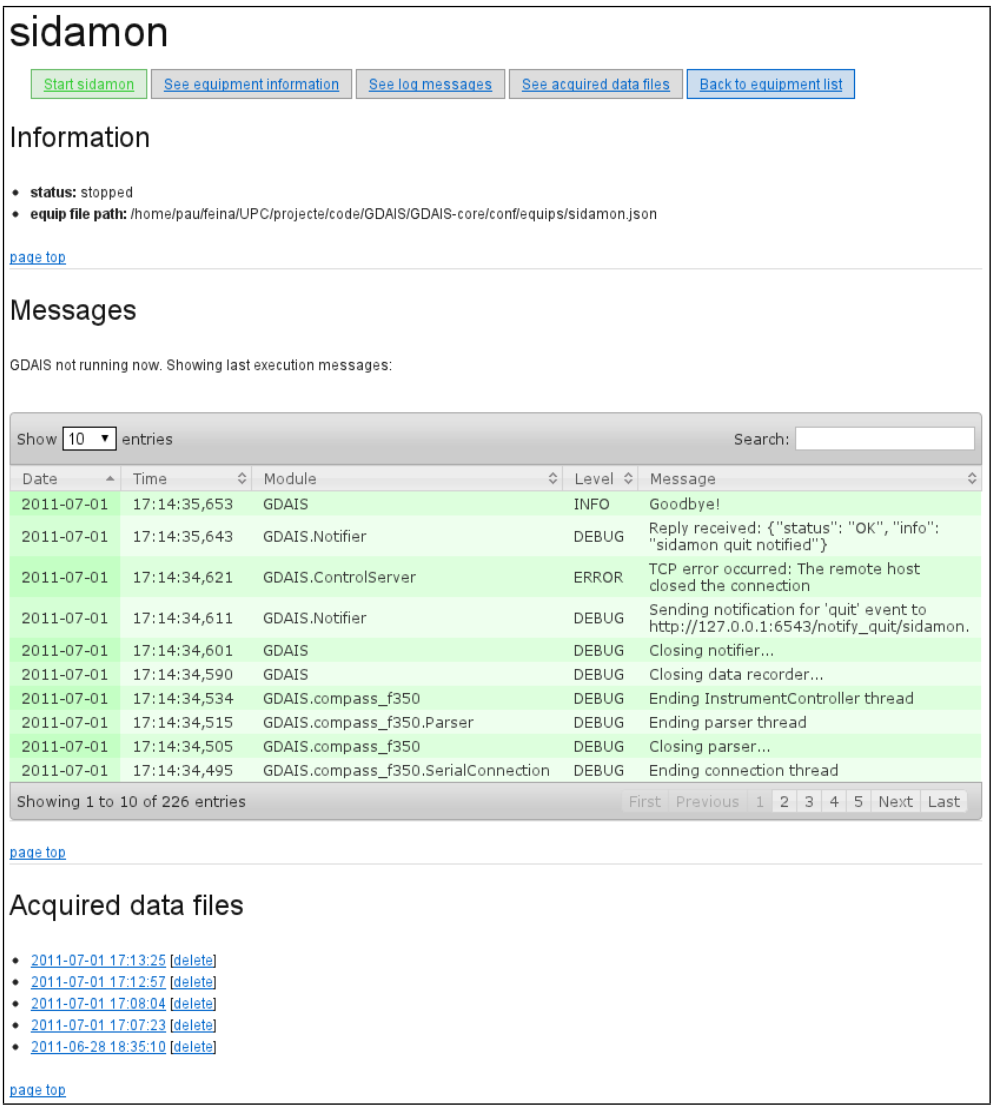


Figure 3.40: Equipment details page when the equipment acquisition is not running

On the other hand, if it is running, the page shown is like in Figure 3.41.

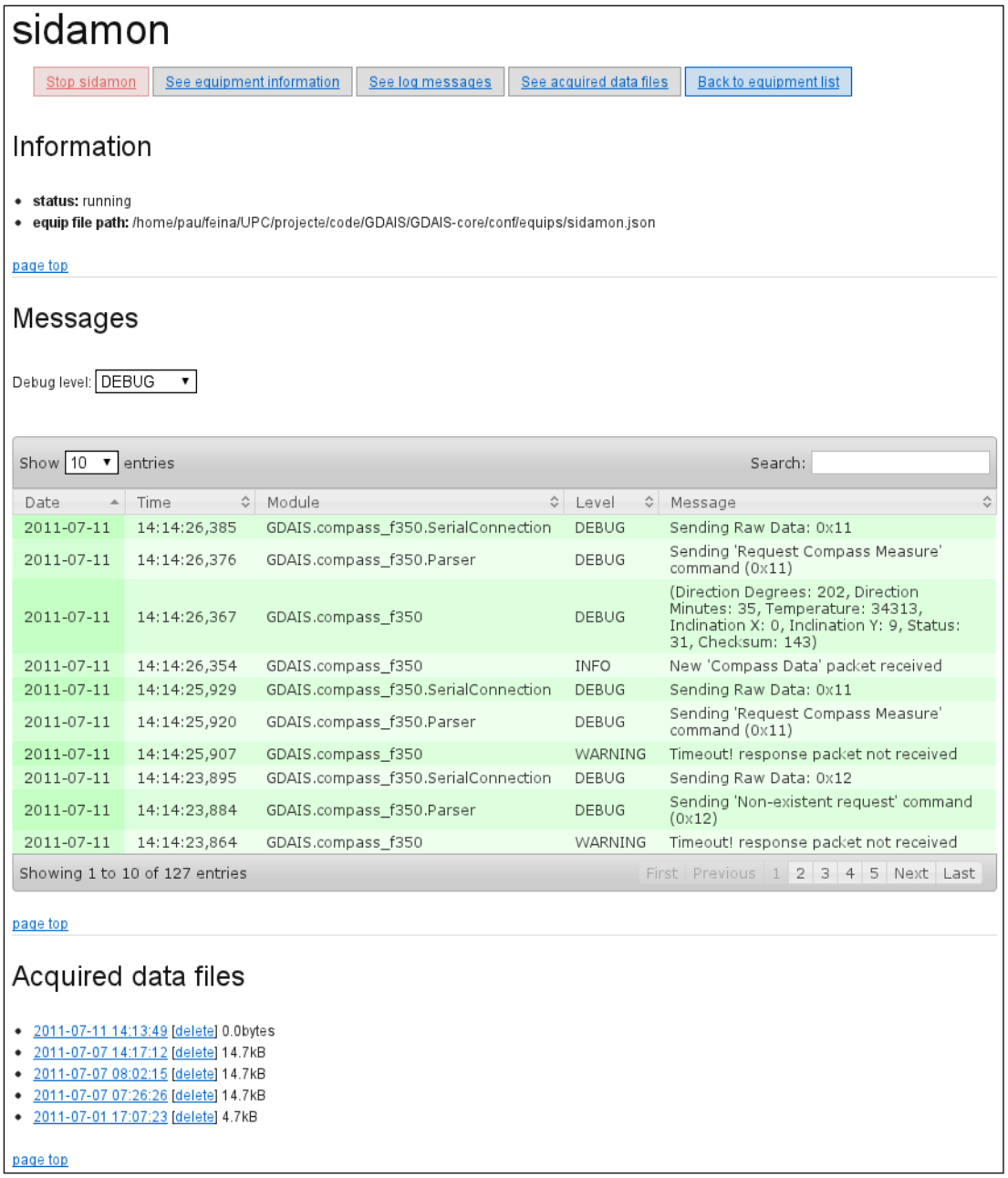


Figure 3.41: Equipment details page when the equipment acquisition is running

view_log

This view is used by the previous one to display the log messages that have been received for the equipment and periodically update them without having to refresh the whole page. This is implemented using a JavaScript library called `JQuery`, which is able to request information to the server and incorporate it to the page, a technique usually known as AJAX. As no direct access is required for this view, just from JavaScript queries, the usual access from a web browser by URL has been disabled.

log

When a new log message is generated in GDAIS-core it makes a request to this view with the information of the log message and GDAIS-control stores it to the database. This view just returns `OK` when completed.

set_log_level

In Figure 3.41, it can be observed that when the acquisition is running a new option appears in the *Messages* section of the page. This selection box allows the user to modify the verbosity level of the log messages notified from GDAIS-core to GDAIS-control. When the value of this selection box changes, a JavaScript call requests the `set_log_level` view with the log level number associated to the selected log level, as in Table 3.3. This view notifies the GDAIS-core block that the log level should be changed sending the command described in section 3.6.4.

stop

When an equipment is running a red link appears on the top of the details page to finish the acquisition. The link is also displayed in the equipments list. Clicking this link calls the stop view, which sends the `quit` command to GDAIS-core, as described in section 3.6.4. Once executed the equipment details page is shown with a message informing that the command to finish the acquisition has been sent. If there has been an error, it is displayed instead of the message. An example of the message shown when no error occurs is displayed in Figure 3.42.

notify_quit

This view is used to receive the notification from GDAIS-core that an equipment acquisition has finished. GDAIS-control logs that the equipment is not running anymore and returns a reply in JSON format, as in `notify_start` view. An example of the returned reply for *sidamon* equipment would be:

```
{"status": "OK", "info": "sidamon quit notified"}
```

download

Figure 3.42: Message shown when the acquisition quit command is correctly sent to GDAIS-core

At the bottom of Figure 3.41 page a list of *aquired data files* is included. Clicking any of the items in the lists calls the download view with the equipment name and the date and time of the acquisition. Then, this view streams the contents of the file to the user, which can store it and view it using an HDF5 viewer or from MATLAB.

Features for advanced download managers are provided. These features include a hash tag of the file, the last time it was modified and positional access, being able to continue the download from anywhere on the file. This would be very useful in applications of GDAIS in remote locations, where bandwidth is usually very limited and the connection may fail from time to time. As most times the acquisition files are quite large, loosing connection would require the download to be restarted from the beginning each time. Thanks to this feature of being able to continue the download from where it was when the connection was lost, this problem is eliminated.

delete

Also in Figure 3.41, at the right of each acquired data file there is a link to delete this file. When the link is clicked, the delete view is called with the equipment name and the date and time of the acquisition. Then, the view removes the file associated with this information and displays a message at the top of the equipment details page. Figure 3.43 shows an example of this message.

3.6.6 Command-line control interface

In addition to GDAIS-control block, another method is provided to control GDAIS-core, without relying on the web interface. Therefore, this method eliminates the performance cost that may suppose having a web server running along with GDAIS-core main acquisition block, and it should provide better performance on resource-limited devices, like in

Figure 3.43: Message shown when an acquisition data file has been deleted

embedded systems.

This alternative control system is provided as a set of two Bash scripts to start and stop GDAIS-core, a log file that is generated for each acquisition with GDAIS-core and a folder from which the acquisition data files can be obtained. These files are intended to be used from a local console or remotely, through SSH or Telnet.

This CLI consist of the following parts:

`run.sh` script

This script takes care of starting a GDAIS-core process for the given equipment. It should be executed from the GDAIS-core folder, along with the Python code files. The command line for an example equipment *sidamon* would be:

```
$ ./run.sh conf/equips/sidamon.conf
```

Executed in the previous form, the script starts the process and displays all the log messages from GDAIS-core. The script keeps executing until the GDAIS-core process exits.

Alternatively, the script can be called with the `-d` parameter. Then, the script start GDAIS-core and it returns back to the command line, not displaying any log message. These can still be accessed through the `debug.log` file. The command line in this case would be:

```
$ ./run.sh -d conf/equips/sidamon.conf
```

`stop.sh` script

This scripts sends the `quit` command to GDAIS-core through a TCP connection it creates. This stops the acquisition and exits GDAIS-core process. The command line to call this script would be:

```
$ ./stop.sh
```

`debug.log` file

In any execution of GDAIS-core this file is created with all the log messages the program creates while it is running. This can be used to check if any error has occurred and to debug the execution of the system if it is not working as expected.

`data/` folder

The acquired data files from each acquisition session are stored in this folder, classified by equipment short name. If GDAIS-core is run through the CLI, the produced acquisition files can be downloaded from this folder.

3.7 Licensing

The development of GDAIS system has been an important effort, but would have been extremely much difficult to implement this system without all the different technologies on which it is based, see section 3.5. Reusing all these available libraries and features has been possible thanks to their permissive and open-source licenses.

Moreover, this project has been a real attempt to provide a generic solution to the proposed problem. Therefore, sharing the results of this development is expected to promote its reuse in multiple situations, proving that the initial objective to provide a truly generic instrument control and data acquisition system has been completely satisfied.

For all these reasons, in order to allow other projects profit from this development effort and to promote GDAIS system, the decision was made to license all the code developed in this project under an open-source license. Different licenses were evaluated and finally the GNU General Public License (GNU GPL) version 3 was chosen. This license, in addition to making the GDAIS system open-source, also adds a restriction on its redistribution: derived versions of the system must be licensed under this same license, thus preserving the open-source status of the development. This is commonly known as a *copyleft* license. The whole license terms can be found in Appendix A.

3.8 Conclusions

This chapter has reviewed the whole GDAIS system development, from the initial analysis of the requirements of the project to the final design of each of the modules that form each block of the system. The obtained system provides all the planned features in its

implementation, but some testing is required to check that all the functional and non-functional requirements are covered on the resulting system.

The next chapter focuses on testing the implementation of GDAIS system. Through two different experiments, most of the features of the system are checked.

Chapter 4

Comprehensive tests of GDAIS

This chapter consists of the analysis of some equipments developed at the RSLab. Using the GDAIS system, a solution to control and acquire data from each equipment is developed and tested. The tested instruments and the main test goals for each of them are the following ones:

Airborne L-band Radiometer
: The goal is to test GDAIS system with an equipment that has a control interface already implemented and that requires a fast acquisition rate.

W-SMIGOL
: The objective is to check how GDAIS can be used to implement an acquisition system, adapting to a system that has slightly different requirements than the ones that were considered in the original GDAIS design.

4.1 Airborne L-Band Radiometer

This section analyses the use of GDAIS system to control and acquire data from the Airborne RadIomEter at L band (ARIEL) instrument, which is carried by a radio-controlled aircraft [16], as seen in Figure 4.1.

Firstly, the equipment and its applications are introduced, with a brief review of the existing solution to control and acquire data from it; secondly, the steps to reimplement this system using GDAIS are explained; and finally, the new system is tested and evaluated.

Figure 4.1: ARIEL radiometer during a flight test in Ripollet, Barcelona [16]

4.1.1 Introduction

This system originated from a RSLab project to design an airborne radiometer capable of generating soil moisture images to investigate its possibilities. The whole platform was self-designed and developed for the specific requirements of the project, including the remotely controlled aircraft.

The aircraft is designed to be capable of carrying a 5 kg payload with an autonomy of 30 minutes to scan an area over land and/or sea. As it can be observed in Figure 4.2, in addition to the aircraft instrumentation, an hexagonal 7-patch array antenna is mounted under the aircraft, which feeds the signal to the radiometer, and an on-board nadir-looking video camera is also embedded to help with the interpretation of the data acquired by the payload. The inner part of the fuselage is prepared for loading the necessary devices like a GPS receiver and attitude sensors.

Figure 4.2: Aircraft with the radiometer embedded and the antenna at the bottom [16]

In the latest version of the system [17], an on-board embedded computer was included to collect the measurements. The control software that runs on this computer acquires data from each of the sensors in the aircraft, which include the ARIEL radiometer and

a commercial Inertial Measurement System (IMS) to geo-locate the acquired radiometric data with the attitude information. For this purpose, the embedded computer acquires the position and attitude data synchronously with the radiometric data.

ARIEL system architecture

Figure 4.3 shows the whole ARIEL system diagram. For this analysis, the relevant part of the system is marked with a red rectangle, which is the one related with the airborne equipment. As it can be observed in the figure, it consists of two sensors and the software controller block which runs on the embedded computer. The original output of this block is a raw file that contains the synchronously acquired data from both instruments: the MTi-G, which provides GPS time, position, velocity and attitude of the platform; and the ARIEL radiometer, which provides antenna brightness, reference temperature and physical temperature measurements. This data file is the input for the next block, the ARIEL processor.

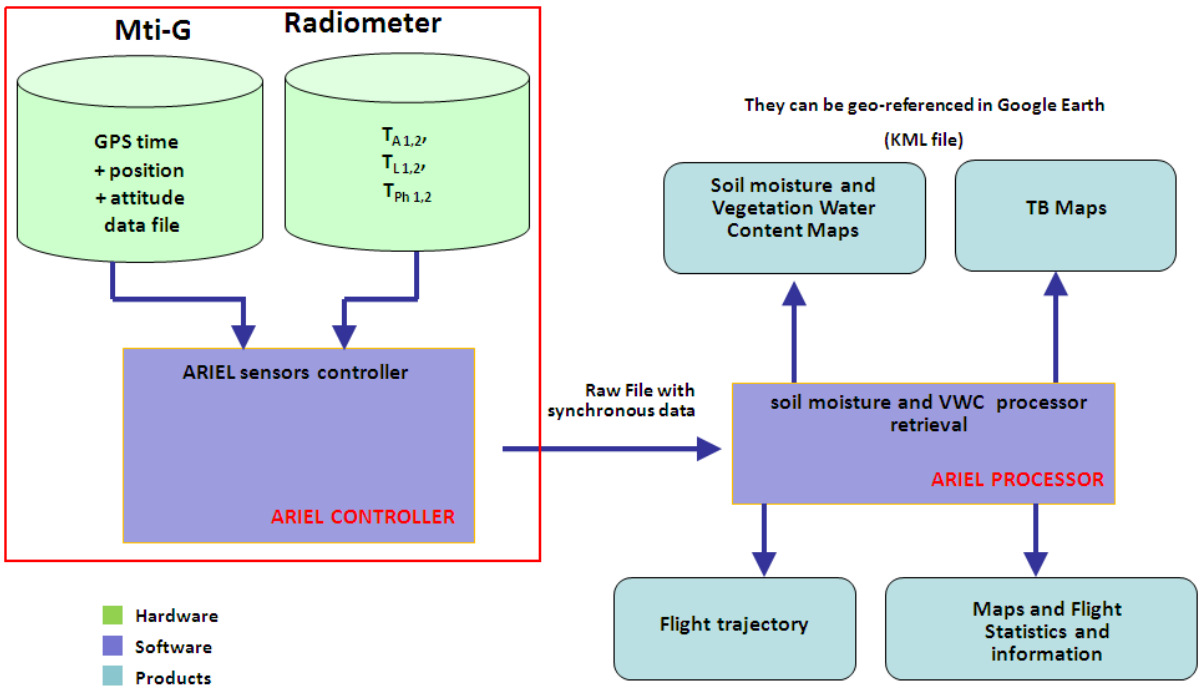


Figure 4.3: ARIEL system architecture diagram [17]

GDAIS integration

The aim of this experiment is to substitute the previously described ARIEL sensor controller which runs in the embedded computer with GDAIS system. For the final applica-

tion to be completely function with this solution, essentially two changes to the original architecture would be required:

1. Substitute the ARIEL controller with GDAIS-core block to provide the instrument control and data acquisition functionalites.
2. Adapt the existing ARIEL processor, so that it can obtain the acquired information from the generated HDF5 file instead of the raw file used in the original system.

The most complex change is the first one, as the second one should be quite simple using any of the available libraries to load HDF5 files and extract data from them. The resulting block of ARIEL processor should be much more simple, than the existing one that has to extract the information from a raw file by itself.

The following sections concentrate on the configuration of GDAIS system in order to provide the same features that ARIEL controller.

4.1.2 On-board instruments

Xsens MTi-G sensor

This equipment consists of a GPS receiver, which records the position and speed, and attitude and static pressure sensors, which are used to geo-reference the aircraft. Figure 4.4 shows the external overview of the MTi-G system packaging.

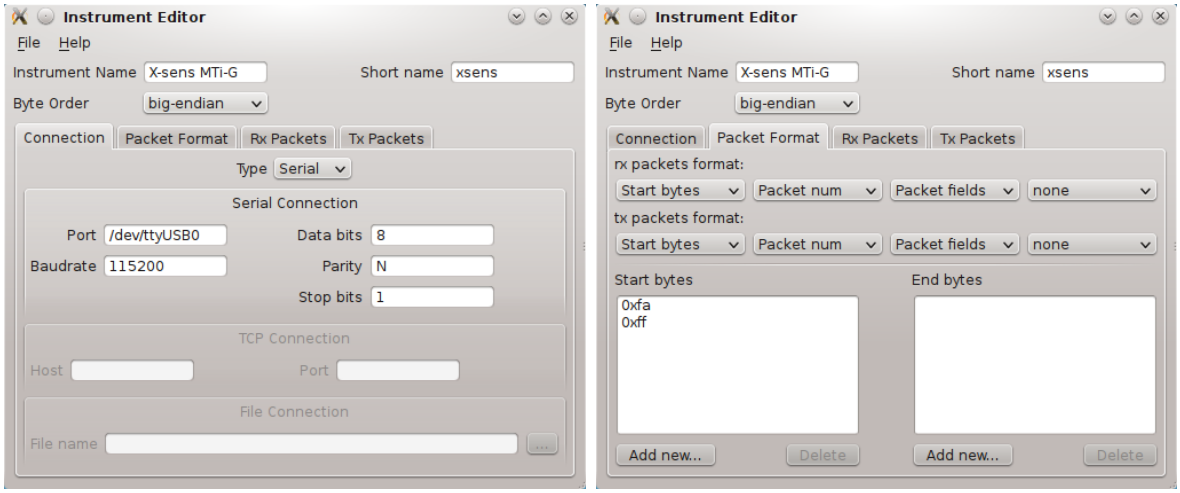
Figure 4.4: External view of the Xsens MTi-G packaging

The MTi-G's Inertial Measurement Unit (IMU) sensors consist of 3 axis magnetometers, 3 axis accelerometers and 3 axis gyroscopes. The data is collected and processed in real-time, with the device embedded Digital Signal Processor (DSP) that runs a fusion algorithm to provide enhanced attitude/heading and inertial enhanced position/velocity data. To ensure the data accuracy, each sensor is individually calibrated for temperature to detect any 3D misalignment and sensor cross-sensitivity.

To access all this information, a serial RS-232 communication interface is included, which provides a maximum update rate of 120 Hz. The X-sens company provides to the user two different ways to interface the MTi-G through the serial protocol RS-232, the first one uses a high abstraction level layer and the second one provides a low-level communication protocol. For this application the low-level layer was chosen, due to the possibility to control the different communication stages and its lower complexity.

The low-level protocol consists of a set of messages that can be sent and received to/from the instrument. Internally, the MTi-G has two different states: the configuration and the measurement states. Changing between them is achieved by sending an specific command of the protocol.

To integrate this instrument into GDAIS system, the packet structure of the protocol messages had to be mapped to the instrument description format defined. Fortunately, the format of these packets can be represented with the already available in GDAIS packet format description. Therefore, a description of this instrument could be easily created just using the GDAIS instrument editor, as it can be observed in Figure 4.5. In it, the connection settings tab for the instrument and the packets format description are shown.

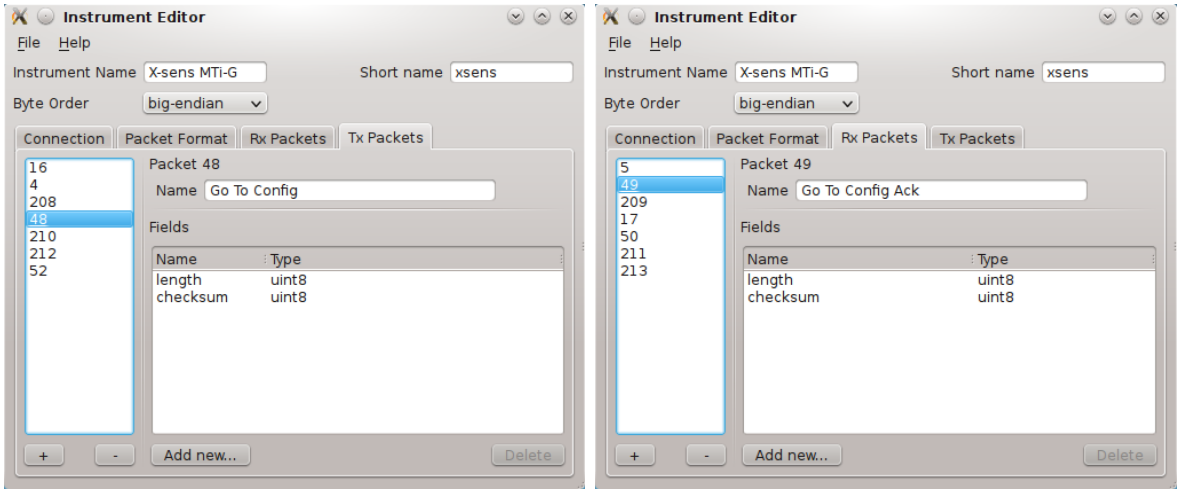


(a) Connection settings (b) Packets format

Figure 4.5: MTi-G instrument description

In Figure 4.6, the command to change to the configuration state of the device is shown, together with the reply message to acknowledge that the command was correctly received and executed.

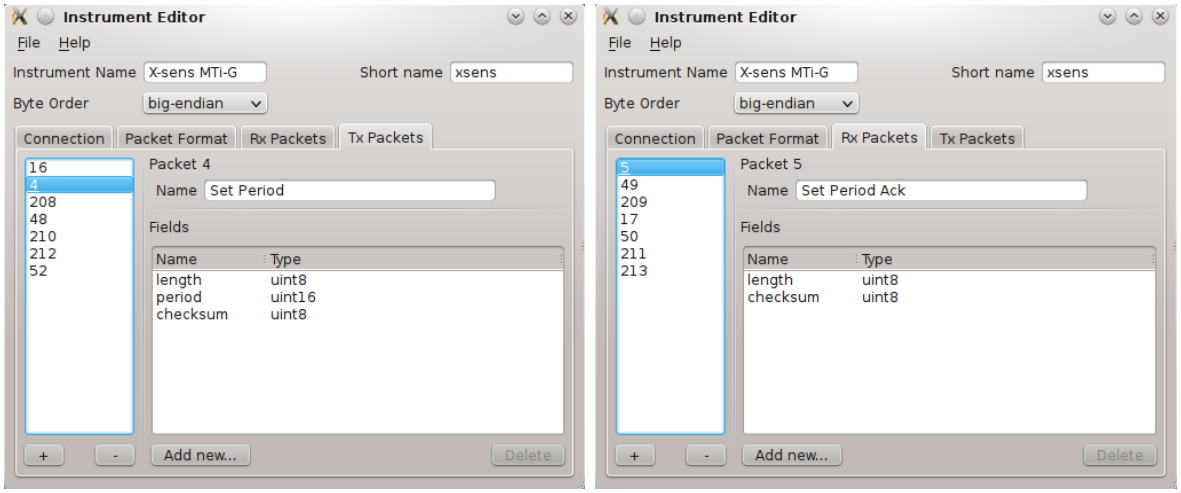
While in this configuration mode, some commands can be sent to the device to change the settings. For example, Figure 4.7 shows the command to set the period between each measurement, with the associated acknowledgement message.



(a) *Go To Config* transmitted packet description

(b) *Go To Config Ack* received packet description

Figure 4.6: MTi-G *Go To Config* protocol messages



(a) *Set Period* transmitted packet description

(b) *Set Period Ack* received packet description

Figure 4.7: MTi-G *Set Period* protocol command and reply

When all the configuration modifications are completed, the *Go To Measurement* command can be used to return the device to the measurement mode. From there, the *Request Data* command can be used to retrieve new measurements when desired. The returned message contains all the information provided by de device, as configured in the settings mode. For the configuration used in this application, the received packet format description is shown in Figure 4.8. As it can be observed, all the previously described parameters of the device are obtained from this single message structure.

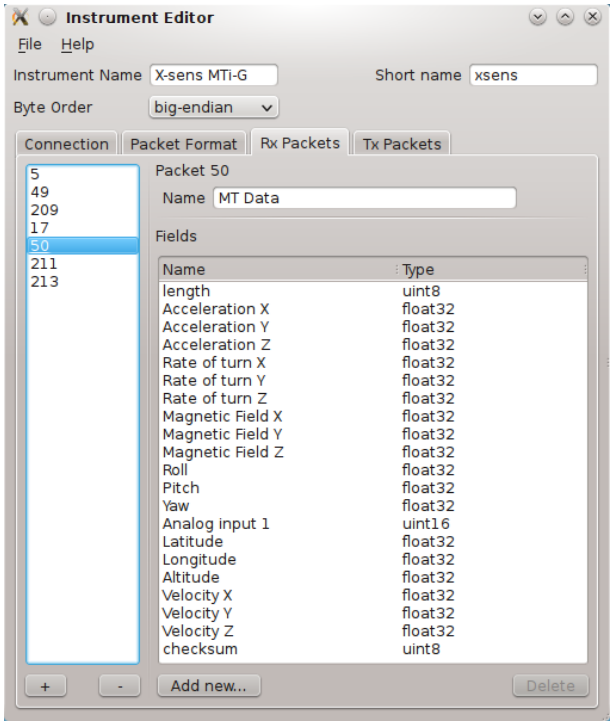


Figure 4.8: MTi-G *MT Data* message with measurements information

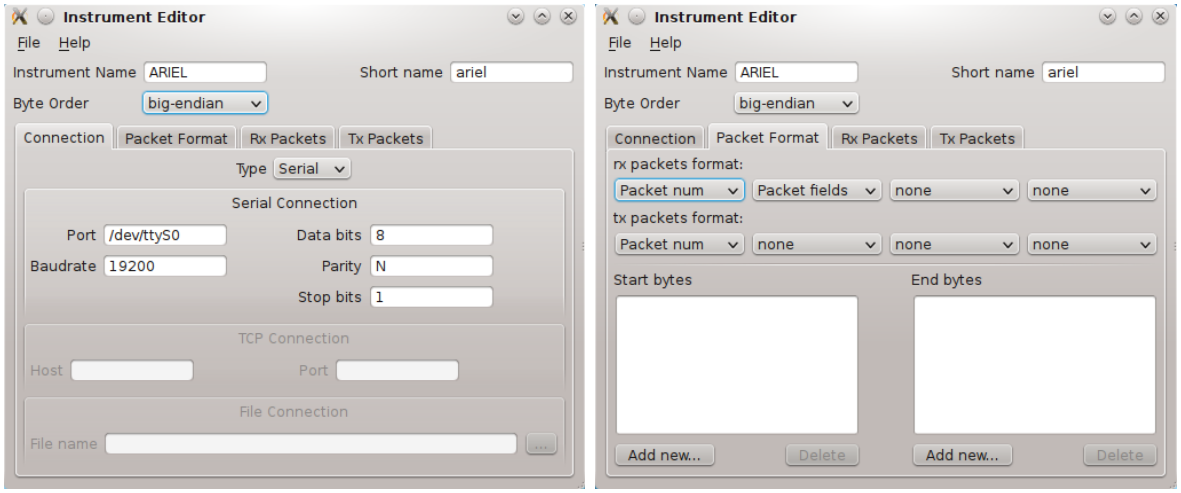
ARIEL radiometer

The ARIEL is a RSLab self-manufactured radiometer with two possible topology configurations, the Total Power Radiometer (TPR) and the Dickie Radiometer (DR), that works at the remote sensing reserved L-band (1,400 – 1,427 MHz) [16].

In the radiometer on-board system, there is a microcontroller (PIC-18F4520) which interfaces the radiometer with any external device using a set of command through the RS-232 serial connection. The communication protocol with this device is simpler than the previous one, as there are no different states and there is a command-reply pair for each possible measurement the instrument can be requested. The available measurements include: brightness antenna temperature, reference temperature, physical reference temperature, physical internal temperature, physical external temperature and brightness

temperature of the Dickie's topology.

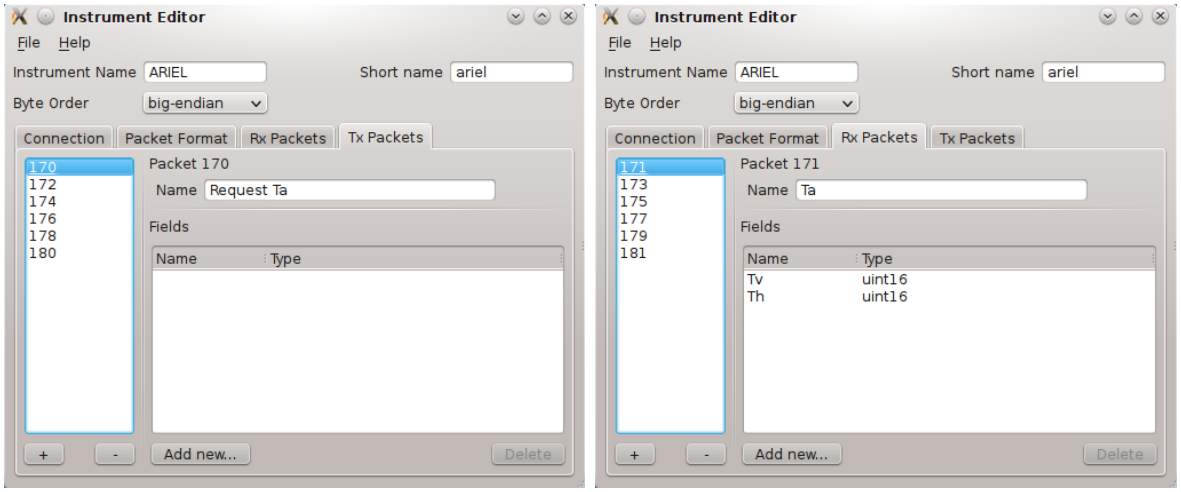
Integrating this instrument with GDAIS system was also as simple as creating its description with the instrument editor GUI. Figure 4.9 shows the serial connection settings and the simple packet format for this device, which as it can be observed just consists of an identifier for the commands and an identifier followed by some fields for the reply messages.



(a) Connection settings (b) Packets format

Figure 4.9: ARIEL instrument description

Since all the measurements that can be requested are similar, an example of one of them, the brightness antenna temperature measurement, is provided in Figure 4.10. The other commands and replies follow the same structure, just changing the meaning of the reply messages fields.



(a) *Request Ta* command description (b) *Ta* measurement reply description

Figure 4.10: ARIEL protocol messages to request brightness antenna temperature

4.1.3 Equipment Configuration

Once each instrument has been separately described, the equipment that integrates both instruments can be described with the equipment editor GUI. In it, the configuration and operation mode for each device is specified.

Xsens MTi-G sensor

As previously commented, this instrument has an special configuration mode where the system can be set up. Before each acquisition session, the instrument has to be properly configured to ensure that the acquired measurements have the expected format and provide the desired information. The sequence of commands is the following one: *Go To Config*, *Set Output Mode*, *Set Output Settings*, *Set Period*, *Set Output Skip Factor*, and finally *Go To Measurement* to return to the state where measurements can be requested.

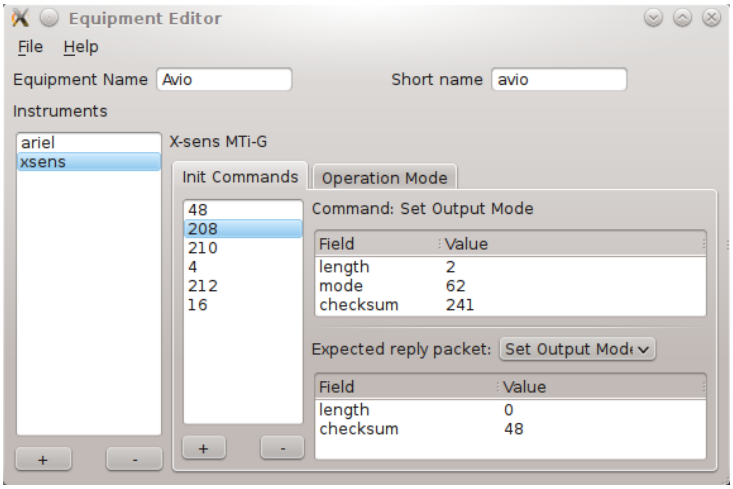


Figure 4.11: MTi-G instrument initialization commands

Figure 4.11 shows the equipment editor initialization commands tab with the desired configuration introduced for the *Set Output Mode* command, and the corresponding expected reply, to check that the command was correctly received and interpreted.

Regarding the operation mode, as acquiring as much data as possible is desired, it is set to *Blocking Sequences*. As it can be observed in Figure 4.12, only the *Request Data* command command is configured to be repeated infinitely during the acquisition.

ARIEL

Regarding the ARIEL radiometer, there is no initialization sequence required for it. Once started, it is ready to reply to acquisition commands. As in the MTi-G, also the operation

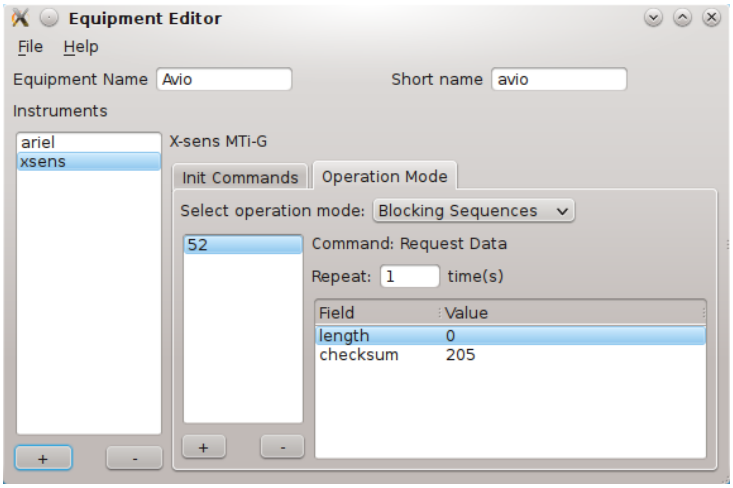


Figure 4.12: MTi-G instrument operation mode settings

mode is set to *Blocking Sequence* to acquire as much data as possible. As it can be observed in Figure 4.13, three measurement request commands (*Request Ta*, *Request Tref* and *Request Tph_ref*) are configured to be sent cyclically to the instrument infinitely.

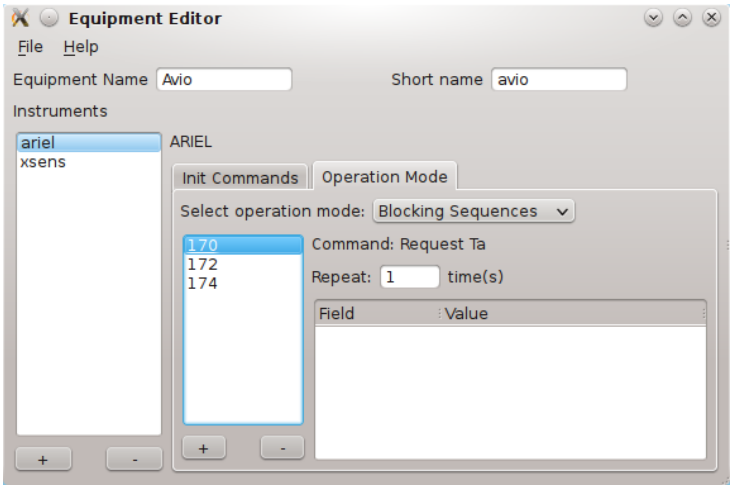


Figure 4.13: ARIEL radiometer operation mode settings

4.1.4 Testing the system

Once the two instrument descriptions and the whole equipment description were completed, everything was ready for testing. Since the current version of GDAIS runs only on a PC, the test were done in the laboratory, connecting both devices to a computer. Therefore, most of the measured values are meaningless, for example the GPS information from the Xsens MTi-G. This does not affect the results of the experiment, as the aim of it was to check that the system worked as expected and the performance of it.

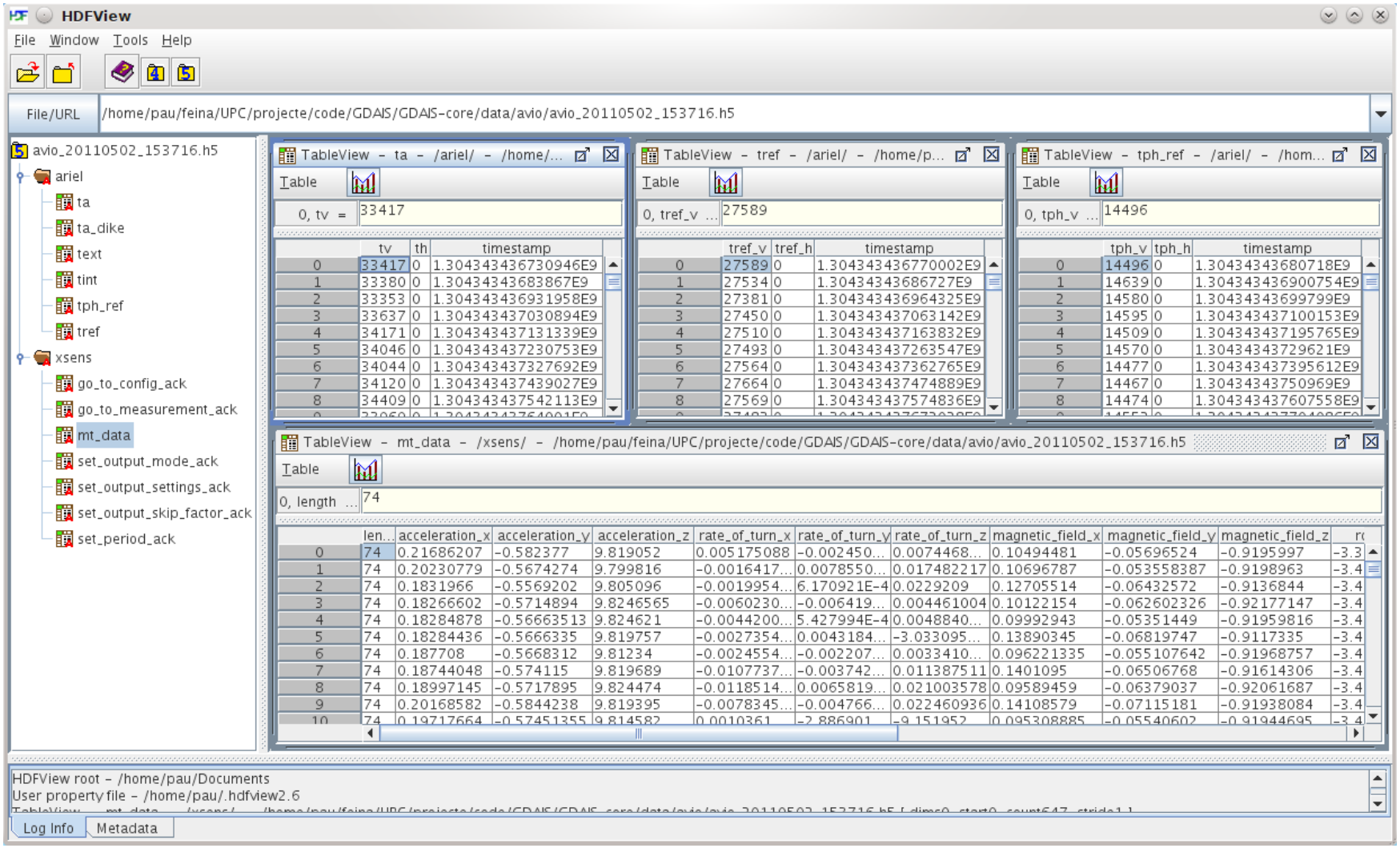


Figure 4.14: Visualization of ARIEL acquisition results in HDFView HDF5 viewer

Figure 4.14 displays a snapshot of the *HDFView*[1] program, which is a multi-platform HDF5 GUI viewer that can be used to navigate through any data file that complies the HDF5 format, like the ones that GDAIS-core produces.

In this snapshot, the acquired data from one of the tests is shown. As it can be observed, at the left side of the capture, two folders are displayed, one for each instrument, and inside these folders there is an item for each possible received packet from the instrument. For the ARIEL instrument, the six different measurements it provides are shown; and for the Xsens MTi-G, the only relevant packet is the *mt_data* that contains the values returned for each measurement request command. The other packets displayed in the Xsens MTi-G folder are the acknowledgment replies to the configuration and mode changing commands, which do not contain relevant measurements.

On the right side of the snapshot, the values received for four different packets are shown. The three packet lists at the top correspond, from left to right, to the brightness antenna temperature, reference temperature and physical reference temperature measurements provided by the ARIEL radiometer. These are the only packets of this instrument with acquired data, as in the equipment description, the request commands for these measurements were the only ones added to the commands list in the operation mode settings tab, see Figure 4.13. The displayed values for each packet show the measurements for the vertical polarization correctly. The values for the horizontal polarization are all zero because this field was added to the packets for generalization when the protocol was defined, but ARIEL does not provide values for this polarization. The last column corresponds to a time stamp added by GDAIS to each measurement when it is recorded.

The other packet values list, the one at the bottom, shows the acquired measurements from the Xsens MTi-G device. In the snapshot the acceleration, rate of turn and magnetic field values for each axis (x, y, z) are displayed. Specifically, the acceleration in the z axis values, which are approximately 9.81 m/s^2, show that the system is receiving and parsing measurements correctly, as this value corresponds to the gravity acceleration.

Once the acquisition session was completed, the measurement rate for each instrument was calculated. GDAIS was able to acquired from the ARIEL radiometer at a rate of 29.2 measurements per second and from the Xsens MTi-G at a rate of 81.4 measurements per second.

[1] http://www.hdfgroup.org/hdf-java-html/hdfview/

4.1.5 Conclusions

The main conclusion extracted from this experiment was that the system worked as expected, being able to acquire data from two different instruments and generating a single output file with all the measurements of the session. This was achieved without adding or modifying any part of GDAIS source code, just using the GUI utilities provided with the system. Therefore, the main project goal was achieved.

In a more detailed analysis, the description of each instrument was correctly interpreted by the system, configuring the serial ports and binary packet details. Moreover, after some failed tries, the more advanced instrument, Xsens MTi-G, was correctly initialized, providing the measurements in the format introduced in the initialization sequence and correctly recognized by the packet parser of GDAIS.

Another feature of the system was tested in this experiment, the capacity to acquire from multiple instruments in parallel at the same time, thanks to the multi-threaded implementation of the acquisition system, running each instrument connection and parser in independent threads. This was one of the requirements of this system, as the measurements from the Xsens MTi-G were used together with the ones from ARIEL to infer the ground footprint that was measured by the radiometer.

Finally, the measurement rate achieved in this experiment was measured. Therefore, it could be compared with the results achieved by the previous control system for this equipment. In the original system, specially designed and implemented to control and acquire from this two instruments, the measurement acquisition rate was 32 Hz for acquiring one measurement from each instrument. In this experiment, the combined achieved rate was 29.2 Hz, a bit less than the original value, but considering the time required to implement the original solution compared to the time required using GDAIS system, this was a very positive result. Moreover, as the acquisition rate achieved for Xsens MTi-G instrument was higher, 81.4 measurements per second, the resulting rate was limited by the ARIEL radiometer delays added to the ones from GDAIS.

4.2 W-SMIGOL

This section analyses the use of GDAIS system to control another remote sensing equipment. As in the previous one, firstly an overview of the device is provided, with the requirements for controlling it; secondly, the adaption of GDAIS to this concrete application is reviewed; and finally the whole system is tested and evaluated.

4.2.1 Introduction

Actually, W-SMIGOL is an upgrade of a previous instrument called SMIGOL [18, 19]. This original device is a reflectometer developed at the RSLab, that uses opportunity GPS signals and a principle called Interference Pattern Technique (IPT) for remote sensing applications. These applications include:

- Retrieving soil moisture
- Measuring vegetation height
- Generating topographic maps
- Measuring snow thickness
- Water level monitoring

(a) (b)

Figure 4.15: SMIGOL reflectometer at Palau d'Anglesola, Lleida, Catalonia [19]

As seen in Figure 4.16, SMIGOL is implemented using a commercial GPS receiver connected to a self-manufactured antenna, which has a beam width of 90°, therefore it just can take measurements from a quarter of its surroundings. To store the acquired GPS data, a data logger hardware device is included in the system, which uses an SD card to save the measurements. Finally, the control of this whole system is implemented with a

PIC[2]. This microcontroller takes the responsibility of starting and stopping the acquisition, which usually runs for 6 continuous hours and then it keeps idle for another 6 hours, repeating this pattern infinitely.

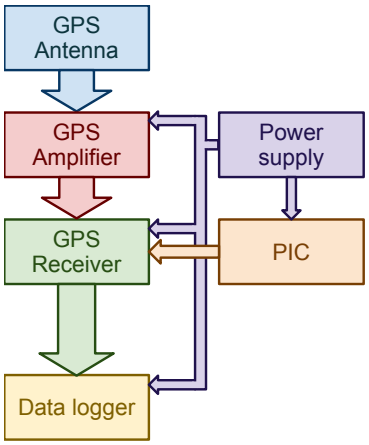


Figure 4.16: Internal structure of SMIGOL [20]

W-SMIGOL [20] is an upgrade of the original SMIGOL device to extend its range to 360º and provide a wireless access system to download the acquired data stored in it, instead of having to open the system and get the SD card and then downloading the collected data into a computer. This wider range version is achieved by replicating 4 times the original SMIGOL antenna, GPS receiver and data logger, in order to cover the 4 faces of its cubic structure.

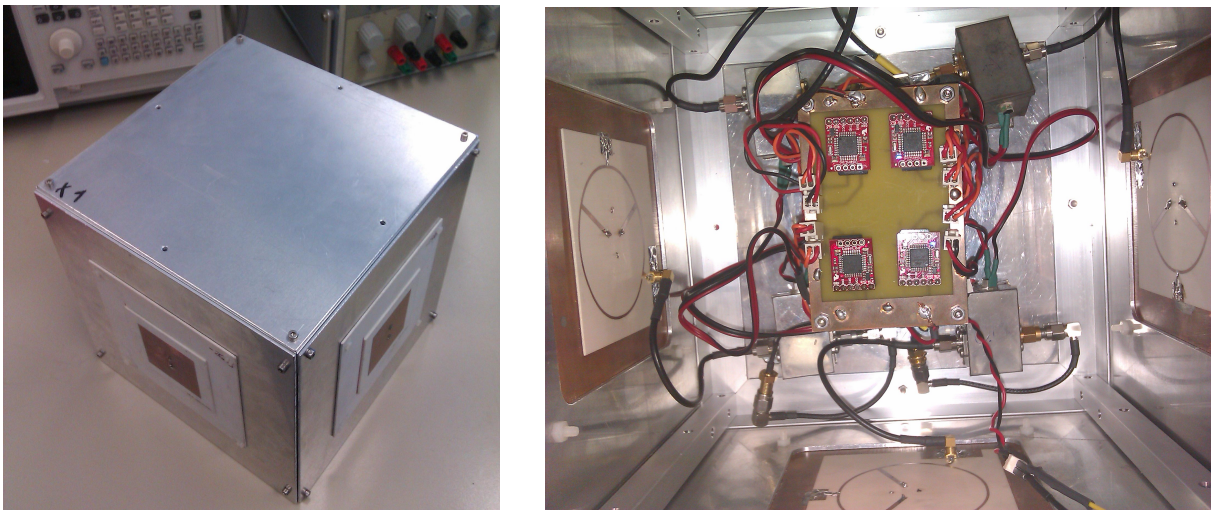
4.2.2 W-SMIGOL system architecture

As it can be observed in Figure 4.17a the external structure of W-SMIGOL shows the 4 sides of the device, which are like the original SMIGOL antenna, but replied in each face. The centered square on each side is the antenna responsible for receiving the GPS signal, which is fed to each of the 4 GPS receivers.

Figure 4.17b shows a photograph of the internal structure of the equipment, and Figure 4.18 shows a diagram of this same structure with the four replied acquisition chains. In each chain, the signal from the antenna that enters the GPS receiver is analyzed to produce some configured measurements, which are sent through the serial port of the receiver. Since the GPS receiver is directly connected with the data logger, this information is immediately stored in the SD card.

Like in the initial SMIGOL version, the whole system is controlled by a PIC microcontroller. However, as it has 4 reception chains instead of only one, W-SMIGOL PIC

[2] http://www.microchip.com/en_US/family/8bit/index.html



(a) External view (b) Internal view

Figure 4.17: Internal and external views of the W-SMIGOL equipment [20]

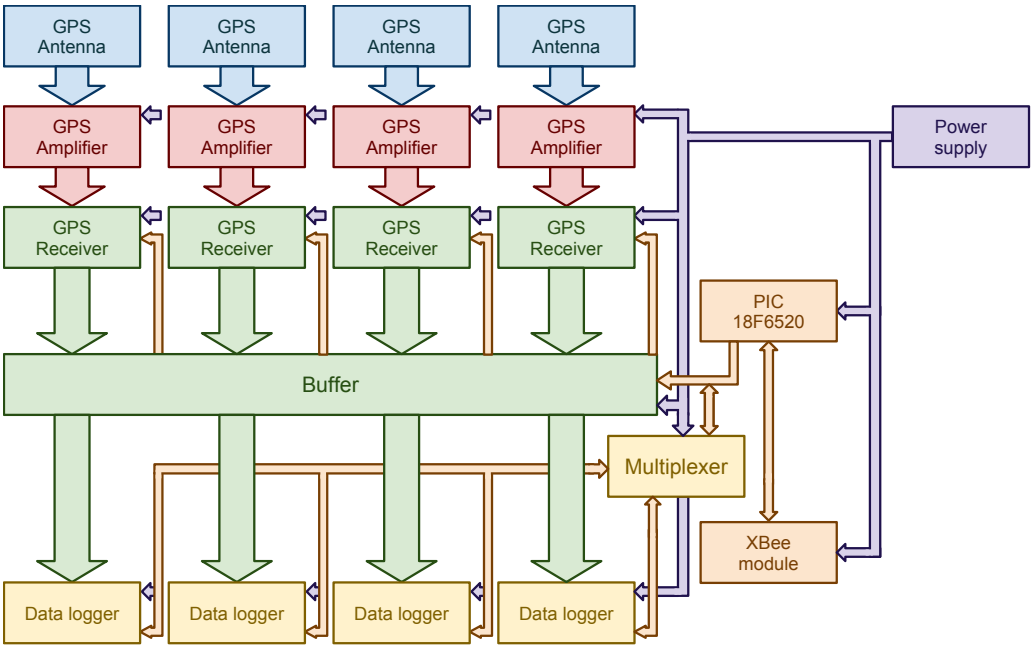


Figure 4.18: Internal structure of W-SMIGOL [20]

software has to manage the acquisition start and stop for each chain. Moreover, as the wireless connectivity was added, the PIC is also responsible for managing the download of acquired measurements through its serial port when it is not acquiring data.

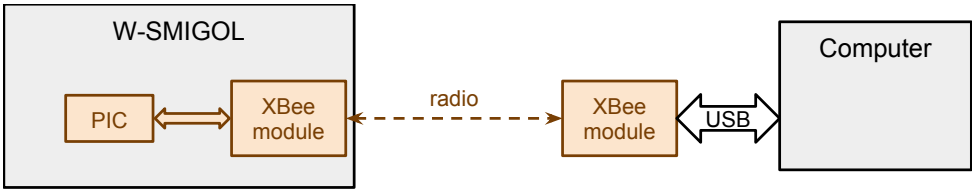


Figure 4.19: Communication interface between W-SMIGOL and a computer

Once the acquisition phase is completed, the PIC enters into a state where an external request can trigger the download of the acquired data from each of the data loggers. Figure 4.19 shows the communication interface introduced for this purpose. As it can be observed, the serial port of the PIC is connected to a XBee[3] module which communicates with the other end XBee. This last device is connected through a serial port to the computer which is being used to download the information.

As the data is stored in 4 independent data loggers and there is a single download channel, a protocol has been implemented for the computer to download each data logger measurements separately. It consists in, once the serial connection is established and the W-SMIGOL is not acquiring data, sending `'DL1'` character sequence from the computer. Then, the PIC gets all measurements stored in the data logger of the first chain and sends them to the computer. When all data from the data logger has been transmitted, the PIC sends the `'END DL'` sequence to inform the computer that the end has been reached. When the computer receives this mark, it can send the command to download the measurements from the next data logger, which is `'DL2'`. This procedure is repeated for each data logger until all the information has been retrieved from all four devices.

4.2.3 GDAIS integration

After describing the equipment and its communication interface, this section concentrates on the implementation of a data acquisition system for this device using the GDAIS system. In this occasion this is not as simple as the previous one, mainly because of having 4 independent measurement sources multiplexed through a single connection. Therefore, the original architecture design of GDAIS does not apply to this system, as GDAIS identifies each data source with an independent connection. For this reason, a new approach has

[3] http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-series1-module.jsp

been developed, which extends GDAIS system by adding a wrapper application between the serial connection and the GDAIS-core block.

W-SMIGOL wrapper

A new application has been developed and implemented to integrate W-SMIGOL with GDAIS-core. The main task carried out by this application is providing a single serial interface for the W-SMIGOL equipment on one end, and in the other end 4 independent ports that can be connected to GDAIS-core, thus having the information from each data logger separately. Figure 4.20 contains a representation of this structure.

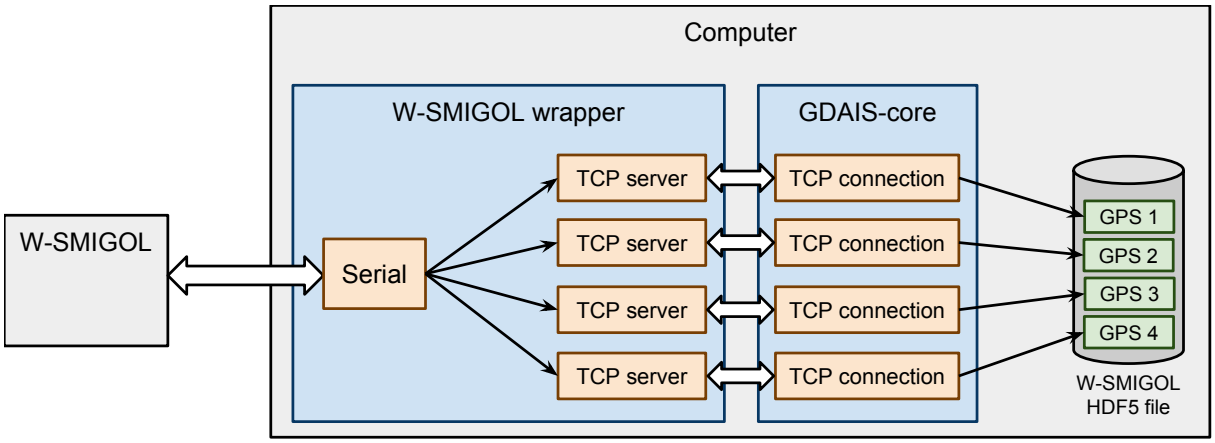


Figure 4.20: W-SMIGOL wrapper architecture and environment

When new data is desired, this application is started. Then, GDAIS-core is also started and the 4 ports are connected between both applications. Once the connections are ready, W-SMIGOL wrapper initiates the equipment protocol by sending the `'DL1'` command and redirects all bytes received from the W-SMIGOL serial port to the first connection with GDAIS-core, until the end sequence, `'END DL'`, is received. Then, the wrapper sends the command to download the information from the next data logger, but this time it redirects all the bytes it receives to the second connection with GDAIS-core. The process is continued until all data loggers are read and their information has been redirected through the assigned connection to GDAIS-core. Finally, the wrapper application exits.

To implement the connections between the wrapper and GDAIS-core, TCP was chosen. TCP is a well-tested protocol that can be used to connect different processes running in the same computer.

The internal design of the application is represented in Figure 4.21. As it can be observed, there is a `ControlServer` class which listens for control commands on a TCP port to start and stop the wrapper. There are also four `ProxyServer` class instances that create

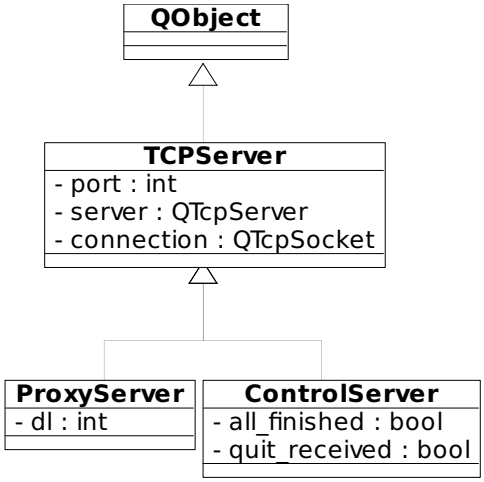


Figure 4.21: W-SMIGOL wrapper class model

a TCP server each, thus providing four connection points to GDAIS-core. Through this connections, each `ProxyServer` instance retransmits the binary data stored in one of the W-SMIGOL data loggers.

As both `ControlServer` and `ProxyServer` classes have a common part — the TCP server — this is implemented in the generic `TCPServer` class from which these two classes inherit. This generic class inherits from Qt `QObject` class to provide the useful functionalities of Qt library, like the *signals and slots* mechanism. The actual TCP server is also implemented using Qt, through the `QTcpServer` class, which provides a `QTcpSocket` instance for each connection that is established at the port where the server is listening. The main execution block of this application uses the `QCoreApplication` base class to provide the environment for the application and all the other instances of Qt classes.

The application workflow is the following one: after the application has started and GDAIS-control has connected to the four TCP servers, when `ControlServer` instance receives the start command, a *signal is emited*, which is received by a *slot* of an instance of `ProxyServer` class. This instance is already connected to GDAIS-control through TCP and it creates a serial connection with the W-SMIGOL serial port. Then the protocol to acquire data from the first data logger is started, redirecting all the bytes received from W-SMIGOL to GDAIS-core through the TCP connection. Once the end of data message is received from W-SMIGOL, the `ProxyServer` instance closes the serial port and *emits a new signal*, which this time is *connected to a slot* of the next `ProxyServer` instance. This creates again a serial connection with W-SMIGOL and downloads the information from the next data logger, redirecting it to GDAIS-core through the second TCP connection. The process continues until all data loggers have been read and their data has been sent to GDAIS-core, through each independent TCP connection. The *signal* from the last `ProxyServer` instance is connected to the `quit` *slot* of the `QCoreApplication`, which

closes all the TCP servers and exits the application.

GDAIS-core required changes

When this experimental application of GDAIS was developed, the system only provided the file and serial connections. As the chosen connection between the W-SMIGOL wrapper and GDAIS-core was TCP, it had to be implemented. This provided an opportunity to check whether the reusability and extensibility requirements of the project had been fulfilled. The following parts of the system had to be extended:

Instrument editor GUI

This modification consisted in adding the configuration settings for this new connection type to the instrument editor GUI. As it can be observed in Figure 4.22a, the TCP settings added included an Internet Protocol (IP) address and a TCP port. Also, this new connection type had to be added to the selection box at the top. All these modifications could be easily made with Qt designer, just reusing the existing blocks for the serial connection and changing the names. In addition to the pure GUI changes, the class that manages the GUI content load and storage had to be extended to manage the new settings for this connection.

As for the already implemented connections, a new `TCPConnectionCfg` class had to be added to the instrument description module, that inherited from `ConnectionCfg`. This class is used both by the instrument editor and GDAIS-core blocks to obtain the details of the connection. `TCPConnectionCfg` class was already displayed in Figure 3.22 class diagram. As it can be observed, it has the IP address and TCP port attributes.

New `TCPConnection` class

In addition to providing a description of the new connection type, the class that implements this new connection type had to be created. It was defined along the already existing ones, using the `QTcpSocket` Qt class to connect to the remote TCP server. This new `TCPConnection` class inherits from the `Connection` class to use the common connection methods to read and write binary data, as it was already shown in Figure 3.33.

It should be noted that none of this modifications required duplicating any functionality of the system, just adding the new connection type to the parts of the system that describe connections was enough. Therefore, it can be concluded that the system is extensible in connection types.

Control script

In order to automate the whole download process, a control script was designed to start the W-SMIGOL wrapper and GDAIS-core, send the command to begin the download and close everything when the process was completed. It was implemented in Bash shell scripting language. The sequence of actions it provides is:

1. Start W-SMIGOL wrapper in background
2. Connect to the `ControlServer` instance of the wrapper
3. Start GDAIS-core using the CLI script `run.sh` (see subsection 3.6.6)
4. Connect to GDAIS-core control server
5. Change GDAIS-control logging level to `WARNING`, reducing the verboseness
6. Send the *start* command to the wrapper
7. Wait until the background wrapper process exits, when all data has been downloaded
8. Send the *quit* command to GDAIS-core
9. Inform the user that the download was completed

With this script, executing the download of all the information from the data loggers is just a matter of running the script and waiting for it to finish. Then, the acquired data can be retrieved from the `data` folder of GDAIS-core.

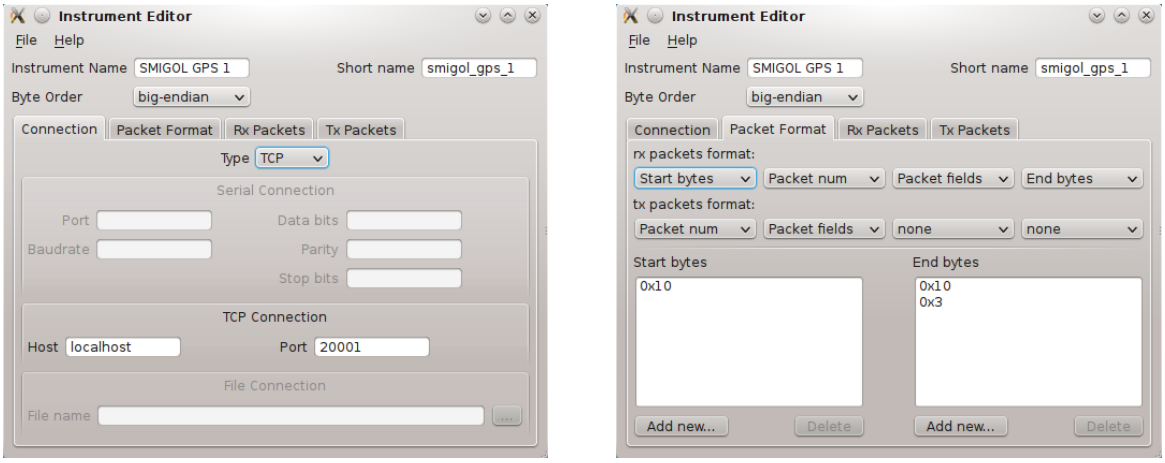
4.2.4 GPS instrument

Once GDAIS system was adapted for the new requirements, the usual procedure for acquiring data form an equipment using some descriptor files was possible.

The first part of this procedure was creating the instrument description. In W-SMIGOL device there are four instruments, but they are all the same one in fact. Therefore, all four generated instrument descriptions were quite similar, the only difference being the TCP connection port to the wrapper.

Even thought the binary information is not received directly from the GPS receivers by GDAIS-core, instead coming from the data loggers, for the system this does not matter as the binary sequence is the same. This binary information follows the Trimble Standard Interface Protocol (TSIP), which is the same protocol that used the GPS receiver used to test during the prototype iteration. Therefore, the description of this new instrument was based on that previous description. This was possible using the *Open* and *Save as...* options provided in the instrument editor menus.

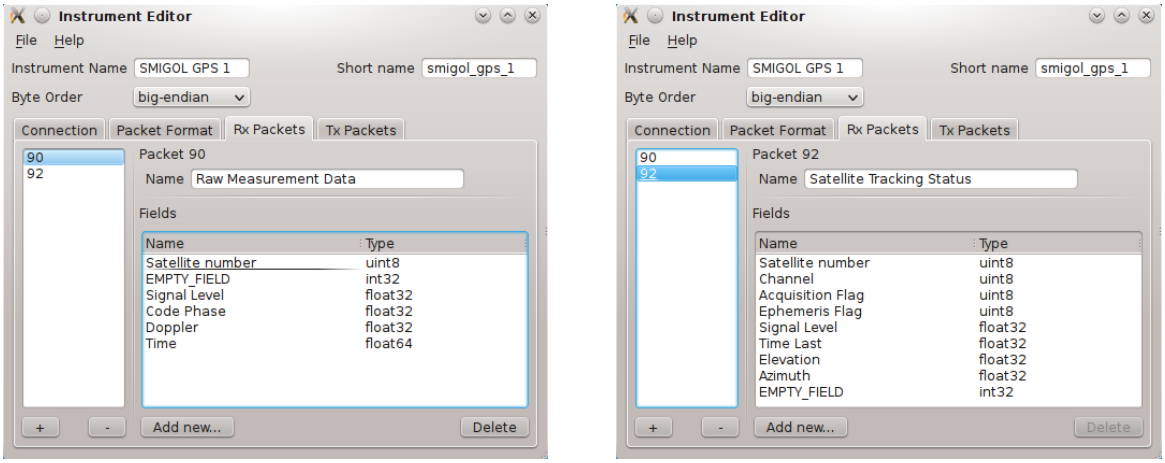
Figure 4.22 shows the instrument editor GUI for the first GPS instrument description of the W-SMIGOL. As it can be observed in Figure 4.22a, the connection type is set to



(a) Connection settings (b) Packets format

Figure 4.22: W-SMIGOL instrument description

TCP with the port set to 20001. The following instruments in the equipment are assigned consecutive ports: 20002, 20003 and 20004. In Figure 4.22b the format of TSIP packets is described, which consists in a start mark (byte 0x10), followed by the packet number and fields, and completed with an end mark (bytes 0x10 and 0x03). Finally, two TSIP packets are described with their fields in Figure 4.23.



(a) *Raw Measurement Data* TSIP packet (b) *Satellite Tracking Status* TSIP packet

Figure 4.23: W-SMIGOL instrument packets description

4.2.5 Equipment configuration

After creating the description for each instrument, the equipment description could be generated using these descriptions and the equipment editor GUI. In this occasion this was very simple, as it only consisted in loading each instrument description. No initialization commands were needed, as no control was required, and the operation mode was left as the

default *Periodic Commands* with no command defined, as the system should just listen for packets, not sending any command. Figure 4.24 shows a snapshot of the GUI with the four instruments loaded and ready to be saved to the in-disk JSON representation of the equipment.

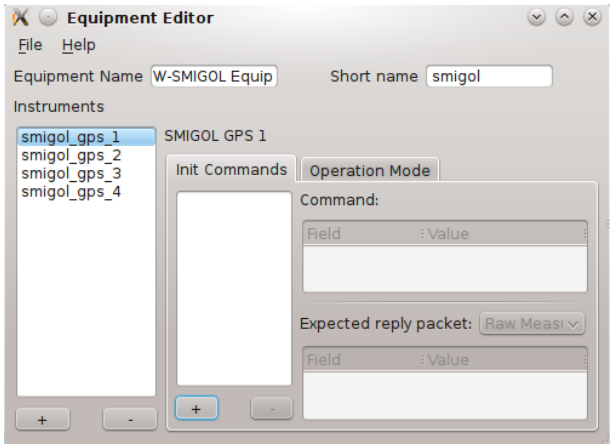


Figure 4.24: Equipment editor GUI with the W-SMIGOL description

4.2.6 Testing the system

In order to check that the newly implemented wrapper, GDAIS-core modifications and the control script worked as expected, an experiment with the W-SMIGOL was carried on. When the test was run W-SMIGOL did not have the XBee connection implemented, so the device was connected directly to the computer, skipping the wireless link. This did not affect the experiment results, as the connection to the computer was still a serial connection with the same settings as the expected by W-SMIGOL wrapper.

In the control script, the location of the equipment description file was introduced, which would be latter used when starting GDAIS-core process to provide it as a parameter. Once everything was set up the control script was executed. After some tries and error fixes, the system worked as expected, providing an output file in HDF5 format with the data stored in the W-SMIGOL data loggers.

Figure 4.25 shows a snapshot of a HDF5 viewer application, *HDFView*, showing the contents of the generated data file. As it can be observed at the left part of the image, for each instrument a folder appears with two items, one for each TSIP packet that was previously configured in the instrument description. On the right side some of the packets of both types received from the first data logger are shown. As it can be observed, each field is identified and displayed in different columns. Each row corresponds to a measurement received and parsed by GDAIS-core. Moreover, the types of each field are

correctly identified. For example, in the *Satellite Tracking Status* table view the *channel* is represented as an integer value and the *signal level* is shown as a floating point value. The *time* and *timestamp* fields are not displayed correctly as the format is not recognized by this viewer, so they are displayed as if they were floating point values.

Finally, a test was done to check that the system operated as expected with a longer file, thus requiring a longer download time and processing by GDAIS-core. For this test, the first data logger SD card was filled with a 6.2 MiB file before starting the control script. Then the download was executed and another HDF5 file was generated, downloading data for 1 hour and 53 minutes, with a size of 5.0 MiB. As it can be observed from this values, download speed was quite slow. This was determined by the serial connection of W-SMIGOL, which is set to 9,600 bps. Moreover, the size of the resulting file was smaller than the raw GPS data file inserted in the SD card. This was the result of parsing the TSIP data and storing only the relevant values, discarding all the other non-relevant bytes, like the packet start and end marks, and also some non-identified packets that had not been added to the instrument description with other information from the GPS receiver.

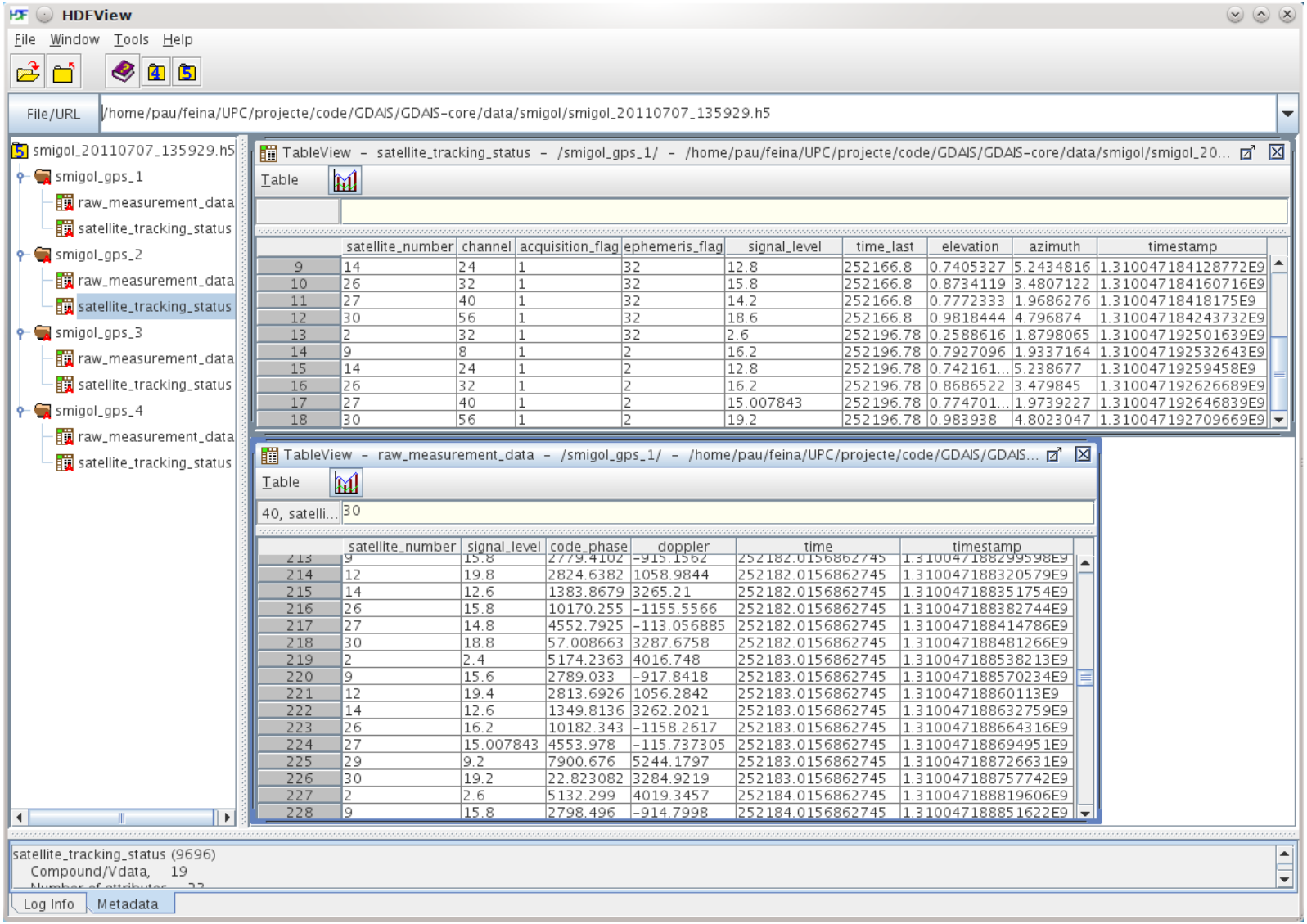


Figure 4.25: Visualization of W-SMIGOL acquisition results in HDFView HDF5 viewer

4.2.7 Conclusions

Using GDAIS system to acquire data from W-SMIGOL instrument provided several interesting results for this project.

Firstly, all the preparation of GDAIS system to adapt to this equipment, which supposed a use case of the system not considered in its initial design, was an interesting check of the flexibility of GDAIS. The chosen solution to adapt to this new situation, the W-SMIGOL wrapper, was quite simple to implement using the same technologies as the ones used in GDAIS implementation.

Moreover, adding the new TCP connection type required for the wrapper did not suppose any major difficulty once all the places where the connection information was implemented were located. No modification was required to the other classes of the system, proving that the implementation of each part of the system is decoupled from the others. Also, thanks to GDAIS-core external interface and the CLI control scripts, the automation of the whole download process was possible, which included executing GDAIS-core and W-SMIGOL wrapper, controlling them, and stopping the processes when the download was completed.

Secondly, the experimentation with the produced system was successful and proved that the solution worked as expected. Data was correctly downloaded and stored in a file using the HDF5 binary format. Analyzing the generated files showed that the obtained file size was smaller than storing raw data, even though all the information was classified and values had the correct format.

Furthermore, some more general conclusions on the acquisition features of GDAIS were obtained. As the capacity of converting raw data not coming directly from a running instrument, but previously stored in a memory card, was proved to work as expected; using GDAIS to convert any raw data file to the HDF5 format should be possible, with just describing the format of the binary packets contained in it. Moreover, this could also be used to merge the information from different already acquired files into a standard HDF5 file for a better handling of it.

Another interesting conclusion is that, in some applications that provide several different binary packets, if only the desired packets are added to the instrument description file, GDAIS-core automatically discards the non-identified packets, resulting in a smaller data file with only the interesting information for the user.

Finally, it should be noted that implementing a similar solution from the ground that interfaced with W-SMIGOL equipment and produced a HDF5 file will all the measurements, would have required a much higher effort if it was implemented from the ground,

not using the data acquisition features provided by GDAIS. Therefore, the main project goal of easing the development of such system for equipment developers has been achieved.

Chapter 5

Conclusions and future work

This final chapter reviews the whole project achieved objectives and conclusions once completed. Also, some future work lines are presented based on the open continuation points detected during the project development.

5.1 Conclusions

This work has succeeded in designing a reusable and versatile system that is able to control and acquire data from multiple instruments, both locally and remotely, through a website that provides a controlling interface and also allows downloading acquired data. In addition, two GUI applications have been developed to help describing measurement instruments and equipments, which simplify this process and provide an enjoyable user experience.

The main objective of this project was developing a generic system for instrument control and data acquisition from remote sensing equipments, which could be used to implement the control and acquisition parts of future or present devices developed at the RSLab, helping the equipment designer concentrate on the hardware part of the system.

Chapter 2 provided a deep analysis of this problem, starting from the definition of the most important concepts; continuing through the more specific characteristics of already designed systems, analyzing three specific instruments developed at the RSLab; and finally evaluating some existing solutions designed to solve this problem or similar ones. At the end, it was concluded that none of the existing solution completely fulfilled all the requirements previously found.

Therefore, Chapter 3 concentrated on the development of a new system from scratch to

perfectly satisfy the requirements of this project. To assure that the resulting system had the expected quality, a formal software development model was adopted. This meant defining in detail the system requirements and functional specification. From this description, a first simple prototype was implemented to test the feasibility of the solution and some technologies. With the experience obtained from this first development, the final design of the system was started. This involved defining a complex architecture that could adapt to all the requirements and also involved designing each part of this architecture in detail. For this final design process, some technologies had to be selected, since these determined the implementation of each part.

Having completely developed the system, Chapter 4 focused on the testing and validation of it. Two equipments were tested, which included analyzing each equipment in detail, adapting the GDAIS system control and acquisition features with the existing parts of the equipment, and finally running some acquisition sessions and checking that everything worked as expected.

For the first experiment, it was checked that GDAIS was able to acquire measurements at a high rate from two different instruments in parallel and simultaneously. Furthermore, it proved that complex instruments that require an initialization to provide the desired measurements was possible. The most important conclusion of this experiment was that, for the applications GDAIS system was designed for, the control and acquisition from an equipment could be achieved without writing a single line of code, just entering instrument and equipment descriptions through a GUI.

From the second test, complementing conclusions could be extracted. In this occasion the equipment required a workflow operation that was not considered during the initial system design, as the measurements from different instruments came from the same connection. For this reason, a simple wrapper application was developed to multiplex the measurements from each instrument to a different connection with GDAIS. This also included adding a new connection type to the system. All this adaption of GDAIS to correctly integrate with this equipment finally resulted in obtaining a single HDF5 file with all the data from the different instruments.

This last experiment proved that GDAIS is quite flexible and that, with a simple addition, it can adapt to a completely new requirement. Also, the final result of a single binary data file with all the information proved its usefulness in interpreting any protocol and generating structured files with meaningful values and the associated description.

5.2 Future work lines

At the end of this work, some future work lines have been envisaged as possible continuation paths from the current stage.

A promising and useful idea that has not been explored yet is extending the instrument-equipment model to allow an equipment include other equipments as if they were instruments. Using this feature, a network of equipments could be controlled from a central point or base station, concentrating the storage of all the acquired data into a single file at this central station.

Connected to this first proposal, some investigation has already been carried on about the synchronization of such a system. Network Time Protocol (NTP) seems as the right solution to help the control systems of all the equipments work at the same time, as that is what this protocol was designed for.

Some complex commercial instruments, which use standard protocols like VISA, can not be controlled with the current version GDAIS. Therefore, an interaction interface that is able to understand and use this protocol could be added to the already included connectivity options. As VISA library functions provide numeric values instead of the binary data usually obtained from a connection, the recommended approach would be creating a new class that included the features of `Parser` and `Connection` GDAIS classes through calls to the VISA library, directly providing the values to the `Recorder` class. In order to implement this new class inside the GDAIS system, PyVISA[1] Python library could be used, which provides high-level object oriented bindings to the VISA library. Having this feature built into GDAIS system would allow to interact with many commercial instruments that are currently not accessible.

Continuing with protocols, there are some that include an escape character, which is used to differentiate between a byte that is part of the protocol structure and the same byte in the data that is being transmitted. For example, in TSIP protocol, the byte 0x10 is used as a mark of the end of a packet. When the data being transmitted contains this byte, it has to be doubled to prevent that it is recognized as the end mark. In this case, the same byte used to mark the end is used to escape it when it does not have to be interpreted as the end mark. The current version of GDAIS does not implement this mechanism, so it may think some packets are wrong because of this doubling not being interpreted correctly. To provide the most coverage to possible protocols, this feature could be added to the system.

Another possible work line is adding a mechanism to limit the size of the acquired data

[1] PyVISA http://pyvisa.sourceforge.net/

files, creating a new file when this limit is reached. This would be useful in long-running remote applications where no human interaction is possible. In this situation, if any error occurs in the file, all data might be lost. However, if the file is truncated into smaller files, only a single file would be lost. Therefore, the loss of information would be much less serious.

Finally, a continuation path that should definitely be explored is running GDAIS system on an embedded system. From the information collected during the project development, this should be possible without changing much of the implementation of the system. The advantages of running GDAIS in an embedded computer would be enormous, as this device could be placed next to the equipment it is controlling and would provide a remote interface to the system without having to add this functionality directly to the equipment instruments.

The whole RSLab team and myself invite anyone who desires to continue extending this development and adding new features to the project, to do so under the GNU GPL or other compatible license. The source code developed in this work is available at `http://gdais.googlecode.com/`, from where it can be freely downloaded under this same license terms.

Bibliography

[1] M. Mussaif Pradas, "Diseño e implementación de la interfaz de usuario y controlador de un radiómetro con beamforming digital," Master's thesis, ETSETB, December 2007.

[2] X. Bosch i Lluis, "On the design of microwave radiometers with digital beamforming and polarization synthesis for earth observation," PhD thesis, UPC–RSLab, April 2011.

[3] "IEEE standard 488-1975," *Acoustics, Speech, and Signal Processing Newsletter, IEEE*, vol. 33, p. 3, May 1975.

[4] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *Computer Graphics and Applications, IEEE*, vol. 10, no. 4, pp. 76–82, 1990.

[5] D. C. Wells, E. W. Greisen, and R. H. Harten, "FITS - a Flexible Image Transport System," *Astronomy and Astrophysics Supplement Series*, vol. 44, p. 363, June 1981.

[6] E. Valencia i Domènech, "Implementació en FPGA d'una unitat de correladors i estimadors de potència per a un radiòmetre de síntesi d'obertura," Master's thesis, ETSETB, March 2007.

[7] I. Ramos-Perez, X. Bosch-Lluis, A. Camps, E. Valencia, J. Marchan-Hernandez, N. Rodriguez-Alvarez, and F. Canales-Contador, "Preliminary results of the Passive Advanced Unit Synthetic Aperture (PAU-SA)," in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium*, (Cape Town, South Africa), 2009.

[8] D. Albiol Garcia, "Contribucions al disseny i caracterització del reflectòmetre de l'instrument PAU," Master's thesis, ETSETB, 2007.

[9] E. Valencia, A. Camps, J. Marchan-Hernandez, X. Bosch-Lluis, N. Rodriguez-Alvarez, and I. Ramos-Perez, "GPS Reflectoemeter Instrument for PAU (griPAU): Advanced Performance Real Time Delay Doppler Map Receiver," in *2nd Workshop*

on Advanced RF Sensors and Remote Sensing Instruments, (ESTEC, Noordwijk, The Netherlands), November 2009.

[10] National Aeronautics and Space Administration, "Open Source Software Agreement." http://ti.arc.nasa.gov/opensource/nosa/.

[11] Free Software Foundation, "Various licenses and comments about them. NASA Open Source Agreement." http://www.gnu.org/licenses/license-list.html#NASA. Retrieved June 17, 2009.

[12] T. J. Ames, K. B. Sall, and C. E. Warsaw, "NASA's instrument control markup language (ICML)," in *ASP Conference Series* (D. M. Mehringer, R. L. Plante, and D. A. Roberts, eds.), vol. 172, p. 103, Astronomical Society of the Pacific, 1999. Also available as http://www.adass.org/adass/proceedings/adass98/sallkb/.

[13] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[14] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Technical Papers of Western Electronic Show and Convention (WesCon)*, 1970.

[15] B. Boehm, "A spiral model of software development and enhancement," *SIGSOFT Softw. Eng. Notes*, vol. 11, pp. 14–24, August 1986.

[16] R. Acevo, "Sistemas de Teledección Activos y Pasivos Embarcados en Sistemas Aéreos no Tripulados para la Monitorización de la Tierra," PhD thesis, UPC–RSLab, April 2011.

[17] X. Bosch i Lluis, "Acquisition and Processing Software for an Airborne Soil Moisture Mapper L-Band Radiometer," Master's thesis, UPC–RSLab, July 2010.

[18] N. Rodríguez, X. Bosch, A. Camps, M. Vall-llossera, E. Valencia, J. F. Marchán, and I. Ramos, "Soil Moisture Retrieval Using GNSS-R Techniques: Experimental Results Over a Bare Soil Field," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 47, pp. 3616–3624, November 2009.

[19] N. Rodríguez, A. Camps, M. Vall-llossera, X. Bosch, A. Monerris, I. Ramos, E. Valencia, J. F. Marchán, J. Martinez-Fernandez, G. Baroncini-Turricchia, C. Perez-Gutierrez, and N. Sanchez-Martin, "Land Geophysical Parameters Retrieval Using the Interference Pattern GNSS-R Technique," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 49, pp. 71–84, January 2011.

[20] A. Alonso Arroyo, "Diseño, desarrollo y test de una red de sensores inalámbrica para medir humedad," Master's thesis, ETSETB. on going.

Appendix A

GNU General Public License

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable

work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

(d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source

may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work)

from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit

to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF

SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.