The Design and Implementation of a Distributed Data Capture and Processing Framework for Ground Penetrating Radar

Allen Bruce Wallis

A dissertation submitted to the Department of Electrical Engineering, University of Cape Town, in fulfillment of the requirements for the degree of Master of Science in Engineering

Cape Town, February 2001

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author .....

Cape Town

5 February 2001

Abstract

This dissertation describes the development of a distributed data-capture and data-processing framework for use with a network-aware ground penetrating radar. The software that was developed addresses weaknesses in existing data processing software, with the main focus being on the distributed capabilities of the framework. The framework was designed from an object oriented perspective, using the Unified Modeling Language to describe the architecture. The Java programming language was used to implement the design. The Common Object Request Broker Architecture was used as the messaging protocol, however the framework was designed such that an alternative messaging protocol such as Java Remote Method Invocation could also be added at a later stage. The Extensible Markup Language was used for storing data as well as configuration information.

The framework was designed to be modular such that additional functionality could be added later in the form of modules. Three modules were developed, a data-viewer module, a data-persister module and a radar controller module.

Performance tests were completed to measure the maximum number of profiles that could be transmitted per second for two scenarios, one involving a standalone machine, and another involving two networked machines using 10 Mbit ethernet. The highest data rate of 346 was achieved when there were no viewer modules active in the system and no processing being applied. For a more useful scenario involving the inverse Fourier transform and a data viewer, it was found that the highest data rate was measured when the radar server was located on a separate machine to the processing framework. The maximum data rate measured under these circumstances was 177 profiles per second. Since the maximum data rate that the radar hardware can currently support is ten profiles per second, the data rate for the framework is more than sufficient.

Acknowledgements

This dissertation and the project which this dissertation describes was completed within the Radar Remote Sensing Group (RRSG) at the University of Cape Town. I would like to thank Prof. M. R. Inggs for the opportunity of working within the RRSG, and for the facilities that were provided.

I would also like to thank Alan Langman for the technical supervision, for the encouragement and for the constant flow of new ideas.

I wish to acknowledge my parents' support throughout my University career, for giving me free reign in my choice of degree and place of study while offering constant financial support.

And to Taryn for her constant love and understanding, for being the motivating force in my move to Cape Town and hence subsequent enrollment at the University of Cape Town.

Contents

List of Figures

List of Tables

Nomenclature

CORBA Common Object Request Broker Architecture

IDL Interface Definition Language

API Application Programmers Interface

RMI Remote Method Invocation

GPR Ground Penetrating Radar

SFCW Stepped Frequency Continuous Wave

UML Unified Modeling Language

OMG Object Management Group

XML Extensible Markup Language

GUI Graphical User Interface

VM Virtual Machine

Chapter 1

Introduction

1.1 Summary

This document describes the development of a distributed, data-capture and data-processing software framework for a network-aware ground penetrating radar (GPR). The system acquires data from the GPR and pipes the data through a web of interconnected nodes. Nodes can be distributed across the network on co-operating computers, and each computer can host any number of individual nodes. Data is processed as it arrives at each node in the web. The system is designed to be highly configurable so that only the necessary components need to be installed on any particular machine.

The system is written using the Java programming language. The Common Object Request Broker Architecture (CORBA) is used to distribute components across the network, as well as to provide connectivity between the framework and the GPR.

The system addresses four key areas:

- Data capture
- Data processing
- Data display
- Data persistence

A modular system was designed, containing modules that accomplished these tasks which can then be combined at run-time to satisfy specific user requirements for varying situations.

Benchmarking tests were performed in order to obtain estimates for the performance of the system under various scenarios. The data transfer rate measured for the most common scenario involving two computers, one behaving as a radar server and the other as a data processing and presentation machine, was 134 data profiles per second. This offers an estimation for the fastest rate at which

the software can accept data. Currently the radar hardware can only produce data at a rate of 10 data profiles per second, and hence the data rate measured for the scenario indicated is acceptable.

The next section discusses the history of the GPR systems that are involved in this project. An understanding of the history behind the current radar system is useful in order to understand the development of the requirements specification that was developed for this framework. Finally, this chapter ends with an outline to the rest of the document.

1.2 Background

1.2.1 History and Motivation

The Radar Remote Sensing Group (RRSG) at the University of Cape Town has been involved with Ground Penetrating Radar since 1990. The Subsurface Research Group at the RRSG has designed and built a number of Stepped Frequency Continuous Wave (SFCW) GPR systems. The GPR hardware that has been developed captures raw data, which is then passed on to a PC for data processing. For the case of SFCW GPR data, it is generally necessary to perform some form of processing on the raw data before it can be presented to the user. Initially, the radar hardware was connected to a PC via the serial port. An application was written (in C++) that allowed radar users to control the data acquisition process from the PC, and provided the required data processing and data presentation. The application provided a Graphical User Interface (GUI) to the user, while handling the serial communications to the radar hardware. This application was called MDR.

As the radar hardware was developed further, it was decided that the radar hardware be designed to be network-aware. This would allow the radar itself to be attached to a network, and hence be reachable from anywhere else on the network. The long term goal in this exercise would be to have the radar hardware running an operating system on which a radar server could run. This radar server would provide all the low level communications with the rest of the radar hardware, while presenting one or more high level software interfaces to the rest of the network. The radar would then be distributed in a client-server paradigm. Client applications would be responsible for data processing and display.

It was therefore necessary to modify the existing MDR application, since under the new paradigm this application was performing both server-side functions (serial communications) as well as client side functions (data display and processing). A radar server was written that would take over the serial communications function of MDR, however it would not provide any processing functionality. Instead, it would provide a software interface to which a client application could connect. It was decided to use the CORBA standard, which would hence allow client applications written in any programming language to still connect to the server. A Java version of MDR was then written which offered data processing and data display functionality, but connected to the radar

server via the CORBA interface. Java was chosen for the client-side application since it is a platform independent language.

The Java version of MDR (known as JMDR) was frequently modified to accommodate new requirements until it offered a range of processing options along with three different modes of data-capture. The process of writing JMDR, and then maintaining it and revising it highlighted some of the problems with its design, and provided a useful means of developing a set of user requirements for a replacement application.

A client-side system was proposed that would replace the existing JMDR client software. The system would be responsible for data acquisition, data processing, data persistence and data presentation. It was proposed that this client side system should satisfy the following two design goals:

1. The system should be modular so that modules could link together at run-time, providing a particular user with a functionality tailored to his requirements.
2. The system should be distributed. Modules should be able to run on various hosts simultaneously, co-operating in the task of data capture and processing. This would take full advantage of the fact that the radar itself is network-aware, allowing sharing of data, and consequently sharing of knowledge.

The scope of this dissertation is limited to the design and implementation of this client-side system.

1.3 Thesis Outline

The rest of this dissertation shall focus on the design and implementation of the proposed client-side system. As far as possible, this dissertation uses the Unified Modeling Language (UML) as a standardised means for describing the design of the system.

A review of the technologies and methods used in the dissertation is given in chapter 2. These technologies include CORBA, Java, XML and the UML. Chapter 2 serves only to give a brief overview of each technology, the reader is referred to appropriate texts if a more thorough understanding is required.

The system requirements are developed in chapter 3. This chapter firstly reviews existing GPR software, and then gives a brief summary of the processing requirements for stepped frequency GPR data. A requirements specification is then developed, and from this, a set of use cases are developed along with the associated use case diagram.

A conceptual analysis of the system is developed in chapter 4. This results in a conceptual model for the system being developed which helps introduce and define terms that are used in the subsequent design.

Chapter 5 describes the specification model for the system, which is basically a definition of the interfaces that the system should implement, as well as a description of how these interfaces should co-operate.

The implementation model is presented in chapter 6. This chapter describes how the system was implemented, using UML class diagrams to document the components of the system.

Chapter 7 describes the results that were obtained. Firstly, the results of the implementation of the design are discussed. Secondly. the results of a couple of performance tests are discussed in which the data rate was measured under two different scenarios.

Chapter 8 is the concluding chapter, and it includes a section describing future work that could be completed for the system.

Chapter 2

Review of Technologies and Methods Used

This chapter introduces the technologies that are used to construct the system. The purpose of this chapter is to familiarise the reader with the technologies that were used in the system, before they are encountered in later chapters. It should be noted that the order in which the chapters of this document occur does not directly follow the order in which the system was designed and constructed, and hence this chapter should not be seen as part of the design process. In particular, the technologies mentioned in this chapter were selected for their applicability to the design, the design was not developed to suit the technologies specifically.

The core technologies are discussed initially. They are CORBA and the Java programming language. The Extensible Markup Language (XML) is then introduced. A discussion of the Unified Modeling Language (UML) is also included in this chapter, and following that is a discussion of the software development method that was applied.

2.1 Common Object Request Broker Architecture (CORBA)

CORBA is an industry wide standard for communication in a distributed object-oriented environment. CORBA allows two systems written for different operating systems in different programming languages to successfully communicate with each other. The CORBA standard was created by the OMG, however the OMG leaves the implementation of the standard up to various vendors. Thus, many vendors can provide CORBA implementations, however these different implementations must all be able to intercommunicate for them to adhere to the CORBA standard. This frees developers from getting locked into a particular vendor's product. There are plenty of books written on CORBA, a good introduction to CORBA can be found in the book by Orfali and Harkey [12] which introduces CORBA from a Java programmer's perspective. However, perhaps the most up to date information is available directly from the

CORBA web site: `http://www.corba.org` and from the OMG's home page: `http://www.omg.org`.

CORBA offers an object-oriented approach to distributed system design. Using OMG's Interface Definition Language (IDL), one describes an interface for an object. This interface is completely implementation independent. The IDL definition is then compiled to a particular implementing language, and an implementation of the interface is written in that target language. The implemented object is then able to publish its interface on the network, and other applications implemented in different languages will be able to communicate with this object via the interface described by the IDL. It should be noted that although CORBA is object-oriented by design, it is quite simple to write a wrapper for legacy systems, and have the wrapper implement an IDL interface thereby incorporating the legacy system into a larger, distributed, object-oriented system.

For these reasons, it was decided that CORBA should be considered for the middleware in designing the data-acquisition and processing system.

2.2 The Java Programming Language

The Java programming language was developed by Sun Microsystems and announced to the public in 1995 [9]. The goal of the Java language is to be completely platform independent. That is, programs written in Java and compiled under one operating system, should run without the need to recompile under another operating system. Platform independence is achieved through the use of a Virtual Machine (VM) that is written for a particular operating system. This VM then interprets the byte codes that a Java program is compiled down to. This means that a Java program can only be run on an operating system for which there is a VM described. There are however a large number of VMs available for existing operating systems. Both Windows and Linux are supported by Sun Microsystems who provides development kits for both these platforms. The development kit includes the VM along with development tools such as compilers. While portability is dependent on the existence of a VM for a particular platform, Java still offers the most comprehensive solution to the problem of code portability. As long as a Java application has been written from the start to take advantage of Java's platform independent features, and it is run on a VM that adheres to the standard Java Application Programmers Interface (API), then the application will be platform independent.

Java's platform independence holds many more consequences that are not immediately apparent. Firstly, an extensive Graphical User Interface library is provided that is available on every platform. This is a huge step from C/C++, where one can easily get locked into using some vendor's GUI toolkit under Windows, and then have to rewrite the GUI completely when the application is ported to Linux. Java also provides an extensive networking package that is distributed as a core part of every VM. Again, this standardises the way of writing applications that communicate via the network. A developer does not have to search for libraries for Linux so that he can port his Windows application. For more information on features that are part of the Java VM, the Java home page is the most up to date location: `http://www.java.sun.com`.

Java's platform independence combined with CORBA's language independence make a powerful combination. These two technologies were selected to be used in the solution to the distributed data acquisition and processing problem for the following two reasons:

1. Java frees both the user and the developer from a particular operating system.
2. CORBA frees the developer from a particular language.

The second point might seem a bit obscure at first, especially since Java has been selected as the programming language to be used. It should be remembered however that the system discussed in this dissertation is simply the client side of a larger system. By using CORBA to describe the interface to the radar server, it will be possible to completely rewrite the implementation of the server in a new programming language without having to alter anything in the client system. For the case of the GPR described in chapter 1, the radar server was required to manipulate platform dependent hardware, and hence Java was not an appropriate language for the server. However, since CORBA was being used as the middleware, it was still possible to write the client data capture and display software (JMDR) in Java.

2.3 Extensible Markup Language (XML)

XML is a markup language very similar in structure to HTML. HTML is actually viewed as a subset of the more general XML. XML is simply a standard format for textual data. XML is more accurately described as a metalanguage - a language that can be used to define new markup languages. XML by itself does not achieve anything of value - it is not possible to develop a program using only XML. The power of XML is in what a developer chooses to apply it to. One of the main advantages of XML is that it is a platform independent format, which allows programs developed for different operating systems to use the same data format. There would be no need for data formats to be converted between operating systems. HTML is an example of how useful this platform independence can be. Web servers can run under Unix/Linux, while serving HTML pages to Microsoft Windows clients as well as to Apple Macintosh clients.

A good reference describing XML and how Java applications can best use XML is the book by Brett McLaughlin [11]. There is also plenty of information to be found on the internet, a good starting point for this is the following website: `http://www.xml.org`.

2.4 The Unified Modeling Language (UML)

As was mentioned in chapter 1, this dissertation shall rely on the use of the UML to document the design process. The UML was developed by Booch, Rumbaugh and Jacobson, and was standardised by the Object Management Group (OMG)

in 1997 [1, pg 4]. It is an object-oriented modeling language, and as such does not specify or promote a particular software development process, it is simply used to describe a system in the same way an architect's blueprint describes a building.

The UML was selected as the modeling language of choice since it has been standardised, and is programming-language independent. The rest of this dissertation is written under the assumption that the reader is familiar with the basics of the UML notation. The UML is mostly a graphical language, and as such much of it can be understood without a thorough understanding of the UML. For a concise and informative introduction to the UML, the reader is advised to read Martin Fowler's *UML Distilled* [1].

2.5 The Software Development Method

A software development method describes the steps taken during the development of a software project. A software development method usually consists of a process and a notation [1, pg 1]. The notation is a graphical language for describing concepts used in the process. Many books have been written on development methods. The OMG developed a book [4] that summarises 21 methods of software development. Jacobson [2], Booch [6] and Rumbaugh [3] have all authored books covering this subject. With the bewildering choice of software development methods available, it is difficult to select one that is applicable. It is further proposed by Fowler [1, pg 14] that development teams develop their own methods tailored to their particular needs. Therefore, in developing the system described in this dissertation, there was no strict adherence to any particular documented development method. Rather, ideas and concepts were taken from the documented methods and used to solve problems encountered during the design process. Since the development of this system was conducted by a very small group consisting of two people, the application of some method intended for a larger group of developers would have been inappropriate.

The many methods that have been proposed over the years each offer an accompanying notation. Since the notation is important for communication, a move was made to develop a standardised notation, and the result was the UML. The UML therefore does not specify a particular process. The UML can be used successfully with any development process, and in fact should be used since it enables better communication between developers. For this reason, the UML has been used in the development of the system described in this dissertation.

The development process that was adopted is based loosely on a process described by Fowler [1, pg 13]. The process described by Fowler is an iterative process. It is generally accepted that the development process must be an iterative process. Rumbaugh [3, pg 166] makes the following statement: "The entire software development process is one of continual iteration; different parts of a model are often at different stages of completion". Further, Fowler's process is incremental. The construction phase of the process is characterised by a number of iterations, with each iteration ending in the release of production-quality software that satisfies a subset of the project requirements. This process of satisfying a subset of requirements by the end of each iteration is also found in

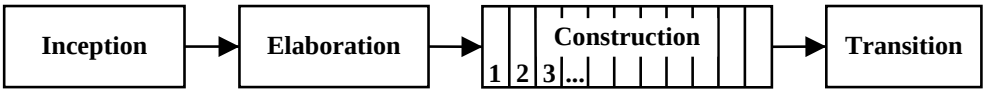

Figure 2.1: Outline Development Process

the Extreme Programming (XP) method developed by Kent Beck [5]. Fowler's process is summarised in Figure 2.1.

Figure 2.1 shows the construction phase as a series of steps, where each step is an iteration resulting in a production quality solution to a subset of the user requirements. Although the figure does not explicitly indicate iterations in the other phases, this does not preclude any iterations occurring during the inception and elaboration phase. In fact, there will most likely be iterations during the elaboration phase which is where the user requirements are developed.

2.6 Conclusion

For the benefit of the reader, this chapter introduced the various technologies that have been applied in the construction of the system. However, the final decision on which technologies would be suitable for the design was not taken before considerable thought was given to the system. It is only in retrospect that this chapter is able to introduce these technologies up front, and indicate reasons for the choice of technology.

The Java programming language was selected as the programming language in which to develop the entire system. Java was chosen since it is platform independent. Note that the system being developed is effectively the client-side of a larger system which includes the radar server and radar hardware.

CORBA was selected as the middleware for the system. The main reason for using CORBA is its language independence. This feature is useful if portions of the system need to be implemented in a different programming language. This was particularly convenient for making the connection to the radar server which already supported a CORBA interface.

XML was selected as a platform and language independent format in which to store data and configuration information.

The UML was used to document the design of the system. The UML was selected since it is a standard modeling language for describing object oriented systems.

The software development process that was followed was loosely based on a process described by Martin Fowler [1].

Chapter 3

Requirements Analysis

This chapter develops the system requirements. The driving force in developing the requirements for this system was the experience obtained while developing and maintaining the JMDR application. As JMDR was developed, more requirements were requested by the end users, and other requirements were identified by the developer. However, JMDR was originally developed as a Java version of MDR, and both applications operated in a similar way to many other existing GPR data processing applications. Therefore, this chapter begins with a review of existing GPR data processing software. The aim of this review is to highlight some of the requirements that this system shall attempt to address that other applications (including JMDR) have not attempted to address. This review is completed with a look at the limitations of these systems.

Section 3.2 then offers an overview of the processing requirements for Stepped Frequency GPR (SFGPR). SFGPR requires more processing than conventional GPR and this extra processing must be reflected in the user requirements.

The sections described above are then used to formulate a requirements specification which is given in section 3.3. This is a formalisation of the ideas and goals introduced in sections 3.1 and 3.2.

Finally, the UML is used to document the use cases for the system as a whole. The actors for the system are identified along with the use cases which summarise how the actors interact with the system.

3.1 Review of Existing Software

3.1.1 Overview of available software

There are generally two types of software that can be found for GPR data processing.

1. GPR system specific software: Commercial GPR system providers offer proprietary software with their GPR systems. This software provides data

acquisition functionality, and usually offers some data processing functionality as well.

2. Generic processing software: Software packages are available that offer various processing options that can be applied to a number of data types. This type of software would not offer data acquisition options, but allows users to apply post-processing to data sets that have already been acquired.

This section will discuss existing software packages from both categories. No attempt is made to review any of the software products listed below since most of the products are commercial and hence are unavailable for testing. The information gathered in this section is simply information made available by the various organisations on their respective websites.

The following organisations offer software of the type mentioned in item 1 above. They offer complete GPR systems, radar hardware as well as supporting software:

Sensors And Software

Sensors and Software is a commercial company focussed on providing various systems for GPR. Their GPR systems at present are all based on the pulsed-radar technique. The software that comes with the radar system is proprietary, and offers data acquisition along with various processing options.

`http://www.sensoft.oc.ca`

Ingegneira dei Sistemi S.p.A (IDS)

IDS is a commercial company that manufacture and distribute their own GPR system. There are three software applications that are offered with their radar, a data acquisition application, and on-site data processing application, and a powerful off-site data elaboration application.

All packages run under Windows, with each application supplying the following functionality:

1. IDSGrasWin: Multi-channel data capture; IDS Georadar offers support for multichannel data acquisition.
2. IDSGresWin: On-site data visualization and basic processing.
3. IDSGred V.4.0: Off-site data elaboration and generation of cartographic outputs.

IDSGred is a powerful data processing application that requires 2 monitors and a database in order to function. This is why IDS specify IDSGred as an off-site application, while the simpler IDSGresWin application is specified as an on-site application.

`http://www.ids-spa.it`

Geophysical Survey Systems, Inc. (GSSI)

GSSI is a commercial company focusing on the development and manufacture of GPR systems as well as electromagnetic induction instruments. They market a variety of GPR systems with various features, however they all follow a similar trend in terms of data acquisition and processing. The radar hardware for each GPR includes a portable monitor on which the data can be viewed as it is captured. The data can then be transferred to a PC where it can be processed under their proprietary post-processing software. The package that GSSI has developed for this purpose is called RADAN. It operates under Windows, and offers various processing options such as noise filters and gain functions. GSSI also supply add-on modules that are designed for specific applications such as Bridge Assessment and Road Structure Assessment.

`http://www.geophysical.com/`

The following organisations supply software only:

Roadscanners

Roadscanners is a commercial company specialised in condition surveys and rehabilitation design of roads, bridges, airports and railways. They focus on the use of GPR to facilitate these activities, and have developed various software applications to support their consulting activities. Their most powerful tool is named Road Doctor. Road Doctor runs under Windows, and provides a standard Windows GUI. Apart from plotting the actual GPR data in a variety of ways, it also provides the ability to play back videos of the actual data acquisition process. It can also show maps of the site under inspection. The processing tools available with Road Doctor are geared toward road condition analysis.

`http://www.roadscanners.com`

Interpex

Interpex is a commercial software company that produces software for the processing, interpretation and display of geophysical data. They offer two software products developed particularly for GPR data, GPR IXeTerra and GRADIX. IXeTerra runs under Windows while GRADIX runs under DOS. Both products are used for post-processing, in that data is first captured from a radar and stored on disk, and is then later imported into the programs to be processed. Both products are able to import data in various formats including the formats produced by GSSI, Sensors and Software and Mala Geoscience radars.

`http://www.interpex.com/radar.htm`

Center for Wave Phenomena, Colorado School of Mines (CWP)

CWP maintain a package called Seismic Unix (SU) that is an instant seismic processing and research environment. SU can be compiled to run under any

Unix system, and is offered free of charge. SU consists of a number of small utilities that accomplish very specific tasks. In order to accomplish a more complex task, the utilities are piped together in Unix fashion. This means that while the name of the package implies that it offers only seismic processing, the individual processing utilities might be combined to accomplish a range of wave-related processing. SU is a departure from the type of software discussed up to this point in that it does not offer a central GUI from which to process and view data. Rather, the individual utilities form extensions to the Unix operating system, and are piped together on the command-line, or in a script, to accomplish a particular task.

`http://www.cwp.mines.edu/cwpcodes/index.html`

Sandmeier Scientific Software

Scientific Software maintains Reflexw, a data processing and presentation program for various types of wave data. Reflexw is modular in design, and GPR specific modules can be incorporated that provide useful GPR-specific processing options. Reflexw imports data from various formats, including formats produced by GSSI and Sensors and Software radars. As with Interpex software, Reflexw is used for post-processing of data that has already been acquired.

`http://www.ka.shuttle.de/software/index.html`

Parallel Geoscience Corporation

Parallel Geoscience Corporation is a commercial company focusing on developing geophysical data analysis software tools for the field and desktop computing environments. They have developed the Seismic Processing Workshop (SPW) which is a family of interactive seismic data processing packages. They offer a specialised package developed for GPR data processing. This package is able to import data from GSSI, Sensors and Software, Mala and Koden radars.

3.1.2 Limitations

As was described in chapter 1, an application called JMDR was developed that loosely resembled many of the GPR data processing applications described above. The process of firstly writing the application, and then maintaining it through various feature requests from the end users highlighted the various limitations inherent in many GPR data processing applications. Some of these limitations are discussed in the points that follow.

Graphical User Interface dependence

JMDR was written around its Graphical User Interface (GUI). This in itself is a limitation since this is a poor design technique. However, even though other GPR applications might not be as tightly coupled to their GUI as JMDR is, their GUI is still the main focus of the application. For applications that

might be operated in the field in order to provide real time processing, this can be limiting since display facilities in the field are often quite limited. If no processing at all is required in the field, then it is still necessary to have a display simply to show the GUI to the user so that the data acquisition process can be controlled.

Complicated functionality

Many of the GPR applications, JMDR included, offer quite a variety of tasks that can be accomplished. The tasks range from acquiring data from the radar, processing the data, viewing the data, and saving the data to some storage device. Each task requires its own set of GUI controls. A recurring problem while maintaining JMDR was the organisation of the GUI to best group these controls. Whether they are concentrated in one window, or whether they are spread between multiple dialogs, the problem still remains that there are many GUI controls available for the user to try and understand. The more functionality an application offers, the more complicated the GUI ends up being, and this can result in new users requiring extra time and training to become acquainted with the application, even if the new user is not going to be using the extra functionality. The solution to this problem is to cut out all of the GUI controls that are not necessary for a particular task. The user is then not confronted with a bewildering array of buttons and option boxes when all he wants to do is capture some data from the radar.

Limited data sharing

JMDR and other similar applications do not offer much opportunity to distribute data to other applications and other users. For the most part, data is acquired directly from the radar, it is processed by the application, and saved to disk. If another application requires the data, then this data is imported from file that it was written to. There is no option for another application to obtain the data directly from the GPR application. Furthermore, data obtained on-site cannot be shared immediately with users off-site. In the case where an expert in GPR data interpretation is unavailable on-site, the data has to be transferred to a place where the expert can view it and offer advice in its interpretation. It would be more convenient if the expert user were able to view the data as soon as it is acquired from the radar. One GPR expert could then oversee the data interpretation occurring at multiple geographically remote locations. This would only be possible given that there is a suitable communications link to the site.

3.2 Overview of Stepped Frequency Continuous Wave GPR Processing Requirements

This section offers a brief overview of the processing requirements for Stepped Frequency Continuous Wave (SFCW) signals. The processing requirements for

SFCW GPR data differ slightly from pulsed GPR, and consequently affect the user requirements of the system being designed. Since most of the GPR software mentioned in section 3.1 is written for pulsed GPR, this section aims to clarify the specific data processing requirements that the system will have to address.

Mensa [16] and Wehner [15] both describe the use of Stepped Frequency Continuous Wave (SFCW) signals for conventional radar. They cover the mathematics required for applying this technique. Noon [13] and Farquharson [14] describe SFCW applied to ground penetrating radar. These texts develop the mathematics applied to the GPR context. This section will simply discuss the results in order to familiarise the reader with some of the processing requirements for SFCW GPR. This is necessary since the processing requirements impact on the design of the software.

Pulsed GPR systems operate by transmitting a pulse of electromagnetic energy, and measuring the delay between the transmitted pulse and the reception of the reflected pulse. This delay can be converted into a depth measurement if the velocity of light within the medium is known. Pulsed GPR systems hence attempt to measure the impulse response of the medium by approximating the transmission of an impulse waveform.

SFCW radars operate by transmitting a single tone that is stepped through a range of discrete frequencies. At each frequency, the reflected signal's magnitude and phase relative to the transmitted signal is measured. A convenient means of describing the principle behind the SFCW technique would be to say that SFCW GPR attempts to measure the frequency response of the medium by sampling the frequency response at discrete frequencies. This sampled frequency domain representation can then be transformed using Fourier theory into the time domain to achieve a time domain signal which is therefore an approximation to the impulse response of the medium. This impulse response is the same impulse response that conventional pulsed GPR attempts to measure directly.

One of the advantages of the SFCW technique is that a larger bandwidth can be synthesised using the SFCW waveform than can be achieved using a pulsed waveform. A possible disadvantage to the SFCW technique is that raw data obtained directly from the SFCW radar requires processing before it can be presented visually to the user. Since raw data obtained from pulsed GPR radars is already in the time domain, it is generally possible to make sense of the data without any processing.

Thus, in order to visualise SFCW GPR data there is the minimum requirement that the data should be processed using an inverse Fourier transform. Coupled with this Fourier transform is the optional use of a windowing function to limit the sidelobes that are generated during the transform.

Once the inverse Fourier transform has been applied, the SFCW GPR data is essentially time domain data, and there is the potential to apply other data processing routines that are applicable to pulsed GPR data, such as range-gain control and background subtraction.

3.3 Requirements Specification

This section summarises the requirements that have been developed for this system. The set of use cases that are developed explore these requirements in more detail. This section is provided mainly to offer a high level overview of the problems that this system is required to address.

Ivar Jacobson [2] states that two models are developed during the analysis of a problem; the requirements model and the analysis model. It is important that the requirements model be the real base from which to develop the system. In developing the requirements model, little heed should be paid to the implementation environment since this guarantees that the resulting system design is based upon the problem, not upon the conditions prevailing during the implementation.

The requirements specified here have been developed to overcome two problem areas. The first problem area is the general requirements of a GPR data processing application, for example, that the user must be able to process the data, and save the data. The second problem area is the solution of specific problems that have been identified, some of which have been discussed in section 3.1.2.

Related requirements have been grouped together in the following subsections:

General General expectations of the system from a user's point of view.

System specific Requirements that are system specific.

3.3.1 General User Requirements

These requirements reflect the general expectations that a user of any GPR software would have of the system.

Data Acquisition The system shall offer a means by which a user can acquire data directly from the radar hardware.

Data Processing The system shall process the acquired data by means of a set of predefined routines that the system shall provide. These routines must include at minimum a windowing routine, and an inverse Fourier transform routine.

Data Persistence The system shall offer a means of persisting the data. The data along with the processing history shall be persisted together so that at later time the user can import the data and examine the processing that was applied to it.

Data Display The system shall have the ability to offer the user a display of the data under examination. The view of the data should include a pseudo-colour representation of the data.

3.3.2 System Specific Requirements

These requirements are specific to the system being designed, in that they are requirements that stem from particular hardware features, and they are also requirements that are derived from the limitations discussed in section 3.1.2.

Network collaboration The system should be network aware. Users connected to the network should be able to share data within the framework of the system. A user should be able to obtain data directly from other participating users connected to the network, process that data and then pass it on to other users connected to the network..

Remote control A user with the correct authorisation should be able to control the system from anywhere on the network. This includes control over the radar hardware, and over the data distribution process, as well as configuration of any remote host controlled by the user.

Display independence It should be possible for the system to be configured according to the environment in which it will run, so that if there is no display, then the system simply runs without a display. Interaction with the system when there is no display shall then be achieved via the network. For example, a user situated in front of a desktop PC would be able to interact with a GUI front-end that communicates across the network with the system running on a computer in the field lacking a suitable display.

Functionally configurable The system should be configurable to suit the functional requirements of a particular class of user. For example, a user who is only going to be capturing data in the field and has no experience in actual data interpretation should not even be offered the option of performing some complex data processing. The simpler the interface offered to such a user, the quicker such a user is able to master the use of the interface. This configuration should be done once by an experienced user, possibly via a configuration script, so that when the inexperienced user starts the application up the required interface is already operational.

3.4 UML Use Case Analysis

The requirements described above were mapped to UML use cases and illustrated using use case diagrams. A use case is described by Fowler [1] as a typical interaction between a user and a computer system. A use case diagram is a graphical means of relating use cases to actors.

An actor is a role that a user assumes with respect to the system [1]. One user might take on various roles while interacting with the system, depending on what goal he wishes to achieve. A use case diagram is used to indicate how actors interact with the system by showing which use cases each actor uses.

In developing a set of use cases, it is important that the use cases reflect the goals required from the user's point of view, not from the developer's point of view. Fowler [1, pg 44] differentiates between these two points of view by using the

terms *user goal* and *system interaction*. Larman [8] also makes this distinction by identifying use cases as being either *essential* or *real*. An essential use case is a use case that achieves a user goal, hence Fowler's *user goal* is synonymous with Larman's *essential* use case. In particular, Larman states that an *essential* use case remains relatively free of technology and implementation details [8, pg 58]. On the other hand, a real use case "describes the process in terms of its real current design, committed to specific input and output technologies, and so on." [8, pg 59]. Larman's *real* use case is therefore equivalent to Fowler's *system interaction*. Larman's terms shall be used from this point on simply because they can be used more clearly to identify the nature of a particular use case.

In order to design a system that satisfies the end users' requirements, it is clear that one should first develop the essential use cases before defining real use cases. The essential use cases will reflect abstract goals that users might require of any GPR data processing package. The set of essential use cases should therefore offer a summary of the high level capabilities of the system. The essential use cases for the system are developed in section 3.4.2.

The set of real use cases offer insight into how the essential use cases are achieved by the particular application. It is the set of real use cases that will show how this system aims to differ from existing GPR applications. Specific issues that this system aims to address, such as live data distribution across the network, are reflected by the real use cases. Section 3.4.3 develops the set of real use cases for this system.

However, Fowler suggests that in developing the requirements for a system, one should begin by identifying potential actors in the system before describing the potential use cases. Identifying potential actors in a system is the starting point of an iterative process through which the descriptions of actors are explored and the use cases are defined. This iterative nature cannot be reflected in a document which is linear by nature, and so the reader should bear in mind that the definitions given later in this chapter for the actors and the use cases in the system were not achieved in a single step. Section 3.4.1 describes the actors that were identified for the system

3.4.1 Actors

An actor definition describes the role that a user might assume when interacting with the software. The most concise representation that was found consisted of three actors; an Observer, an Operator and a Power User. They are defined such that the Operator is a generalisation of the Power User, and the Observer is a generalisation of the Operator. They are defined more clearly as follows:

Observer An Observer is a user that requires the most basic level of interaction with the system. An observer might be able to view GPR data made available to him/her, process it locally for him/herself and possibly store it locally.

Operator An Operator is an actor that extends the abilities of the Observer actor. This means that an Operator would be able to potentially do anything that an Observer is able to do, along with some added abilities.

The main interaction an Operator would perform with the system would be to control the data acquisition process. The data acquisition process is the actual transfer of data from the radar server into the application itself.

Power User A Power User is an actor or role that an experienced user might assume in order to accomplish a specific task. As an example, an Operator is simply a user given the power to control the data acquisition process. There is no implied skill that the Operator must have. In an extreme case, an Operator might only understand how to start taking a measurement and how to stop taking a measurement. If the radar requires configuration that this Operator is not capable of providing, then an experienced user who understands the radar configuration would interact with the system in the role of a Power User in order to manage the radar configuration. The Power User actor is an extension of the Operator actor, since the Power User is also able to perform any operation that the Operator may perform, which hence also implies that the Power User is able to perform all the operations that the Observer is able to perform.

It can now be seen that the requirements described in section 3.3 that depend on the level of experience of the user can now be described more easily by the use of the three actors defined above. It should be noted that a user would most often interact with the system in the role of an Observer. It is only when more specialised goals are required by the user that the user would assume a different role. This shall be clarified once the use cases have been discussed in the following section.

3.4.2 Essential Use Case Analysis

As was pointed out at the beginning of this section, there are two types of use cases; essential use cases and real use cases. This section shall develop the set of essential use cases required for the system. The essential use cases developed here are then used in the derivation of the real use cases discussed in section 3.4.3.

Section 3.3.1 outlines the general user requirements for this system. These requirements are used to develop the set of essential use cases. The four requirements map directly to four essential use cases as can be seen below. Two more essential use cases were identified that support the other four use cases, but which would not necessarily be identified immediately as user goals. These two use cases are the *Configure* use case, and the *Distribute Data* use case. The essential use cases are therefore:

1. Acquire Data
2. Process Data
3. View Data
4. Persist Data

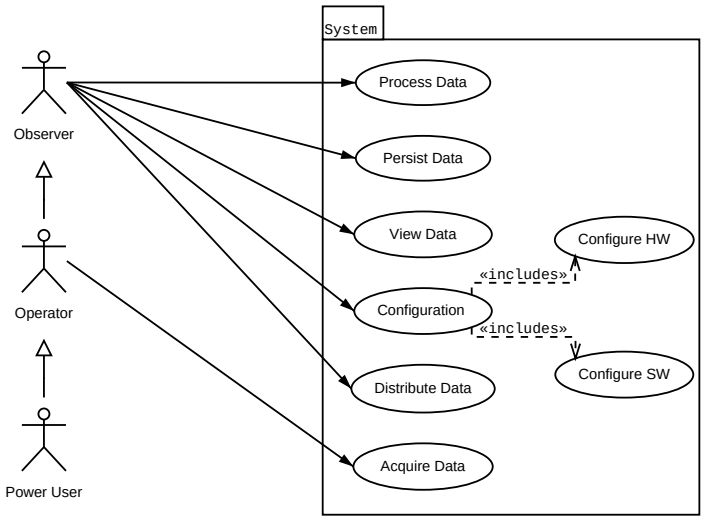

Figure 3.1: Use Case diagram showing actors and the basic functionality they require

5. Configure
6. Distribute Data

This set of use cases represents the end goals that are required by users of the system. Each use case captures a specific goal that a GPR user might wish to achieve. The use cases specified above do not imply the use of any specific technology. As an example, consider use case 4, *Persist Data*; no storage device is specified, and so the use case could be implemented using a simple file, a database or any other means of long term storage.

Table 3.1 describes the set of essential use cases in a format similar to the format used by Larman [8, pg 49].

There are many more actual operations that a user might want to perform, however each of these operations can be related back to one of these six use cases. As an example, the use case *Setup Processor* could be identified. However, this use case can fall under the more general use case of *Configuration*. Hence, these six use cases form the key use cases for the entire system.

In Table 3.1, the *Type* field specifies that each use case is both essential and primary. The value *primary* is used to indicate the priority of the use case. The priority is used in order to determine which use cases should be implemented first in the construction phase. All of the above use cases are prioritised as primary since each use case is required in the final construction.

Figure 3.1 shows the basic use case diagram for the entire system. This diagram shows the six use cases that were specified in the previous section, as well as a further two use cases that are used to illustrate how more specific operations are related back to one of the six primary use cases.

1	Use Case	**Acquire Data**
	Actors	Operator
	Type	primary, essential
	Description	An Operator initiates a measurement session with the radar. The Operator maintains control over data acquisition process until the Operator terminates the measurement session.
2	Use Case	**Process Data**
	Actors	Observer
	Type	primary, essential
	Description	An Observer indicates the type of data processing required. When data is distributed to the Observer, the data is processed according to the Observer's requirement.
3	Use Case	**View Data**
	Actors	Observer
	Type	primary, essential
	Description	An Observer is presented with different views of the data. The Observer has limited control over which views are presented.
4	Use Case	**Persist Data**
	Actors	Observer
	Type	primary, essential
	Description	An Observer indicates the data that should be persisted. This indication can be implied, resulting in automated data persistence.
5	Use Case	**Configure**
	Actors	Observer
	Type	primary, essential
	Description	An Observer indicates specific configuration options. The actual configuration options that are available to an Observer are limited according to the level of ability of the particular Observer.
6	Use Case	**Distribute Data**
	Actors	Observer
	Type	primary, essential
	Description	An Observer specifies that he wishes to share data across the network. The Observer is able to import data from across the network as well as indicate what data should be made available across the network.

Table 3.1: Primary Use Cases

The two extra use cases are *Configure Hardware* and *Configure Software*. They are related to the *Configure* use case through the use of the *include* UML stereotype. The *include* stereotype indicates that the functionality of the two included use cases is used in the *Configure* use case.

Figure 3.1 shows how the actors defined above interact with the system. From the diagram it might appear that the Observer actor can perform more operations than the Operator and Power User actors are able to, but it must be remembered that the Operator and Power User actors can in fact interact with all the use cases that the Observer actor can since they are extensions of the Observer actor. This is indicated on the diagram by the generalisation arrows linking the three actors. At this level of abstraction, the Power User does not appear to extend the Operator actor at all. The use cases that are accessible only by the Power User are visible at lower levels of abstraction. For example, only a Power User should perform the radar hardware configuration, depicted by the *Configure Hardware* use case. In the implementation of the more general *Configure* use case, the *Configure Hardware* use case will only be included if the actor involved is a Power User. This conditional behaviour does not get captured in the Use Case diagram, since it is modeled at a lower level by UML activity diagrams.

3.4.3 Real Use Case Analysis

The real use cases discussed by Larman often include detail describing the exact interactions that the user makes with the system. For instance, where there is a GUI involved he will specify what fields in the GUI the user must complete and which buttons the user should press. This is relevant where it is possible to define at the start what mechanism the user shall use to interact with the system. The difficulty with this system is that there is no predefined GUI that the user should be offered. There might not even be a GUI, the user might be interacting with the system via a command line interpreter. Since the design of this system is expected to be modular, with certain modules being implemented later in the system's life cycle, these types of low level use cases will have to be defined closer to the time. As an example, consider the requirement that the system be configurable through the use of a configuration script. This configuration script may describe how the GUI itself will appear to the user at run-time. It would therefore be inaccurate to describe how the user should interact with the GUI when the GUI itself might change from execution to execution.

For this reason, the real use cases described here might still appear more abstract than one might expect. Specific modules will require their own set of use cases to be developed. Furthermore, since the precise specification of many of the real use cases depend upon concepts developed in the conceptual analysis of the problem, the details of these use cases shall not be discussed here.

Six real use cases were identified that closely match the six essential use cases. The six real use cases are described below in the high-level format proposed by Larman [8].

1	Use Case	**Obtain Data**
	Actors	Observer
	Type	real
	Description	The Observer chooses a host on the network from which to receive data from. The Observer makes a connection to the host, and all the data that is available on the host is transferred tot he Observer's platform. The connection is maintained until the Observer explicitly disconnects from the host. Data is transferred to the Observer's platform as an entire data set if an entire data set is available, otherwise data is transferred profile by profile, as each profile becomes available.
2	Use Case	**Offer Data**
	Actors	Observer
	Type	real
	Description	The Observer indicates the data which he wishes to make available on the network.
3	Use Case	**Configure Processing**
	Actors	Observer
	Type	real
	Description	The Observer specifies the processing that he requires by choosing from a set of predefined available processing routines. The Observer has control over which particular routines shall be applied, the setup parameters for each routine, and the order in which the routines are applied to the data.
4	Use Case	**Persist Data**
	Actors	Observer
	Type	real
	Description	The Observer indicates how the data shall be persisted, whether it should be persisted on a profile-by-profile basis, or whether it can simply be persisted on completion of a data set. The Observer does not need to explicitly indicate when the data should be persisted, data should automatically be persisted once the user has indicated the method by which this persistence shall occur.

5	Use Case	**View Data**
	Actors	Observer
	Type	real
	Description	The Observer is offered a view of the current data set. The Observer has controls such as zooming and panning. The view should update to include new data as data is obtained from another host.

6	Use Case	**Acquire Data**
	Actors	Operator
	Type	real
	Description	The Operator controls the acquisition of data from the radar hardware. The Operator begins by specifying the azimuth step size, and any comments about the measurement site. Profiles are then obtained either automatically or manually, and the Operator then indicates that the measurement session is closed.

3.5 Conclusion

This chapter has developed the user requirements for the system. The requirements were developed mainly from the experience gained through the development of the JMDR application, as well as from limitations identified while using JMDR and investigating existing data processing applications. Among the limitations identified were the reliance of the applications on their GUIs, their complicated functionality and their limited capabilities for the sharing of data.

The requirements specification was then developed to address these limitations, as well as to address the further processing that SFGPR data requires. The requirements were divided into two areas: a set of general user requirements, and a set of system specific requirements. The system specific requirements were identified as those requirements that were unique to this particular system.

A set of actors and use cases were identified and described in section 3.4. Two sets of use cases were described: a set of essential use cases and a set of real use cases. The essential use cases summarise the abstract goals that a user expects from the system, while the real use cases summarise more concrete goals that are more dependent on the implementation environment (for example, GUI interactions).

The set of real use cases that were developed above are the link between the abstract, essential use cases and the actual design and implementation of the system that is discussed in the following chapters.

Chapter 4

Conceptual Analysis

4.1 Introduction

This chapter focuses on using object oriented concepts to describe the conceptual model of the system. The conceptual model is a model that describes how the system can be viewed from a completely abstract point of view. No implementation details are presented in this chapter since the conceptual model is used to obtain an understanding of the problem domain, not to solve the underlying problem.

Fowler [1, pg 55] introduces three perspectives from which to approach the design of an object oriented project. He identifies the conceptual, specification, and implementation perspectives. The conceptual perspective concentrates on the concepts of the domain under study. UML class diagrams are usually used for this perspective, and while there is a relation between the classes described from the conceptual perspective and the classes that implement them in the implementation perspective, there is usually no direct mapping. Fowler states that "a conceptual model should be drawn with little or no regard for the software that might implement it, so that it can be considered language-independent". Larman refers to the model that is drawn from the conceptual perspective as the conceptual model, defining the conceptual model as the "representation of concepts in a problem domain" [8, pg 87]. For the purpose of this chapter however, strict UML notation will not be used since some of the ideas can be easily expressed without UML class diagrams. This is beneficial to the reader that does not have a full grasp of the UML notation.

This chapter describes the conceptual model for the system. Concepts that are used throughout the design are introduced here. Firstly, the abstract concept of data flow is introduced. It is shown that data flows between various points distributed geographically, and that this data is modified as it flows. The discussion of data flow introduces objects that participate in this data flow. Section 4.3 formalises the definitions for these objects, defining a set of key concepts which make up the conceptual design.

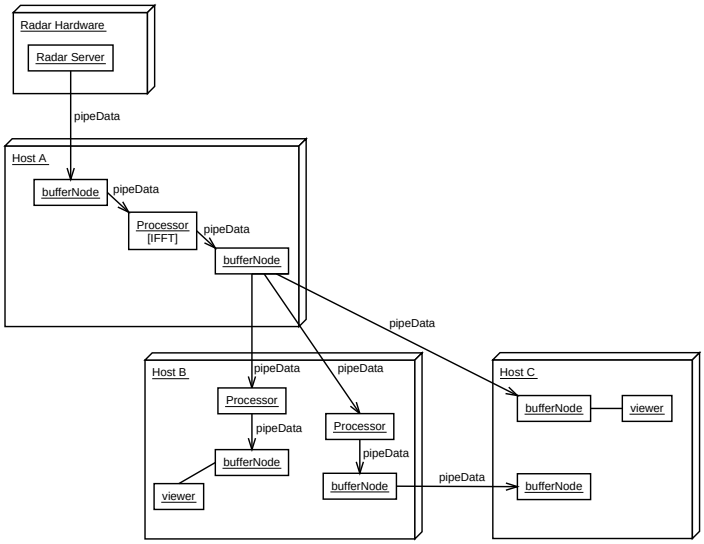

Figure 4.1: Conceptual Model

4.2 Data Flow

One of the key concepts for this system design is the concept of data flow. Data can be said to flow from the radar hardware out to the interested users. Data flows through points along the way where it can be processed, and where the data flow can fork into two separate paths. Figure 4.1 illustrates these ideas by proposing a hypothetical situation in which there are four physically distinct hosts or platforms. One of these platforms is the radar hardware (or radar server), and this platform is the source from which the data starts flowing. The three other platforms could be desktop computers. The following can be seen from the diagram:

1. Data is generated by the radar hardware.
2. The data flows to a point on a PC labelled Host A.
3. The data flows through a processor (which performs the inverse Fourier transform).
4. Data flows to another point on Host A.
5. At this point, the data flow forks into three flows. Two flows are directed to Host B, and one flow is directed to Host C.
6. Host B and Host C are now free to direct the incoming data-flows in their own, customised fashion.

Figure 4.1 is really a simplified UML collaboration diagram, where each box indicates a unique object. The objects depicted in the diagram are described as follows:

Radar Hardware The platform on which the radar server is running. This could be a desktop computer, however ultimately this would be an embedded operating system with a link to the network.

bufferNode A bufferNode is a point in the data-flow where data is buffered before being transmitted further. The data that is buffered at a bufferNode can then be used for various purposes such as viewing and persistence. A bufferNode has one input and multiple outputs, allowing one data flow to be forked into multiple flows.

processor A processor is an object that performs some processing on the data flow. A processor has one input and one output.

viewer A viewer is an object that provides a graphical interpretation of the data. It does not interact with the actual data flow at all. It connects to a bufferNode and displays the data that is buffered at the bufferNode. This is an example for how other objects would interact with the bufferNode objects. For example, an object that performs some form of data persistence would connect to a bufferNode in the same way as the viewer object. The persister object would only interact with the bufferNode and would not interact with the data flow either, persisting only the data that is buffered at the bufferNode.

The concept of the data flow should be kept in mind when considering the rest of the system, however this concept is purely abstract. It exists through the interaction of the objects depicted in the figure, but there is no actual "data flow" object. The objects depicted in the figure are the basic concepts from which the system is designed. The next section defines these concepts more clearly.

Figure 4.1 shows the four distinct platforms only to emphasise the fact that this system is required to be distributed. From the point of view of the bufferNodes shown in the diagram, and the data-flows themselves, the behaviour of the system should remain standard whether data is being transported between physically distinct hosts or whether all operations are occurring on the same physical host. This is evident in the diagram since there are no explicit links directly to the boxes representing the various platforms.

By using this conceptual view of the system, the opportunity for modular design is evident. The diagram shows the basic infrastructure required for data flow to take place on one host platform, or across host platforms. Data processing forms part of this required infrastructure since data processing modifies the data, and thus should interfere with a data flow. This infrastructure can exist on any host, and satisfies the requirement that the system should be able to run without a display. From Figure 4.1 this is demonstrated by the fact that viewer objects are only indicated on hosts B and C. Host A is acting simply as a data processor.

The potential for a modular design is hence also visible, since modules can be created that interact with the buffer nodes, without having to obstruct the data flow. The viewer object is an example of this, as well as the data persister that was mentioned above. Other modules could also be defined as new requirements become apparent.

4.3 Key Conceptual Components

From the above discussion, it is now possible to clearly define some conceptual components of the system. These components are described below. Also included in the description of each concept is a reference to the actual implementation class in the implementation model, this is included simply for clarity and can only be done in hindsight since the implementation model is developed chronologically later than the conceptual model.

Host There should be the concept of a host which participates in the whole distributed system. It is necessary to abstract the implementation details of how the system is distributed, however it is not necessary to abstract the fact that the system is distributed. For this reason there should be the concept of a host. The implementing class for this concept is the `Bus` class, and it is discussed in section 6.3.

Node The concept of a Node is indicated by the bufferNode objects illustrated in Figure 4.1. A Node is a point in the flow of data where the flow can fork into any number of flows. A Node is also a point where data is buffered after being processed. A Node offers a connection point at which access is granted to the data that the Node buffers. This access does not interfere with the data flow, it could be said that this access is read-only. The implementing class for the Node concept is the `LocalNode` class, and it is discussed in section 6.3.1.

Processor The processing that is applied to the data flowing from point to point should be encapsulated in the concept of a processor. The actual type of processing that is done is irrelevant at this stage, as long as it is the processor that does the processing. The processor is an object that directly alters the data that is flowing from point to point. The implementing class is the `Processor` class and it is discussed in section 6.3.1.

Module Any component that adds functionality to the system for a particular user should be viewed as a module. The Host, Node and Processor objects are essential for ensuring that data can be communicated from point to point, however these concepts only offer a realisation of a couple of use cases. It is distinct modules that offer realisations of most of the use cases. For example, viewing data should be a function that is implemented by a particular module. Another module would offer a radar operator the ability to capture data from the radar hardware. Each module has a distinct function that a particular user will find useful. Not all modules need to be available at every host, only the modules that a particular user requires. The module concept is used to indicate that there may be many different types of modules that accomplish various things, but all these can be abstracted into the concept of a module. The `Module` interface is the software interface that defines the base behaviour that any module must exhibit. This is discussed in section 6.6.

It is with this conceptual design that the class diagrams from the specification and implementation perspectives can be developed.

4.4 Conclusion

The conceptual model for the proposed system was developed in this chapter. The problem domain was discussed, introducing the concepts around which the actual design was developed. Firstly, the concept of data flow was discussed. It was found that, although data flow as a concept is important for visualising the behaviour of the proposed system, the data flow concept is something that exists only due to the interaction of a set of key concepts. This key set of concepts was described fully in section 4.3.

The next chapter considers the specification analysis of the actual design. The specification analysis concentrates on describing clear interfaces to all the components making up the system. There will therefore not be a direct mapping from the concepts described in this chapter to the interfaces described in the next. It shall be seen that there might be many interfaces to a component representing a single concept from this chapter. The concept of a Node is an example of the case where the concept is realised by a single implementation class, however it implements more than one interface.

Chapter 5

Specification Analysis

5.1 Introduction

This section describes the specification model for the system. Fowler [1] states that the specification perspective differs from the conceptual perspective in that the specification perspective relates directly to software entities such as classes specified in software. However, the specification perspective focuses on interfaces, not implementation. This is important since one of the keys to object oriented development is the difference between interface and implementation. It is through the definition of clean interfaces that it becomes possible to abstract details such as how the system communicates over the network.

This chapter is brief since it merely describes the interface specifications that were defined for the system. Once again, the reader should be aware that the process of defining the interfaces is iterative. This chapter only indicates the final solution to the iterative process of interface definition.

5.2 Interface Definition

Figure 5.1 defines the specification model that was derived for the system. The UML class diagram is used to illustrate how the various interfaces are related to each other. This model describes the basic infrastructure around which the system has been designed. This infrastructure by itself only really satisfies the requirement that data should be distributed, however this infrastructure provides the potential for more specific functionality to be added later.

A base interface has been defined, and every other interface in the diagram inherits from this interface. The base interface is the `Located` interface drawn at the top of the diagram. The `Located` interface provides methods that allow a client of a `Located` object to request its location within the network. The method by which an object is located is dealt with when the implementation model is developed.

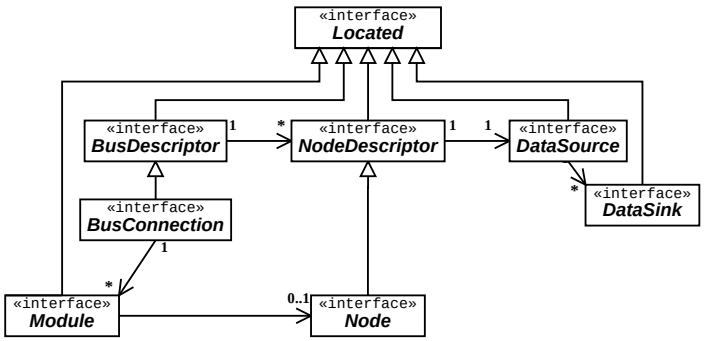

Figure 5.1: Specification Model

The fundamental relationship in the infrastructure described by Figure 5.1 is the relationship between the `DataSource` interface and the `DataSink` interface. These two interfaces provide the structure by which data is transferred. The diagram indicates the multiplicity of the relationship between a `DataSource` and a `DataSink` object. A `DataSink` object can have only one `DataSource` object, while a `DataSource` object can have any number of `DataSink` objects. It is this definition that allows the data-flows described in the conceptual model to be forked into parallel flows. As soon as a `DataSource` object has more than one `DataSink` object, then a fork in the data-flow occurs. This `DataSink/DataSource` relationship holds even when the `DataSource` and the `DataSink` are on physically separate hosts. This is achieved in the classes that actually implement the interfaces and is discussed in the implementation model.

The `DataSource` and `DataSink` interfaces only specify how data is transferred. Further functionality is provided through the `Node` and `NodeDescriptor` interfaces. In the conceptual model, reference was made to a bufferNode object. The bufferNode was described as a point in the system at which data is buffered and made available for use to other components. The `Node` and `NodeDescriptor` interfaces identified in the figure are different interfaces to the same conceptual bufferNode. This is illustrated more clearly in the implementation model. In fact, such a bufferNode object would also have to implement the `DataSource` and `DataSink` interfaces in order to be able to form part of the conceptual data-flow. The extra functionality that the `Node` interface provides would include a means by which an external component or module might get access to the data buffer in order to display or persist the data. The reason that both a `Node` interface and a `NodeDescriptor` interface were defined becomes apparent later, but it should be noted that the `NodeDescriptor` interface is a generalisation of the `Node` interface, and as such, the `Node` interface offers extra functionality over that offered by the `NodeDescriptor` interface. Basically, a `NodeDescriptor` interface provides methods for exploring the underlying object, while the `Node` interface provides methods that offer a tighter coupling between the objects involved. These issues are covered in more detail in the implementation model.

In the same way that the `Node` and `NodeDescriptor` interfaces are different interfaces to the same conceptual object, the `BusDescriptor` and `BusConnection` interfaces are also interfaces to the same conceptual object. In the conceptual model, four different hosts were mentioned and depicted in the diagram as three

dimensional cubes. Rather than abstracting the actual hardware, it is more useful to introduce the concept of a *bus* and define it as the environment in which the implementation of the specified interfaces can exist. A bus therefore glues all the components together. While typically there would be one bus existing on a particular host computer in which many bufferNodes might exist, there is no reason why more buses could not be started up on the same host. The bus provides the environment in which data transfer between `DataSource` objects and `DataSink` objects can occur. The term *bus* is used since this object can be viewed as the software equivalent of a hardware bus; it offers connectivity between many different components. The `BusDescriptor` interface provides a means of querying the bus object, while the `BusConnection` interface provides a more tightly coupled means of configuring the bus along with the components that are connected to that bus.

Lastly, Figure 5.1 shows a `Module` interface. The `Module` interface is a generic interface for any module that might connect with a bus. Such an object does not form part of the conceptual data-flow, but simply exists on the bus to provide some added functionality. The diagram indicates a relationship between the `Module` and the `Node` interfaces. The multiplicity shown on the diagram indicates that this relationship is optional for the `Module`. This is more easily described by means of an example. One obvious module that will be implemented is a viewer module. A viewer module is a module that provides a user with a graphical representation of the data. Such a module would have to connect to a bufferNode object via the `Node` interface in order to get access to the buffered data so that an image could be created from this data. However, another module that might be developed is a chat module that allows users to send messages to each other. Such a module would not require any access to a bufferNode object, however it would still have to exist on a bus object. This is why the multiplicity indicated on the diagram shows that a `Module` can optionally interact with one `Node`.

5.3 Conclusion

This chapter described the specification model that was developed for the system. A set of interfaces were described in UML to indicate the interfaces that the system should implement. The interfaces that were defined mainly described how the conceptual data flow should occur. The core interfaces responsible for achieving this data flow are the `DataSource` and `DataSink` interfaces.

The `Node` and `NodeDescriptor` interfaces were defined as interfaces to the same conceptual object - the *Node* concept that was described in section 4.3.

The `Module` interface was defined as a generic interface for modules that implement specific functionality, such as persisting data. More specialised interfaces for these modules are created when required, but all specialised interfaces extend the base `Module` interface. This allows the rest of the system to treat all module implementations identically.

The next chapter is devoted to the implementation of the interfaces defined in this chapter. The details of network communication are explained, as well as the details of how the data processing is achieved.

Chapter 6

Implementation Model

6.1 Introduction

The implementation model relates directly to the software classes that are defined during the implementation of the system. Since this model deals with the implementation of the interfaces described in the specification model, it is a more detailed and comprehensive look at the system. The implementation model will be discussed in portions since there would be too much detail to include in a single class diagram.

Since the implementation model deals specifically with software classes, it is necessary to specify the software tools that will be used to implement the system. Section 6.2 summarises the tools that were employed in the implementation.

The core infrastructure is described in section 6.3. The core infrastructure is made up of classes responsible for data transfer, as well as classes implementing the framework in which data processing can occur.

Section 6.4 explains how the actual network communications is achieved using CORBA. This section describes the class structure that makes it possible for other messaging protocols such as RMI to be implemented at a later date.

The data structure is described in section 6.5. This section describes the wrapper classes used to hold the data, as well as the CORBA structures that are required for these wrapper classes to be transferred across the network.

The data-viewer, data-persister and radar-controller module implementations are discussed in section 6.6. This section describes how these modules are incorporated into the system, and how other modules may be added at a later date as requirements develop.

Finally, section 6.7 discusses the two scripting languages, JPython and XML, that are used in the framework.

6.2 Software Tools

The technologies that were used in this design have all been introduced in chapter 2; this section simply clarifies for which parts of the system each tool is used.

Java was selected as the programming environment in which the interfaces presented in chapter 5 would be implemented. Implementation classes would be written in Java to perform all the functionality of data processing, data viewing and data persistence.

CORBA was selected as the middleware for the system. In particular, the Orbacus ORB was selected to provide the connectivity between software entities. It should be noted though that while the solution incorporates a CORBA ORB to provide the connectivity, this connectivity could also be provided by another technology such as Java Remote Method Invocation (RMI). These details are explained more fully in the section that covers distributed communications.

XML was selected as the scripting language to be used for configuration scripts and data persistence. XML compliments Java and CORBA in that XML is both platform independent and programming language independent.

Finally, JPython was used to provide a more powerful scripting environment. For example, JPython was used to script together a set of GUI components to create a customised GUI.

6.3 Core Infrastructure

The core infrastructure provides the environment in which the data-flow described in the conceptual model can occur. The core infrastructure is therefore the structure in which data transfer and data processing takes place. The interfaces used in this structure have been defined in the specification model. This section now discusses the classes that implement those interfaces to create the required infrastructure.

There are two classes that form the core infrastructure. These classes are the `Bus` class and the `LocalNode` class. These classes are defined as follows:

LocalNode The `LocalNode` class is an implementation of the conceptual bufferNode discussed in the conceptual model. A `LocalNode` instance therefore provides a point at which data is buffered in the data-flow. In order to fit into the data-flow, the `LocalNode` must implement the `DataSource` and `DataSink` interfaces. The `LocalNode` class also provides an implementation of the `Node` interface, and consequently also of the `NodeDescriptor` interface. By implementing these interfaces, a `LocalNode` object allows other modules to the access the data buffer that is managed by the `LocalNode` object. The `Node` interface also provides further functionality, but this is covered in more detail later. A `LocalNode` instance can only exist on a `Bus` instance. The `LocalNode` class is also responsible for data processing. A processor object is managed by the `LocalNode` instance, and as

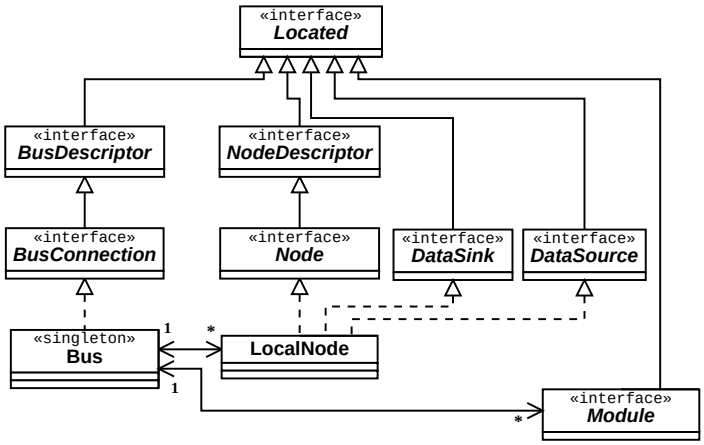

Figure 6.1: Core Infrastructure Class Diagram

data arrives at the `LocalNode` instance, it is first processed before being placed into the buffer owned by the `LocalNode` instance.

Bus The `Bus` class roughly encapsulates the environment on a particular host computer in which a conceptual data-flow can occur. A `Bus` object would contain a number of `LocalNode` instances, as well as any modules that were being used. Typically one `Bus` object would exist on a particular computer, however it is possible for more than one `Bus` instance to exist on the same computer, as long as each `Bus` instance has a distinct name associated with it. The `Bus` class implements the `BusConnection` interface, and consequently also implements the `BusDescriptor` interface.

These two classes provide the basic behaviour for the entire system. In order to achieve any of the user goals mentioned in chapter 3, there would have to be at least one `LocalNode` instance existing in a `Bus` instance on the user's machine. These classes together offer complete implementations of the data transfer and data processing behaviour required, however they have been defined to interact through the purely abstract interfaces defined in the specification model in order to decouple the two classes from each other, and to decouple instantiations from each other.

Figure 6.1 shows the class hierarchy for the classes discussed above. The `Module` interface is also shown at this level, however it is discussed in a section on its own. The diagram indicates the multiplicity between the `Bus` and the `LocalNode` classes, as well as between the `Bus` class and the `Module` interface.

6.3.1 The `LocalNode` Class

The `LocalNode` class is discussed here since the requirements and consequent implementation of the `LocalNode` class affect the design of the related infrastructure in which `LocalNode` instances exist and behave.

The `LocalNode` class is an implementation of the conceptual buffer node that was discussed in the conceptual model. The `LocalNode` class therefore has the following responsibilities:

1. Maintain a data buffer containing all the data profiles making up a data set.
2. Process data coming in through the `DataSink` interface according to a configurable set of routines.
3. Ensure that the newly processed data is available to `DataSink` instances through the `DataSource` interface.
4. Provide the `Node` interface through which components can access the data buffer.

The need for a buffer node object was identified from an early point in the design process, since it was required that data should be processed in stages in order that various views of the data could be obtained to satisfy different user processing requirements. The `LocalNode` class was defined to provide the implementation of the buffer node.

The conceptual model shown in Figure 4.1 illustrates a processor object as being a separate object in the flow of data. A `Processor` class was therefore defined that would perform the task of data processing. While conceptually the processor object can be considered completely separate, from the implementation perspective it is necessary to bind the processor object more closely with the buffer node. The main reason for this relates back to the fact that processing data can be time consuming. The issues surrounding the timing of data processing shall be dealt with in a separate section, for the time being it is enough to note that a `Processor` class was defined that would be responsible for data processing, and that an instance of this class is associated with every `LocalNode` instance.

The core infrastructure needs to provide the mechanism by which data can be propagated as rapidly as possible without any loss of data. Thus, if data is being generated at five data profiles per second, then the infrastructure must be able to accept the data at this rate, and attempt to propagate it at this rate. However, since data processing can be a time consuming task, it may be that data is being generated faster than the data can be processed. This is one of the main reasons that a `LocalNode` instance must buffer data. Data can then be buffered before it is processed, so that the data capture process can occur at the rate required by the radar operator, and the processing can occur at a different rate. The buffer maintained by the `LocalNode` instance acts as a dam in the data-flow. This already implies that the data propagation is likely to be asynchronous, since forcing the data flow to be synchronous would require data to be generated by the radar server at a rate that was compatible with the most time consuming processing occurring in the system.

The implementation perspective therefore differs from the conceptual perspective in that from the implementation perspective, data flows from one `LocalNode` instance directly to another `LocalNode` instance. The `Processor` instance

that handles the data processing still acts directly on the data flow, however the `Processor` instance is encapsulated by the `LocalNode` instance. The `LocalNode` instance first uses its `Processor` instance to process the data before it places the processed data into its buffer. Data is therefore exchanged between `LocalNode` instances, through the `DataSource/DataSink` interfaces. The `LocalNode` class keeps a list of references to any number of `DataSink` objects. Data is piped to all these references, and it is through this structure that a data flow can be forked.

The behaviour of the `DataSource/DataSink` relationship is fundamental to the design of the system. This behaviour must describe how data is actually transferred between nodes in the data-flow. There were two possibilities for this behaviour; either a *push* paradigm could be employed where data is pushed onto the `DataSink` by the `DataSource`, or else a *pull* paradigm could be employed where data is pulled from the `DataSource` by the `DataSink`. It is natural to think in terms of the push paradigm for this system since data is generated by the radar server at a rate required by the radar operator. However, it should be remembered that the data flow must be asynchronous in order to allow for the time required by the various `Processor` instances, and hence the pull paradigm was employed. The reasoning is described in the following scenario:

Consider two `LocalNode` instances, X and Z. X is the `DataSource`, and Z is the `DataSink`. Data must therefore flow from X to Z. Both instances own separate `Processor` objects, however it is only Z's `Processor` object that will affect the data flow between these two instances, since Z's `Processor` object will process the data that arrives from X. Since a `LocalNode` object should only have one data buffer, Z can only receive data at the rate at which it can process data. This rate should not propagate back up the data flow, so X should be able to receive data at a faster rate than Z can handle. It therefore makes sense that instead of X trying to force data onto Z at X's data rate, Z should fetch data from X when Z is able to. If Z catches up with X, then Z will simply wait until X receives more data. This also simplifies the case where there are multiple `DataSink` objects connected to X. Instead of X trying to keep track of the data rate required by each `DataSink` object attached to it, each `DataSink` object can keep track of its own data rate, fetching data when required from X. This also allows a `DataSink` object to connect to X after a measurement session has already started, and the new `DataSink` object will be able to catch up to the current profile. This is possible since all `LocalNode` objects have a data buffer. It should also be evident at this point that a multithreaded environment will be required. A `LocalNode` object will require its own thread in which fetching and processing data can be handled.

This behaviour is achieved by the definition of the `DataSource` and `DataSink` interfaces indicated in Figure 6.2.

Only the important methods are shown in the diagram. The `DataSet`, `DataHeader` and `DataProfile` classes alluded to above are described by their names, no more detail is required about them at this point. A typical measurement session would proceed as follows:

1. A radar operator begins a measurement session; he will describe properties such as azimuth step size which will be incorporated into the `DataHeader`

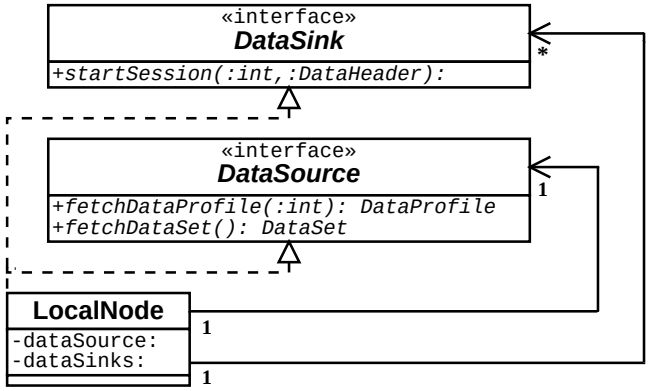

Figure 6.2: `LocalNode` Class Diagram

object. This results in the `startMeasurement` method call propagating through the data flow. This is essentially an asynchronous message, since at each `LocalNode` instance in the data flow, a new thread is started allowing the `startSession` call to return.

2. Each `DataSink` object responds to the `startMeasurement` message by calling `fetchDataProfile` method on its `DataSource` reference. This `fetchDataProfile` method will block until data is available.

3. As soon as the `fetchDataProfile` method returns, the `DataSink` object calls the `fetchDataProfile` method again. This process loops until the end of the session, when a `DataProfile` object is returned that has a flag set indicating that the session has ended.

A UML sequence diagram is shown in Figure 6.3 illustrating how the `DataSource` and `DataSink` interfaces described above behave.

Figure 6.3 illustrates the implementation of the concept of the data flow mentioned in the conceptual analysis. It is difficult to illustrate the multithreaded behaviour of `LocalNode` instances on the sequence diagram, however an attempt has been made by using multiple life lines where necessary.

The diagram shows a `DataSource` instance to which a `LocalNode` instance has connected. There are multiple `DataSink` instances connected to the `LocalNode` instance. In the diagram, time flows from the top to the bottom indicating the sequence in which method calls are made.

When no measurement session is in progress, then the entire system is idle with no active threads. As soon as a measurement session is started, the `DataSource` instance calls the asynchronous `startSession(...)` method. This method indicates to all `DataSink` objects that a session has started, and that each `DataSink` object should begin requesting data. As can be seen in the diagram, the `startSession(...)` method is propagated by the `LocalNode` instance to all connected `DataSink` objects.

As soon as a `DataSink` instance (including the `LocalNode` instance) receives the `startSession(...)` call, it must start up a new thread in which the

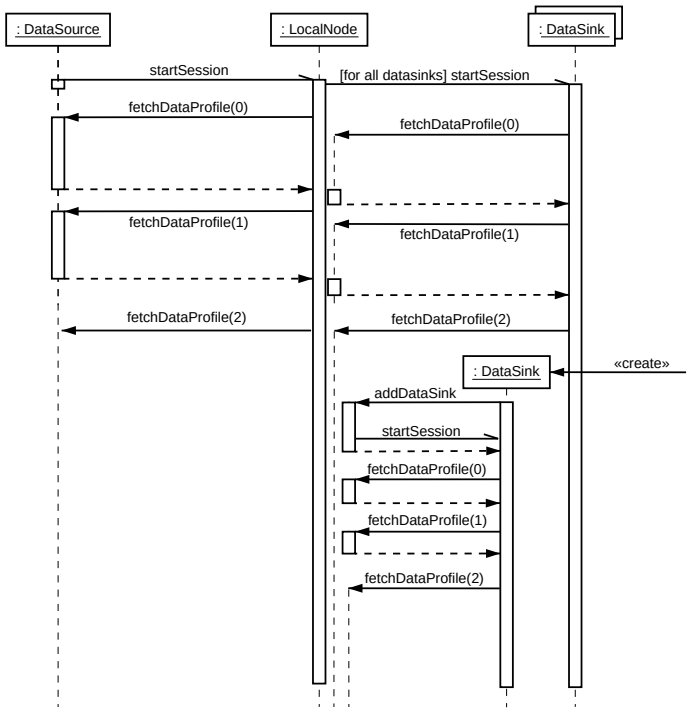

Figure 6.3: Sequence diagram showing `DataSource` and `DataSink` interaction.

`fetchDataProfile(int profile)` method can be called on the appropriate `DataSource` instance. The integer argument in the `fetchDataProfile(int profile)` is used to tell the `DataSource` object exactly which `DataProfile` is needed. In this way, it is up to each `DataSink` instance to keep track of which profile is required, which allows each `DataSink` instance to be in the process of retrieving a different profile from the same `DataSource`.

When the `fetchDataProfile(...)` method is called and the required profile is not available at the `DataSource` yet, then the `fetchDataProfile(...)` method will block until the profile is ready. This blocking is achieved by putting the calling thread to sleep. When the profile arrives, the thread is notified, and execution continues. This is indicated on the diagram by showing only the dotted lifeline when the thread is being blocked. When the thread is notified, the activation symbol (the narrow rectangle) is shown to indicate that the thread is active. As can be seen, this occurs once the `DataSource` instance has returned a profile to the `LocalNode` instance.

The diagram also shows how a `DataSink` object is created and connected to the `LocalNode` instance in the middle of the measurement session. The `DataSink` object connects to the `LocalNode` instance via the `addDataSink(...)` method. Since a measurement session has begun, the `LocalNode` instance immediately calls the asynchronous `startSession(...)` method on the new `DataSink` object. The new `DataSink` object then starts up a new thread and calls `fetchDataProfile(0)` on the `LocalNode` instance. Since the `LocalNode` instance already has profiles `0` and `1`, these profiles can be returned immediately to the

new DataSink object. The new DataSink object has now caught up to the rest of the DataSink objects connected to the LocalNode instance.

When the measurement session is completed, a data profile is returned from the radar server indicating that the measurement session is finished. This profile does not contain any data, and acts as a data footer, however for efficiency reasons this footer was packaged as a DataProfile. When a DataSink instance receives this footer it closes the extra thread, waiting for the next startSession(...) method call.

Note that the focus of Figure 6.3 is on the implementation of the LocalNode class. The other objects in the diagram might or might not be instances of the LocalNode class, however, whatever they are at the implementation level, they offer the required DataSource or DataSink interface. So for example, the DataSource instance shown in the figure might actually be the radar server, or a connection to the radar server but it is offering data via the DataSource interface.

Finally, Figure 6.2 shows that the DataSource interface also specifies the fetchDataSet(). This method allows an entire data set to be transferred in one step. This is useful if the entire data set is available since it is more efficient to transfer it as a whole instead of profile by profile. There are two cases where the entire data set might be available. The data may have been retrieved from some storage facility such as a file. In this case the system might have been configured to operate as a stand-alone application used for post-processing. The second case where the entire data set is available might occur when a DataSink object is connected to a DataSource object after a measurement session has been completed. The entire data set is then available at the DataSource, and the DataSource instance can then inform the DataSink object to use the method call fetchDataSet() instead of the fetchDataProfile(...) method. The mechanism is still the same however - that is, the fetchDataSet() method is still called from a new thread spawned by the DataSink instance.

6.3.2 The Bus Class

The Bus class is not complicated since it is mainly a holder class for other classes. Figure 6.1 shows enough detail for this section. The main responsibilities for the Bus class can be summarised as follows:

1. Maintain a list of LocalNode instances.
2. Maintain a list of Module instances.
3. Provide a management interface through which the LocalNode and Module instances can be configured.

As can be seen from Figure 6.1, the Bus class implements the BusDescriptor and BusConnection interfaces. The BusDescriptor provides an interface through which a Bus instance can be queried. This allows users to find out about a Bus object that has been made available on the computer/network. For example,

this interface specifies a method that lists the names of all the `LocalNode` instances managed by the `Bus` instance.

The `BusConnection` interface extends the `BusDescriptor` interface to add management functionality. There are reasons for splitting the functionality of the `Bus` class between two interfaces. There is the issue of restricting the functionality for particular users. A general user should be able to get information from the `Bus` instance regarding what `LocalNode` objects are available, however that same user should not necessarily be allowed to alter the configuration of the `LocalNode` objects. The other reasons relate to the implementation of the network adapters that do the work of distributing the `Bus` instance on the network and will become apparent later.

6.3.3 The `Processor` Class

The `Processor` class was mentioned while discussing the `LocalNode` class. The behaviour of the `Processor` class in conjunction with `LocalNode` class will be discussed in this section.

The `DataSource/DataSink` interfaces define how data is transferred between nodes in the conceptual data flow. It was pointed out that in the implementation model, the processor object would be encapsulated by the `LocalNode` class.

While the `Processor` class does not participate in the data flow via the `DataSource/DataSink` interfaces, it does modify the data flow directly. The `LocalNode` is responsible for ensuring that the data received is processed by a `Processor` instance before being added to the `LocalNode` object's data buffer.

The `Processor` class contains a list of routines that are applied consecutively to the incoming data. Each routine is an object that performs some specific signal processing. One routine may be an implementation of an inverse Fourier transform, while another routine might perform a windowing operation. A `Processor` instance would then chain these two routines together to obtain a processor that applies first a window routine followed by an inverse Fourier transform on each profile that arrives at the `LocalNode` instance.

The structure is actually an implementation of the *Chain of Responsibility* design pattern referenced in the book *Design Patterns* by Gamma et al [10]. The class diagram for the `Processor` class is shown in Figure 6.4.

The diagram shows that the `LocalNode` class contains a reference to one `Processor` instance. The `Processor` class is a concrete implementation class while the `Routine` class is an abstract class. The `RoutineLink` is a pure interface. This structure allows a `Processor` instance to contain a list of any number of `Routine` objects, where each `Routine` object is an instance of a different concrete implementation of the `Routine` class. Each `Routine` object in the list contains a reference to the next `RoutineLink` instance in the list. The last `Routine` instance in the list contains a reference back to the concrete `Processor` class also through the `RoutineLink` interface implemented by the `Processor` class. This allows the processor to receive the processed data back from the list of `Routine` objects. The processor is then able to place the processed data into the `LocalNode` object's data buffer.

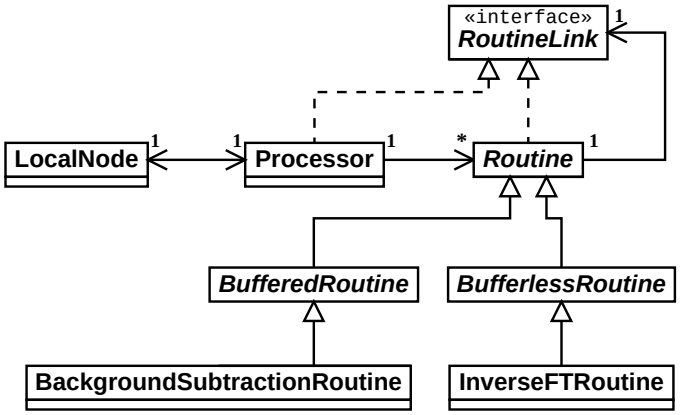

Figure 6.4: Class diagram of the `Processor` class

Since it is the `LocalNode` class that is responsible for creating a thread in which processing can take place, the `Processor` class must be completely synchronous. This simplifies the implementation of the processing routines.

Also shown on the class diagram are the abstract classes `BufferedRoutine` and `BufferlessRoutine`. These classes are used to further simplify the implementation of various processing routines. The `BufferlessRoutine` class describes a routine which only requires the current profile in order to process the output profile. An example of such a routine would be the inverse Fourier transform which is applied to each profile.

The `BufferedRoutine` class describes a routine where the output profile is dependent on the processing of several input profiles. Any routine that uses some kernel to process the output profile would fall into this category. Figure 6.4 uses a background subtraction routine as an example of a concrete implementation of the abstract `BufferedRoutine`. While the `BufferedRoutine` class is abstract, it implements the buffering of a specified number of profiles. Concrete implementations of this class then only need to implement the processing that is performed on the kernel, and need not worry about implementing the buffering as well.

The behaviour of the `Processor` class along with the two abstract `Routine` classes is best illustrated in a UML sequence diagram. Figure 6.5 shows the sequence diagram for the case where there are two routine objects. One routine object is an implementation of the `BufferlessRoutine` class, while the other routine object is an implementation of the `BufferedRoutine` object.

A sequence diagram gives an indication of the sequence in which messages are exchanged between objects. The diagram shows a `LocalNode` instance receiving the asynchronous `startSession(...)` message. Included in this message a `DataHeader` object containing information about the measurement such as the azimuth step size. This data header must be processed by each routine object since each routine needs to add information to the data header. The `LocalNode` instance therefore calls the synchronous `applyDataHeaderProcessor(...)` method on the `Processor` instance. The `Processor` instance

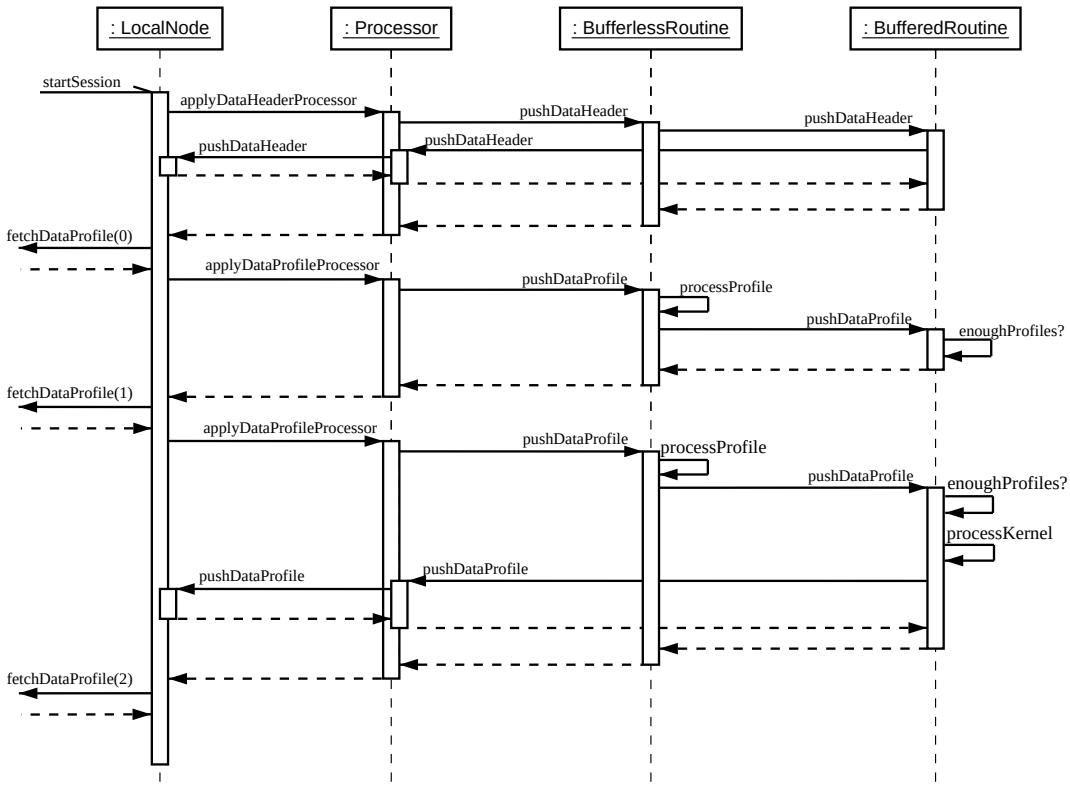

Figure 6.5: Sequence Diagram for the `Processor` class

then calls the `pushDataHeader` method on the first `RoutineLink` instance in the list. In this example, this next object is the `BufferlessRoutine` instance. This `BufferlessRoutine` instance processes the data header, and then passes the data header to the next `RoutineLink` instance in the list. The next instance is the `BufferedRoutine` instance. The data header is processed once again, and pushed onto the next `RoutineLink` instance in the list. The next `RoutineLink` instance turns out to be the `Processor` object itself. The `Processor` object is now able to place the data header into the `LocalNode` data buffer. This is done by calling the `pushDataHeader(...)` method on the `LocalNode` instance. Only after this method returns, can the `applyDataHeaderProcessor(...)` method complete and return control to the `LocalNode` instance.

On completion of the `applyDataHeaderProcessor(...)` method, the `LocalNode` instance is able to request the first data profile. This data profile is returned by the `fetchDataProfile(0)` method. Once the data profile is returned from the source, the `LocalNode` instance is able to process it by calling the `applyDataProfileProcessor(...)` method on the `Processor` instance. Note that after calling this method, the `LocalNode` instance relinquishes responsibility for that data profile. The `Processor` instance is now responsible for ensuring that the profile ends up in the data buffer of the `LocalNode` object. This design is necessary since the `LocalNode` instance cannot keep track of the types of processing routines that might exist in the routine list. The diagram illustrates this point by showing a `BufferedRoutine` instance. This routine can only process a new data profile once the required number of profiles have been received. In the diagram, the first profile is sent into the list of routines, however when the profile is received by the `BufferedRoutine` object, it is stopped since the routine does not have enough profiles to process the result. The method is still synchronous however, and returns control to the `LocalNode` object without the `LocalNode` object being aware that the profile is actually buffered in a routine awaiting more data. The next profile that is fetched is then also sent into the list of routines, and when this profile arrives at the `BufferedRoutine` instance, the routine checks and finds that it has enough profiles with which to process the new profile. The routine processes the new profile and passes it on to the next `RoutineLink`, which happens to be the `Processor` object, which can then add the profile to the data buffer of the `LocalNode` object.

While the diagram illustrates the means by which data profiles are processed individually, the same process is applied to the case where there is an entire data set available to be processed. The transfer of an entire data set was discussed near the end of section 6.3.1. A data set that has been transferred as a unit like this should also be processed as a unit since it is more efficient. The `RoutineLink` interface specifies the `pushDataSet(DataSet set)` method which would be called if a data set was available. In this case, instead of each profile in the data set being sent through the list of routines individually, the entire data set would be passed between each link in the list.

6.3.4 Core Infrastructure Summary

The core infrastructure is made up primarily of the `Bus` and `LocalNode` classes. The `Bus` and `LocalNode` classes enable data to be transferred between a web

of connected `LocalNode` instances while the `Processor` class describes the sequence in which data is processed. This core infrastructure is what gives the main functionality to the system.

6.4 Network Communications

One of the requirements for this system was that it should be distributed. It should therefore seem that this section should come under the previous section as part of the core infrastructure, however it is discussed in its own section due to the nature of the requirements.

The requirements were that the system should be distributed, however there was no requirement that a particular technology be used to achieve this distribution. In fact, the requirement was that the system should not be entirely dependent on a single technology. It was indicated earlier that CORBA would be used to achieve the distribution of the system, however, the solution should not be tied to the CORBA technology. It should be possible to replace the CORBA specific functionality with another distributed technology such as RMI. For this reason, this section has been separated from the core infrastructure.

The `DataSource/DataSink` interfaces described earlier specify the manner in which data transfer will take place, but no implementation is specified. It is therefore possible to define a set of classes that implement these interfaces, and adapt the interfaces to another set of interfaces that can be used to achieve distributed communications. This design pattern is known as the *Adapter* pattern, and it is well documented in the *Design Patterns* book [10].

Using this pattern, it is possible to define a class that performs communications over the network via CORBA, but that implements the `DataSource` interface so that the core infrastructure can communicate with this class without being aware that it is in fact a CORBA implementation. Similarly, one could define a class that uses RMI to communicate over the network, and these two classes could both be used by the core infrastructure with no alteration required in the core infrastructure. Figure 6.6 is a class diagram showing how CORBA adapter classes can be defined to adapt the interfaces on both sides of a network connection to hide the network details from the core infrastructure.

The diagram shows that the *Adapter* pattern is applied twice, once for the `DataSource` interface and once for the `DataSink` interface. The `LocalNodeAdapter` class adapts the `DataSink` interface to a `CDataSink` interface while the `CNodeAdapter` adapts a `DataSource` interface to a `CDataSource` interface. The `CDataSource` and `CDataSink` interfaces are interfaces that are specified in the IDL definition. The actual Java source files are then generated automatically from the IDL definition.

At first, this may appear to be a rather round about way of achieving the network communications. In fact, it might appear that the above structure seeks to achieve what CORBA already achieves since the two adapter classes in the above diagram appear to function as CORBA stubs and skeletons function. One option that was considered in the design of the system was to simply implement the interfaces declared in the IDL file directly. From the diagram,

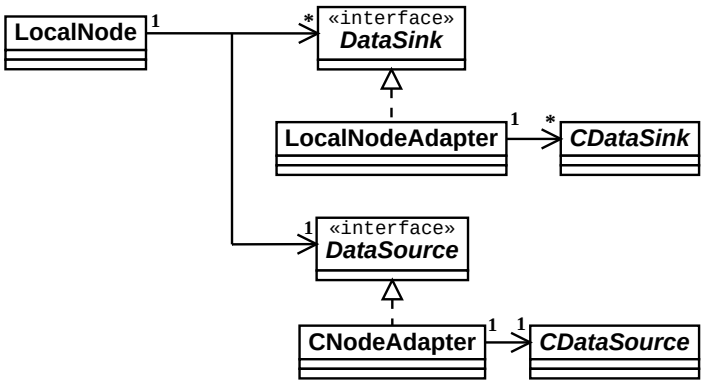

Figure 6.6: Adapter Class Diagram

this would mean that the `LocalNode` class would simply implement the `CDataSource` and `CDataSink` interfaces directly. This would eliminate the need for the two adapter classes, and the `LocalNode` class could then communicate directly with the stubs generated by the CORBA ORB. However, this would then limit the implementation to CORBA. The `LocalNode` class would be tightly coupled to the CORBA classes, with the following consequences:

1. It would be very difficult to add another distributed technology such as RMI to the system.
2. The CORBA libraries would always be required when running the system, even if the system were to run on a stand-alone computer with no network capabilities.

There is also the issue that the `CDataSink` and `CDataSource` interface source files are automatically generated from the IDL file through the use of a separate compiler. If suddenly a different technology to CORBA were to be used, then these interfaces would have to be rewritten from scratch.

By implementing the design as shown in Figure 6.6, one could simply replace the two adapter classes with RMI adapter classes, and the core infrastructure would not require any modification.

This section will next concentrate on how inter-node communication is achieved using CORBA, and will then go on to describe how `Bus` instances communicate over the network.

6.4.1 `DataSource` to `DataSink` Communications

This section describes the implementation of the adapters that perform the network connection between a `DataSource` instance on one computer, and a `DataSink` instance on another computer.

Two adapter classes were defined, the `LocalNodeAdapter` class and the `CNodeAdapter` class. The `LocalNodeAdapter` class is used to take an existing `LocalNode` instance, and make it available on the network as a data source. The

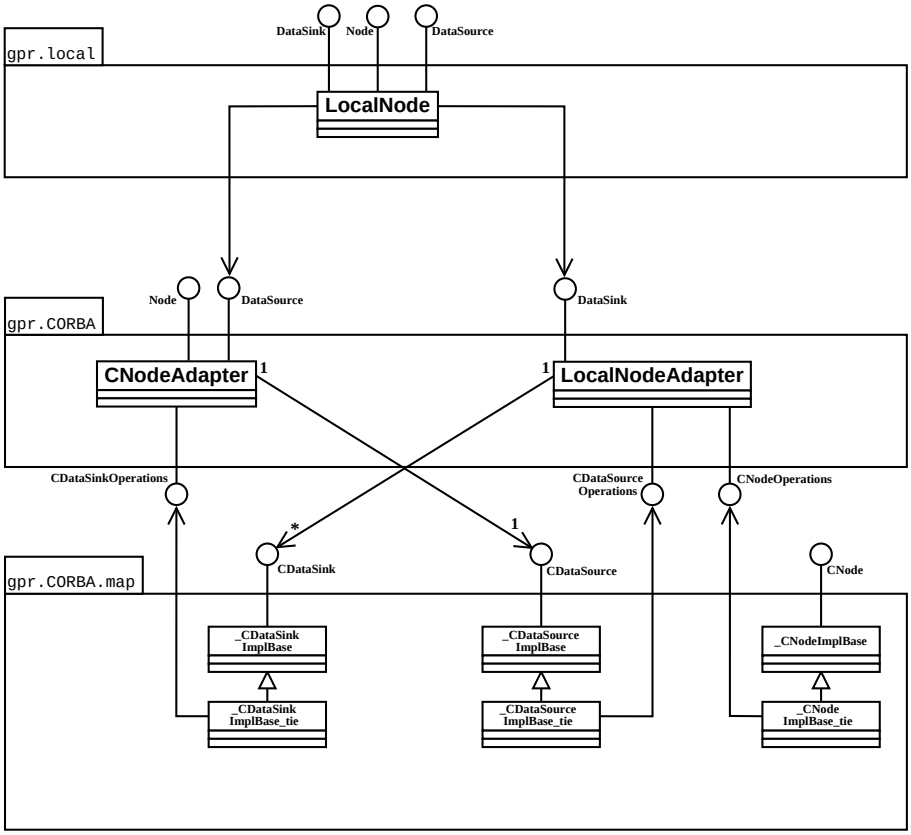

Figure 6.7: Class Diagram of the CORBA node adapter classes

`CNodeAdapter` class is used to complete the network connection to the data sink. A `CNodeAdapter` instance will connect to a remote `LocalNodeAdapter` instance, while offering the standard `DataSource` interface locally.

The class diagram of this design is shown in Figure 6.7. The class diagram also shows the package information overlayed. This is useful to illustrate how the interfaces decouple implementing classes. The packages are described as follows:

`gpr.local` The package in which classes implementing the core infrastructure resides.

`gpr.CORBA` The package in which CORBA specific classes such as CORBA adapters reside.

`gpr.CORBA.map` All the classes that are generated by the IDL compiler are stored in this package. This package includes both the interfaces that generated from the IDL source, as well as the CORBA stub and skeleton classes that are generated by the IDL compiler.

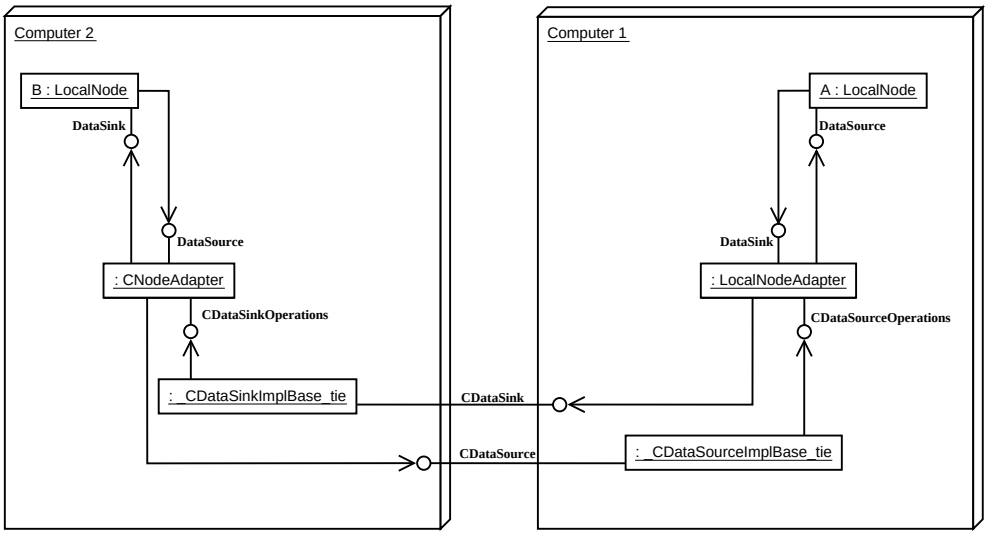

Figure 6.8: Collaboration Diagram showing the collaboration of the CORBA node adapter classes

The diagram shows how it would be possible to replace the `gpr.CORBA` package with a `gpr.RMI` package without disrupting the `gpr.local` package.

A UML Collaboration diagram offers a means of visualising how the various classes described in Figure 6.7 collaborate to achieve the `DataSource/DataSink` communications. Figure 6.8 illustrates how a `LocalNode` instance on computer A can offer data through its `DataSource` interface to a `LocalNode` instance on computer B. The collaboration diagram also illustrates which interfaces are used to archive the communications.

A further advantage to the above design is the fact that the CORBA adapter classes are not constrained to implement the same pull paradigm that is described for the higher level `DataSource/DataSink` interaction. The `LocalNodeAdapter` and `CNodeAdapter` class actually implement a push paradigm for their interaction. This is necessary to avoid having the processing thread in the `DataSink` instance blocking across a network connection. Instead, the local `CNodeAdapter` instance blocks the processing thread while the `CNodeAdapter` instance waits for data to be pushed onto it from the remote `LocalNodeAdapter` instance. The implementation at the network level is therefore completely independent of the higher level `DataSource/DataSink` implementation.

The `LocalNodeAdapter` and `CNodeAdapter` classes provide a CORBA implementation of the `DataSource` and `DataSink` interfaces. However the `LocalNodeAdapter` and `CNodeAdapter` classes were actually written to be CORBA proxies for the `LocalNode`. This means that the `CNodeAdapter` class must implement the `Node` interface as well so that objects that interact with a `LocalNode` instance through the `Node` interface can interact just as easily with the `CNodeAdapter`. The `LocalNodeAdapter` class must provide the link between the `CNodeAdapter` class and the `LocalNode` class. This double-sided adapter pattern is used extensively throughout the design of the system.

6.4.2 Inter Bus Communication

This section explains the implementation of the adapters used to distribute a Bus instance on the network. A Bus instance is used to manage a list of LocalNode objects on a particular host. The Bus instance also maintains the list of modules that the user is currently using. The main responsibility of the Bus class however is to provide a standard means by which objects (nodes and modules) can find one another and connect to one another. A LocalNode instance on Bus A will use BusA to find a DataSource instance on a separate host.

As was stated earlier, the system should not be dependent on a particular distributed messaging technology. The previous section has shown how the implementation of the inter-node communication can be hidden behind generic interfaces, however there has to be a class somewhere in the system that is responsible for creating the CORBA-specific adapter classes that adapt the generic interfaces to the underlying communication medium. Such a class would obviously have to be tightly coupled to the implementing classes for the particular communication medium. In order to decouple the Bus class from such a class, one could define a general interface for that class. The Portal interface was defined for this purpose.

A class that implements the Portal interface would be responsible for creating the adapter classes for a specific communication medium. The methods of the Portal interface return the higher level interfaces specified in the specification model. In this way, a Portal implementation can be written for any network messaging technology that might be used in the future. For this system, a CORBAPortal class was written that implements the Portal interface, however one could also write a RMIPortal class that would also implement the Portal interface. A Bus instance could then keep a reference to instances of both Portal implementations. When the Bus instance is asked to find a particular node, the Bus instance would ask both Portal implementations for a reference to the required node.

Figure 6.9 is a class diagram illustrating how the Bus, Portal and LocalNode classes are related to one another. Package information is overlayed on the diagram to indicate how classes in different packages co-operate. The following are the important points that Figure 6.9 attempts to illustrate.

1. The Bus class can have a reference to zero or more Portal instances. A Portal implementation class must exist for each network messaging technology that is supported by the system.

2. The Bus class maintains a list of zero or more LocalNode instances. Each LocalNode instance has a reference to one DataSource instance and any number of DataSink instances. The diagram illustrates how these DataSink and DataSource references might be to adapter classes in the gpr.CORBA package, thereby achieving a link to a LocalNode instance that exists on a separate host.

3. Apart from the tight coupling between the Bus class and the LocalNode class, all other references occur through interfaces defined in the gpr package. The diagram shows that classes in the gpr.local package will com-

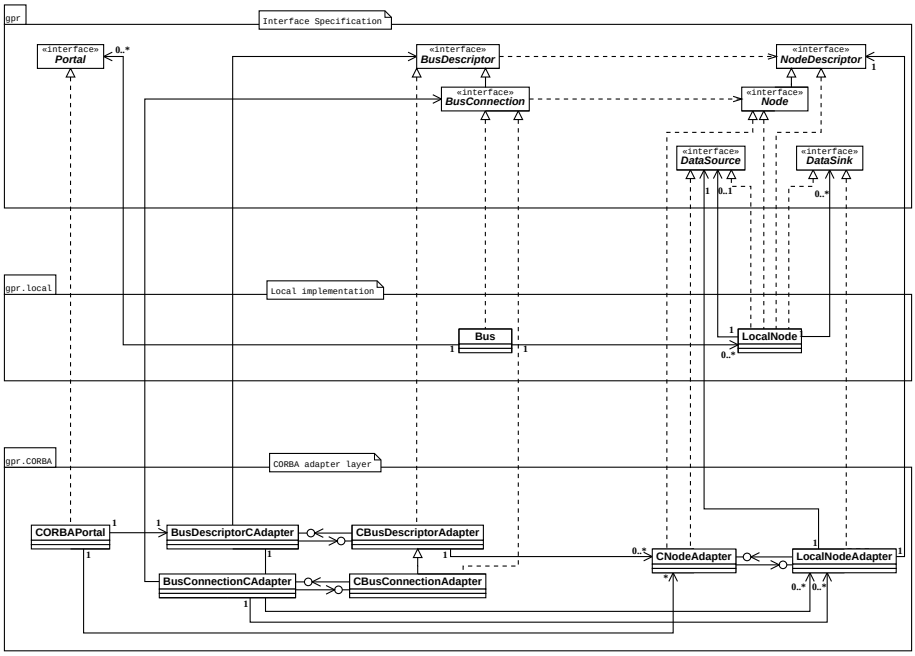

Figure 6.9: System Class Diagram

municate with classes in the `gpr.CORBA` package via interfaces in the `gpr` package.

6.5 Data Structure

This section describes in more detail the structure that has been defined for the data that is transferred and processed within the core infrastructure, as well as the manner in which data is transported between the CORBA adapters.

6.5.1 Core Data Structure

It was evident from the requirements analysis that a more complex structure would be required for representing the GPR data than simple arrays of numbers. One data profile consists of many individual samples, and each sample is a complex number, so at the lowest level it is necessary to store two floating point numbers to represent one sample. But apart from the actual numerical data, there was the requirement that the processing history should accompany the numerical data as well, so that it would be possible to rebuild the processor that was applied to a given set of data. The information required for the processing history would vary depending on the type of processing routines that were applied - different routines require quite different parameters, and these parameters would need to be stored as well. There should also be the possibility for including comments embedded in the data that describe certain sections of

the measurement, for instance the radar operator might wish to comment on surface anomalies that might affect the profile taken at a specific point. The data structure must therefore provide for all this information.

It was decided that the smallest element of a set of data would be a data profile consisting of a number of complex samples. This was decided since this is the logical manner in which data is captured using the radar hardware. It is then more efficient to transfer one profile as a unit rather than transferring many individual complex samples. The class used to represent a profile is the `DataProfile` class. This class is a wrapper for the following information:

- The numeric data consisting of a number of complex samples.
- A comment associated with the measurement, this comment can be empty.
- The profile's position within the entire set.

A number of `DataProfile` instances would make up a set of data, and the `DataSet` wrapper class was defined to encapsulate a set of `DataProfile` instances.

The `DataHeader` class was defined to encapsulate all the information required for the processing information. This information was separated from the individual profiles since the processing information is applicable to an entire set of data, and it would be inefficient to repeat the processing information with each profile. The `DataSet` class also encapsulates the `DataHeader` class. The `DataSet` class is in fact used as the buffer for the `LocalNode` class. `DataProfile` instances can be appended to the `DataSet` buffer, allowing the `DataSet` instance to grow as more data arrives at the `LocalNode` instance. A `DataHeader` instance is the first data element to be transmitted from `DataSource` to `DataSink`, and it is transmitted through the exact chain of routines that the rest of the data set is transmitted through. This is necessary since the `DataHeader` instance allows each routine in the chain to append its own configuration to the `DataHeader` instance.

These three data classes are used by all classes in the core infrastructure. The following section describes how these data classes are transferred between the CORBA adapters.

6.5.2 CORBA Data Structure

The data structure described above must be communicated through whatever network communications protocol is used. There are various ways of doing this. Firstly, using Java's built-in ability to serialise objects, it would be possible to simply serialise each data profile into a stream of bytes. The `LocalNodeAdapter` instance would perform this serialisation, and would then transfer this byte stream via a suitable IDL interface to the `CNodeAdapter` instance. The `CNodeAdapter` instance could then deserialise the byte stream back into a `DataProfile` instance, and this `DataProfile` object could be passed to any registered `DataSink` objects.

The difficulty with this scenario is that it limits the data transfer operation to Java implementations since only a Java implementation would be able to

reconstruct a `DataProfile` instance from the stream of bytes. This is also applicable to instances of the `DataHeader` class, which would be transferred through object serialisation as well. It was therefore decided to define a set of structures within the IDL definition of the CORBA interfaces. These CORBA structures would be parallels for the data classes described above. The `LocalNodeAdapter` instance would then have to map a `DataProfile` instance to a type that is defined in the IDL interface definition. This would be transferred through the IDL interface to the `CNodeAdapter` instance, which would then transfer the IDL structure back to a `DataProfile`. This is perhaps not as elegant as the Java object serialisation solution, however it has a great advantage in that programs developed using different languages can connect directly into the flow of data. Three IDL structures were defined, a `CDataProfile`, a `CDataHeader` and a `CDataSet`. It might seem that these structures should simply be used instead of the classes described in the above section, however it must be remembered that these IDL structures are specific to CORBA, and should therefore be kept independent from the core infrastructure.

6.6 Module Implementations

This section discusses in more detail the means by which a `Module` implementation connects with the `Bus` and `Node` classes. As was indicated in Figure 6.1, a `Bus` instance can maintain any number of references to `Module` implementations. The `Module` interface by itself does not specify any functionality that directly satisfies any of the use cases described in the requirements analysis. It offers methods that the `Bus` class requires in order to host it, in particular, the methods specified by the `Located` interface.

Any other methods that a particular `Module` implementation wishes to publish must be specified in an interface that extends the `Module` interface. This can be achieved in various ways, and the actual structure employed for each `Module` implementation may differ from implementation to implementation. This is more clearly illustrated by considering certain modules individually.

6.6.1 The Data Persister Module

The data persister module fulfills the requirement that data must be persisted in some form. Such a module should for the most part operate automatically; it should not be necessary for a user to explicitly tell the module that it should persist data. In general, the module should persist data when a session has been completed.

It is possible to define a generic data persister interface that extends the `Module` interface and adds only one extra method. The method `persistDataNow()` overrides the module's automatic behaviour and explicitly persists the data. By defining an interface, it is possible to have many different implementations that use different technologies to achieve the data persistence. Furthermore, it is possible to then provide a CORBA interface as well, and the module could be controlled from over the network.

A simple data persister module was written that would write the data in XML format to a file on the local file system. From Figure 5.1 it should be noted that a module can have a reference to zero or one node. A data persister module clearly requires a reference to a node so that it can gain access to the data buffer. The data persister module therefore implements the methods of the `Module` interface that allow it to be connected to a `Node` reference. Note that this connection is not the same as the connection between `DataSource` and `DataSink` objects. A module is connected to a `Node` reference simply by maintaining a reference to that `Node` instance. The `Node` instance will not physically transfer data to the `Module` instance. Instead, the `Module` reference can get access to the `Node` instance's data buffer through the `getDataReference()` method specified in the `Node` interface. The `Node` interface also provides a means by which a module can register itself to receive notification of when new data arrives at the node. This is simply a notification informing the module that there is new data in the buffer. This saves the module from having to poll the data buffer.

The persister module is simple in that there is little direct interaction required between the module and an end user. In the future however, it might be required that there is a GUI that allows an end user to configure the module in a more intuitive way. This GUI should then be defined in a separate class. The GUI could then access the module through either the `Module` interface, or else through an extension of the `Module` interface. It is also then possible to write the appropriate adapter classes for the module that make the module available on the network. The GUI can then control the module from across the network.

6.6.2 The Data Viewer Module

A data viewer module is a module that provides a visual interpretation of the data on a particular node. The data viewer module is more complicated than the persister module since the viewer module must provide some form of GUI. The GUI should be a compact component that can be treated as other GUI controls within a larger GUI. This larger GUI should be configurable from a script file. Instead of defining one class that implements the `Module` interface directly, it was decided to split the data viewer module. The GUI class itself would simply take a `DataSet` reference and display the data in that `DataSet` object. A separate class would provide the implementation of the `Module` interface, and this class would then control the GUI class. The GUI class is realised by the `DataSetViewer` class, while the `Module` implementation is provided by the `LocalViewerModule` class. This structure allows the `DataSetViewer` class to be dependent only on the `DataSet` class, making it more reusable in the future. The `LocalViewerModule` class does the work of communicating with the `Node` reference and controlling the `DataSetViewer` accordingly.

6.6.3 The Radar Controller Module

The radar controller module is a module that provides an interface specific to the radar. This module does not define a GUI, it rather provides a connection to which a GUI might connect. The connection would be achieved through the controller interface that is defined for the radar.

A radar controller module would be responsible for two main functions. Firstly it must provide an interface to which a radar operator might connect - that is there should be methods that accomplish functions such as telling the radar hardware to take a scan. Secondly, a radar module must be responsible for importing data from the radar hardware into the system. It is therefore likely that there might be a different implementation of a radar module for each different kind of radar supported. However, each of these implementations would implement a common radar controller interface which would be derived from the `Module` interface. For example, in order to support the existing radar hardware, a radar controller module would be defined that controls the existing radar through the existing CORBA interface, but that implements the `DataSource` interface, so that the `LocalNode` instances could connect directly to the radar module. If even older radars were being supported then a radar module would have to be implemented that would perform the serial communications directly with the radar hardware. Future versions of the radar hardware will support TCP/IP directly, and it may even be possible that the radar hardware could implement a CORBA interface compatible with the `DataSource`/`DataSink` paradigm. If this becomes possible, then the radar controller module would no longer be required to import the data from the radar hardware, however the radar controller module would still be required to offer an interface for controlling the radar hardware.

Firstly, a basic interface was defined that could be applied to any type of radar. The methods that make up the interface allow an operator to achieve the following:

1. Start a new session, specifying an opening comment as well as the azimuth step size that will be used for the session.
2. Explicitly take a single scan.
3. Indicate that the radar hardware should start scanning continuously - this allows the hardware to control the rate at which data is acquired.
4. Take a fixed number of scans. A number of scans is specified, and the radar hardware is allowed to set the rate at which the scans are obtained.
5. Stop the radar hardware from scanning continuously. This reverts control of the acquisition rate back to the system.
6. Comment the current scan. It is possible to buffer each scan as it comes from the radar hardware. Instead of transmitting the scan into the data flow immediately, the module can buffer the scan until the next scan arrives. This allows the operator to add a comment to a scan before it is transmitted into the data flow.

These options should be possible regardless of how the radar controller module communicates with the radar hardware. The `RadarControllerModule` interface was defined to support these functions.

A more specific interface is required for the existing radar, since there are other options that can controlled, for example switching on the odometer. Furthermore, since the current radar hardware is controlled via a server that implements

a CORBA interface, it is necessary to connect to this server. The server publishes its location through writing a stringified reference to a file. The radar controller module must be given this stringified reference in order to make a connection to the radar server. These additional methods have been defined in the `MercuryBRadarControllerModule`. This interface extends the `RadarControllerModule` interface. The implementing class for the `MercuryBRadarControllerModule` interface is the `LocalMercuryBRCModule`. Once again, it is useful to split the implementation from the interface in this way, since it is now possible to define adapters that allow the radar controller module to be controlled from anywhere else using the same interface. For example, a GUI will have to be designed for radar operators. This GUI would then only need a `MercuryBRadarControllerModule` reference, however the implementation might be achieved by some CORBA adapters, allowing the GUI to be operated from one host, while the radar controller module is located on a separate machine.

6.7 Scripting Languages

The system relies on configuration scripts that are used to describe the structure of the system. A configuration script would capture all information related to number of nodes, processor parameters and which modules are present, and how the interconnections between components are made. A format for these configuration scripts was required. It was decided that XML should be used for this format since it is a standardised data format. Furthermore, since it is standardised there are parsers already written. A set of XML tags were defined that are specific to the system. Appendix B includes an example of a configuration file in which each tag that has been defined is shown in relation to the rest of the tags.

The configuration script currently is used to specify the logical structure for the system. A script will describe each node along with the processing that must be performed, and it will specify the source to which the node must connect. The script will also describe each module that must be instantiated. All of this information is of a structural nature, describing how system core must be constructed. However there also must be provision for creating a GUI for the system from a configuration script. Currently, this is achieved through the use of JPython. JPython is a language based on the Python language, however JPython is written in Java, and can be used in conjunction with existing, compiled Java classes. JPython allows for rapid prototyping, and as a consequence was used for scripting the GUI components together. Ultimately, a more elegant solution would be to have the entire system, GUI included, described in XML, however this requires that another XML derivative for the GUI components be developed, and this is a more complicated task than developing the XML derivative for the logical structure of the system.

6.8 Conclusion

This chapter described the implementation model for the framework. The core infrastructure describing how data is transferred was discussed. This core infrastructure does not detail the underlying messaging protocol to be used. A structure of adapters was defined that would allow developers to include the use of other messaging protocols such as RMI. The concept was illustrated with a set of CORBA adapters.

The data structure was described in detail, as well as how this data structure is mapped to CORBA structures for transmission across the network.

The implementation of a set of modules was described. The data viewer, data persister and radar controller modules were discussed, as well as how new modules could be written and used. Finally, the roles of the two scripting languages, JPython and XML, within the framework were discussed.

Chapter 7

Results

7.1 Introduction

This chapter discusses the results obtained from the development of the system. Section 7.2 describes the results of the design itself relating to the functionality provided by the system.

Section 7.3 then discusses the performance results that were measured for various scenarios. The results that are listed in this section only offer a very basic idea of the performances that can be expected under certain scenarios, however they are necessary in order to show that the system is viable.

7.2 Design Results

The system described in the preceding chapters was implemented. The core infrastructure along with a complement of CORBA adapters was developed. This core infrastructure, supported by the CORBA adapters, provides the following:

- Data distribution between nodes on local and remote hosts.
- Data processing framework allowing routines to added without the requirement that the system be recompiled. Two reference routines were written (a windowing routine and inverse Fourier transform routine).
- Ability to be configured at startup through a XML configuration script describing the number of nodes and the type of processing required at each node.
- Ability to be configured while running through remote interfaces.

In addition to the core infrastructure that was developed, the following modules were written:

Data viewer A module that is used to display data in various forms. The actual GUI component that the user interacts with was implemented separately to the viewer module so that the GUI component was not dependent on the core infrastructure or the structure of the viewer module. This GUI component is not a stand-alone component and must be included in a larger GUI.

Data persister A persister module was written that takes data from a node and writes it as an XML file.

Radar controller module A radar controller module was written that would connect to the existing Mercury B radar server. This module imports data from the radar server into the system, while offering a remote software interface to which a GUI or command line program could connect.

The XML configuration script that describes the structure of the system at runtime is used to specify which modules should be loaded. When a new module is written, the new class files for that module are included with the existing class files for the system, and the configuration script is updated to include the name of the new module class. The system will then attempt to instantiate the new module when it is started up. This process is the same for writing new processing routines. A routine is written and compiled, and the resulting class files are then added to the existing system class files. The name of the new routine is then used in the configuration script, and the system is able to instantiate the new routine.

7.3 Performance Results

The most important numeric result is the rate at which data can be transmitted through the system. However, such a result will vary widely from application to application, and from environment to environment. This dependence is largely due to the following factors:

Data processing The application in which the framework is used will determine the nature of the processing that must be set up. Since the time requirements for the different types of processing routines vary from routine to routine, the processing configuration will affect the rate at which data can be transmitted in an unpredictable fashion.

Host computing power The power of the computer hosting components of the framework will affect the time taken for processing to occur, and consequently the rate at which data can be transmitted.

Network traffic When framework components are distributed physically, the system becomes dependent on the bandwidth of the network. Since the radar server is accessed across the network, the network bandwidth can limit the data transmission rate at the source.

It was decided that a couple of scenarios would therefore be simulated in order to get an idea of the performance of the system. Firstly, the mercury B radar controller module was used to connect to a radar simulator. No processing was applied at any node. This gives an idea of the fastest rate at which data can be acquired from the radar server. Secondly, a typical scenario was considered that would most commonly be used. Such a typical scenario would consist of the radar controller module, a node at which windowing and inverse Fourier transform processing could be applied, and to which a viewer module could be connected.

The same radar simulator was used for both scenarios, and the same data profile size was used. The size of the data profile was a set of 256 complex floating point samples - resulting in an array of 512 floating point numbers. The tests were completed using two computers interconnected via a 10 Mbit ethernet connection.

7.3.1 Scenario 1

This scenario gives an idea of the fastest rate at which data can be fetched from the radar server. A basic simulator was written in Java that would implement the mercuryB IDL interface. This simulator would simply return a fixed array of data for each scan request. Since there is very little overhead within the simulator, the speed at which each scan can be fetched from the server will not be limited by the implementation of the simulator, but rather by the implementation of the node.

The setup for the scenario consisted of the radar simulator, one node, and the radar controller module. No GUI components were specified since only the components that are essential for importing data into the system were instantiated. A node is required since the radar controller module needs to connect to a node in order to import data from the radar server into the node (the radar controller module appears to the node as a `DataSource`).

Two cases were considered for the configuration of the node. Firstly, no processing was described for the node. The node would simply fetch raw data from the radar controller. Secondly, the standard windowing routine along with the inverse Fourier transform were configured for the node. The XML configuration file for these two tests is given in Appendix A.

The scenario described above was repeated for two different topologies. Firstly, the radar simulator and the node were both located on a single machine, and the two tests were executed. The following results were obtained:

Node Location	Simulator Location	Processing	Profiles/second
Local	Local	none	346
Local	Local	windowing and IFT	322

The scenario was repeated for the case where the simulator was located on one machine, and the node was located on another. The radar controller module then connects to the radar simulator across the network. The following results were obtained:

Node Location	Simulator Location	Processing	Profiles/second
Local	Remote	none	177
Local	Remote	windowing and IFT	171

A decrease in the rate of data transfer is noticed when the connection to the radar simulator is made across the network.

7.3.2 Scenario 2

The second scenario that was considered was a more typical case where a user might wish to see a visual account of the data acquisition process. The windowing routine as well as the inverse Fourier transform routine were also added to this scenario since a visual representation of the data is more useful after this processing has been applied. The scenario hence consisted of the following components:

- Radar simulator
- Radar controller module
- Two nodes
- Data viewer module

Two nodes are required, one to form the buffer for the raw data that the radar controller module acquires from the radar simulator, and one to form the buffer for the processed data. This second node would be the node to which the data viewer module would connect. The XML configuration file for the above scenario is included in Appendix [A].

Once again, two situations were considered. Firstly, all the components above were located on a single machine. This would be an example of how the system might be used in a stand-alone mode, where the radar server is running on the same machine, and the radar hardware is connected directly to the same machine. This situation is similar to many of the current radar setups, where a laptop computer is used to connect to the radar hardware, and a display is shown on the laptop's screen.

The second situation that was considered was more representative of where the system would be better employed. The radar simulator was located on a separate machine simulating the goal of the radar hardware, that it should be connected directly to the network. The rest of the components were then located on a single machine. These components would form the typical set of components that a user would require in order to acquire raw data from the radar, process it, and view the results while writing the data to disk for future reference.

The following table summarises the results that were recorded.

Node Location	Simulator Location	Processing	Profiles/second
Local	Local	windowing and IFT	10
Local	Remote	windowing and IFT	134

In the above table, the Node Location column refers to the entire system consisting of both nodes, controller module and data viewer. Only the location of the radar simulator itself was altered.

7.3.3 Discussion

The results described here are merely introduced to provide some idea of the currently available data rate. Obviously if more powerful computers were used, the data rate measured for these scenarios would increase.

The most noticeable feature of the above results is the significant loss of performance that occurs when the data viewer is used and the system and the simulator execute on a single machine. The reason for this loss of performance is primarily due to the overheads associated with the use of Java. Since both the system itself as well as the radar simulator were written in Java, when they execute in the same host and connect via a CORBA ORB that is also written in Java, there is a lot of contention for the system resources. As the results show, when the radar simulator is moved to a separate host, the data transfer rate is significantly increased. Orfali and Harkey [12] explain this phenomenon when similar results are described in their book. A similar improvement should also be noted if the simulator were written in a native programming language such as C and allowed to execute on the same host as the rest of the system.

One factor affecting the performance for these scenarios is the interface exported by the mercury B version of the radar. This is the interface that was implemented by the radar simulator used in the above scenarios. The mercury B interface is based on a pull paradigm. The radar controller calls a method on the radar server that returns a single profile. This method blocks until the radar server is able to return data. Since the radar hardware will normally be in control of when a scan occurs, a better paradigm would be the push paradigm, where the radar server actually calls a method on the radar controller when data is ready. This is better since the data rate can then be set by the radar server. This is one improvement that will be made in newer versions of the radar.

Currently, the radar hardware is able to produce data at a rate of 10 scans per second. This means that the radar controller must be able to acquire data from the radar server at a rate of at least 10 scans per second. With the mercury B design, this is very important since, if the hardware is producing data at a faster rate than the radar controller can manage, data profiles will be skipped. In future, the radar server will also buffer data, and the consequence of not matching the hardware data rate will not be as severe since the data will be buffered, however the radar controller should still be able to acquire data at the rate at which it is being produced. As is evident from the results, the slowest rate that was measured was also 10 scans per second. This is a very close match to the rate supported by the radar hardware, however it should be noted that the true mercury B radar server is in fact written in C/C++, and hence if the system were configured to execute on a single machine as the second test scenario was, a higher data rate could be supported by the system.

Finally, some comments should be made about the repeatability of the above tests. The data rates were obtained by measuring the time taken for the node

to retrieve two thousand data profiles from the radar controller. This retrieval process was repeated until the average data rate over the retrieval of two thousand data profiles reached a constant. This typically involved running through the set of two thousand data profiles about three times. Typically, the data rate measured for the first set of two thousand data profiles is slightly slower than the rate measured for the second and third sets. Thereafter the data rate reaches a constant. There are two factors that contribute to this behaviour. Firstly, the Java virtual machine incorporates a *Hot-Spot* technology that optimises the execution of the Java byte code while executing the program. This means that if there is a portion of code that is frequently executed, then this portion of code will execute more efficiently as time progresses. Since the process of retrieving data from the radar controller is repetitive, this process becomes more optimised the longer the program executes.

The second factor affecting this performance increase is the recycling of data structures. When the first two thousand data profiles are retrieved, the system needs to create two thousand new objects - a new `DataProfile` object for each data profile that is obtained. However, when the next measurement session is started, the two thousand `DataProfile` objects are not simply discarded, they are recycled to hold the new data profiles that are retrieved. This saves on the object creation time that would otherwise be incurred.

7.4 Conclusion

Two sets of results were discussed in this chapter. Firstly the results of the system design were discussed. The functionality that the system achieved was discussed as part of this set of results. It was found that the design requirements for the framework were met. Secondly, the performance results were discussed. Two scenarios for measuring the performance were considered. The fastest data transmission that was recorded was made when the radar server was executed on a separate machine to the node at which data was being processed. Since this would be the typical deployment for the framework, the data transmission rate was quite adequate - being more than ten times the required data rate of ten data profiles per second.

Chapter 8

Conclusions And Recommendations

8.1 Conclusion

A framework for distributed data capture and processing was developed for ground penetrating radar. The framework provides a means for distributing data obtained directly from a ground penetrating radar to users connected on a network. The framework allows processors to be configured at various points in the network. The framework allows modules to connect to points in the framework. It is these modules that provide behaviour customised to a particular requirement.

The following modules were developed:

- Data viewer module
- Data persister module
- Radar controller module

The framework by itself is not a stand alone application, it requires scripts that define how it will operate. XML is used as the scripting language to achieve this. These scripts are used to customise the behaviour of the framework to suit particular users' requirements.

Two scenarios were considered in which different system configurations were constructed, and an attempt was made to measure the maximum data transfer rate. The fastest rate obtained was 346 scans per second. This rate was obtained when no processing was applied, and only one node was present. This was the fastest rate at which data could be transferred from a radar simulator into the system itself via a CORBA interface. This test was conducted on a single machine. It was found that when processing consisting of a windowing routine and an inverse Fourier transform was configured, and a data viewer module

was configured, the data rate was faster when the simulator was relocated to a remote host. The fastest rate achieved in this case was 134 scans per second. These data rates are suitable for the existing mercury B radar hardware which can produce data at a maximum rate of 10 scans per second.

8.2 Future Work

The focus for the framework was on the infrastructure for processing and communicating data from point to point. In order for the framework to be of use to a wider range of users with different levels of computer experience, more work will need to be applied on the GUI side of the development. The goal is that the GUI will also rely on a configuration script that defines its appearance. An experienced user will then write a script to configure the framework for a particular group of end users. Once this is done, the end users should not have to edit or view the configuration script since the GUI will be customised to their requirements. Presently, this is achieved through the use of JPython, however ideally the same language should be used for all configuration scripts.

The framework was developed to support ground penetrating radar data. The interfaces that were specified for the data transfer mechanism rely on a data type that was defined specifically for GPR data. The reason for defining a specific GPR data type was that it should be more efficient since no parsing would be necessary. However, this is restrictive in that only data that can be formatted as GPR data can be transferred in the framework. An alternative would be to transfer data in XML format. The advantage to this would be that completely new data formats could be described within the XML message, and the framework would not have to be concerned about this. The disadvantage would be the extra overhead required for parsing XML data and formatting data as XML. These advantages and disadvantages should be compared, since it is possible that the advantage in abstracting the data format may outweigh the disadvantage of the overhead required for parsing. This might be especially true for cases where the size of the data is small.

Appendix A

Configuration File Listings

The XML configuration files used for testing the system are included here.

Scenario 1

The following file was used to configure the system for the first scenario. The processor section of the following listing was commented out for the case where no processing was applied, but otherwise the listing was unchanged even though the radar simulator location was altered.

```
<?xml version='1.0' encoding='us-ascii'?>
  <bus name="Bus1A">
     <portal name="CORBAPortal" class="gpr.CORBA.CORBAPortal" publishbus="true">
      <parameter name="orbConfigFileName" value="orbconfig.cfg"/>
     </portal>
     <node name="radarnode">
      <processor>
        <routine class="gpr.local.WindowRoutine">
         <parameter name="type" value="rectangle"/>
        </routine>
        <routine class="gpr.local.InverseFTRoutine">
         <parameter name="points" value="256"/>
        </routine>
      </processor>
      </node>
     <module name="MercBController" class="gpr.modules.LocalMercuryBRCModule">
      <sourcenode node="radarnode"/>
      <parameter name="stringifiedreference.file" value="mercuryB.ref"/>
     </module>
  </bus>
```

Scenario 2

The following file was used for the configuration of the system for the second scenario. Again, the same file could be used for both cases where the location of the radar simulator was altered.

```
<?xml version='1.0' encoding='us-ascii'?>
  <bus name="Bus2">
     <portal name="CORBAPortal" class="gpr.CORBA.CORBAPortal" publishbus="true">
      <parameter name="orbConfigFileName" value="orbconfig.cfg"/>
     </portal>
     <node name="radarnode">
     </node>
     <node name="processednode">
      <sourcenode node="radarnode" bus="Bus2"/>
      <processor>
        <routine class="gpr.local.WindowRoutine">
          <parameter name="type" value="hamming"/>
          <parameter name="beta" value="8.2"/>
        </routine>
        <routine class="gpr.local.InverseFTRoutine">
          <parameter name="points" value="256"/>
        </routine>
      </processor>
     </node>
     <module name="MercBController" class="gpr.modules.LocalMercuryBRCModule">
      <sourcenode node="radarnode"/>
      <parameter name="stringifiedreference.file" value="mercuryB.ref"/>
     </module>
     <module name="viewer" class="gpr.modules.LocalViewerModule">
      <sourcenode node="processednode" bus="Bus2"/>
     </module>
  </bus>
```

Appendix B

XML Configuration File Skeleton

The following listing shows a skeleton version of a configuration file.

```
<?xml version='1.0' encoding='us-ascii'?>
  <bus name="BusA">
     <portal name="CORBAPortal" class="gpr.CORBA.CORBAPortal"
      publishbus="true/false"/>
      <parameter name="orbConfigFileName" value="orbconfig.cfg"/>
     </portal>
     <node name="NodeA">
      <portal name="CORBAPortal"/>
      <sourcenode node="NodeX" bus="BusX"/>
      <processor>
        <routine class="gpr.local.WindowRoutine">
          <parameter name="type" value="hamming"/>
          <parameter name="beta" value="8.2"/>
        </routine>
        <routine class="gpr.local.InverseFTRoutine">
          <parameter name="points" value="256"/>
        </routine>
      </processor>
     </node>
     <module name="MercBController" class="gpr.modules.LocalMercuryBRCModule">
      <sourcenode node="radarnode"/>
      <parameter name="stringifiedreference.file" value="mercuryB.ref"/>
     </module>
     <module name="viewer" class="gpr.modules.LocalViewerModule">
      <sourcenode node="processednode" bus="Bus2"/>
     </module>
     <module name="persistor" class="gpr.modules.LocalFSDataPersistorModule">
      <sourcenode node="NodeA"/>
      <parameter name="directory" value="/home/allen/projects/measurements"/>
```

```
    </module>
  </bus>
```

The following table lists the tags and their attributes:

Tag	Description	Attributes	Description	Rqrd?
`bus`	Indicates the presence of a bus.	`name`	Used as the identifier for the bus on the network	yes
`portal` (1)	This tag has a slightly different meaning depending on its context. In this context it must appear as a direct leaf of the bus tag. It then specifies a portal that the bus must instantiate.	`name`	Used as a simple identifier for the portal, so that it can be referred to later by name.	yes
		`class`	Specifies the absolute class name for the required portal.	yes
		`publishbus`	Specifies whether the bus should be made available on the network or not. It's value can be `true` or `false`	yes
`node`	Specifies the presence of a node	`name`	The identifier for this node on its host bus	yes
`portal` (2)	In this context, the tag appears within an enclosing `node` tag. It is then used to specify which portals the particular `node` must be published through.	`name`	Indicates which of the declared portals must be used to publish this node.	yes
`parameter`	A tag used to specify a named parameter. The parameter is generally required by the enclosing tags.	`name`	The name of the parameter.	yes
		`value`	The value of the parameter.	yes
`sourcenode`	Used to specify the source to which the enclosing tagged object should connect. This tag is used for both `nodes` and `modules` to indicate which node should be the source.	`node`	The name of the source node. This name can refer to a remote node, as long as the remote node's bus is also specified by the next attribute.	yes
		`bus`	The bus on which the source node occurs. If this attribute is left out, then the source node is assumed to occur on the local bus.	no

Tag	Description	Attributes	Description	Rqrd?
`processor`	Specifies the processor. This tag is used to enclose any number of `routine` tags.			
`routine`	Specifies a routine. The order in which routines are listed reflects the order in which the routines are applied. The `parameter` tag should be used to list routine specific parameters.	`class`	The fully qualified class name of the implementing class.	yes
`module`	Specifies a module.	`name`	Specifies the name of the module so that it can be identified and referred to.	yes
		`class`	The fully qualified class name of the implementing class.	yes

Bibliography

[1] Fowler, M. *UML Distilled*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 2000.

[2] Jacobson, I. *Object-Oriented Software Engineering, A Use Case Driven Approach*, Workingham, England: Addison-Wesley, 1993.

[3] Rumbaugh, J. et al *Object-Oriented Modeling and Design*, New Jersey, USA: Prentice Hall, 1991.

[4] Hutt, A. T. *Object Analysis and Design: Description of Methods*, New York, USA: John Wiley & Sons, Inc., 1994.

[5] Beck, K. Embracing Change with Extreme Programming, *IEEE Computer* Vol 32 issue 10 page 70-77.

[6] Booch, G. *Object Oriented Analysis and Design with Applications*, 2nd ed. California, USA: The Benjamin/Cummings Publishing Company, Inc., 1994 .

[7] Nierstrasz, O & Tsichritzis D, *Object Oriented Software Composition*, Hertfordshire, UK: Prentice Hall, 1995.

[8] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, New Jersey, Prentice Hall Inc. 1998

[9] Naughton, P & Schildt, H, *Java 2: The Complete Reference*, Berkeley, California: Osborne/McGraw-Hill, 1999.

[10] Gamma, E. et al *Design Patterns*, USA: Addison Wesley, 1995.

[11] McLaughlin, B. *Java And XML*, USA: O'Reilly & Associates, 2000.

[12] Orfali, R & Harkey, D. Client/Server Programming with Java and CORBA, New York, USA: John Wiley & Sons, Inc. 1998.

[13] Noon, D. A. *Stepped-Frequency Radar Design and Signal Processing Enhances Ground Penetrating Radar Performance*,PhD thesis, University of Queensland, 1996

[14] Farquharson, G. *Design and Implementation of a 200 to 1600 MHz, Stepped Frequency, Ground Penetrating Radar Transceiver*, M.Sc. thesis, University of Cape Town, 1999

[15] Wehner, D. R. *High-Resolution Radar, 2nd Edition*, Norwood, UK: Artech House, 1995

[16] Mensa, D. L. *High Resolution Radar Cross Section Imaging*, Norwood, UK: Artech House, 1991