RHINO ARM Cluster Control Management System

by

Valerie Edith Chiriseri

Supervisor: Dr Simon Winberg

Co-supervisor: Prof. M.R. Inggs

May 14, 2014

Declaration

I understand the meaning of plagiarism and declare that all work in the dissertation, save for that which is properly acknowledged, is my own. This dissertation is being submitted in fulfillment of the requirements of a degree of Master of Science in Electrical Engineering at 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

South Africa

May 14, 2014

Abstract

The purpose of this project is to design, develop and implement a software framework that aims to increase the usability of a bespoke reconfigurable platform called the Reconfigurable Hardware Interface for computiNg and radiO (RHINO) to non experienced reconfigurable computing programmers, by providing a remote platform control and cluster management framework. The RHINO platform was designed by the Software Defined Radio Group (SDRG) at the University of Cape Town; this platform and its computing operation, is planned for use for developing Radio Astronomy (RA) and Software Defined Radio (SDR) applications in an educational setup. The framework aims to abstract the RHINO's functionalities and will be the first software application that will be designed to run specifically on this board.

The framework for the RHINO will be implemented as a multi-server single-client application with an Application Programming Interface (API) designed to run on both applications. The server applications would run on the RHINO boards and the client application will run on the control computer. Users will be able to control and manage a collection of networked RHINOs through the API, with the API designed to focus on development of Linux-based C code. The developed framework consists of six main parts, the server and client applications, two API libraries that will be hosted on the server-client application, a control protocol and lastly a user interface to enable ease of use of the framework.

The evaluation of the framework was done through a series of experiments; starting with tests that evaluate the implementation of the code starting from a cluster that has a single RHINO connected to the control computer's client application. This was then followed by black box testing that determined if the developed framework met all of its specified requirements. The last set of experiments involved expanding the system to a larger networked system involving three RHINOs each running the server application connected to the client application. The results include a reflective discussion of the main challenges encountered during the design, and effective solution strategies used to work around these. The conclusion provides a summary of the findings and discusses future work that will take the project further for applications in RA and SDR.

Acknowledgements

- *To the Lord God almighty for whom without Him, none of this would have been possible.*
- *To the National Research Fund (NRF) through the Square Kilometer Array (SKA) who funded this project.*
- *To my supervisor Dr Simon Winberg for his continual support and positive feedback.*
- *To my co-supervisor Prof Michael Inggs for direction on how to approach and structure my project.*
- *To Dr Marc Welz, Shanly Rajan and his family and the SKA team for their time and advice which made a huge difference in this project.*
- *To my family and friends who encouraged me when I felt I could not continue and for all their support and guidance.*

Contents

List of Figures

List of Tables

Nomenclature

RACCMS - RHINO ARM Cluster Control Management System

API - Application Programmable Interface

SDR - Software Defined Radio

UCT - University of Cape Town

SKA SA- Square Kilometer Array South Africa

RHINO - Reconfigurable Hardware Interface for computiNg radiO

ROACH - Reconfigurable Open Architecture Computing Hardware

USRP - Universal Software Radio Peripheral

FPGA - Field Programmable Gate Array

ARM - Advanced RISC Machine

BORPH - Berkeley Operating System for Re-programmable Hardware

ADC - Analogue to Digital Converter

DAC - Digital to Analogue Converter

OPB - On-Peripheral Bus

OSI - Open System Interconnection project

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

Chapter 1

INTRODUCTION

The purpose of this dissertation is to present the design, development and implementation of a framework that will enable the management and control of specific reconfigurable computer platforms from a remote location. This will be achieved through the development of the Reconfigurable Hardware Interface for computiNg and radiO (RHINO) ARM Cluster Control Management System (RACCMS) framework which will be utilized at the University of Cape Town. The RACCMS framework is the subject of this dissertation.

This chapter will begin with a background and motivation to the project, followed by the the presentation of the project scope and then the objectives of the dissertation and lastly the chapter closes with an overview of the contents of dissertation.

1.1 Background of Investigation

With South Africa's recent success in being awarded the bid to build the largest telescope in the world, there has emerged a growing need to develop local technical knowledge and skills in the back-end processing of the astronomical data using reconfigurable computing platforms; that is there is a need to invest in local human capital is required to successfully run the SKA project. With this increase in usage of RCs and the human capital need created by the SKA SA, the University of Cape Town developed the Reconfigurable Hardware Interface for computiNg and radiO (RHINO) in order to provide an affordable platform that would be used as a teaching aid in Software Defined Radio (SDR) and Radio Astronomy [66]. It was mainly designed to grow local South African skill and innovation by allowing inexperienced and experienced designers to develop and prototype radio systems and SDR applications on it [66][57].

The RHINO board is a FPGA based platform with an ARM processor that runs the BORPH OS, which is a Linux based operating system with the kernel modified for RC

platforms [35]. It is an open source project currently running at the University of Cape Town [57]. The board, which is still in its prototype stage, is a custom designed processor board and a hybrid platform for which its current tool-flow is still under development [44]. Limited documentation is available on how the board operates or on how to configure and debug it. It is also important to note that the board currently has no management or control software running on it and hence makes it only usable by low-level programmers with experience with such boards and development environments. However in order to make the board accessible to users who are not highly experienced in using and controlling such platforms, a framework was required in order to abstract the complexities involved in using such a board.

1.2 Project Scope

Reconfigurable Computers (RCs) are computer architectures that have one or more general purpose processors coupled with one or more reconfigurable fabrics [60]. Their increase in popularity in fields that require high processing power has been attributed by some performance bottlenecks that have been reached by the Von Neumann architecture PCs [54]. The processing power on RC platforms is provided through the FPGAs that perform all the heavy lifting processing [35] and they provide processor control on the same platform which offers the same flexibility of a general purpose computer architecture. Even though RCs have many promising characteristics such as their high computational power, many users are still reluctant to use them due to a lack of experience in programming and controlling such platforms [34]. This has resulted in a need for high level design languages, simple development environments and software that enables easy learning and use of these RC platforms [56]. The aim of this project is to develop a framework specifically for the RHINO board that improves the usability of the board for university students with little to no experience using such a platforms when developing SDR and Radio Astronomy applications. The framework developed in this project will be the first software to run on the RHINO board and will be used mainly for teaching and academic purposes. The framework will also assist in the management of RHINO boards connected together in a cluster and will enable the control of running gateway designs on the board through the ARM processor. It should be noted that a gateway design is The abstraction provided through the framework for communicating with the boards peripherals and registers will increase the current usability of the board in the teaching and academic community.

1.3 Objectives

The main objectives were set at the start of this project and formed the basis from which the user and functional requirements were derived from.

The main objective of this project was to design and implement a software based framework that would run on the RHINO board and enable inexperienced users to utilize the board. From this main objective, the first sub-objective derived from it was to design and implement a simple method for controlling set parameters on the FPGA, from the ARM processor, through the framework remotely. The second sub-objective was to design and implement an abstracted means of allowing users to program the FPGA from the ARM processor using the same given framework. The last sub-objective was to implement a cluster management system to enable the utilization of many RHINO boards at any given time.

From the objectives mentioned above, several user requirements and functional requirements were derived to make sure that all given objectives were met.

1.3.1 User Requirements

The following user requirements were specified for the development of framework and are labeled [U1] to [U7] for easy reference throughout the rest of this dissertation. The user requirements were developed by Dr Simon Winberg, Professor Michael Inggs from the UCT Software Defined Radio Group and Square Kilometer Array representatives. The defined user requirements are as follows:

U1. The RACCMS framework should allow users to write user defined "C" based code to access and manipulate the RHINO ARM and FPGA. This would be done through a library comprising of the framework's API functions that can be included in the user's code using *"#include<header file>"*.

U2. User should be able to access the RHINO from a remote location. This should be achieved through the design and implementation of a form of server-client application that would enable access to the RHINOs from a remote location. Secondly, this user requirement would involve the development of a control protocol between the RHINO in the cluster and a control computer to facilitate the communicate and exchange data.

U3. User must be able to read and write to register values while a gateway process is running on the FPGA, from a remote location. This functionality should also include the ability to read and change one bit of a given register value using a few network operations.

U4. User should be able to load configuration settings onto the FPGA, i.e. to load a ".bof" file onto the RHINO using the developed framework.

U5. User should be able to select a RHINO or a set of RHINOs from the cluster. The allocation of RHINOs should be done using the management aspect of the framework.

U6. The cluster management system must have some form of a lock manager, in order to lock a RHINO in the cluster for use.

U7. User must be able to query the status of a particular RHINO node. This would entail being able to check that the RHINO is up and running on the cluster, whether it has been locked for use by another user, and lastly to be able to check if there is a hardware process running on it or not.

Figure 1.1 below provides an overview of how the framework would be expected to operate within a cluster set up in order to meet all the user requirements specified above.

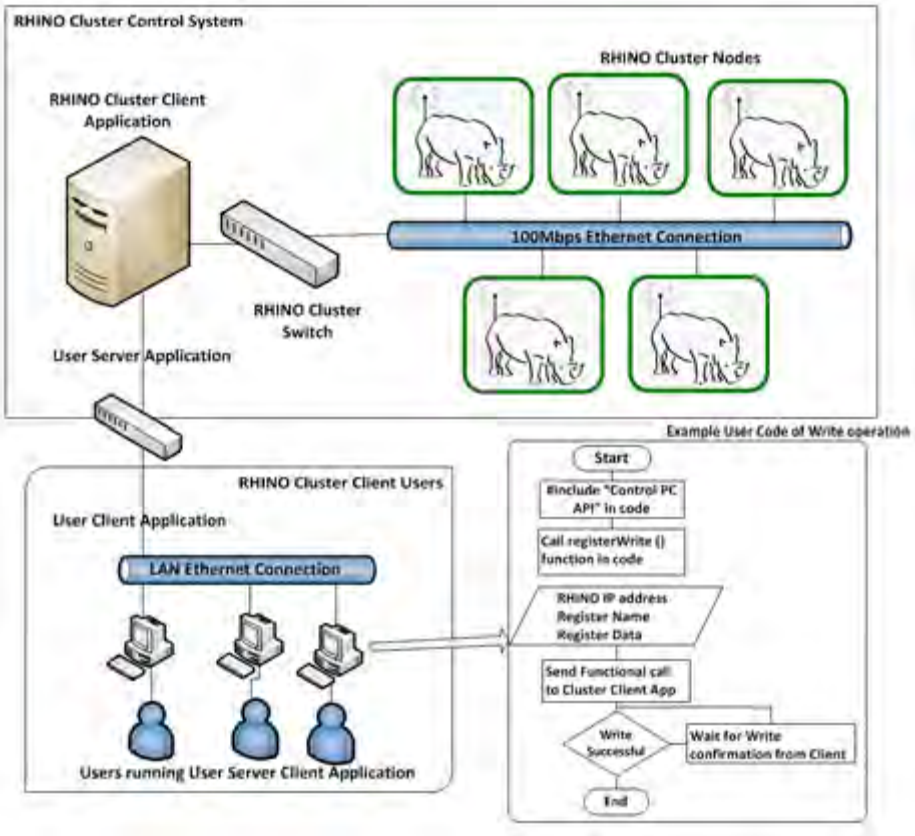


Figure 1.1: Overview Illustration of the RACCMS Framework and how it is to be utilized within a RHINO cluster

Figure 1.1 illustrates five RHINO boards connected to a control computer using an Ethernet connections through a switch. It also shows how students sitting in a remote lab will be able to develop applications that include the framework's API functions, and then executes them on a selected RHINO/ RHINOs using the server-client application. The framework developed in this project was dubbed the RHINO ARM Cluster Control Management System (RACCMS) and will be called as such from here on.

1.3.2 System Functionality

The following functional requirements were formulated based on the set user requirements mentioned in the above section. They have been labeled [F1]-[F7] for easy refer-

ence throughout the rest of this dissertation. These functional requirements will form the basis on which the main building blocks for the RACCMS framework will be built on and are as follows:

F1. Server-Client Application This application will be used to access the RHINO boards from a remote location. The server application will be hosted on the ARM processor of the RHINO platform while a single client application will be implemented on the control computer.

F2. FPGA Programming This functionality would involve configuring the FPGA on the RHINO board through its serial programming interface with a user designed gateway file. Only one gateway file will be able to execute at any given time.

F3. ARM-FPGA Control and Communication A means to control and communicate with the FPGA from the ARM. This could be done using a block driver to enable the framework to access and manipulate the registers on a running gateway file. It is important that the block driver should have a standard interface in order to maintain compatibility with existing code and to ensure that the framework will work with other concurrently running RHINO projects.

F4. DHCP Client Will enable each RHINO platform to get a unique static network IP address when the board is starting up. The DHCP client will connect to the DHCP server on the control computer and be handed its unique IP Address based on the board's MAC address.

F5. API library functions The API will be implemented in two forms, one for the server application sitting on the RHINO board and the other one for the client application sitting on the control computer. The API functions will be accessible to users as a C based library which will consist of all the control and management functions implemented by the RACCMS framework.

F6. Control Protocol This will define standard network packet structures which the client on the control computer and the server application on the RHINO will use to interact and exchange commands with.

F7. User Interface This will be implemented in the form of a user friendly GUI (graphical user interface) which will be used to interact with certain framework functions. It should be noted that not all API functions will require GUI interaction, only a portion will.

Figure 1.2 illustrates all the functional blocks that were to be implemented on the RHINO board and shows how they would function together within the RACCMS framework.

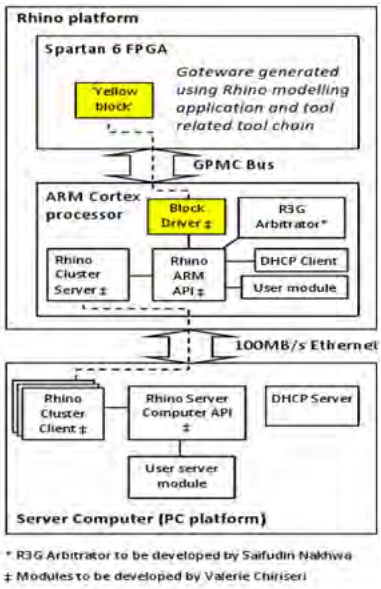


Figure 1.2: Overview of the RACCMS represented in a block diagram. This diagram shows where the different functional blocks will sit on overall system. It also shows the RHINO cluster Client and the RHINO Cluster Server API library. Blocks tagged with "‡" are to be developed specifically for the RAACMS framework

Table 1.1 below shows how the functional requirements listed above meet the user requirements. It can be seen from the table that each functional requirement would meet one or more user requirements.

Function Number	Function Description	User Requirements
F1	Server-Client Application	U2
F2	FPGA Programming	U4
F3	Block Driver	U3
F4	DHCP Client	U2
F5	API Library Functions	U1, U3, U4, U5 and U7
F6	Control Protocol	U3, and U4
F7	User Interface	U1

Table 1.1: Functionality summary Table

1.4 Project Limitations

The framework to be designed will be a multiple server-single client framework. The RHINO board will be set up to host the server application as it will be the device that provides a service to the users of the framework. The client application will be designed to be hosted on the control computer. The control computer's client application will enable users of the framework to request for services from the RHINO board through its server application.

The framework setup can be thought of in two major systems namely system 1 and system 2 which are illustrated in Figure 1.3 below.

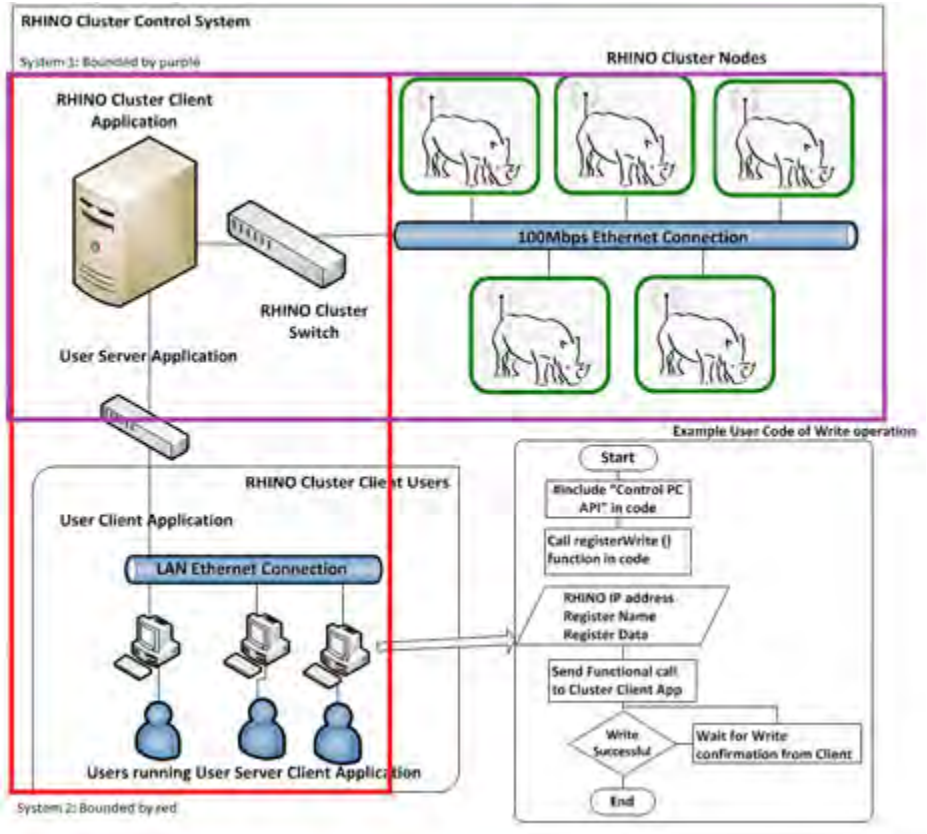


Figure 1.3: Illustration of the RACCMS framework with the two different systems defined

System 1 can be seen in the Figure 1.3 bounded in purple. This system consists of the control computer running the cluster client application and the RHINO boards connected to the control computer. This represents a server-client application, where the server runs on the RHINO's ARM processor, which will provide services from the FPGA to the client application running on the control computer. The control computer requests for services from the RHINO. Such services that can be requested from the RHINO include reading and writing to a register or starting and stopping a gateway design on the FPGA. In this case the system would have a single-client multiple-server application.

System 2 is the one bounded in red in Figure 1.3 which also can be modeled as a server-client application. This system consists of users or students sitting in a lab in a remote

location logging into the control computer sitting in a server room. The server application will be implemented on the control computer, as it would provide services to the users that log into it in order to utilize the RHINO boards, the control computer is connected to. The users would all run client applications in order to connect to the server running on the control computer. This system would result in a single server-multiple client application.

The focus of this project was restricted to the development and design of System 1 which is the system bounded in purple.

1.5 Dissertation Overview

The last section in this chapter describes the layout of the rest of this dissertation and provides a brief summary of each of the steps that were taken in the execution of this project. It includes summaries of each of the chapters and the work presented in them.

Chapter 2

This chapter covers the relevant literature background on which this project was based on. The chapter begins with a section on Reconfigurable Computing (RC) which presents an overview of reconfigurable computing, FPGAs and their design, common RC platforms and lastly presents an RC based OS. The next section covers cluster systems and includes an overview of RC cluster systems as well as an overview of Beowulf type cluster systems and their uses. This is followed by a discussion on two software and hardware based cluster control and management systems and then common cluster network topologies. Lastly the section concludes with a presentation of cluster communication methods in RC systems. The literature review chapter ends off with a section on communication networking. In this section the basics of network layers are presented followed by the highlighting of differences between TCP and UDP transport layer protocols. The definition and explanation of network ports and sockets is then discussed and the section ends off with a presentation on the dynamic host configuration protocol (DHCP) and its uses.

Chapter 3

This chapter presents an overview of the RACCMS framework design process and shows the steps taken in the execution of this project. This includes the phases the project took and it also presents a section that discusses the testing procedures that were used to test the framework. It follows all the processes that were undertaken from the formulation of the user requirements to the final tested project and conclusions. The design process that

was used in this project was the waterfall model and Figure 1.4 illustrates this process structure.

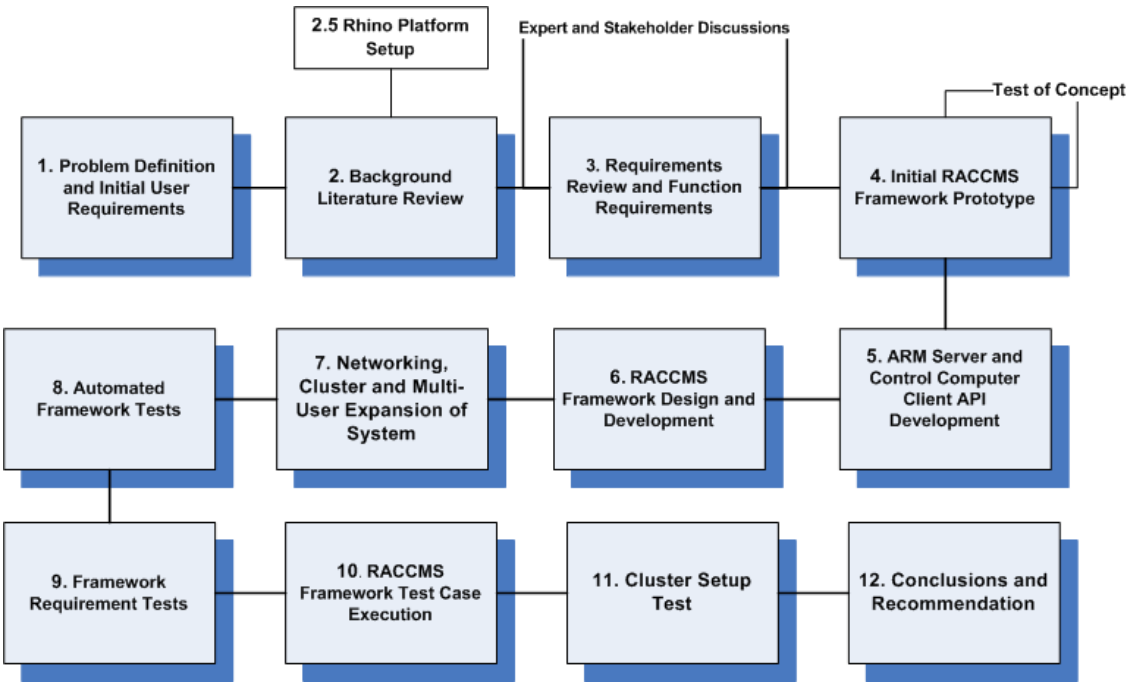


Figure 1.4: Overview of Project Development

Each of these 12 steps illustrated above are discussed in detail in this chapter. It concludes with the final full system tests that will be implement for the framework.

Chapter 4

This chapter discusses the framework setup and system configurations required for the project. It starts off with a review of some of the requirements specified in chapter one, followed by a presentation on some of the design constraints that were placed on the project. The chapter then presents the steps involved in setting up the RHINO board to a point where the developed framework would be able to execute on it. This section is then followed by a discussion on how to design the gateway files using the BORPH OS and how the framework would utilize given aspects of this process. The chapter then concludes with a discussion on how BORPH handles hardware-software communication that will be used by the framework.

Chapter 5

This chapter presents the RAACMS framework's software design and architecture which is presented in five main stages. The chapter starts with a discussion of the framework's

use case diagrams which were stage one of the framework's design. This is then followed by a presentation on the framework's class diagrams that discusses both the client and server application function classes. The chapter then presents the framework's sequence diagrams that describe how the each of the RACCMS function calls utilize the classes to perform a given functionality. The third step is then discussed which presents the framework's flow and state diagrams and then the chapter presents how all the designs were implemented as code. The chapter then concludes with presentation of the RACCMS graphical user interface which is an alternative interface to the command-line interface.

Chapter 6

In this chapter the tests described in chapter 3 are implemented and the results from these tests are discussed. The tests are implemented in 2 main sections, white box testing of the framework code and the black box testing. The white box testing includes a discussion on the framework's unit tests followed by a section which presents the RACCMS integration tests. The white box section then discusses the frameworks boundary and state diagram tests. Lastly the section concludes with the memory tests that were performed on the framework to make sure no memory leaks were present. The black box testing section includes discussions on network integrity test, database tests, transaction tests, function performance tests and ends with a presentation of user acceptance tests. The chapter concludes with the testing of the graphical user interface and the framework using a test case application called the gmpc_test.

Chapter 7

The final chapter of this dissertation covers the conclusions that were drawn in order to determine the success of the project. This chapter will provide a comparison of the results from the tests and results chapter and compare them with the user requirements and project objectives presented in chapter 1 to determine whether or not they have been met.

This chapter closes with a discussion on future work that can be implemented to enable the system to be more versatile and usable for the development of SDR applications in a university setup.

Chapter 2

LITERATURE REVIEW

This chapter presents a background to the important technologies, techniques and theories on which this project was built on. The chapter begins with a discussion on reconfigurable computing. In this section an overview of the topic is presented followed by a section on what FPGAs are and how they function. Section 2.1.2 proceeds to describe common reconfigurable platforms available to date, and this is then followed by a discussion on an Reconfigurable Computing (RC) based OS that runs on the mentioned reconfigurable platforms. The next section 2.2 presents a background to cluster systems with an overview on Beowulf as well as reconfigurable clusters. A description of popular cluster topologies is then given, followed by cluster communication methods used in most RC systems. The last section in this chapter, section 2.3, presents a review on communication networking. In this section networking layers are discussed as well as the transport layer communication protocols TCP and UDP. This is then followed by the presentation of network ports and sockets and lastly the concepts behind dynamic host configuration protocols are given.

2.1 Reconfigurable Computing (RC)

A reconfigurable computer platform is defined as a computer architecture that combines the flexibility of software with high performance hardware. It offers vendors the ability to produce peak performance values and increase application performance by several orders of magnitude as compared to standard microprocessors [37]. The first part of this subsection gives an overview of reconfigurable computing and then proceeds to discuss FPGAs, which most RCs implement as the high speed computing fabric of the platform. The subsection then proceeds to discuss 4 reconfigurable platforms that are common to date, followed by an operating system that was designed to make using reconfigurable computers easier.

2.1.1 Reconfigurable Computing: An Overview

Reconfigurable Computing is increasingly becoming a popular topic in many disciplines that require major computations at accelerated speeds. Their ability to fulfill this niche is because they offer a combination of hardware, that is able perform large parallel applications at very fast rates, and a microprocessor that handles all the software requirements of the application. A Reconfigurable Computer (RC) is a system that consists of one or more general purpose processors that is coupled with one or more reconfigurable fabrics [60][37]. RCs are designed to bridge the gap between hardware and software, in order to produce better performance than software and yet still maintain the higher level of flexibility of hardware [25]. RC systems have many advantages such as speed in computations, reduced energy consumption and power consumption in comparisons to microprocessors and general-purpose computers. The RC system can be split into two parts the hardware part that is the reconfigurable logic and the software part, which is the general-purpose processor [25]. The two sections of the RC system are connected through a system I/O bus. The data bandwidth in this I/O bus is usually the bottleneck of the RC system [33].

2.1.1.1 General Purpose Microprocessor

General-purpose microprocessors perform the operations that reconfigurable logic cannot perform efficiently. Such operations that are not efficiently executed by the programming logic include variable-length loops and branch control. General-purpose microprocessors are also used to control the reconfigurable logic at start up or during execution. The processors can be a physical processor that is made of hardware or be a soft processor. A soft processor is one that is software based and programmed as needed by the application using the reconfigurable hardware[33].

2.1.1.2 Reconfigurable Logic

The Reconfigurable Logic performs the calculations for applications that are computationally intensive. It consists of reconfigurable Functional Units (FU), reconfigurable interconnections and an interface that will connect the fabric to the overall system [60]. The reconfigurable FUs can be classified into two forms based on their granularity:

i. **Coarse Grained:** Reconfigurable fabric that is of this form is larger than that implemented with fine-grained FUs. It consists of arithmetic and logic units (ALU) and a lot more storage than what fine-grained systems provide.

ii. **Fine Grained:** Typically implement a single function or single bit. They are mostly in the form of Look Up Tables (LUT) that determines the logic of the FU. It should be noted

that reconfigurable fabric made of LUT are highly flexible and used to implement digital circuits.

Reconfigurable systems have been around since the mid 1980's [51]; however there are other alternatives solutions to handle computationally intensive applications. Such systems include Application Specific Integrated Circuits (ASIC), which are circuits designed to perform a specific computation. They are hence much faster and efficient when they execute the exact computation they are designed for [25][60]. However a disadvantage is that they are costly to alter after fabrication. Another alternative to RC systems is high performance microprocessors that offer much more flexibility than the ASICs. They are programmed to execute a set of instructions to perform given computations. However they have high execution overhead for each operation as it involves reading and decoding the command before execution.

2.1.1.3 Field Programmable Gate Array (FPGA)

FPGAs are defined as "A chip composed of an array of configurable logic cells (also called logic blocks). Each can be configured or programmed to perform one of a variety of tasks. FPGA logic cells can be used as building blocks to implement any kind of functionality desired from low-complexity state machines to complete microprocessors" [22]. The introduction of the first Static Random Access Memory (SRAM) based FPGA by Xilinx was in the mid-1980s [51] and with their development, they have ushered in the era of reconfigurable devices. They were designed to be the intermediate device between the Programmable Array Logic (PAL) and the Mask Programmable Arrays, in that they are fully electrically programmable and yet still be able to perform difficult computations per processing chip [25].

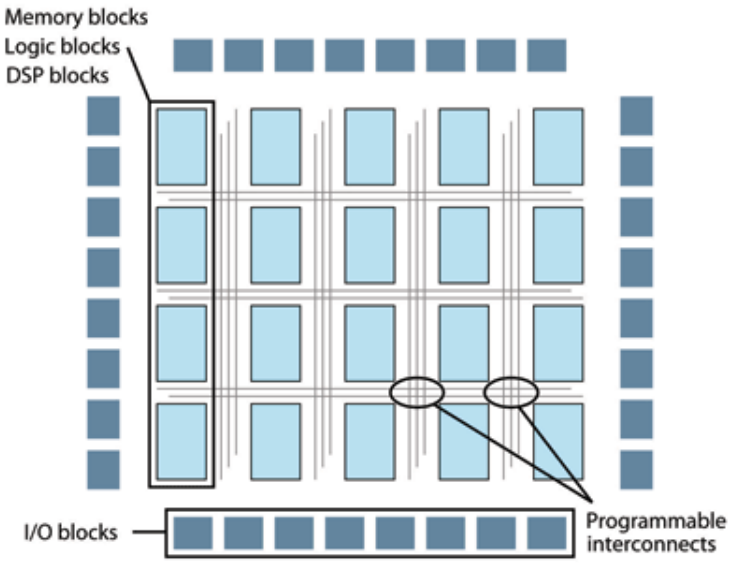


Figure 2.1: FPGA Architecture (Image courtesy of [7])

As can be seen in Figure 2.1 above, an FPGA consists of a number of logic cells, configuration logic block slices, Digital Signal Processing (DSP) slices, BRAM blocks, memory controller blocks (MCB), and I/O banks. The logic cells make up the most basic programming element in the FPGA. They contain a single look-up table (LUT) and flip-flop. It should however be noted that the logic cell metric is rarely used anymore in industry as the Configurable Logic Block (CLB) slices are the preferred logic for measuring logic resources in FPGAs [53][67]. The CLB is made of two slices where each slice has four LUTs and eight flip-flops and represent the FPGA die structure. The DSP slices handle the digital signal processing in the FPGA and contain 18x18 multipliers, an adder and accumulator. The FPGA also has 18KB of Block RAM (BRAM) built in it as well as MCB, which is specialized circuitry on the FPGA used for interfacing with with external SDRAM integrated circuits. Lastly, the FPGA has I/O banks which are a group of user I/O pins powered from a command voltage rail. It is important to note that the larger the number of user I/O banks available the bigger the pool of user I/O pins.

2.1.1.4 Reconfigurable Computing in Radio Astronomy

Radio astronomy can be defined as the identification and study of celestial bodies, by detecting radio waves emitted by these bodies using radio antennae [19][53]. This means that radio astronomy applications require hardware that is able to process several gigahertz of bandwidth in real time. FPGAs, as well as reconfigurable computers that have FPGAs, have the parallel nature that enables them to process large amounts of data in real time, making them ideal hardware for radio astronomy applications. There are FPGA based devices used in radio astronomy to perform correlations, beamforming and wideband spectroscopy operations [19]. With each radio astronomy instrument taking 3-5 years to design, construct and deploy, the use of FPGA based platforms offers an opportunity to shorten development time.

2.1.2 Reconfigurable Computing Platforms

This subsection discusses four popular reconfigurable platforms used in a different fields from software defined radio to radio astronomy.

2.1.2.1 USRP

The USRP platform is FPGA based board designed specifically for software defined radio by Ettus Research and stands for Universal Software Radio Peripheral (USRP)[28]. It comes in two models, the N200 and the N210. These two flavors of the USRP boards can support a wide range of RF front-end devices also known as daughter boards, therefore

allowing the USRP to handle applications up to frequencies of 6Ghz [53]. Figure 2.2 shows a picture of the USRP N210 board.

Figure 2.2: Picture of USRP N210 Board [13]

The N200 and the N210 both have low cost Xilinx Spartan 3A FPGAs on them. The N200 USRP has a MicroBlaze soft core processor on it which is a RISC processor programmed into the FPGA fabric. This soft core processor is used to manage the boards Ethernet communication, configure the daughter boards and control the flow of data to the FPGA itself. The soft core processor can also be used to process software data by running C programs on the processor. However it should be noted that neither of the two versions of the USRP contain a physical processor. The boards also have in built ADCs and DACs which enable RF front ends to be placed on them.

A major advantage of this board is that it is relatively cheap, costing USD$1700 per board. As well the use of a soft processor as opposed to a physical processor enables a more flexible interface between the FPGA data path and the processor. However a disadvantage of this board is that because it uses a soft processor that utilizes one third of the logic and memory of the FPGA, the amount of signal processing that can be performed by the FPGA is thus limited.

2.1.2.2 BEE4

The Berkeley Emulation Engine 4 (BEE4) is a full-speed FPGA prototyping platform. The BEE project was started to enable a speed up in development on new processor architectures. The BEE4 is the fourth generation BEE and is said to “enable researchers to rapidly prototype a variety of architectures in a relatively short amount of time by using a repository of low-level component designs” [38]. Although the initial BEE was developed by the University of California Berkeley, the BEE4 was developed by BEEcube. The BEE4 offers a large variety of high-performance, real-time implementations in a wide range of fields including SDR, MIMO radar and High Performance Computing.

The BEE4 has 4 Xlinix Virtex -6 FPGAs interconnected in a ring bus structure, with each FPGA connected to two DDR3 SDRAM memory and 20Gbps Ethernet I/O connections[53]

[21]. This set up results in the symmetrical appearance on the board. The processor, through 1Gbps Ethernet link, is used to configure and monitor the FPGA and is an Intel Atom processor. The datapath control on the board is done by a Microblaze soft core that is hosted on one or more of the FPGAs. Each of the cell on the BEE4 also has a FMC (FPGA Mezzanine Card) to interface with a single ADC or DAC card. An image of the BEE4 can be seen below in Figure 2.3.

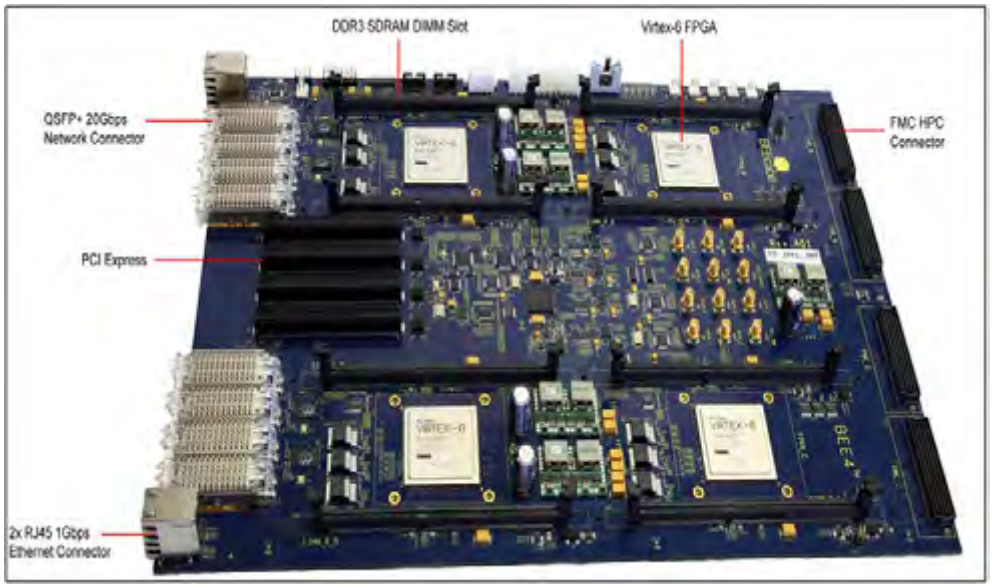


Figure 2.3: Image of the BEE4 board (Image courtesy of [21])

2.1.2.3 ROACH

ROACH stands for the Reconfigurable Open Architecture Computing Hardware, which is a reconfigurable platform designed by the Square Kilometer Array South African group (SKA-SA) [4]. The platform forms a collection of FPGA boards used for signal processing by CASPER[1]. The ROACH's main use is as a packetized correlator and as such has been designed to process high data rates on both the ADC I/O side and on the network side [53]. An illustration of the ROACH can been seen in Figure 2.3 below.

The ROACH has a Virtex-5 FPGA which handles all the data processing on the platform. It also uses to Z-DOK connectors to interface with the ADC and the DAC cards. In order to handle its fast networking requirements, the board has four CX4 connectors each with its individual 10Gbps Ethernet connection.

Additionally the ROACH has a PowerPC chip that is used for programming the FPGA and controlling the platform once it has been programmed through a parallel bus. The

[1]Casper is an international collaboration of radio astronomers who aim to streamline and simplify the design flow of radio astronomy instrumentation by promoting design reuse through the development of platform independent, open-source hardware and software

Figure 2.4: Picture of ROACH board designed at MeerKAT in South Africa Image Courtesy of CASPER website [10]

PowerPC runs a modified version of Linux called BORPH, which enables communication with the processor to be done via its 100Mbps Ethernet connection.

A major advantage of the ROACH is that the Virtex-5 FPGA provides extremely good performance with regards to speed, which results in it meeting most of its processing requirements when running applications. However the board is relatively expensive to purchase costing around USD$5300, with the software required to build gateware designs for them costing USD$5250.

2.1.2.4 RHINO

The RHINO platform was designed by the University of Cape Town (UCT), for the purpose of developing local South African skills in SDR and radio astronomy[66]. The platform combines an ARM processor and a FPGA, which allow for users to gain skills in both FPGA HDL coding and microprocessor based programming.

A block diagram of the platform can be seen in Figure2.5 below.

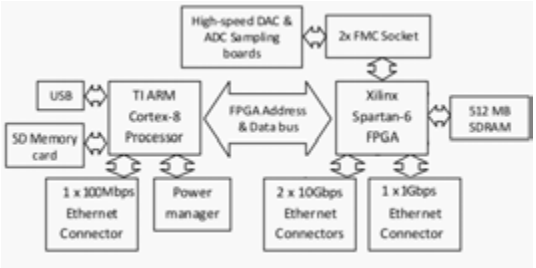


Figure 2.5: Rhino Board Overview

ARM Processor The RHINO board has a Texas Instrument's ARM 3517 physical processor on it to handle the operations that cannot be efficiently handled by the FPGA. It provides the configuration, control and interface functions to the FPGA, but does not do any computationally-intensive processing itself. It is also used to monitor the status of the board. The high performance ARM Cortex-A8 microprocessor also supports the BORPH OS Linux operating system as its ROACH counterpart.

Spartan 6 FPGA The FPGA that is on the RHINO platform is the Xilinx Spartan 6 XC6SLX FPGA. It is used to do much of the digital front-end [66] and any computationally intensive processing that may be required by the application.

GPMC Bus Interface GPMC stands for the General Purpose Memory Controller, which is a 16-bit bus interface on the ARM3517 processor chip [58]. It is connected directly to both the FPGA_PROC_BUS and the NAND flash memory. The GPMC can be configured to support any bus protocol. The parallel bus between the ARM and FPGA, connects directly to the GPMC interface and enables the BORPH operating system to access registers and peripherals on the FPGA [53].

The GPMC has 16 data lines, 10 address lines and large number of control lines. The address space can be increased by multiplexing the 16 data lines to act as address lines too. It also has 8 chip select lines, allowing for 8 external devices (each of 128MB), to be addressed. Table 2.1 below illustrates how each of the device memories are mapped onto the processor's address space. Access to these devices are provided by reading and writing to a given mapped address. The GPMC can only use addresses 0x00000000 to 0x3FFFFFFF.

CS number	Connected to	Start address	End address	Region Size
CS0	NAND flash	0x30000000	0x37FFFFFF	128MB
CS1	FPGA	0x80000000	0x0FFFFFFF	128MB
CS2	FPGA	0x10000000	0x17FFFFFF	128MB
CS3	FPGA	0x18000000	0x1FFFFFFF	128MB
CS4	FPGA	0x20000000	0x27FFFFFF	128MB
CS5	FPGA	0x28000000	0x2FFFFFFF	128MB
CS6	FPGA	0x38000000	0x3FFFFFFF	128MB
CS7	FPGA	-	-	Unused

Table 2.1: The 8 Chip Select Regions for the GMPC [53]

As can be seen in Table2.1, memory CS1 to CS7 is dedicated to the FPGA meaning that 768MB of the processor's GMPC space is allocated to the FPGA. Lastly it should be noted that the GMPC is not used for configuring the FPGA. The FPGA programming is done using the boards serial programming interface.

The FPGA Mezzanine Card (FMC) Connectors The platform has 2 FMC connectors that allow for commercial off the shelf components such as Analog to Digital Converters (ADC) and DACs to connect up to platform.

Communication Peripherals The RHINO board has two main communication links. The ARM processor has a 100Mbps Ethernet link which enables users to control, monitor and program the board and it is used to copy files to the ARM using the network. The Spartan 6 has a 1Gbps Ethernet transceiver used to transfer data directly to the FPGA.

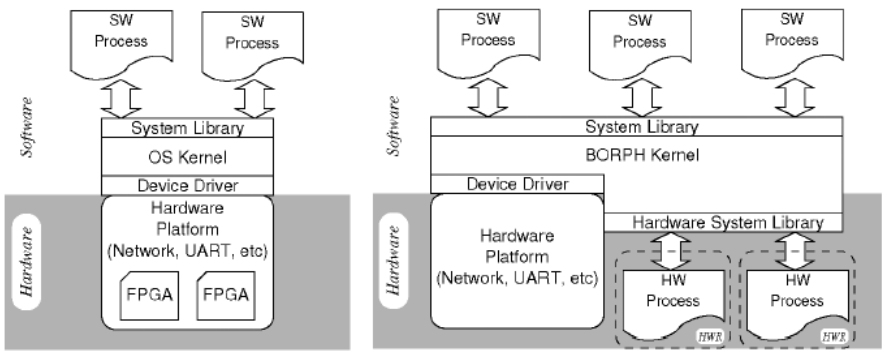
The system will be described in greater detail in Chapter 4, as it forms the platform which is used by the software developed as part of this dissertation.

2.1.3 Reconfigurable Operating System

This subsection presents an operating system that is usable on most of the platforms described in subsection 2.1.2 excluding the URSP board.

2.1.3.1 BORPH: An Operating System for reconfigurable computing platforms

BORPH (Berkeley Operating System for Re-programmable Hardware), is an operating system that was designed specifically for FPGA-based reconfigurable computers. It achieves this by extending the standard Linux kernel to allow for support for FPGAs in a reconfigurable platform [35]. It was designed with the aim of easing and accelerating development of high-level applications on RCs [34]. BORPH distinguishes between the hardware resources used by user applications such as the FPGA, and those associated with the platform which are called hardware regions. Applications that run in these regions are called hardware processes as such.



(a) Traditional coprocessor-based FPGA system: SW/HW has master-slave relationship.

(b) BORPH-managed FPGA system: SW/HW has a peer-to-peer relationship.

Figure 2.6: Diagram of how BORPH manages the SW/HW peer to peer relationship [35]

Previously operating systems for reconfigurable hardware before BORPH treated hardware designs and software processes with a master-slave relationship as shown in Figure 2.6(a). This meant that in order for a software process to communicate with a hardware design running on the FPGA, it had to rely on on set of system libraries as well as device drivers. However, BORPH treats hardware processes with no difference to software processes and hence establishes a peer-to-peer relationship [35] as seen in Figure 2.6(b). This has resulted in the simplification of the communication between hardware designs and software process to simple file system operations. BORPH is the first OS to offer runtime support directly to hardware processes as well as modeling the FPGA gateware designs to UNIX semantics [35]. This adoption of the UNIX semantic allows developers from any research background to easily use FPGA based RCs [34].

A hardware process communicates with the BORPH kernel through a predefined message passing network that acts in a similar manner to software system calls. This is useful as when a gateware design is run on the FPGA as a hardware process, the constant UNIX interface allows the BORPH kernel to handle it in the same manner as it would a software process. One can communicate with a hardware process using the IOREG interface which encapsulates conventional memory mapped I/0 concepts with a virtual file system [35].

2.2 Computer Based Clusters

Computer clusters can be described as computer based devices linked together using a network to perform a given task. This next section presents an overview of the Beowulf cluster system which is currently amongst the popular cluster models. This is then followed by an overview of what a reconfigurable cluster is with an example, and lastly the section closes with a an discussion on some of the cluster communication methods.

2.2.1 Beowulf Type Clusters

Beowulf type clusters are a network of workstations connected together with the purpose of solving a single problem [45]. The workstations connected to the cluster can range from one with homogeneous nodes to ones built from a variety of workstation. They have recently emerged as very cost-effective supercomputers used to solve a variety of problems as they provide the high performance speed that supercomputers do at a fraction of the cost.

However a disadvantage of Beowulf clusters is a general lack of cutting edge network, memory , processor, I/O subsystems as a super computer. This limits its ability to perform certain classes of problems such as those similar to Fast Fourier Transform [62].

2.2.2 RC Clusters

A reconfigurable cluster can be defined as a pool of reconfigurable platforms connected together to perform a given task. There are not that many RC clusters however BEE family of devices are good example of a reconfigurable clusters.

The system architecture of the BEE family of devices can be compared to that of a cluster of networked workstations. It enables multiuser experiences by allowing many users to share its resources while still maintaining high computational throughput [24]. This enables, for example, many different radio astronomy applications to share a given BEE2 resource pool. This in turn enables multiple applications to run simultaneously on the compute modules on the back-end of the signal processor.

For applications that require random communication between the compute nodes, the BEE2 device uses a commercial network switch technology called Inifiniband connectors. However the compute modules can be connected to form any other network topology simply by altering the connectors. Having a system designed with multiple BEE2 compute modules on the front-end and back-end of a signal processor greatly reduces the system down time and hence improves the overall system reliability.

2.2.3 Cluster Control and Management Systems

A computer management system can be defined as "a computer utility that allows the user to access system tools in the computer that include the device manager, local users on the system and removable storage" [15]. A control system on the other hand is a unit that manages, commands, directs and regulates the behavior of a computer machine. A cluster control management system combines both of the qualities of the management and control systems in a single system or device. This means that cluster control management systems can be in the form of hardware or software. Hardware based control and management systems are usually custom designed for a particular function, which results in them being expensive to develop and use. Software based control systems are cheaper but are highly dependent on the operating system they run on and if the OS crashes they seize to work. The next subsection below present two control and management systems.

2.2.3.1 The TaskManager Control Software for ALICE High Level Trigger

The ALICE program is a Large Ion Collider experiment for the Large Hadron Collider (LHC) built at CERN [59]. It operates on a PC cluster of 400 to 500 nodes arranged in a multiple hierarchy level with a processing chain of software components distributed over the cluster nodes [61]. In order to manage the large number of software components

the TaskManager control software for the ALICE High Level Trigger was designed. It provides flexible and hierarchical program control of the cluster of PCs.

The TaskManager control software consists of three main parts, the Configuration Engine (CE), the Program State and Action Engine (PSAE) and the Program Interface Engine (PIE). The CE is used to read the TaskManager's configuration files which other modules in the system can use to query information about the control system from. Each configuration file contain specifications on how each of the programs in the control system can be started, Python code to be executed for specific events and interface libraries used for communication. The second main element of the TaskManager system is the PSAE. It is the Python interpreter used to describe the different tasks that are to be performed given a change in program state. It is what is used in place of standard state machines that are used in most control systems. The last part of the TaskManager is the PIE. This is what the TaskManager uses, through interface libraries, to provide support for all the programs running on it. The libraries have a standard calling interface and provide functionality for querying the status of a program and its data as well as enabling the sending of commands to the running program. There is also a special interface that provides support for the master-server communication within the TaskManager.

2.2.3.2 Cluster Interface Agent: A Hardware Based Cluster Control and Management System

The Cluster Interface Agent (CIA) is a hardware based cluster management and control system card developed at the Kirchhoff Institute of Physics in Germany [49]. It was designed to address some of the challenges that cluster administrators face when monitoring large PC based clusters such as ALICE that are prone to computer errors and to enable installation and configuration of the cluster in order for it to start up.

The CIA card is able to remotely configure and control a single computer or the entire cluster of PCs and can act completely independent from the cluster node through its 10/100 Mbps network interface. Each node connected on the cluster will have a CIA card which will be connected in a network completely different to the network the cluster will be connected on. The card performs two main functions, the first being it returns the related PC host information such as the CPU state, hardware scans or the read content from the hosts memory. This enables the PC with the CIA card to be monitored to the detect failure of the host or be configured with a given program. The second function the CIA card performs is controlling the computer hosting it. This is achieved by the card emulating I/O devices such as the mouse, keyboard and floppy disk. This enables the control of data and video on the CIA card host computer. It is important to note that the administrator has full control of the hardware network through the networked CIA cards and can install any new software on the boards as well as monitor them.

2.2.4 Cluster Network Topologies

A network can be represented by a graphical connection of nodes, in which the nodes represent switching points in the graph and the edges as the communication links between the nodes [29]. Network topologies can be classified into two main categories:

1. **Static Topologies:** Are topologies in which the links between the processors and buses in the network cannot be reconfigured for connection to other processors once the initial connections have been established.
2. **Dynamic Topologies:** These are network topologies in which the links can be reconfigured by setting the network's active switching elements to change according the the network connection needs.

The main types of topologies that are used are static topologies. These can be classified as star, bus, tree and ring topologies and can be seen in Figure 2.7 below.

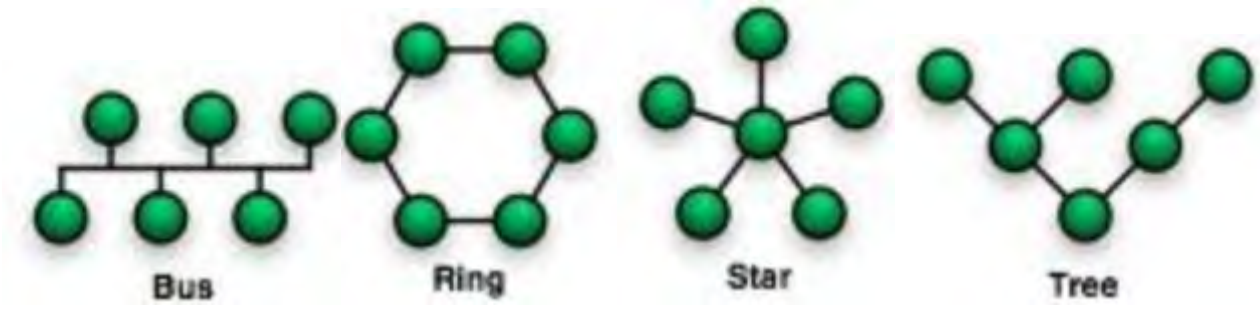


Figure 2.7: Common Static Network Topologies [Diagrams courtesy of www.interfacebus.com]

Bus Topology In this topology all the nodes are connected to a single cable known as the trunk in the network. Data sent on the network is delivered to all the nodes on the trunk. Each trunk on the node examines every packet and only accepts packets addressed to it. The main disadvantage of this network topology is that the performance of the network degrades as more nodes are added to it. As well it is hard to trouble shoot the bus, which could result in one fault bringing down the entire network. Advantages of the bus topology is that it is easy to understand and is an inexpensive, simple network.

Ring Topology This topology comprises of nodes connected on a single circle of cable which is usually seen as token ring or as a fiber optic network. A token or a message packet is sent from one node to the next, around the ring with the node with the token being the only that can transmit at any given time. In this set up each node acts as a repeater so as to keep the signal sent around the token ring strong. An advantage of this

setup is that no single node can have complete monopoly of the network. However a disadvantage of this topology is that a single node failure can affect the entire network and adding or removing a node from the network results in the disruption of the entire network.

Star Topology This topology comprises of nodes connected by cable segments to a centralized node such as a switch or a hub. In this setup a signal can travel from the hub to any one of the nodes it is connected to. It is considered to be the most scalable and reconfigurable of all the mentioned static topologies. However its major downfall is if the centralized node goes down, the entire network goes down and it is more expensive to setup as compared to using the bus topology.

Tree Topology (Hierarchical Topology) This can be considered as the most common network topologies which can be found in most large corporations to date. It is often designed to mirror the different corporate structures in place. Its major advantage is that if a single node goes down the network is able to use alternative routes for communication to end nodes hence the network is not disrupted.

2.2.5 Cluster Communication Methods

This subsection covers the most common multi-system or cluster communication methods. All these communication methods enable multiple platforms to send and receive information between each other in a standardized manner. The chapter begins with a discussion of MPI as a standard to specify the communication between two processes. The second communication method discussed is PVM and the subsection concludes with a presentation of control protocols and examples of them developed at the SKA SA for the use in the MeerKAT project.

2.2.5.1 MPI Overview

The message-passing interface (MPI), is the specification of a standard that is used when designing applications that are portable on different platforms, and facilitates the development of parallel applications as well as libraries [40][31][1]. MPI specifies the communication behavior of processes that make up the parallel application or library through the use of an API. It was designed for implemented on tightly coupled massively parallel machines and network of workstations (NOWs). The MPI Forum that consisted of end users and vendors, library writers and application specialists first defined the MPI standard in

1994. Today many sub-versions of MPI have been developed and evolved to meet the current needs for process communication. It is more appealing than other message passing systems, as it standardizes the way in which the processes communicate with each other making it easier to implement and use on different systems. These standards were derived from the most commonly used functions in existing message passing systems [40]. Due to the fact that it is an interface, it is portable and scalable and can be implemented on both distributed memory systems as well as shared memory systems. MPI can be implemented in both C and Fortran.

2.2.5.2 MPI in High Performance Reconfigurable Computers

In most reconfigurable computers, portability is achieved through the use of either a UNIX based OS (or a variant of it) or through a standard software middle layer such as the Message Passing Interface (MPI) library [17]. MPI is a standard that is used in most applications that require a parallel programming API and also need to be portable on different platforms [31][1]. Where OS based platforms, like the BORPH OS introduced in subsection 2.1.3, treat applications that run on hardware regions as typical operating system processes (hardware process), platforms that utilize a software middle-ware, like the MPI library, treat processes running on hardware regions as MPI processes. The ability to run MPI processes on RCs is important to note as this enables communication between these MPI processes to be done using MPI function calls from the MPI library.

TMD-MPI is a hybrid implementation of the MPI standard that was designed for Multiprocessor System-on-Chips implementations across multiple FPGAs in an attempt to abstract hardware details from the average user [**?**]. It was designed for the Toronto Molecular Dynamics (TMD) machine which is a scalable multi-FPGA configuration designed to accelerate computationally intensive applications. TMD_MPI has a middleware layer that abstracts the communication between the software and hardware to enable portability and is a proof of concept that the MPI standard can be used to abstract such RC architectures. TMD_MPI is made up of low level APIs for the processor-FPGA communication and can be categorized into four major functions; Initialization and termination functions, FPGA management functions, data transfer functions and memory allocation functions. The TMD_MPI software library that provides MPI related functionality can be seen in Table 2.2 below. TMD_MPI assumes that the processes running on the RC platform are MPI processes and not OS based hardware process.

The advantage of using an MPI based system is that it does not dictate to a user how the standard should be implemented, but rather provides the user with the freedom to make trade-offs between performance and functionality as they see fit, enabling efficiency in implementation [31][1]. However the disadvantage that MPI presents is that it does not cover shared memory operations, interrupt-driven messages, remote execution and active messages.

MPI_Init	Initializes TMD-MPI environment
MPI_Finalize	Terminates TMD-MPI environment
MPI_Comm_rank	Get rank of calling process in a group
MPI_Comm_size	Get number of processes in a group
MPI_Wtime	Returns number of seconds elapsed since application initialization
MPI_Send	Sends a message to a destination process
MPI_Recv	Receives a message from a source process
MPI_Isend	Non-blocking send
MPI_Irecv	Non-blocking receive
MPI_Issend	Non-blocking send using synchronous protocol
MPI_Test	Tests if a non-blocking request is complete
MPI_Wait	Blocks the calling process until a non-blocking request is complete
MPI_Waitall	Blocks the calling process until all the non-blocking requests in an array are complete
MPI_Barrier	Synchronizes all the processes in the group
MPI_Bcast	Broadcasts message from root process to all other processes in the group
MPI_Reduce	Reduces values from all processes in the group to a single value in root process
MPI_Allreduce	Reduces values from all processes in the group, and broadcast the result to all the processes in the group
MPI_Gather	Gathers values from a group of processes
MPI_Alloc_mem	Allocates memory for Remote Memory Access (zero-copy transfers)
MPI_Free_mem	Deallocates memory that was reserved using MPI_Alloc_mem

Table 2.2: TMD_MPI Library Functions [17]

2.2.5.3 Parallel Virtual Machines (PVM)

PVM is a combination of tools and libraries that enable a network of similar computer workstations or heterogeneous platforms to appear as though they were a single virtual machine [55][27]. It enables these networked (TCP/IP) machines appear as a parallel virtual machine in order to perform tasks that are optimized through parallelism. The virtual machines are able to exchange data between themselves through simple message passing constructs. It is used in applications such as molecular dynamics, distributed fractal computations, matrix algorithms and in classrooms for teaching current computing techniques.

PVM is made up of two main components, the daemon known as pvmd3 and a PVM library [43][55]. The daemon or pvmd3 is the background process that resides on all computers that make up the virtual machine. The virtual machine is started up by each of the PVM daemons in order to execute a PVM application. It should be noted that multiple users can configure overlapping virtual machines and each user can execute several PVM applications at any given time. The second component of PVM is the library of PVM interface routines. It contains primitives used for cooperation between tasks running on the virtual machine as well as function calls for message-passing, spawning new processes on the virtual machine, coordinating tasks and modifying the virtual machines. PVM also has process and resource control, which enables a user to start and stop tasks running on the PVM. It also enables the user to tell which tasks are running on the PVM and also possibly where they are running. Lastly PVM has a resource management system which allows for the dynamic adding or deleting of hosts to the virtual machine from the system

console or from the user's application.

PVM has a couple of advantages that are associated with it. Firstly it is freely available software which makes it easily accessible to the average user. As well, it handles heterogeneous platforms more efficiently than any other parallel programming software. Lastly the fact that it has resource management and process control enables applications developed for it to be portable and able to run on any cluster of workstations.

2.2.5.4 Control Protocols

The definition of a control protocol can be split into two, that of a control system and that of a protocol. The definition of a control system is a device or a set of devices used to manage, command, direct or regulate the behavior of another device or set of devices. A protocol on the other hand, can be defined in computer science as a set of rules or procedures for transmitting data between electronic devices. In order for two devices to exchange control information there must be a preexisting agreement as to how the information will be structured and how each side will send and receive. This is achieved through a control protocol.

Control protocols can come in two major forms, binary based protocols and text based protocols [26]. A binary protocol is one that is based on the exchange of specific data structures between two devices that speak that particular protocol where as a textual protocol are designed around the exchange of text based packets between the devices that communicate using that protocol. The difference between the two protocols can be seen in Table 2.3 below:

Two unique protocols as presented in next that were both developed by the SKA SA for the MeerKAT project.

KAROO ARRAY TELESCOPE CONTROL PROTOCOL KATCP

The Karoo Array Telescope Control Protocol (KATCP) was developed by the Square Kilometer Array (SKA) South Africa as part of the Karoo Array Telescope project and acts as an interface between the various control and monitoring components of the telescope. It is currently being adapted for the MeerKAT project [64] and is used to handle multiple connections of devices or platforms that communicate using KATCP. However it should be noted that it is not intended for high volume data transportation between devices that speak this protocol. KATCP is a text based protocol that transmits its data between two devices using a TCP/IP connection. The commands that can be executed using KATCP can be seen in Figure 2.8 below [2]:

KATCP is currently the preferred remote command and control interface for the Reconfigurable Open Architecture Computing Hardware, ROACH 1 board. It is used to execute

Protocol Feature	Binary Protocol	Textual Protocol
Verbosity	Binary based protocols have the ability to be highly compact generally, ensuring that the least amount of data is sent across the network	(i) Text based communication has the disadvantage of not being able to representing Boolean values (ii) Numbers that can be represented in single bytes in binary cannot be represented in a similar manner in a text based protocol.
Word Ordering	Binary based protocols use htonl(s) and ntohl(s) to ensure that if two words or data are sent to different receivers, the two sides are able to agree on how to interpret the two numbers.	Text based protocols have an advantage in this area, as they don't have an issue of representing numbers with textual digits i.e. they are endian neutral and the receiver converts received data to a binary format.
Strict Definition (This relates to the protocol having a strict definition of every aspect of the information being transferred back and forth)	Binary protocols are generally defined by structures and ensure that a strict layout of values in memory is obtained. There is not as much opportunity for assumptions to be made in this protocol.	Textual protocols could have non readable characters included in the packet without being visibility obvious to the user. Tests for the protocol may or may not catch this which could eventually lead to the breaking of the firmware using the protocol

Table 2.3: Difference between Binary Protocol and Text Protocol

KATCP Command	Description of Command
read	Reads arbitrary data lengths from a named register. This only works once the FPGA has been programmed. (Data is returned in big- endian format)
write	Writes arbitrary data lengths to a named register. This only works once the FPGA has been programmed.
listdev	Displays all the available device registers. This is only useful once the FPGA has been programmed.
listbof	Displays available BORPH bitstreams in "./boffiles".
status	Displays image statistical information. I.e. if the FPGA is programmed, a string with the path to the hardware registers will be returned, however if the FPGA is not programmed a status error will be issued.
progdev	Programs the FPGAs with a BORPH bitstreams. One gets the bitstream names from the listbof command.
sensor-list	Lists the sensors on the board. Such sensor readings include fan speeds, voltage rails and device temperatures.

Figure 2.8: KATCP available commands

commands remotely on the board and it allows a user to send commands to configure and access the FPGA, as well as to read its device sensor values.

DIGITIZER CONTROL PROTOCOL

SKA-SA have developed a digitizer using the ROACH board with only the FPGA and an ADC on the front used to send data from the telescope to the rest of the system [65]. The development of this digitizer resulted in the PowerPC on the ROACH being removed as it caused a lot of noise interference to data being transferred. This meant an alternative way was required to control and manage the gateware designs running on the FPGA, using the 10Gbps Ethernet ports on the ROACH to communicate with FPGA.

UDP packets with a payload on how to perform the read or write to the register on the FPGA via the 10Gbps Ethernet ports were developed. These UDP packets form the Digitizer Control Protocol which enables the reading and writing of registers on the digitizer from any location on the network. A memory mapped interface of all the registers on the FPGAs is also provided on the network in order to tell what registers are accessible. An advantage of this control protocol is that it eliminates the need for the OP bus interface that was initially between the PowerPC and the Virtex-5 FPGA.

DESIGNING OF PROTOCOLS

Protocol engineering is defined as "determining which methods are best suited in defining the protocol, for verifying that they have the desired properties, for developing an implementation (which could be in different hardware/software environments) and for testing the given protocol implementation for conformance with the protocol definition" [32]. The formal development of a protocol is done in three major steps which are described as follows [42][30][32]:

1. *Protocol Layering (Protocol Specification)* : The first step in the design of any protocol is the determination of the protocol layers which characterize the services it provides and the type of communications it performs. This step is also known as the protocol specification. The protocol layer structure is developed based on the OSI networking model presented in section 2.3.1, with each layer providing a given service to it users. The protocol specification also includes a description of the services the protocol will provide in each particular layer including a description on how the protocol responds (its output) based on a given input. The service specification also includes service primitives which describes the operations the layer provides, for example in a transport service these would include Connect, Disconnect, Send and Receive function calls. It must be noted that the services provided by a protocol layer must be executed in a particular sequence in order to work and these constraints are best illustrated as sequences or by using any state changes as a result of some operation. Formal Description Languages assist in the

definition of the protocol's specification. The most common protocol specification languages are SDL, Estell and Lotos and UML modeling. It is important to note that current modeling tools have integrated SDL into the UML modeling tools.

2. *Protocol Verification:* In this case verification is being able to demonstrate that each of the objects in the protocol meet its given specification. This verification is based only on the system specification and involves logical reasoning. In addition to the verification of the protocol design, the verification of the actual implementation of each protocol entity is also performed. Two main verification methods exist, the first is the reachability analysis which analysis the interactions between entities in a given protocol layer. The second is the program proving approach, which is used to test the correctness of the properties of the protocol given by the initial specifications. A hybrid approach of the two mentioned techniques enables the advantages of both approaches.

3. *Protocol Implementation:* The final step in the design of a given protocol is the actual implementation of the protocol. There are automated compilers for generating protocol code based on protocol specification developed in step 1. Mostly commercial tools have been developed to automatically generate protocol code for growing popular specification languages such as SDL and UML. C language is used mainly for the development of the protocol software.

2.3 Communication Networking

This section provides the background on the communication networking ideas that will be used in the development of framework. This section begins with an overview of the seven layer OSI model, followed by a description of TCP and UDP. Subsection 2.3.3 then presents the concepts of network ports and sockets and how a unique pair of the two can define a device in a network setup. The last subsection provides background on DHCP and how assigns IP addresses to devices on the network.

2.3.1 Networking Layers

The networking layers are best represented in the seven layer OSI model. The OSI reference model aims to abstract the description of interprocess communication between different systems [39][36]. In this context a system refers to one or more autonomous computers and their associated software, peripherals and users. Each layer in this model serves the layer above it and is served by the layer below it.

A diagram of the different layers in the seven OSI model can be seen in Figure 2.9 below.

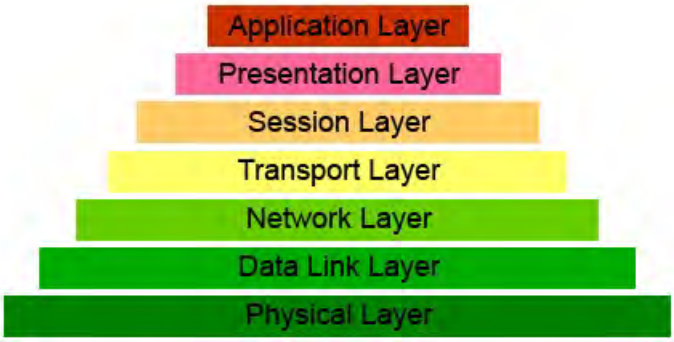


Figure 2.9: Seven Layer OSI model [Image courtesy of www.washington.edu]

Layer 1 [Application Layer] This is the highest layer in the model meaning it does not service any layer. It mainly contains all the application processes in a system. Its primary concern is with the semantics of the applications.

Layer 2 [Presentation Layer] This layer provides independence to the application processes from the differences in data representation [39].

Layer 3 [Session Layer] The purpose of this layer is to provide the mechanics for organizing and structuring the interactions between application processes.

Layer 4 [Transport Layer] The transport layer provides a medium for the transfer of data between two end systems, enabling the upper layers to not concern themselves about data reliability and the cost effectiveness of a data transfer.

Layer 5 [Network Layer]_ The network layer masks from the transport layer, all the details of the actual transfer medium, by providing functional and procedural means of exchanging network service data units instead.

Layer 6 [Data Link Layer] The main purpose of this layer is to provide the functional and procedural means of transferring data between two network entities as well as detecting and possibly correcting errors that may have occurred in the physical layer.

Layer 7 [Physical Layer] The seventh and last layer in the OS model provides the mechanical, electrical, functional and procedural standards used to access the physical medium used to connect the two devices.

2.3.2 TCP and UDP

Both TCP and UDP are two of the most commonly used transport layer protocols. TCP is a connection-oriented protocol sent over an IP network. Its main advantage is that it guarantees that all the packets sent across the network will arrive to the receiver in the order in which they have been sent [41]. Acknowledgements are used to determine if a packet has been correctly received by the receiver and if not, an automatic retransmission of the packet will be done by the sender. However, because of this in built reliability feature, TCP is considered to be much slower and less efficient than its UDP counterpart.

UDP is a connectionless protocol and communication is datagram oriented [47]. Datagram packets sent across the network are not guaranteed to reach the receiver in the correct order or even to be delivered at all. It is utilized mostly for real time communication applications in which a small percentage of data loss is considered to be acceptable as compared to the overhead of using a TCP connection. It is considered to be faster than TCP and more efficient as it doesn't use acknowledgement of packets.

2.3.3 Network Ports and Sockets

Network Ports A port is defined by the RFC 793 as "the portion of a socket that specifies which logical input or output channel of a process is associated with the data" [41]. That is, its a point through which programs running on different devices can communicate information to each other. The devices can be communicating using the Internet or a local network.

A combination of the IP address and the port number enable a network or the Internet to know who's who on the network [9]. Ports are numbered to enable consistency when used. They are grouped into 3 categories:

- **0 to 1023** are the well-known ports. They are commonly used and are dedicated mostly for Internet use. Examples include port 22 which is reserved for either UDP, TCP and the Secure Shell (SSH) protocol, and port 23 used for telnet. More of the protocols assigned in this range of port numbers can be found in [9].
- **1024 to 49151** are registered ports, meaning they can be registered to specific protocols by software corporations.
- **49152 to 65536** are dynamic and private ports

Network Sockets A socket is a network connection that consists of a four-tuple pair made of a client's IP address, port number and a server's IP address and port number. This 4-pair information, when combined together forms a unique TCP connection over the Internet [63].

2.3.4 Dynamic Host Configuration Protocol (DHCP)

The DHCP protocol is a network layer protocol used to configure devices connected to the network to enable them to communicate over the network using the Internet protocol (IP) [48].

The protocol is implemented as a server-client model. The DHCP server maintains a database of available IP addresses on the network as well as configuration information for the client. The DHCP client on the other hand initiates requests for configuration data from the server, such as an IP address, a default route or a DNS server address.

Chapter 3

RACCMS Framework Design Process

This chapter presents the design process used in the development of the RACCMS framework. It provides details of the steps the project underwent, as well as the testing procedures that were carried out on the implemented framework. The frameworks design process starts with the initial user requirements of the framework and works all the way down to the conclusion stage of the project.

The development model used in this project was the waterfall model where each phase of the design process was finished before the next could be started [23]. The project was carried out in 12 main steps which can be seen in Figure 3.1 below.

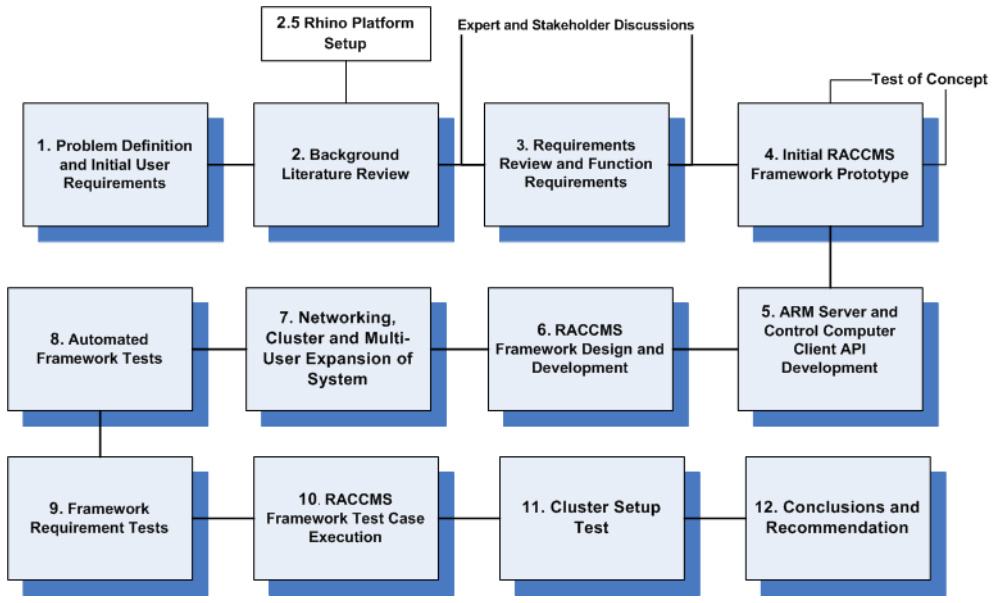


Figure 3.1: Overview design steps taken in the RACCMS project development

A detailed explanation of each of the steps shown in the above diagram is given in the sections below.

3.1 Problem Definition & Initial User Requirements

The first step in this project was the definition of the user requirements. These user requirements were defined in collaboration with Square Kilometer Array team and the academics at the Software Defined Radio Group (SDRG) from the University of Cape Town which included Dr Simon Winberg and Professor Michael Inggs. The initial user requirements were derived from the idea of developing a framework that will be used as an educational tool for students working on laboratory projects, learning about SDR techniques, as well as being used for larger projects requiring the use of several RC platforms. The process of drawing up these initial requirements required meetings with the given stakeholders of the project to discuss and establish the initial requirements. Section 4.1 below effectively documents the results of this step.

During the process of developing the framework, referral back to these initial requirements was made in order to make sure the system being developed met the stakeholders initial requirements of the system.

3.2 Background and Literature Review

The second step in the project's development involved a two-stage process. The first stage involved a background understanding the hardware, as well as the principles behind its operation, and the second stage entailed reading relevant literature.

The first stage in this step was a complete understanding of the platform's operational principles through user manuals and specification sheets, before the initial prototype framework could be built. This stage involved learning how to connect up all the supporting hardware in order to get the platform fully operational as well as an understanding of the different signals exchanged on the platform when performing certain functionalities. Additionally an understanding of how the board's hardware, namely the processor and the FPGA, communicate was done.

The next stage in this step was learning how to upload middleware onto the RHINO platform and understanding how the operating system provides communication to running gateware logic on the FPGA. The last stage in understanding the board used in this project was programming the FPGA with the equivalent of a "hello world" application. This aided in understanding how the communication with the gateware logic functioned and

how it executed using the operating system. This whole process helped determine what was possible on the system and what wasn't.

Although the literature review is mentioned only once in this methodology, it is important to note that this was an ongoing process through out the development of this project. The literature review provided a basic understanding of how similar and current hardware systems work. These two stages provided a good foundation to further refine the user requirements and draw the functional requirements of the system.

3.3 Requirements Review and Functional Requirements

The third step in the design process was to review the given initial requirements based on the background and literature review step. This step involved meetings and email correspondence with the SKA team who developed KATCP and the KAT stakeholders. Although the majority of the work is mentioned in this section of the project, it is important to note that the requirements review was an ongoing process until a user and functional requirements baseline was established and all interested parties were satisfied with them. This process also involved the clarification of any misconceptions on how the platform used in this project functioned. Lastly this step involved establishing performance parameters required for the framework. After the user requirements were finalized by all the stakeholders involved in the development of the framework, a set of functional requirements were drawn up. These were defined in a manner that each functional requirement meet at least one of the given user requirements.

The last stage involved in the refinement of the user and functional requirements was establishing which of the cluster communication methods mentioned in section 2.2.5, best fit the designed framework. This involved matching up the user requirements of the framework with the various communication methods. The RACCMS framework's communication was developed based on the concept of a control protocol. This is because a control protocol enables users to uniquely define how two or more devices will communicate with each other. This enabled the development of a customized framework communication system that would meet the given user requirements. The concepts around the development of the control protocol were based on those similarly used in the development of KATCP discussed in section 2.2.5.3.

Table 3.1 illustrates how each of the cluster communication and control methods mentioned in 2.2.5 meet the given user requirements of the RACCMS framework.

As can be seen in the table above, the reason MPI and PVM were not selected as part of the communication method for the framework was because most of the MPI and PVM defined communication functions, would not have been utilized in the framework. If

User Requirement	Properties	Message Passing Interface (MPI)	Parallel Virtual Machine (PVM)	Control Protocol
U1	Simple and easy to use framework	✔		
U2	Allows user to develop applications using C based code	✔	✔	✔
U3	Enables Remote Access of boards		✔	✔
U4	Reading and Writing to RHINO registers	✔	✔	✔
U5	Loading BOF file to RHINO board			✔
U6	Selecting and Releasing of RHINOs for use	✔	✔	✔
U7	Cluster management in the form of lock management	✔		✔
U8	Querying status of RHINOs connected in cluster			✔

Table 3.1: Comparison of different communication methods discussed in section 2.2.4 against the framework user requirements stated in section 1.3.1

this approach had been followed, a lot of alterations to the MPI and PVM function list would have been required in order to customize the communication functions to suite the framework needs, as compared to the work required to build a unique control framework from scratch. Lastly, MPI hands out tasks to processors or a pool of MPI processors to handle the work and communication. This means that MPI requires the system to be running MPI processes in order to use the MPI API library for communication. However since the RHINO uses BORPH which runs OS based processes and not MPI processes, the use of the MPI library would not have been the most efficient communication method for the RACCMS framework.

3.4 Prototype Design and Implementation Stages of Framework

The fourth step in the project's design process was the design and implementation of several prototypes of framework. This step involved four stages of prototype development and four progressive prototypes of the framework being produced. The framework development was done by the individual components, with each component being linked to the functional requirements of the system. The initial prototype was to establish a proof of concept that such a framework could be developed and further iterations where extensive versions of the framework with modifications to make it more robust and user friendly.

3.4.1 High Level System Design

The RACCMS framework was to be used as an enabling tool to create applications that will run on a cluster of RHINOs that are accessible from a remote location. From Table 3.2 it can be seen that the RACCMS framework was designed to lie in the application layer of the OSI model. It used TCP over IP to transport the information between the server-client application. This is because TCP is a reliable transport protocol, as mentioned in Section 2.3.2. Hence to eliminate the loss or miss ordering of packets during transportation as well as maintaining packet integrity, it was chosen as the transport layer protocol. The link layer was limited to the 100Mbps Ethernet connection as this was the only link that enabled interfacing with the ARM processor for the control and management of the board. Lastly in the physical layer the national semiconductor DP83640 Ethernet PHY was used that supported the IEEE1588 Precision Time Protocol. This protocol synchronizes the time difference between nodes (in this case RHINO boards) on an Ethernet network to within 10ns of each other [46, 53]. This was an advantage as the developed framework was to be for RHINOs connected together in a cluster setup.

Table 3.2 illustrates where in the network layer the RACCMS framework would sit and illustrates the application layers below it that provide services to it in order perform its given functionality .

Network Layer	**Application**
Application Layer	RACCMS Control and Management framework
Transport Layer	TCP/IP
Physical Layer	100Mbps Ethernet (IEEE1588 Precision Time Protocol)

Table 3.2: Table illustrating the location of framework in network layer

A high level view of the designed framework consisted of at least 2 major components, the control computer and the ARM processor on the RHINO platform. The control computer was the platform on which the client application ran on and the users developed their C based code that included the control computer's API. The server application ran on the RHINO platform and enabled the control of the RHINO platform and the registers of a gateware process running on its FPGA.

The two major processing block on the RHINO platform were the ARM processor and the Spartan6 FPGA. The ARM processor hosted the major part of the framework's server application and the ARM-FPGA library functions. The client application communicated using the FPGA-ARM library, via BORPH which uses the GPMC. The Spartan 6 FGPA is responsible for all the heavy lifting processing of data that is required by the RHINO board.

3.4.1.1 System Functional Blocks

The system functional blocks were the different components used to build the RACCMS framework. They were developed based on the functional requirements drawn up from the user requirements. These blocks can be seen in Figure 3.2 with the red descriptive arrows below.

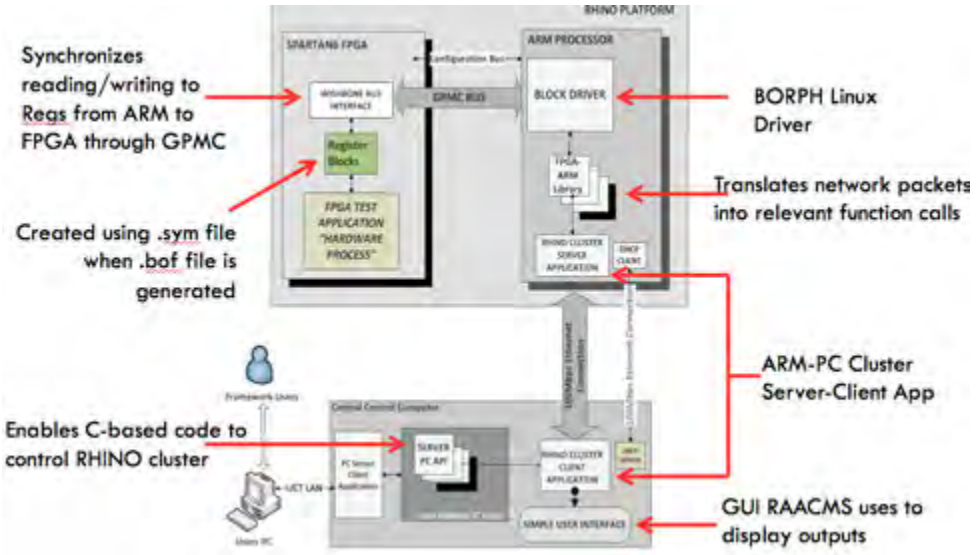


Figure 3.2: Functional diagrams of RACCMS system

3.4.2 Initial RACCMS Framework Prototype

The first prototype was developed to test basic RHINO platform register accessibility from a remote location through the RACCMS. This comprised of developing a simple single server - single client application with the ability to read and write to registers from a remote location. This step involved using the following systems in order to build this prototype:

ARM-FPGA Bus This is the parallel bus that connects the ARM processor and the FPGA and was the main link for communication and data exchange between the two devices. On the ARM side this parallel bus connects up to the GPMC interface which is described initially in section 2.3.4. This parallel bus is what enables BORPH to read and write to registers of a running gateware logic on the FPGA [34]. In order for the GPMC to interface correctly to the FPGA using the parallel bus it had to be configured appropriately [53]. Configuration was done mainly using u-boot which is a simple bootloader environment [53], when the board was starting up. However in order to be able to exchange information between the ARM and the FPGA a WISHBONE bridge interface had to be developed to enable data exchange between the register block and the GPMC interface. It

was developed to bridge the difference between the GPMC bus protocol that takes 2 clock cycles to read or write to a registers and the clock cycles of the actual register blocks. It achieved this by creating gateware registers that operate on a 3 clock cycle protocol to read and write to them.

Server-Client Application Development of the initial server client application was based on the example Linux server client application from [11]. The model developed was a very simple one, with the server application hosted on the ARM processor and the client application running on the control computer. The server application was designed to reside on the ARM processor as the RHINO is the device that receives requests from the control computer, processes the request and returns its result to it. The client application was designed to run on the control computer as this is the central medium by which users send requests for services to the RHINO platforms (e.g. to read or write from a register). This then resulted in a single client - multi server framework, when multiple RHINOs were connected to the control computer as each RHINO would host its own server application.

The API implementation in this prototype was limited to only reading and writing to known registers on an already running gateware process. The protocol implemented was based on single word commands i.e. “read” to read to a register and “write” to write to a register. The main purpose of development of such a simple command set was for a proof of concept that such an application could work on the given platform.

Protocol Selection From the initial prototype of the framework the binary protocol was selected for this framework. This was because it was more strictly defined meaning that the packets sent across the network would be as compact as the designer required them to be in terms of memory space. As well its strict definition meant it was more secure than its text counterpart as what was sent by one side of the server-client application was what was expected to be received on the other end. It was also selected over its textual counterpart as it was able to represent numbers without having to convert them into textual digits at the sender side and undoing the operation at the receiving side. Lastly the binary protocol was seen to be more efficient as it sends numbers as they are and, it is able to send them as single words. It is important to note that structures used and developed for the framework contained no pointers in them, as pointers point to a memory location on a device. However since the packets will be sent across the network to different devices the structures used in the protocol did not contain any.

3.4.3 ARM Server and Control Computer Client API Development

The second prototype development of the framework involved building the majority of the two API libraries shown in Figure 3.2, for both the control computer client application and the RHINO platform's server application.

A portion of the API was designed specifically to enable users without any proper knowledge of the ARM-FPGA bus structure, to read and write data to registers on the FPGA. The rest of the API was designed to be used for administrative purposes. The API itself was designed to be simple single worded commands to abstract the complicated logic involved in the ARM-FPGA and FPGA-ARM communication and register manipulation.

3.4.4 RACCMS Framework Design and Development

The third and final prototype of the RACCMS framework was the software design and implementation of the framework which was done in five main stages. The first 4 stages of the development represented the models that enabled the developer to understand what the framework requirements were and how best to achieve them and the 5th and final stage involved the actual construction of the framework which involved code generation. Standard UML diagrams were used to develop the first 4 models of the framework.

1. Use Case Diagrams: The main aim of this stage was determining who the main users of framework would be and what would be the main actions they would perform to interact with it. The actions and tasks to be performed by the identified users were derived from given user and functional requirements. Standard use case diagrams were used to illustrate these relationships.

2. Class Diagrams: The second stage in the design of the framework involved the splitting of the tasks from the use case diagrams into functions or classes that would perform similar actions. A class diagram was then developed to show how the individual classes would interact together.

3. Sequence Diagrams: From the class diagrams and use case diagrams, sequence diagrams were then developed to represent how the different actions or events would flow from one function, defined in the class diagram, to another. A sequence diagram was developed for each task that was represented in the use case diagrams as well as what would cause the task to occur.

4. Flow Diagrams: The last UML diagrams used to aid in designing the the framework's architecture were the flow diagrams and state diagrams. From the sequence diagrams, flow diagrams for each of the functions in the framework were designed showing each

step the developed code would have to take to. The state diagrams showed how certain elements in the framework that had states transitioned to the different designed states.

5. Code Implementation: The last stage in the prototype development was conversion of the flow diagrams into code. All code developed for the framework was done using the C language and followed the Linux kernel coding style standard [8]. Error handling was also added into the code at this stage of the prototype iteration in order to make the framework more robust. This involved catering for as many foreseen options in which the framework could encounter an error and crash and making sure that if it should crash it would do so gracefully.

3.4.5 Networking, Cluster and Multi-User Expansion of System

The fourth iteration in the development of the framework prototype was expanding the system from a single ARM server-single control computer client application, to a multiple ARM server-single control computer client system. Adding another platform was required, as well as a switch to handle the multiple platform connections from the control computer. Further more, this prototype development step involved extending the server application to two platforms as well as enabling the client to connect up to two boards instead of only one. Extending the system to more than one board also involved using the DHCP server to build a subnet of IP addresses to assign to all possible platforms that would connect up to the cluster. A selection of a suitable range of network ports from the official network port list for the server-client application was use to determine these connections.

3.5 Evaluation of the RACCMS Framework

This step provides an overview of the tests that were performed in order to see if the developed framework performed as it was designed to and that it met all the specified requirements. The tests were performed once the framework had been completely implemented. Tests were developed to test the code (white box testing) as well as the user and functional requirements (black box testing). The white box tests comprised of automated unit tests, integration tests and boundary tests of the code as well as memory checks. The black box testing comprised of tests that were used to determine that framework was performing the way it was intended to. Lastly a test case application was presented that tested the framework as a whole on an existing gateware design for the RHINO board, and tested the ability of the framework to operate in a cluster setup.

3.5.1 Automated Code Framework Tests

This section presents the automated tests used to ensure that the developed framework code executed as specified and designed. The automated tests were designed around the framework's software UML designs presented in section 3.4.4. It tested each of the software models derived to represent the RACCMS framework in an automated manner. The automated tests were developed using a unit testing framework called CHECK Unit Testing Framework [5] where as the memory tests were performed using the Valgrind framework.

CHECK Unit Testing Framework, which was the framework used to perform the automated tests, is an open source framework licensed under LGPL. It is designed in the style of xUnit testing framework [**?**] with its main interface being for C. The CHECK framework is a simple interface for defining mainly unit tests but can also be used in developing integrated tests. The main advantage of using this framework was that the CHECK framework was able to run in a different user space, which made it easier for it to not only catch assertion failures that may have occurred but also catch segmentation faults or other signals that may have been caused by the code.

Unit Tests

The first set of tests that were performed on the framework were the unit tests. This was the most basic test for the framework and formed the foundation of the automated framework tests. The unit tests were designed around the UML flow diagrams generated in section 3.4.4 that showed the way the code was designed to execute given all possible conditions. They tested the structure of the individual functions and possible branch conditions as well. The unit tests were restricted to functions that did not perform any file I/O or that included any actions that required communication between the server-client application. The unit test execution and results were documented as seen in Table 3.3 below:

UNIT TEST CASE TITLE	DESCRIPTION
Test Case	Specifies the API function to be executed and tested.
Test Setup	Describes the components of required to perform the test case, as well as how they will be setup for the test.
Unit to Test	Provides a detailed explanation of what the test is meant to achieve.
Steps to Execute	Provides an explanation of the sequence of steps to be taken to execute the test case.
Assumptions	States any previous actions that need to be performed before executing "Steps to Execute".
Results	These will be the results printed on the command line which are shown as images or tables to illustrate the outcomes of the given test case.
Pass/Fail	Specifies weather the above mentioned test case has been a success or a failure. Pass indicates as success and Fail indicates a failure.
Comments	This is a brief discussion of the results obtained from the function execution and at times images to confirm the correct or incorrect execution of the Framework in response to the test.

Table 3.3: Unit tests Test Case setup and description

Integration Tests

The automated integration tests that were performed on the framework were designed around the sequence diagrams presented in subsection 3.4.4. They were split into two main tests, one for the client application and one for the server application. The software integration approach that was used was a bottom-up integration approach [50]. This is because the components for the subordinate levels were already available and, testing the lower levels of a given functionality for errors first made it easier to detect errors when more functionality was added by the upper layers. The integration tests specifically tested framework functions that were not self-contained and their interfaces. This included mostly functions that perform file I/O operations and those that perform network communication. Individual unit test components were combined to build a given functionality.

Boundary Tests

Boundary tests were performed for functions that dealt with traversing through the array containing RHINOs and the register bit related functions. A range was determined for the various functions and tests were designed around these ranges to see how the framework performed at its boundaries and beyond them.

State Diagrams Tests

State diagram tests were designed specifically to test the transition of the developed framework from a given state it could be in to the next state as expected from the produced stage

diagrams in section 3.4.4. The expected results from the test were compared to those that were produced by the execution of the framework code, and used to determine if the stage diagram test had passed or failed.

Memory Tests

The memory tests done on the framework code were used to determine that there were no memory leaks or corruption due to the implementation and use of any of the given structures and memory allocations. Valgrind was the framework that was used to perform these tests [14]. It is a GPL licensed tool for memory debugging, memory leak detection as well as general code profiling. All the code designed to run on both the ARM processor as well as the control computer were simulated and tested on a PC.

3.5.2 Framework Requirements Tests

These tests were modeled to ensure that the given requirements of the framework were adequately met. Among the tests in this section data integrity tests, network connectivity tests as well as functional tests are performed. A description of these tests is given below:

Network Transfer Integrity Tests

These tests were used to ensure that the communication between the server and the client application was occurring correctly. In order to be able to check this TCPDump was used. This is a command-line based packet analyzer which enables users to intercept and display TCP/IP packets sent or received over the network [3]. The flags used when using TCPDump were -V which when parsing and printing allows to produce a bit more verbose output, and the -X option which when parsing and printing will allow for the data in each packet to be printed in hex and ASCII. The -X option is deemed to be very useful when analyzing new protocols such as the one developed for this project.

Database Tests

These tests were to ensure that the data packets received, stored and manipulated by both the server and client applications maintain their accuracy and integrity. This was done by manually checking that the data packets sent by the client application were the same as those received by the server by printing the packets sent and received and vise versa.

Transaction Tests

These tests were used to determine the processing and transmission of certain function calls and data sent between the client and server application. The results were in the form of the transaction times involved in sending both a server generated packet as well as client generated packet, between the server-client application.

Function Performance Tests [F2]

These tests aimed to analyze the functions involved in the reading and writing of gateware registers using the framework. The analysis was done by calling the different functions on the server from the client application numerous times and recording the average execution time. It should be noted that these tests also made sure that the functional requirement [F2] was performing correctly, by ensuring that the correct data was written and read from the running gateware processes.

User Requirement Verification Tests

The user requirement tests were used to determine if the RACCMS framework met all the user requirements specified by the stakeholders of the framework. The tests involved running the framework and verifying that the outputs provided did what the user requirements had specified. The whole framework was tested as single functional unit to perform a given user request as opposed to the previous tests that focused on either the client application or the server application as separate entities.

3.5.3 RACCMS Framework Test Case Execution

The purpose of this test was to ensure that the framework functioned as required using a test case example. The test case that was used was a gpmc_test application that was developed by Dr Alan Langman for the RHINO board. It was developed to test that the GPMC interface on the RHINO board was functioning correctly [18]. The gpmc_test application has four accessible memory regions allocated to it that represent physical peripherals on the RHINO. These memory regions are registers that are accessible through the IOREG interface. The physical peripherals represented in this test case are the RHINO reg_led, the reg_fmc, reg_files and the reg_word [18]. The reason for choosing this application as the test application was because writing to a register through the IOREG file would result in a physical output representation on the RHINO, i.e. writing to the reg_leds register would result in the corresponding LED lights on the RHINO lighting up.

3.5.4 Cluster Setup Test

The purpose of these tests was to ensure that the framework would be able to function in a multiple server-single client setup. The tests included showing that a basic cluster of RHINOs could be setup and that the framework could be run on it. In this test the DHCP server-client application was checked to see if the RHINOs were being given the correct static IP addresses based on their individual MAC addresses. This was to ensure that functional requirement [F3] which involved a functional DHCP client application on the RHINO was working.

3.6 Conclusion and Recommendations

The final step of this project's design process was the drawing of conclusions on how the framework worked and on the design process used when developing the framework. This involved measuring how many of the user and functional requirements were meet by the developed framework through its test results as well as conclusions on how well the functional requirements met any of the performance conditions placed on the framework. It also discussed any observations that were noted from the development and execution of the framework on the RHINO board. Finally, recommendations for future expansion to the framework were given based on the results and outcomes of the project.

Chapter 4

FRAMEWORK SETUP AND CONFIGURATION

The RHINO board, which is the board that the RACCMS framework is being developed for, is a new bespoke designed platform that to date has currently no working toolflow and very little documentation on it. A lot of work had to be done in order to get the RHINO to a state were it could support any software or gateware applications to run on it. As well, certain parts of the RHINO (such as the how the FPGA based device was accessed via the ARM) were simultaneously under development and hence there were a complicated sets of dependencies that had to be dealt with when developing the framework. It is important to note that the RHINO had no supporting software on it and that the RACCMS framework would be the first supporting software application designed specifically for users with low expertise levels on using such RC platforms. The platform was to be built on top of standard Linux libraries which involved an amount of work in the adapting of the code to make it run on the RHINO board.

This chapter begins with section 4.1 discussing the system functions specified by the stakeholders that required some review and then goes on to present some of the design constraints that were placed on the development of the framework and the project as a whole in section 4.2. A detailed description of the steps that were taken to setup and configure the ARM processor on the RHINO board as well as the interface that enables users to program the FPGA from the ARM processor are presented in section 4.3. This is then followed by section 4.4 and 4.5 which describes the steps that were taken to generate a BOF executable and any resulting entities that accompanied it to be used for the development and testing of the framework. A description of the operating system's communication methods that were used by the framework's API function calls is then given. Lastly, section 4.6 discusses how the RHINO cluster was built so as to test the framework on several RHINOs connected in cluster.

4.1 Requirements Review

As mentioned in section 3.1, some of the user and functional requirements presented in section 1.3.3 were not specific enough for the framework to be developed from. Therefore a review of the initial user and functional requirements presented in Chapter 1 was conducted by the SKA Digital Back End team and the academics involved with the project. Although the initial user and functional requirements provided a good starting point from which to design the framework from, further refinement of some of these requirements was needed. The specific user and functional requirements that needed to be reviewed were functional requirement [F2] and user requirement [U3]. The rest of the user and functional requirements provided sufficient and accurate information from which to develop the framework from. Hence with the modification of these two requirements, the baseline for the user and functional requirements was established.

4.1.1 Review of Reading and Writing to a Register [U3]

User requirement[U3] from section 1.3.1, specified that "users must be able to read and write to register values while a gateware process is running on the FPGA via a remote location". This requirement was refined to include the reading and writing of all 16bits of a given register value at a given time as well as the reading and writing of a single bit of the given 16bit register. The users of the framework are also required to be able to perform the additional bitwise operations as well as any combination of the mentioned functions:

- Clearing a bit value
- Checking if a bit has been set
- Setting a bit value
- Toggling a bit value

4.1.2 Review of ARM-FPGA control and communication Requirement [F3]

The control and communication between the ARM and the FPGA was initially specified to be through a uniquely designed device driver that would be run as part of the BORPH OS. Its main functionality was to enable users access and control to running gateware processors through the ARM. It was established that BORPH had a standard way of performing the functionality required by the framework and hence it was used as ARM-FPGA control and communication interface. As well, due to the RHINO board still being

in its prototype phase, as mentioned in the background section 1.1, any existing drivers currently in use were subject to change. These driver changes would be a result of mainly modifications that are being instrumented on the FPGA side of the control interface by other RHINO projects happening concurrently with the development of this framework. The use of the BORPH driver ensured that the framework used a constant interface so that should any of the device or block drivers be replaced with a different control strategy, no major changes would be required to be made to the framework.

4.2 Design Constraints

This section presents the design constraints placed on this project and the implemented framework.

4.2.1 Hardware Restrictions

The first design constraint was the hardware platform the framework was developed for. The RACCMS framework was designed specifically for the RHINO platform. The board was designed by the University of Cape Town to be used as a teaching aid for students learning about SDR and radio astronomy for research purposes. The RACCMS framework server application was hence cross compiled specifically for the RHINO's ARM Cortex 8 processor and it was designed to interacted with the RHINO's FPGA by using systems calls that were specific to the BORPH OS discussed in section 3.5 below. The data that was be exchanged within the framework was limited to 16bits because of the RHINO board's Processor-FPGA bus which is explained in detail in subsection 4.3.3 below.

4.2.2 Communication Restrictions

The second constraint imposed on the project was the connection used to remotely control and monitor the RHINO board. Once the board had loaded the BORPH OS, the only means to access the RHINO board for control was through the 100Mbps Ethernet connection. This 100Mbps Ethernet connection on the RHINO board was designed specifically to enable users to remotely access, control and monitor the board [53]. For this project, the 100Mbps connection was sufficient as the RACCMS framework was to be used for mainly control and monitoring, and not as a data interface for transferring amounts of data larger than 100Mbps to the board. Lastly, the communication discussed in this project applied to the communication between the RHINO platform and the control computer. It did not include interprocess communication or platform to platform communication.

4.2.3 Operating System Restrictions

The OS that runs on the RHINO ARM is the BORPH OS which is described in section 2.1.3. This is a Linux based operating system with the kernel modified to enable communication with processes running on the FPGA to become standard UNIX system file accesses. The ARM processor is loaded with this OS through u-boot at start up. This OS was designed specifically for boards such as the RHINO and was the OS used in all other subsequent RHINO projects including this one. This would maintain consistency in the board's development process.

4.2.4 RACCMS Data Management Restrictions

The last design constraint was that the framework was limited to the control and monitoring of a RHINO board. This meant that the framework would not be used to transfer or process large amounts of data. Transfers of data larger than 1MB or more, will be done using the two 10GB Ethernet ports connected directly on the FPGA. However, the RACCMS framework API does not provide services for transferring data using these high-speed ports for two main reasons:

1. To utilize the two 10GB Ethernet ports, the ARM processor would need to have access to them through a configured BOF process(which is another name for a BORPH hardware process), during its gateware design.

2. To save space (i.e. the use of Logic Elements on the FPGA), the standard Ethernet protocol stack, which can take a significant amount of space when implemented as HDL on a FPGA, was generally not used. Instead a customized, application defined binary protocols was used. Hence, the API did not provide support for these data transfers as the methods utilized differed substantially between applications.

4.3 Setting Up the Rhino Board Platform

This section presents an overview of the physical components used in the design and development of the framework. A section on the RHINO board and the peripherals used by the framework is presented in subsection 4.3.1. This is then followed by a discussion of the control computer that was used and lastly a description of the ARM-FPGA bus and how it works is discussed.

4.3.1 Physical Board Set Up

Figure 4.1 below shows the RHINO board that would host the RACCMS framework. The board was initially introduced in section 2.1.2.4. It has an FPGA and an ARM processor connected by a bus that is not visible in this diagram. The RHINO also has two, 256MB DDR3 SDRAM integrated circuits with a combined data rate 25.6Gbps, which form the FPGA's memory and a 256MB DDR2 SDRAM integrated circuit which makes up the ARM processor's memory. The 1Gbps Ethernet transceiver is used to connected up to the Spartan 6 FPGA, where as the 100Mbps Ethernet is used to connect and control the ARM processor. The SD card connector is used to load the Linux file system from an SD card as well as some of the FPGA configuration files. The board also has a USB port which is used to configure the ARM processor before the Linux operating system is installed.

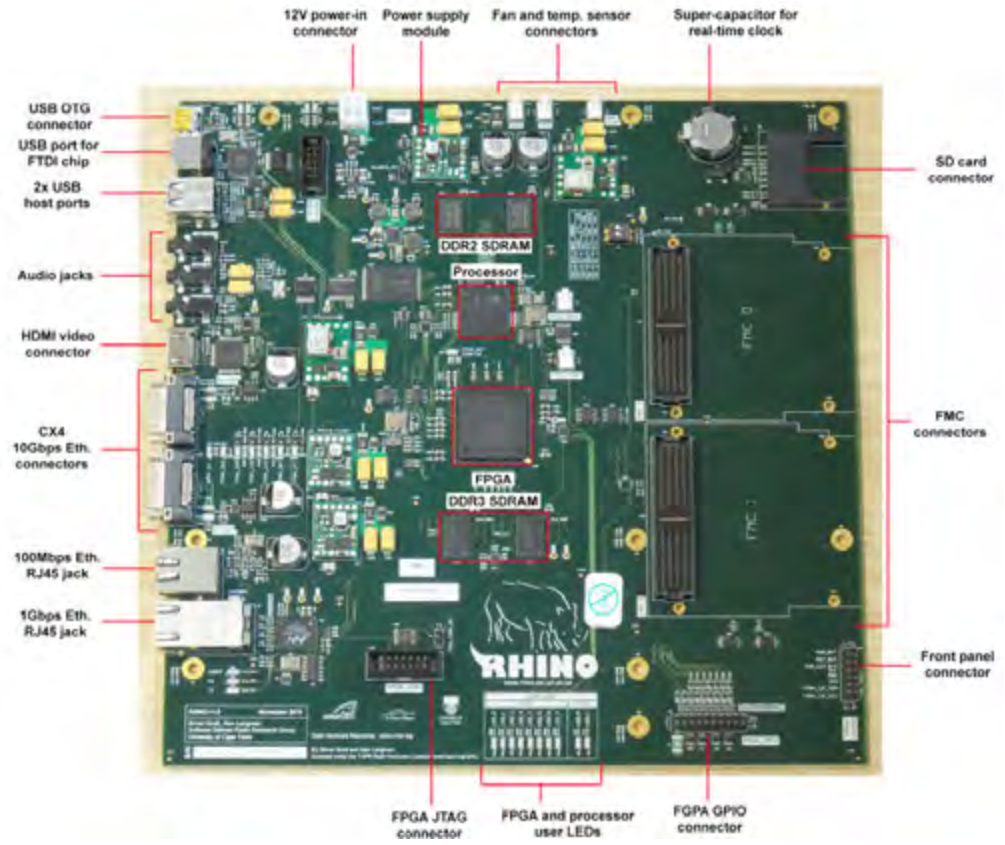


Figure 4.1: RHINO board with annotations showing different components [53]

4.3.2 Control Computer

The client application was designed to be hosted on the control computer that was simulated in this project as standard personal computer (PC). The PC that was used to host the client application was the DELL Inspirion 5520 i5 laptop. It has the following specifications:

- Intel Core I5-3210M 2.5Ghz

- 6GB DDR3 RAM
- AMD Radeon HD7670M 1GB Graphics Card

The large amounts of memory (6GB DDR3 RAM) were mainly needed for running and compiling HDL Xilinx ISE which had demanding memory requirements [68].

In order to connect the control computer to the board, several software applications had to be installed. The RHINO board required an NFS file system to be installed on the computer that it connected to, which in this case was the control computer. As well, the DHCP server mentioned above section 2.3.4, was installed in order to create a subnet for the cluster and to also statically allocate IP addresses to the boards that connect to the control computer.

4.3.3 RHINO ARM processor configuration

The ARM processor was the only way in which to configure, control and monitor the RHINO board as mentioned in section 2.1.2.4. The ARM processor enabled users to initially connect to RHINO board through its TTYS link via the USB port which was used to configure the ARM processor's u-boot and also as a debugging channel for the booting options on the board. The RHINO board had a couple of boot options available and are selected by two small white switches on the board. The first task in getting the RHINO configured was deciding which of the booting options best suited the framework design and provided the simplest development and execution environment for it. An image of the accessing the board via its USB port for configuration can be seen in Figure 4.2.

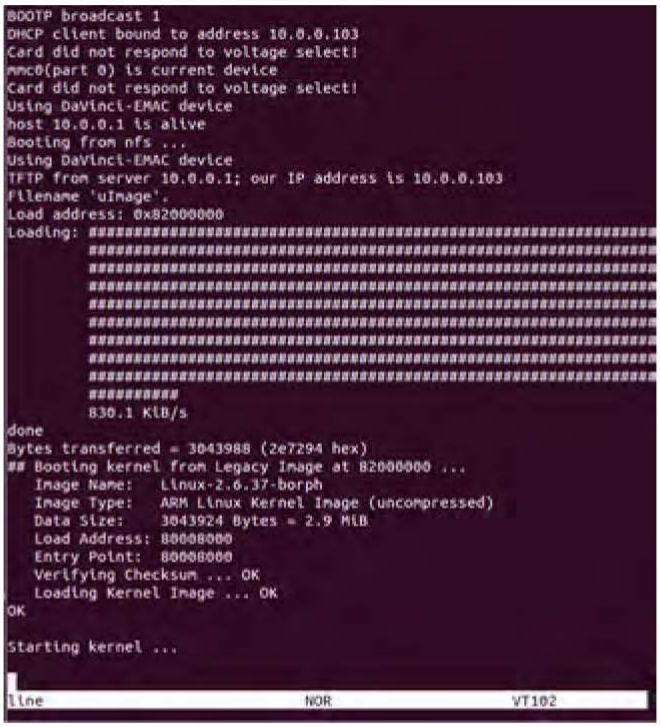


Figure 4.2: Accessing RHINO board via USB port for u-boot configuration

An SD card was used to initially boot the RHINO board. The SD card contained the x-Loader, u-boot and BORPH Linux kernel that board would require to start up. The x-loader and u-boot are both bootloaders with the x-loader used to specifically configure the RHINO board's DDR2 SDRAM and then hands over the booting process to u-boot. These files were then transferred manually to the ARM processor's NAND flash memory through the ARM's serial link and then the board was then switched to boot from the NAND and not the SD card. U-boot was used to load the BORPH OS kernel onto the ARM processor and then control of the ARM was handed to the kernel which would be then accessed via the 100Mbps Ethernet port. Running programs on the board in u-boot without the kernel would have proved difficult as u-boot lacked any default libraries and working files system as its main design purpose was to be a boot-loader environment and not a development environment.

The BORPH kernel was then configured to use the NFS (Network File System) to enable the easy transfer and modification of files from the Control Computer to the ARM processor. The NFS was selected as opposed to using the TFTP protocol transfer files to and from the NAND. The former would have required strenuous transfers to the board involving using the command-line to transfer the files to board specified NAND memory address locations block by block.

4.3.4 Processor-FPGA Bus

The Processor-FPGA bus was used as the main communication link between the ARM processor and the FPGA on the RHINO. It is a parallel bus connected through the GPMC interface on the ARM processor to the FPGA and runs at a bus frequency of 83MHz. This was the communication link the framework utilized to control and manipulate gateware designs running on the FPGA. This parallel bus enables the BORPH OS to read and write to gateware processes running on the FPGA [34, 53] and hence the RACCMS framework utilized it as well.

The bus provides its functionality through the ARM processor mapping reserved memory space onto the GPMC interface. When an address in this region is written to or read from, the GPMC asserts the relevant control line on the FPGA and executes the given command. It should be noted that on the Processor-FPGA bus, an address space reserved on the ARM is identical to the memory addresses on the FPGA. These given address locations are what a gatware designer has to insert in the BORPH IOREGs file to create a register. Table 2.1 illustrates the available ARM memory addresses that are mapped to the GPMC interface. The GPMC interface provides a 16bit data line meaning that the data that is written to and read and from the FPGA is also limited to 2 bytes in a single read or write operation. It is configured by u-boot to support the ARM-FPGA bus when the board is started up. This process involves mapping memory regions to the FPGA as well as configuring the GMPC timings and protocols to interface with the FPGA correctly. The GPMC bus timing diagrams are seen below for both a read and a write operation.

Figure 4.3 illustrates the actions taken by the GPMC interface when performing a read and write operation to a register of a BOF process per clock cycle. As can be seen in Figure 4.2 the bus cycles for a read operation takes 8 GPMC_FCLK[1] cycles, while a write operation takes 6 GPMC_FCLK cycles indicating that a read operation should take more time to perform than a write.

It should also be noted that the Processor-FPGA bus is not used to program or configure the FPGA. That functionally is provided by the serial configuration interface, driven by the SPI port on the ARM processor.

4.4 Gateware Design using BORPH

A hardware process, as mentioned in section 2.1.3.1 is defined as an executing instance of a hardware gateware design [35]. It is created when a BORPH Object File (BOF) with

[1]The GPMC_FCLK is the internal function clock for the GPMC bus and runs at twice the rate of the GPMC clock. This enables events to take place on both the rising and the falling edge of the CLK [53]

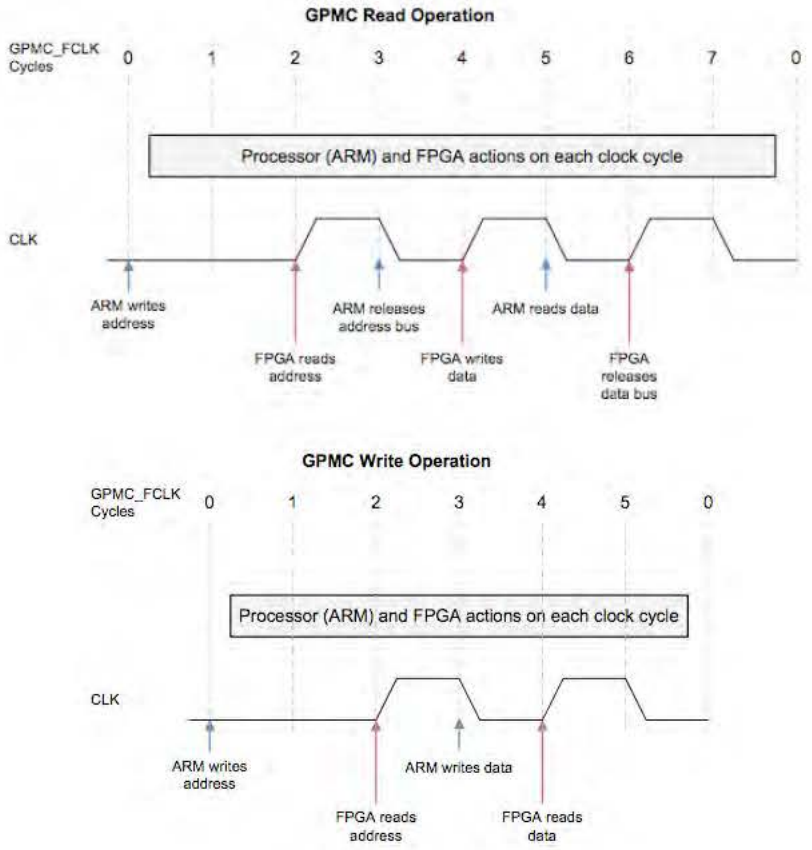


Figure 4.3: GPMC read operation and write operation [53]

the "*.bof*" extension, is executed on the FPGA. A BOF file is compiled into a binary file and is made up of information such as configuration data for the FPGA and a symbol file.

The hardware process has its own memory space and that memory is exported into the user space using the IOREG, which is discussed in detail in section 4.5.1. Each hardware process has its own execution space. Since BORPH is a modified version of UNIX, this means hardware processes can be manipulated and monitored using standard UNIX system calls. This means, for example, similar commands that can terminate a software process, such as UNIX signals SIGKILL and SIGTERM can also be used to kill a hardware process. The steps taken to generate a BOF file are summarized in Figure 4.4 below.

The first step in this process is the actual design of the gateware process using either VHDL/Verilog in Xilinx or using Simulink blocks provided by the Xilinx System generator [35]. In order to be able to interact with BORPH, users have to use data I/O blocks specified in an in-house library. The I/O blocks are then exported and made accessible through the IOREG virtual file.

The next step in the process involves the synthesizing of the designed code using the Xilinx System Generator. This involves clock and reset insertions, the inserting of the processor system micro kernel and the connection of the GPMC interface of all the IOREG blocks specified in the user design. From this step, a symbol file and a FPGA configura-

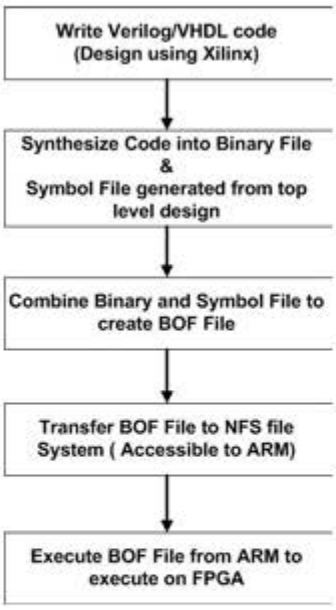


Figure 4.4: Generation process steps of BOF file for execution on the RHINO

tion file are generated.

The third step in this process is the combination of the symbol file and the FPGA configuration file to generate the BOF file. This BOF file is then transferred through the NFS file system on the control computer to the RHINO board via the ARM processor. For the RACCMS framework to work the BOF file and the symbol file need to be contained in a folder named after the particular "*.bof*" file in the user's server folder on the NFS file system, for example *"/username/bofprocessname/bofprocessname.bof*". This is to enable the program to easily locate a particular BOF file and all the files used to generation it.

4.4.1 BOF Process Bin File

This is the FPGA configuration file that is generated by the vendor tool [35]. It is made from the code or blocks designed in Xilinx to be a gateware process that runs on the FPGA. Such generated designs include flip-flops, adders, multipliers amongst the many other logic gates. This ".bin" file is what is combined with the symbol file to make the BOF file that is then executed on the reconfigurable platform.

4.4.2 BOF Process Symbol File

The symbol file is a generated list that contains information about BORPH specified blocks [35] that are shown as the second step in Figure 4.4. The information in the symbol file includes the name of the block, it's memory address on the FPGA, its size and the read and write rights to that particular block. Figure 4.5 shows an example of the symbol file for a gateware design called rhino_adder.

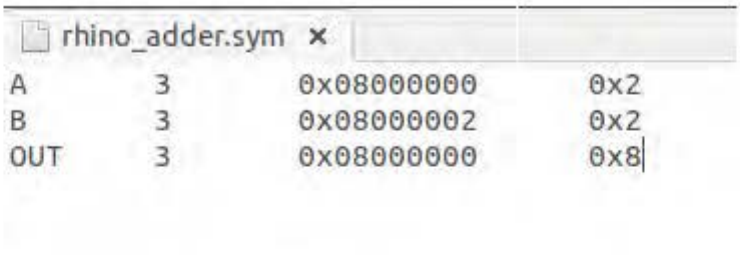


Figure 4.5: Example of symbol file and its contents

It can be seen from Figure 4.5 that the symbol file has the '.sym' extension. The file in this case contains 3 registers named A, B and OUT with information relating to the register separated by tabs in the same line. The first column to the left of the file is the given name of the register. The next column that follows contains the read/write rights to the register. This number can either be a 2 for read only or a 3 for a register that can be read and written to. The third column in Figure 4.5 represents the GPMC memory mapped address location of the register block on the FPGA. The last column on the right represents the size that each register utilizes in memory.

In the design of this project the symbol file is important because the definition of the registers that are accessible via the IOREG interface (which are considered BORPH specific blocks), are defined in it. In order to access information about the registers for a particular BOF file, the symbol file is read by the framework. The symbol file is what is combined during compilation, with the bin file to make up the BOF file. This is important to note as register definition can only be done during the design stage and before the BOF file is generated and not while the gateware process is executing.

4.5 BORPH: Hardware and Software Process Communication [Functional Requirement F2]

The RACCMS framework utilized the BORPH OS communication driver, introduced in section 2.1.3.1, and its hardware-software process communication functionality in its API function calls. The framework abstracts the BORPH communication system calls and implements the whole process involved in executing a BORPH system call into a single phrased API function. This section provides details on how BORPH performs its communication and how the framework incorporated this information into its design.

4.5.1 IOREG Interface

When a hardware process executes in BORPH, it does so with its own memory space as well as execution domain as a software process would. Its shared memory, which contains

the I/O blocks included during the gateware design process mentioned in section 4.4, must then be exported to the IOREG interface to enable access to the running hardware process. The IOREG file is what the framework will accesses to read and write to registers on the FPGA.

BORPH treats hardware and software processes equally and hence the two communicate with each other using standard Linux services and library functions such as file pipes, sockets and signals [34]. The communication between the two processes is normally done through special defined hardware registers, which are memory mapped by software drivers. This process is done by the IOREG interface that encapsulates conventional memory mapped I/O concepts with the Linux virtual file system interface.

The IOREG interface is a subdirectory in the Linux *"/proc"* directory accessible through a standard UNIX file accesses. The *"/proc"* directory contains all hardware specific information about the hardware process. Each virtual file in the IOREG folder represents a characteristic embedded in the user's hardware design. Hence reading from any of these virtual files or writing to one translates to the BORPH kernel reading or writing to the physical construct on the hardware process through a standard message-passing network. Figure 4.6 illustrates how BORPH interacts with the hardware processes running on the FPGA.

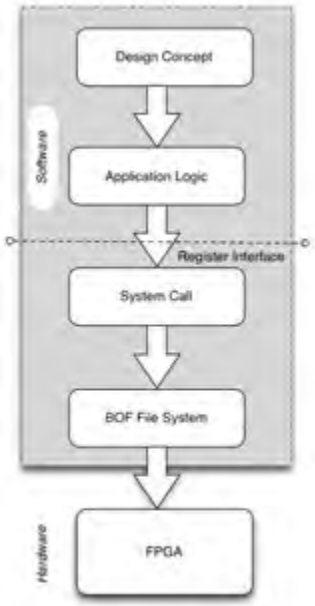


Figure 4.6: How BORPH interacts with FPGA resources [52]

Each virtual file in the IOREG interface has a unique ID, size as well as access information which is all stored in the BOF file header. It should be noted that the IOREG file in this project supports single word registers only as this the largest amount of data the Processor-FPGA bus can transfer in a single operation.

4.6 Cluster Design Details

4.6.1 Network Port Selection

The network port numbers that were used in establishing connections between the server applications on the RHINO boards and the client application on the control computer were between 5507 to 5552 [41]. The use of network ports in establishing connections for remote applications is discussed in section 2.3.3. The selection of these port numbers was because this range provided 45 consecutive free port numbers for use and because they lied in the range of the well known ports used in local and Internet communication [9].

Each server application was assigned a static port number every time it started up and the client application used this port number, obtained from the fileIPAdd.txt, to connect to it.

4.6.2 DHCP Network Configuration [F4]

The DHCP server was setup using the DHCP.config file found in the control computer's */etc/dhcp/dhcp.conf* folder. A subnet was formed using the IP addresses with the 10.0.0.0 bit group and with a netmask of 255.255.255.0 as can be seen in the Figure 4.7 below.

```
# This is a very basic subnet declaration.
#hardware ethernet 00:24:BA:77:A2:8C;
#hardware ethernet 00:24:BA:77:A2:8C;
subnet 10.0.0.0 netmask 255.255.255.0 {

               range 10.0.0.10 10.0.0.49;
}
```

Figure 4.7: Creation of network on which RHINO nodes can connect to cluster from dhcp.conf file

The IP addresses within the subnet were then split into addresses that could be assigned dynamically (when a board connects to the network) and statically. The dynamic addresses were given the range of IP addresses from 10.0.0.10 to 10.0.0.49. The reason for such a small range of dynamic IP addresses is that these IPs would be used by the RHINO boards during its initial configuration when all the boards have a generic MAC address of 00:00:00:00:00:00. Once the board has obtained its unique MAC address, it was then assigned a static address as can be seen in Figure 4.8. The image illustrates a RHINO board with the MAC address of 00:24:BA:77:A2:8C assigned an IP Address of 10.0.0.103. As each board is added to the cluster its MAC address can be assigned a unique static address by adding it to the dhcp.conf file in a similar manner as seen in Figure 4.8.

```
host rhino{
  hardware ethernet 00:24:BA:77:A2:8C;
   fixed-address 10.0.0.103;
}

host rhino1{
        hardware ethernet 00:24:BA:7C:B2:3C;
        fixed-address 10.0.0.104;
}

host rhino2{

      hardware ethernet 00:24:BA:78:12:ED;
      fixed-address 10.0.0.105;
}
```

Figure 4.8: Fixed IP addresses assigned to RHINO boards on DHCP server sitting on control computer from dhcp.conf file

Chapter 5

RACCMS FRAMEWORK SOFTWARE DESIGN

The software design of the RACCMS framework consisted of five main stages. These are namely the development of the framework's use case diagrams, the design of the RACCMS class diagrams, the framework's sequence diagrams, the flow diagram and state diagrams and lastly the RACCMS framework's code implementation. A description of each of the mentioned methods is given in the subsections below.

5.1 RACCMS Framework Use Case Diagrams

Once the baseline requirements were established, the next step was a conversion of user requirements into a software design, architecture and implementation plan. The first step in this process was the development of use case diagrams. The main users of the framework and the task they could perform were identified as follows:

1. ***System Administrator:*** This user also known as the administrator, would be the individual responsible of managing the cluster of RHINO boards connected to the cluster. His roles would include starting up the framework and having the client connect to and disconnect from all the RHINOs on the cluster. The administrator would be able to control BOF processes running on the RHINO and select a board for use and release it. Of all the users of the framework, the administrator would be able perform all the functionality available in the framework (being all of the tasks the framework offers). The administrator was the user specified in requirements [U2], [U3], [U4], [U5] and [U7].

2. ***Student User:*** The student user would consist of all the other users of the framework excluding the administrator. The student user's interaction with the framework would be more limited than that of the administrator. This user's actions

would limited to being able to select and release RHINOs on the cluster and being able to control a BOF process on the board. The student user was the user specified in requirements [U1], [U3], [U4], [U5] and [U7].

3. ***RACCMS framework client application:*** The framework client application was considered a user of the server application on the RHINO board, as it requested services from it. The client application was able to connect and disconnect from a particular board, read and write to a register on a running BOF process, start and stop a BOF process, as well being able to list all the available registers on each board.

Detailed use case diagrams illustrating each of the framework users and their interaction with the software can be seen in Figure 5.1 and Figure 5.2.

Figure 5.1: Use case diagram of administrator user

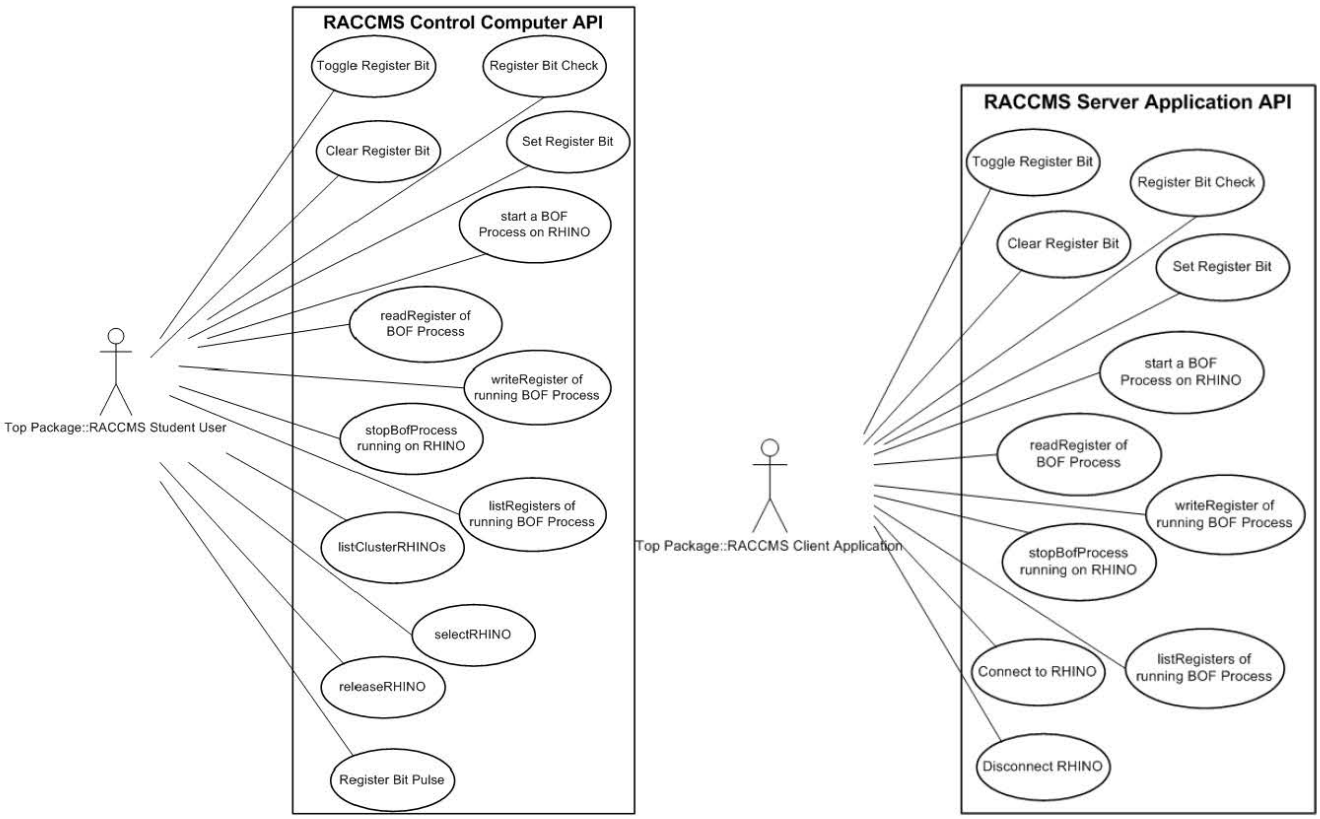


Figure 5.2: Use case diagram of the administrator, student and client application users interaction with RACCMS framework

As can be seen in the use case diagrams above the administrator can perform a total of 16 tasks on the framework where as the student user can only perform 13 functions that do not directly change the cluster's setup status. These diagrams were then used to derive the next stage of the framework's design.

5.2 RACCMS Framework Class Diagrams

The second step in the design of the framework was the derivation of class diagrams from the use case diagrams presented in section 5.1 and the server-client design mentioned in section 3.4.1. It should be noted that the implementation of the framework code was C based and hence did not use true classes.

The ten framework classes were identified, six to develop the client application of the framework and four for the server application. These classes are listed below with a description of their functionality in the framework.

5.2.1 Client Application Classes

This subsection covers the classes that will be available in the client application that will run on the control computer.

1. ***raacms client interface (main.c) :*** This is the main point of contact with the entire framework, that is to both the client application and the server application. It enables users to access all the API functions of the RACCMS framework. It contains one function which is the *main()* function. It is responsible for maintaining the status of the rhino_status_table which keeps a record of all the RHINOs that are connected to the framework at any given time.

2. ***raccmsGUI (raacmsgui.c):*** This is an alternative interface to the RACCMS framework, implemented as a graphical user interface. It provides all the functionality of the raacmsclient interface class in three main windows, which are discussed further in section 5.6.

3. ***client management fctns (managementfctns.c):*** This class was designed to contain all the management related functions of the framework. These functions include connections and disconnections of the RHINOs in the cluster, selecting and releasing boards for use as well as general cluster status information.

4. ***client management support fctns (managementsupportfctns.c) :*** This class provides supporting functions in order for the clientmanagementfctns class to work. The functionality provided by this class include checking if a user has access to a board, creating rhino_status_table elements as well as updating the current rhino status table. It was designed to contain functions that only the clientmanagementfctns would use hence functions in this class will not be available to the users of the framework via its API library.

5. ***client raccms control protocol (raacmscp.c) :*** This class, as suggested by its name, was designed to contain all the control related function calls. This included all the functions that used the designed control protocol to communicate with the server application running on the RHINO. The clientraccmscp class functions include loading a BOF process onto the RHINO board, reading and writing to a register and listing the registers on the board.

6. ***client transport cp (clienttransportcp.c) :*** The clienttransportcp class is responsible for all functions related to the communication between the client application on the control computer and server application on the RHINO. It performs TCP based communication and hence implements functions such as *tcp_send()* and *tcp_recieve()* as well as TCP based connection and disconnect methods.

5.2.2 Server Application Classes

This subsection provides a description of all the classes available in the framework's server application that sits on the RHINO's ARM processor.

1. ***raacms server interface (raacmsserver.c) :*** This is the main point of contact with the server application running on the RHINO board. It is responsible for starting the server application on the RHINO board and handling connections from the client application through its *main()* function. It also provides an interface to interact with the server API functions. It enables a user to access the server's functions through a look_up table implemented as shown in Table 5.1 below.

CLIENT MESSAGE ID	STATE NAME
1	WRITE_ REG
2	READ_ REG
3	LOAD_BOF_FILE
4	LIST_BOF_PROC_REG
5	KILL_BOF_ PROCESS
6	SET_REG_BIT
7	CLEAR_REG_BIT
8	REG_BIT_PULSE
9	REG_INCREMENT
10	REG_BIT_TOGGLE
11	REG_BIT_CHECK
12	REG_TEST_SET

Table 5.1: The server application's look up table states

Based on a given Client Message ID the server application would then call the corresponding function from its API library.

1. ***raccms server transport cp (raacmsservertransportcp.c) :*** This class presents the same functionality on the server application as the client transport cp class on the client application. It is responsible for all TCP based communication between the server application and the client application hosted on the control computer. It implements two connection functions, one for the server application and the second one to handle the client application's connection. It also has TCP based send and receive functions for sending control packets between the two applications.

2. ***raacms server control protocol (raacmsservercp.c) :*** This class contains functions that interact directly with the FPGA and the process running on it. The functions in this class directly communicate with a BOF process on the RHINO's FPGA. These include functions that enable programming and stopping a BOF process on the FPGA, reading and writing a register of a running BOF process and lastly listing registers that the board has access to.

3. ***raccms server management fctns (raacmsservermanagement.c) :*** This class provides basic server management functionality. It enables the server to get network information from the board it is running on, checking if the RHINO has a BOF process running on it and if the user requesting a service has access to the BOARD.

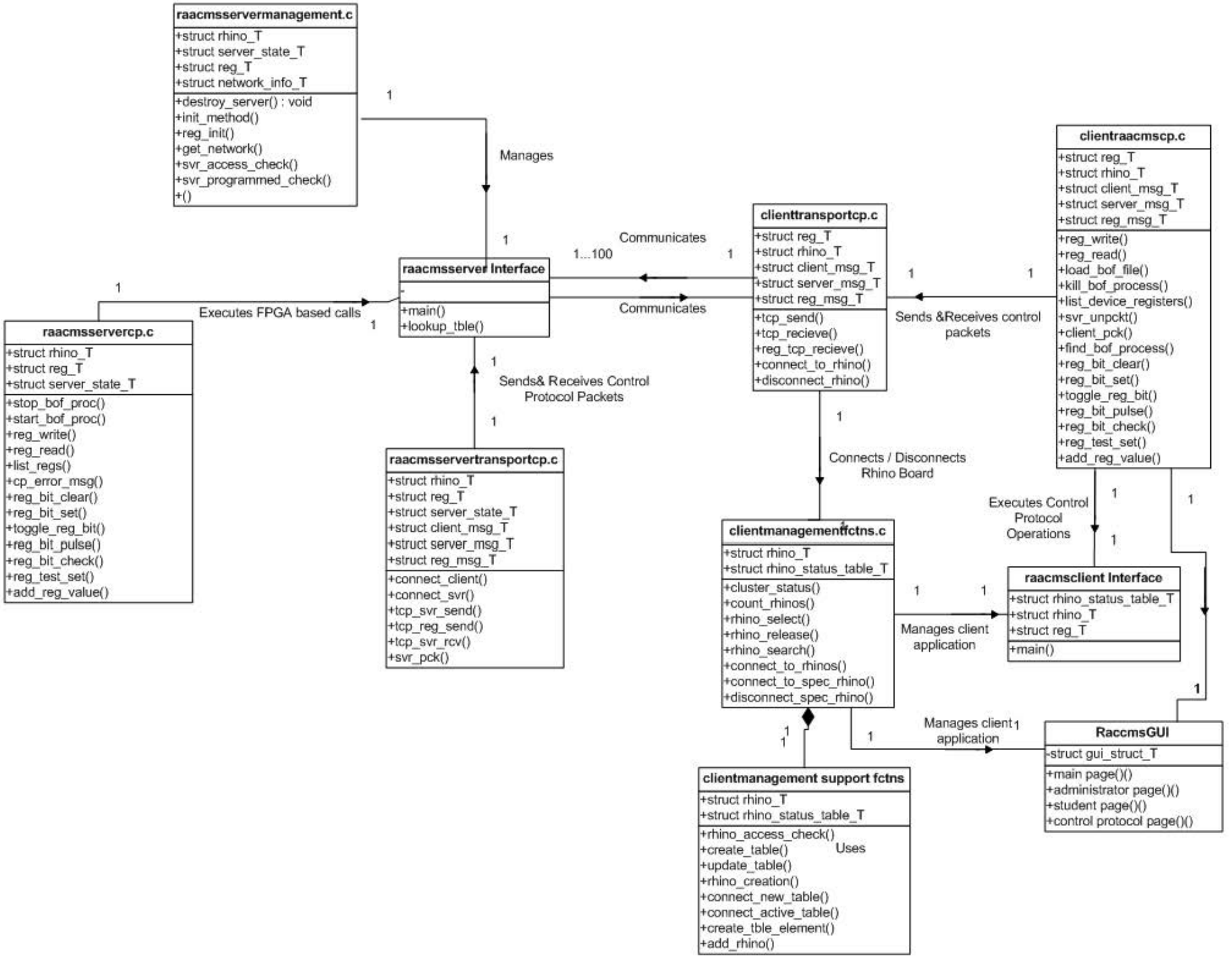


Figure 5.3: RACCMS Framework class diagrams and how they interact with each other

Figure 5.3 above illustrates how all the mentioned classes from both the server application and client application interact together in a class diagram. It can be seen that all the classes in the diagram have a one to one relationship with each other except for the client transport cp class which connects to the RACCMS server interface that has a one to many relationship. The client application is the user of the server application and communicates with the RACCMS server interface header file through its client transport cp header file.

On the client side, the raccmsclient interface is used to call only two of the 5 classes, that is the client managementfctns class and the client raccms cp class. These two classes in turn then call the client transport cp class that will communicate with the server. On the server side, the raccmsserver interface is the central class and calls all subsequent server application classes from it.

5.3 RACCMS Framework Sequence Diagrams

The third stage in the design of the RACCMS framework was the development of sequence diagrams to illustrate how the given tasks identified in the use case diagrams would be performed using the defined classes in section 5.2. Sequence diagrams were used to show how each function in the control protocol would execute and interact with the whole framework in time. Sequence diagrams for the main functions in the framework were developed to aid in the design of the subsequent flow diagrams presented in the section below. The figures below shows the sequence diagrams for the six functions in the framework with the rest of the framework's sequence diagrams are illustrated in APPENDIX C.

5.3.1 Connect RHINOS to Cluster

The first sequence diagram illustrates how the framework would execute a request by a user to connect to the boards on the cluster. This functionality can only be performed by the administrator. The sequence diagram for the connect function is for both the possible connection routes the framework could take and can be seen in Figure 5.4 below.

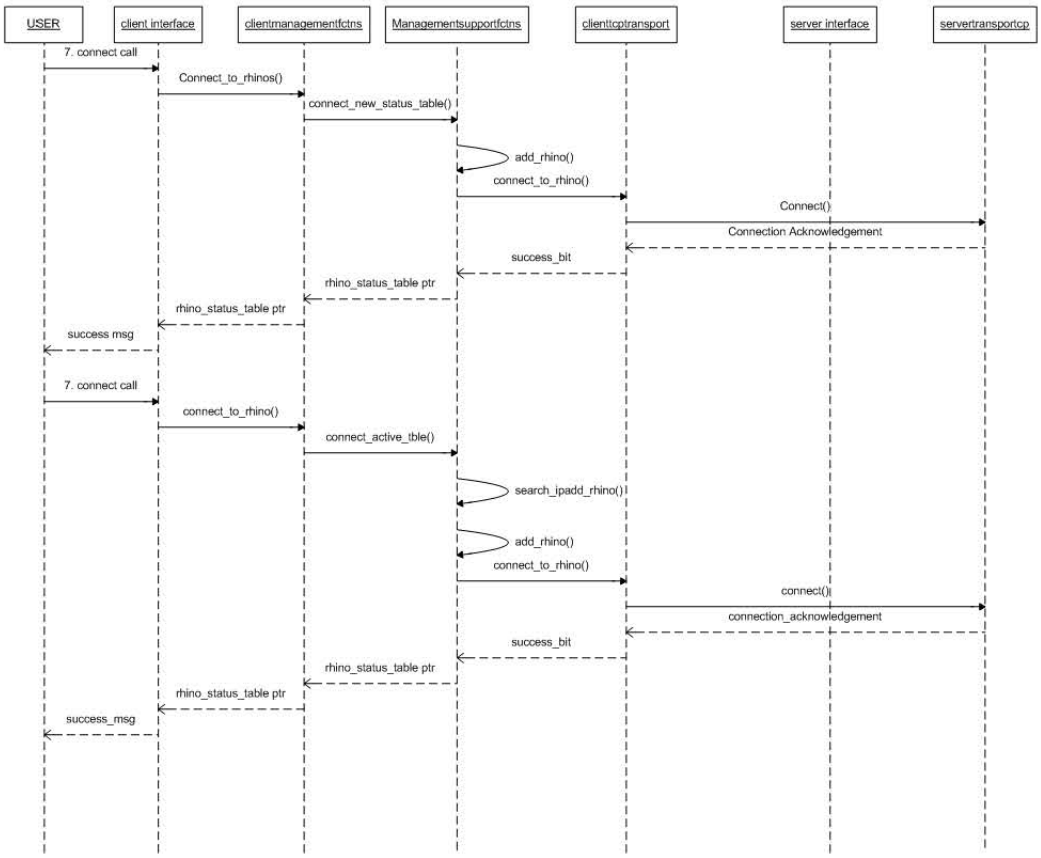


Figure 5.4: RACCMS framework connect() to RHINO sequence diagram

As can be seen in the top half of the illustration above, in order for a user to connect to all the available RHINOs on the cluster, the user would call the connect function. Once the function was called the framework would then call the *connect_to_rhino()* function call in the client interface class. This would then subsequently call the *connect_new_status_table()* and then the *add_rhino()* function from the management function class and the rest of the functions that then follow the step as shown in the sequence diagram above.

The second connection function call that caters for existence of rhino_status_table in the framework, seen in the bottom half of Figure 5.4. Due to the preexistence to a table in the framework this function only attempts to reconnect to boards that had failed to connect in the initial connection phase. It differs from the first connection call in that, in the third phase of the sequence, the *connect_active_table()* function is called in place of the *connect_new_status_table()*. This is then followed by a *search_ipadd_rhino()* function call that searches the cluster to see if the board is not already in the cluster. The subsequent function calls in sequence diagram after this call are then the same as the ones in the first connection function call.

5.3.2 Disconnect RHINO From Cluster

This sequence diagram illustrates how the disconnect function would execute as a function of time when called by the user. This functionality, as with the connection method,

can only be executed by the administrator of the framework. An image of the sequence diagram of the *disconnect()* function call can be seen in Figure 5.5 below.

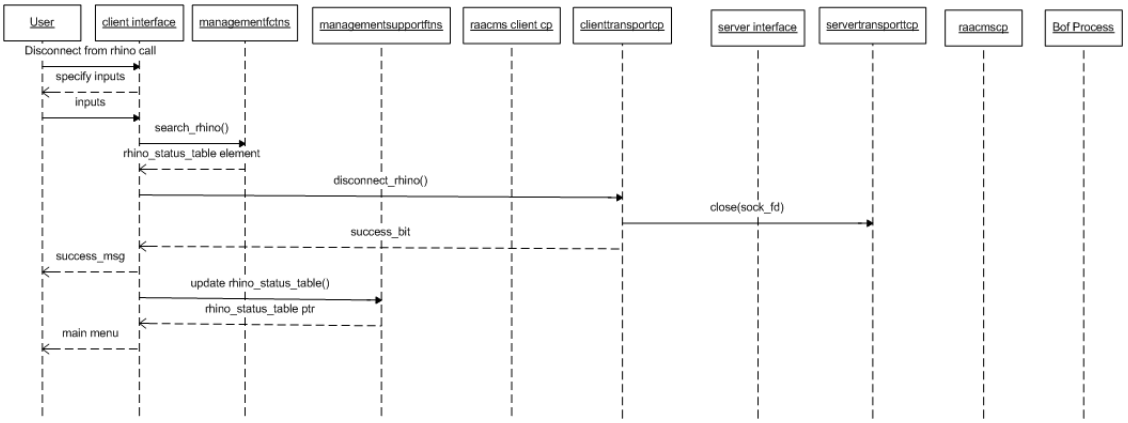


Figure 5.5: RACCMS framework disconnect() sequence diagram

As can be seen in the image above, the user will be requested to enter parameters which would be in this case the RHINO number the user would want to disconnect from. From this the framework would then search for the given RHINO in the cluster. A RHINO from the table would then be returned in the form of a status table element. This element would then be used as input into the *disconnect()* function call, that would then in turn call the *close()* function call that would initiate a disconnection from the server application. As success bit would then be returned from the client transport cp class all the way to the back to the client interface.

5.3.3 Load BOF File onto RHINO Board

The load BOF file function call is a control protocol function, in that it requires the control protocol in order to execute. This function can be called by either the administrator or the student user. The sequence diagram of how it would execute utilizing the given classes in the framework can be seen in Figure 5.6 below.

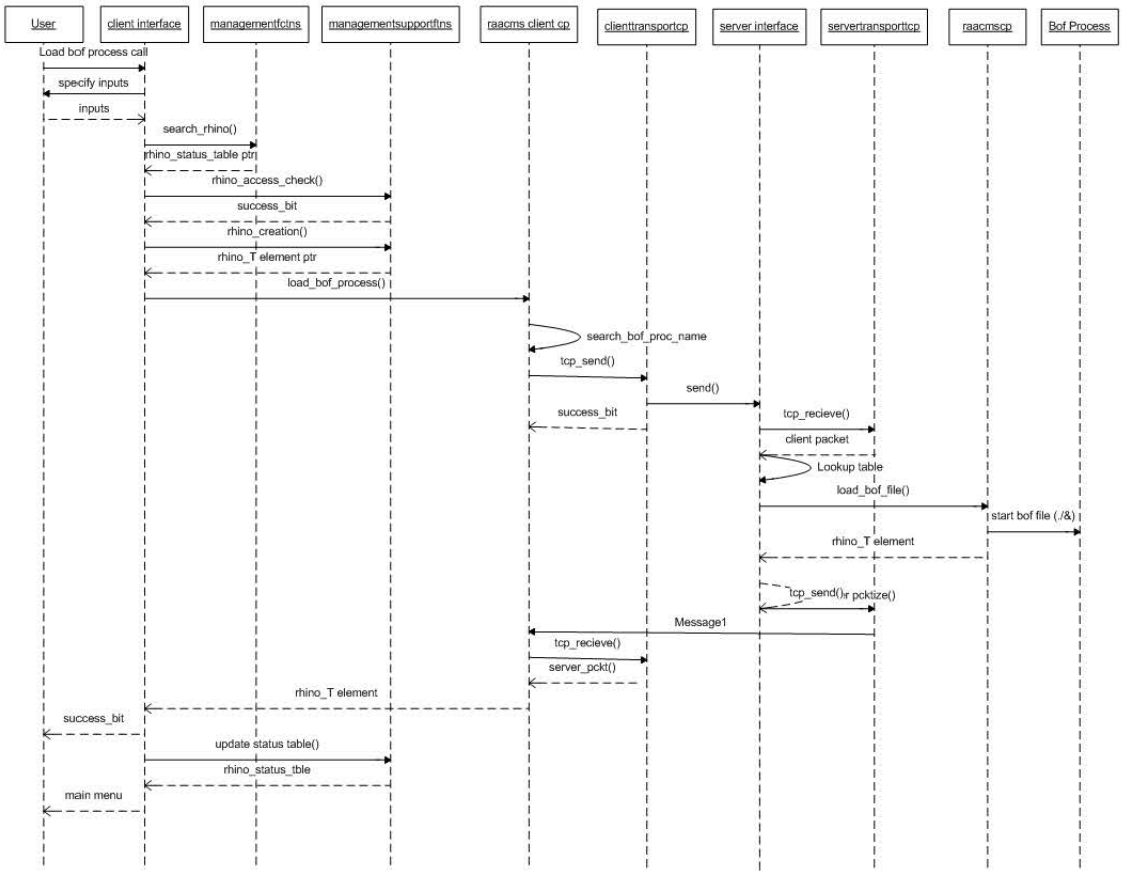


Figure 5.6: RACCMS framework load_BOF() function call sequence diagram

As can be seen in the above image, this function also requires inputs in order to execute which include the RHINO number, the user server name and the BOF process name to be loaded. Once the inputs are given the RHINO board to be used is then searched for in the cluster, and once it is found the *rhino_access_check()* method would then be called by the client interface to see if the user has access to that requested board. Once the success_bit has been sent back to the client interface, a RHINO element is created that contains all the relevant information from the RHINO returned from the rhino_status_table earlier. This is then used as input in the *load_bof()* function in raccms client cp class. The load bof process then searches for the given BOF process name in the NFS server on the control computer and once it has found the BOF process, it then sends the request to the server. Once the server application has finished executing the load BOF function as seen in the diagram above the success_bit is sent back to client interface and the status table is updated and the main menu returned back to the user.

5.3.4 Read Register From BOF Process

The read register function call is also a control protocol function and like the load BOF function above can be called by either the administrator or the student user. Its sequence diagram can be see in Figure 5.7.

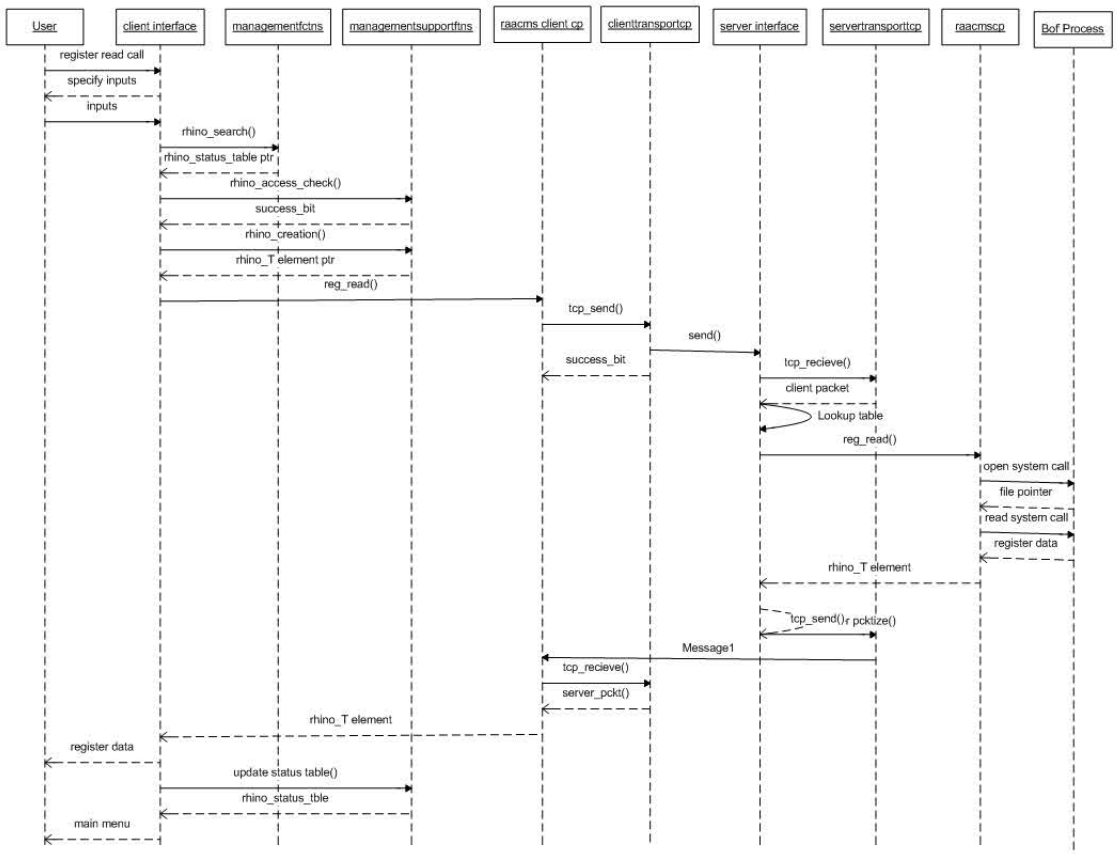


Figure 5.7: RACCMS framework read_register() function call sequence diagram

The read function assumes that a BOF process is already running on the RHINO board and some data has been previously written to the register in order for it to work. If there is no BOF process the read function will terminate at the *rhino_access_check()* with a failure and the main menu restored to the user. The read function also requires inputs, namely the RHINO number and the register name. It then, like the load function above, follows the same sequence until the 7th sequence step that calls the read function. The read function then packetizes the data and sends it using the control protocol and the *tcp_send()* function. From that step the function call then again follows the same sequence of steps as those of the load function specified above except instead of returning a success_bit to the user, data obtained from the register is returned.

5.3.5 Write to BOF Process Register

The write register function call is a control protocol function and like two previous functions above, can be called by either the administrator or the student user. Its sequence diagram can be see in Figure 5.8.

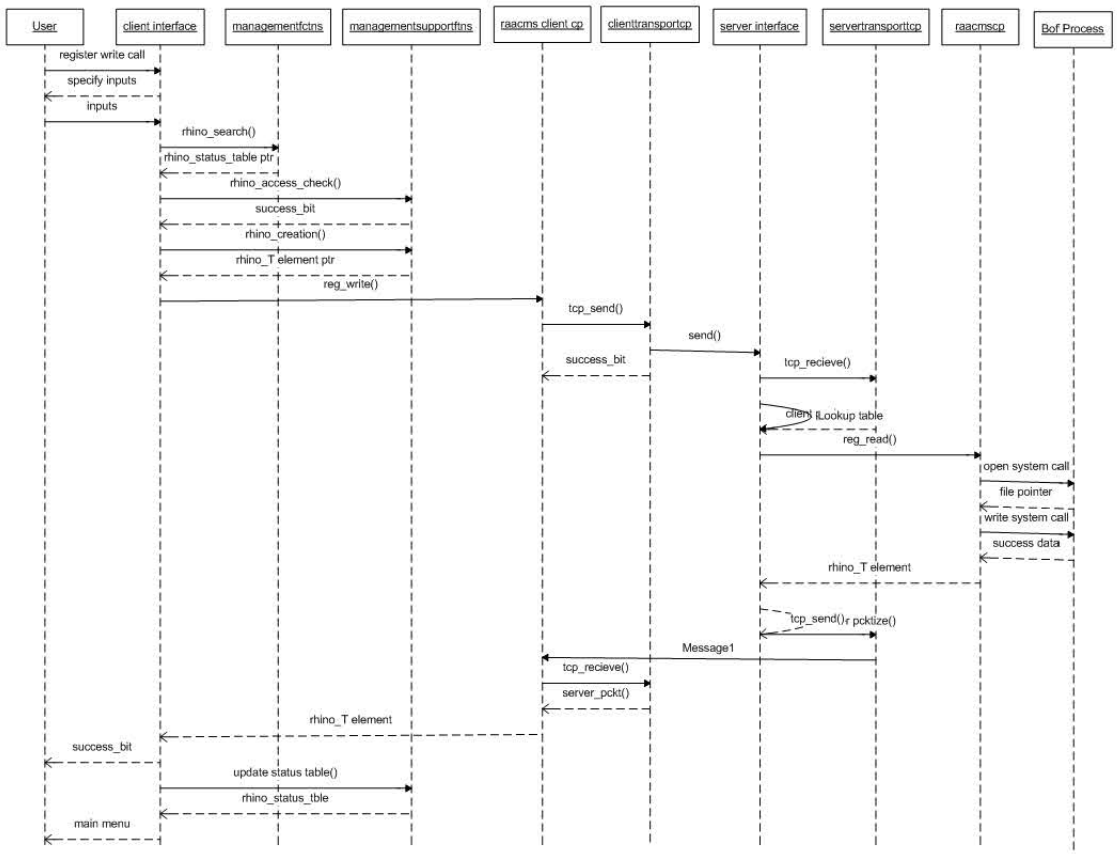


Figure 5.8: RACCMS framework write_register() function call sequence diagram

The write function assumes that a BOF process is already running on the RHINO board in order for it to work. If there is no BOF process the write function will terminate at the *rhino_access_check()* with a failure and the main menu restored to the user. This function also follows the same set of sequences in its execution as the read function above except for the 7th step that will execute the *reg_write()* instead of the *reg_read()* function. As well this function returns a success_bit as opposed to data as mentioned in the read register function call above.

5.3.6 Kill BOF Process on RHINO Board

This sequence diagram illustrates how a running BOF process on the RHINO board will execute and the sequence steps it will take when executing. This function is part of the control protocol functions and like the three function calls above, it requires that a BOF process be executing. The sequence of steps that are followed can be seen in Figure 5.9 below.

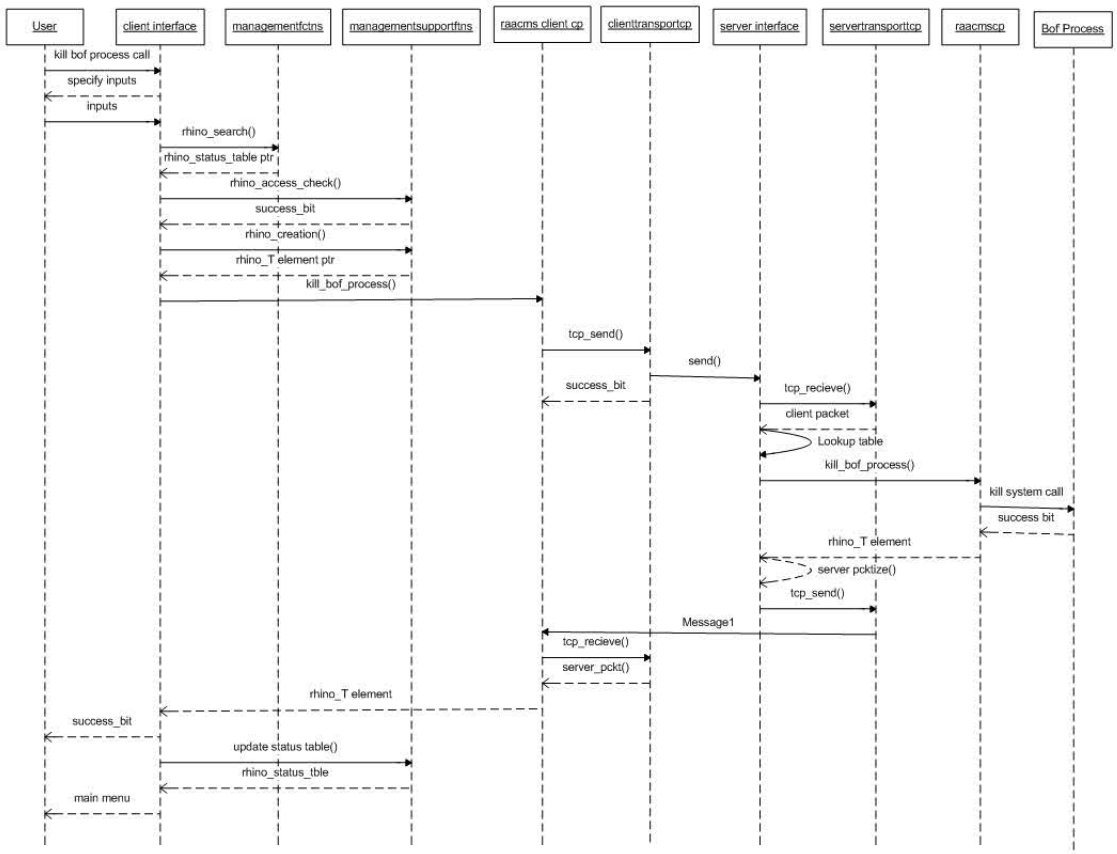


Figure 5.9: RACCMS framework kill_BOF() process sequence diagram

From the image above it can be seen that as with all the above control protocol functions, the sequence of steps of the function call is the same except for the 7th step that calls the *kill_bof_process()* function. A success_bit is returned to the client interface upon successful execution of each of the sequence steps shown in Figure 5.9.

5.4 RACCMS Framework Flow and State Diagrams

The flow diagrams were used to design how the code would be implemented. The flow diagrams were derived from the sequence diagram described above. Each of the main framework functions in the client and server application were represented as a flow diagram before being implemented.

5.4.1 Server Application Flow and State Diagrams

This section describes the developed server application functions. However the flow diagram shown in this subsection are for the control functions that interact specifically with the FPGA on the RHINO or a BOF process executing on the FPGA. Figure 5.10 is an image of the flow diagrams for the control functions. The rest of the flow diagrams for the server application are in APPENDIX B.

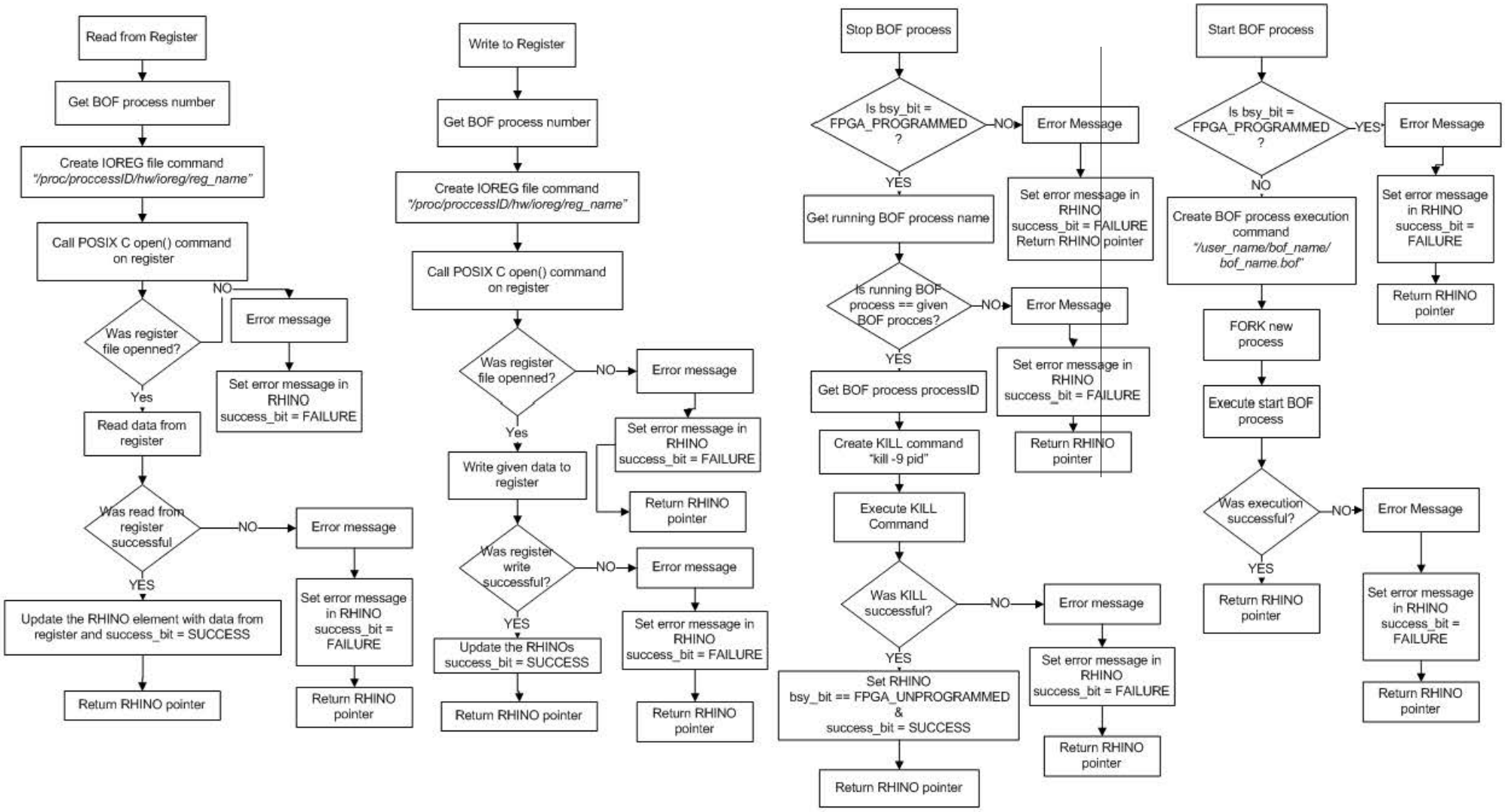


Figure 5.10: Server application control protocol flow diagram

A brief description of the functions in the server application given above and in APPENDIX B are given below:

1. *reg_read():* This function can only be performed once the FPGA has been programmed with a BOF file and the register has been written to. The only parameter the *reg_read()* function call will be able obtain is the data in the register. The data in the register will be obtained through two standard C I/O functions, *open()* and *read()*. The data from the register is limited to 16bits (2 bytes) as that is the maximum size the FPGA_ARM bus can transfer.

2. *reg_write():* As with the *reg_read()* fnuction, the *reg_write()* function requires a BOF process to be running on the FPGA. The only parameter that will be changed in the register will be the data contained in it. All other details about the register that requires change, will need to be specified in the design of the gateware logic and recompiled as mentioned in section 4.4. The data in the register will be written using two standard C I/O functions, *open()* and *write()*. The data written to the register will be limited 2 bytes.

3. *start_BOF_process():* This function checks first if the board is already programmed with a BOF process before it can execute the requested process by checking if the *"bsy_bit=FPGA_PROGRAMMED*". Once this check has been completed the BOF process is then executed by forking a new process and the BOF process is then executed on the new forked process using *"exec"* system call. The bsy_bit is then update from FPGA_UNPROGRAMMED, to *"bsy_bit = FPGA_PROGRAMMED"*.

4. *stop_BOF_process():* This function needs to check first that the RHINO board is programmed. If the board is not programmed the function is unable to execute and terminates with an error message. Over if the board is programmed the function then proceeds to check if the running BOF process is the same as the BOF process requested to be stopped. The stop function then proceeds to call the *"KILL -9 pid"* system call, where the -9 extension calls the *SIGKILL* command that stops the process immediately and can not be ignored. A state diagram of the different states the RHINO board goes through using the start and stop function calls can be seen in Figure 5.11. As can be seen in the image above the two states that a BOF process

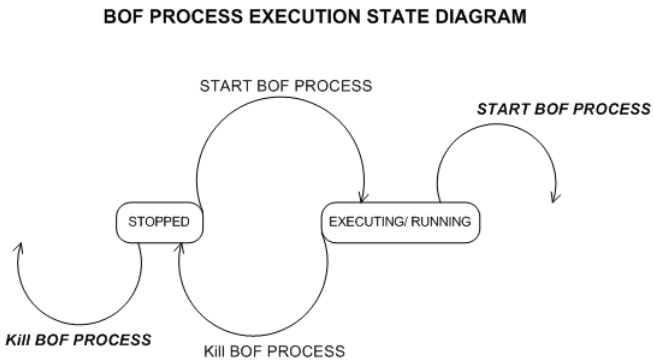


Figure 5.11: RACCMS server application BOF process execution state diagram

can go through is the STOPPED STATE and the EXECUTING/ RUNNING state.

5. *reg_bit_clear():* This function utilizes the register read and write functions and hence has to meet the requirements for these functions in order to work. The reg_bit_clear() function clears a given bit of a named register by first checking that the named bit of a given register is between 0≤bit≤15, and then performing the following action *"reg_data &= ~(1<< bit_to_be_cleared)"* on the data read from register. Once it is done clearing the bit the data is written back to the register using the reg_write() function call.

6. *reg_bit_set():* This function sets a given bit of a named register. It, like the reg_bit_clear() function call, also utilizes the register read and write functions and as such has to adhere to the requirements that are set by these functions. It starts by checking that the named bit to be set is between 0≤bit≤15 and then sets the register data bit using the bitwise OR operator in the MACRO defined as *"reg_data |= (1 << (bit_to_be_set)"*.

7. *reg_bit_pulse():* This function performs a pulsing effect on a given bit from a named register. It achieves this by clearing, setting and then clearing the bit in this exact order. The *reg_bit_clear()* and *reg_bit_set()* functions are used to achieve this. The *reg_bit_pulse()* function also first performs a check to make sure the bit to be pulsed is within the given range of register value.

8. *toggle_reg_bit():* This function changes the status of a given bit by toggling it. This means if a bit is set it clears it and vise versa. It does this by using the XOR operation in the following manner "*reg_data* ^= (1 << *(bit_to_be_toggled)*". It, like functions 5 and 6, uses the *reg_read()* and *reg_write()* function calls and also performs a check to make sure that the given bit value is between 0≤bit≤15.

9. *reg_bit_check():* This function is used to check that a given bit in a named register is set. It uses the AND operator in order to perform this check in the following defined MACRO *"reg_data & (1 << (bit_to_be_checked)"*.

10. *reg_test_set():* This function checks that a given bit of a register value is set and if not sets it. It achieves this by using the *reg_bit_check()* and *reg_bit_set()* function calls mentioned above.

11. *add_reg_value():* This function increments the value of a given register by one. It achieves this by reading the data from the register using the *reg_read()* and then incrementing it by one and writing it back to the register using the *reg_write()* function.

12. *list_regs():* Utilizes the symbol file with the *".sym"* extension mentioned in subsection 4.4.2. The function will access the symbol file using the user's server name and the .bof process name. This function lists the contents of the symbol file using the *"more"* Linux command. This function then reads all the information from the symbol file into the respective fields of the reg_T structure. This structure is then returned to the lookup table for packetization to be sent back to client. If the BOF process has no register, the array will set the reg_T structure to NULL. If the BOF process does have registers, then the last element of the register will be NULL to indicate there are no more registers available. In order for this method to work, it is important to note that elements in the symbol file need to be separated by a tab.

13. *look_up_table():* This function determines which function to call on the RHINO board based on the client message ID number. Once the look up table function is called the table is set with an initial *State* = *NONE*. The State is then set to the message ID from the client protocol message and then using the look up table illustrated in Table 5.1, it will call the appropriate function from the control functions list. Figure 5.12 shows how the look up table handles some of the given states mentioned in the table.

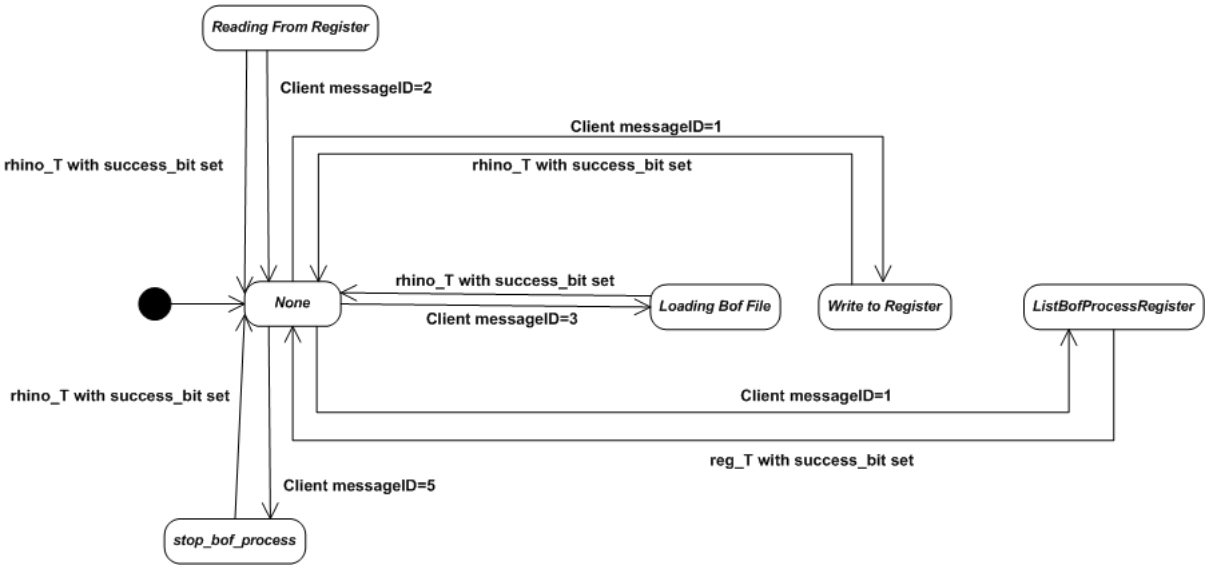


Figure 5.12: Look up table state machine diagram

14. *server_connection() and client_connection()*: These functions are best described in a state diagram form. The server application has 3 states, *"STOPPED"*, *"WAITING FOR CLIENT CONNECTION"* and lastly *"SERVER EXECUTING"*. The server changes from all these states by either calling the *server_connection()*, that connects to the server application, the *connect_client()* function that connects the client application and when the server application just starts running. The *server_connection()* function call takes the server from the stopped state to the waiting for client connection state. To move from that state to the server executing state, the client has to initiate a connection using the *client_connection()* function call. The connect client from the server executing state will move the client to an error state but this will be tested to ensure that this executes as designed. A detailed state diagram explained above is given in Figure 5.13.

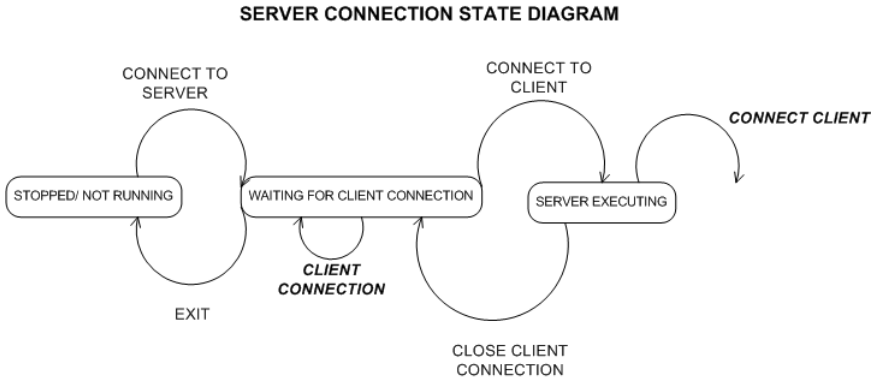


Figure 5.13: RACCMS server application (server) connection state diagram

It should be noted that functions 5 to 11 mentioned above require multiple operations to the same registers and hence implementing all these operations on the RHINO in a single function call reduces the user's network traffic cost. This should result in a much faster execution of the given function by the framework as compared to performing multiple operations across the network to achieve the same functionality.

5.4.2 Client Application Flow and State Diagram

This section covers the design of the functions in the client application using flow diagrams. However the flow diagrams shown in this subsection are for the management function only. These are specifically functions that handle the management of the cluster of RHINOs connected up to it. Figure 5.15 is an image of the flow diagram for the management function. The rest of the flow diagrams for the client application can be found in APPENDIX B.

1. *count_rhino():* This function counts the number of RHINOs that are connected to the cluster at any given time. The RHINO has to be in the "ACTIVE *state = 1"*

on the cluster in order to be counted. The value returned from this function call is an integer representing the numbers counted.

2. *cluster_status():* This function displays the status of the cluster from the RHINO_STATUS_TABLE structure. This function lists all the *"ACTIVE"* and *"INACTIVE"* boards on the cluster. The output is displayed on the GUI or command line interface.

3. *rhino_select():* This function selects a RHINO board from the cluster. The framework first checks if the board is on the cluster and is "ACTIVE" then it selects the board, sets the actv_bit to the "INACTIVE" state and returns the updated pointer to the status table.

4. *rhino_release():* This function releases a RHINO that has been selected using the *rhino_select()* function. It uses the users program_id to determine if the user has access to board. It then releases the board by setting the actv_bit back to "ACTIVE" making it available for use. A state diagram representing the state shifts can be seen in Figure 5.14. The select RHINO and release RHINO calls in bold are transitions to unknown states that will be tested in the testing chapter.

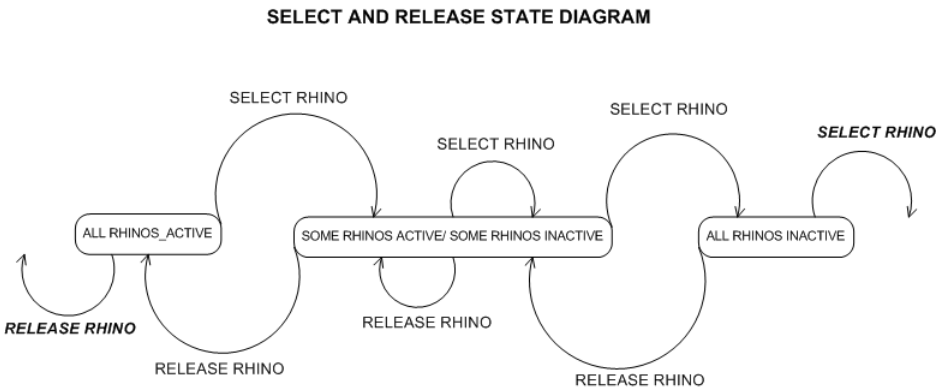


Figure 5.14: State diagram for selecting and releasing RHINOs in the cluster at any given time

1. *rhino_search():* This function searches for a RHINO board in cluster using a given RHINO number. A pointer to the found RHINO will be returned else a NULL pointer will returned if the board could not be found.

2. *connect_to_rhinos():* The *connect_to_rhino()* function illustrated in APPENDIX B, Figure B4, present two options for connecting to RHINOs in the cluster. The first option for connection will be an initial connection of the framework to the cluster, and the second option is the second attempt to connection to a status table that already exists. The number of RHINOs and their IP addresses and preferred port number for connection will be specified in a text file that will only

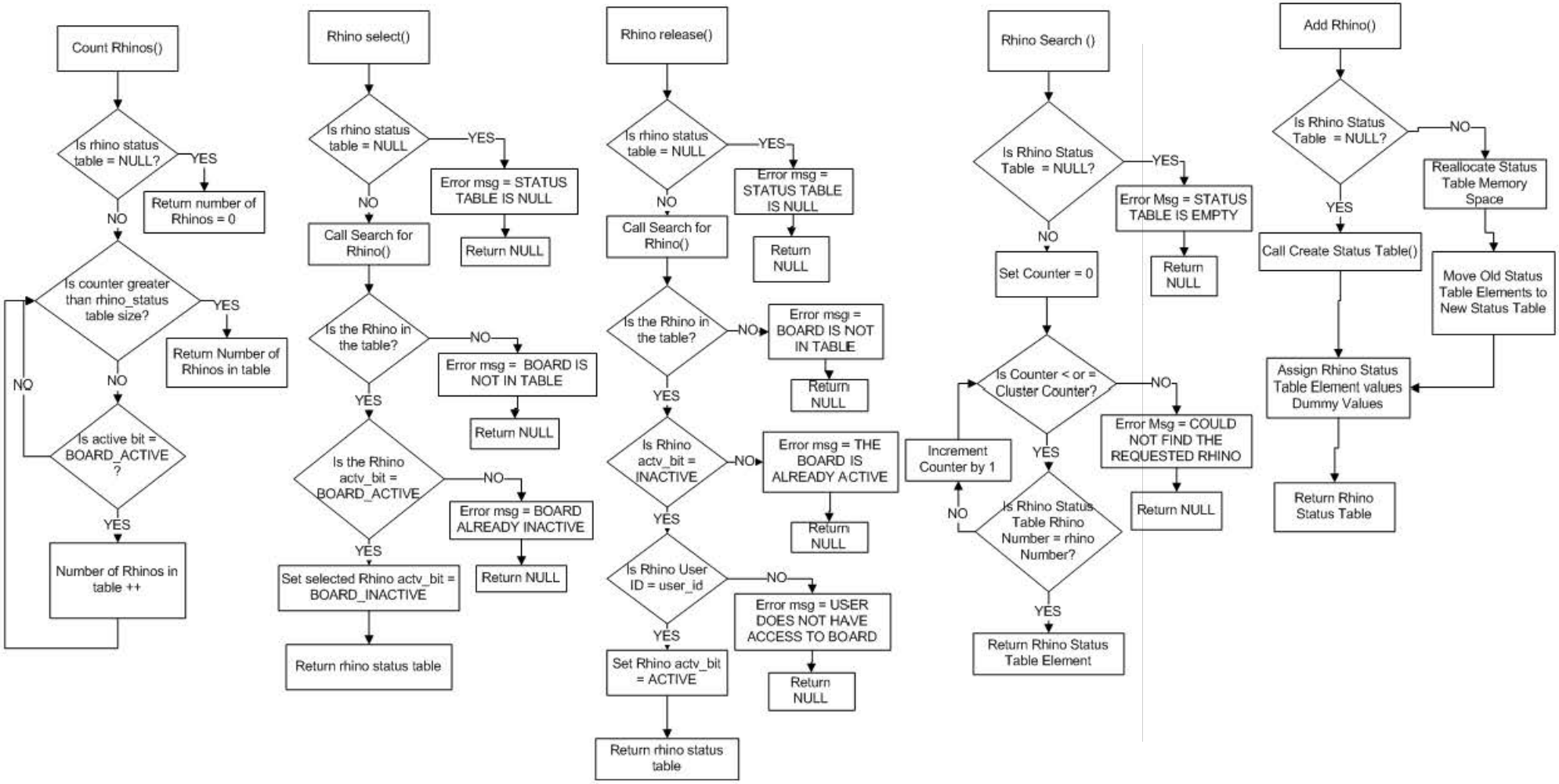


Figure 5.15: Client application control protocol framework functions

be accessible to the cluster's administrator. For this project the file used was the *"fileIPAdd.txt"*. Once a successful connection has been made to a RHINO, that RHINO's actv_bit is set to "ACTIVE" else the board is added to the cluster but the actv_bit is set to *"INACTIVE"*.

3. *connect_specific_rhino():* This method allows for a user to connect to a specific RHINO on the cluster that is already on the cluster or not yet added to the cluster. Should the IP address and port number not already be assigned to an element in the table, it will create an new element else it will attempt to reconnect to the specified board already in the cluster. The connect RHINO and disconnect RHINO state in bold represent transitions to no states that will be tested in the testing chapter 6.

4. *disconnect_rhino():* This function disconnects a specific RHINO from the cluster by using the *close()* function on a given socket id number. The function will have to be in the cluster and *"ACTIVE"* for the disconnect function to succeed.

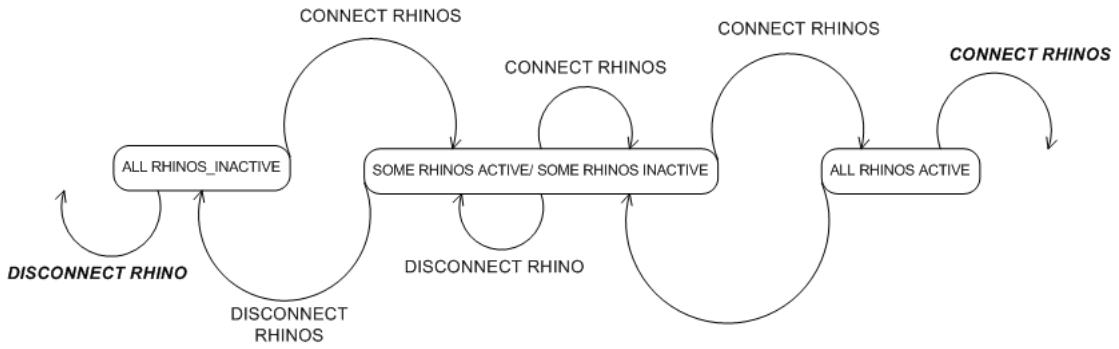


Figure 5.16: Diagram illustrating how the framework moves from each state using the connection methods and disconnect methods.

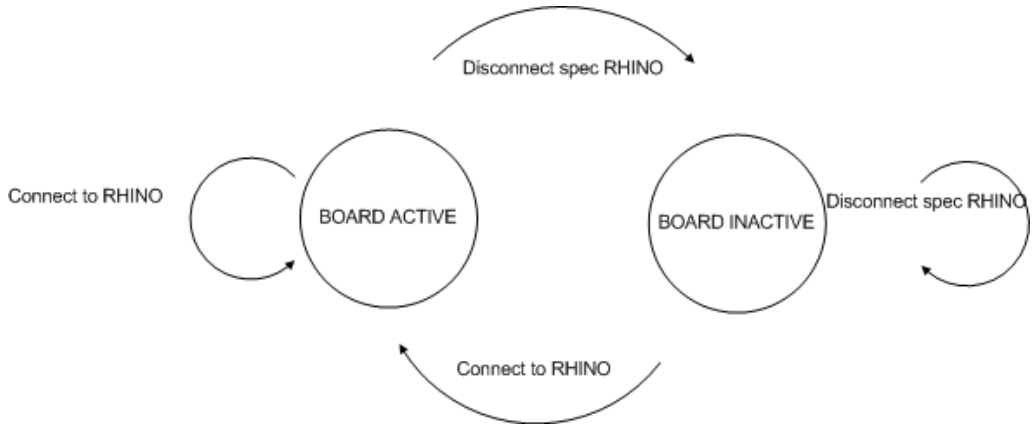


Figure 5.17: RHINO board active bit state transitions based on the connection and disconnect function calls

5.5 RACCMS Framework Implementation

The RACCMS framework was implemented based on the above designs. It was implemented using the C language while following a Linux coding standard. However for the server application that would run on the ARM process, a cross compiler was required to convert the C coded application into an executable that was compatible with the processor. The cross compiler that was used was developed by Denx Software Engineering specifically for the ARM Cortex 8 family [6] and was called ELDK 5.4. The implementation of the code was done in 4 major steps. The development of the framework structures, the design and implementation of the control protocol structures, the implementation of the client API functions and lastly the implementation of the server API functions.

5.5.1 RACCMS Framework Structures

These structures described the main attributes of the framework ranging from the main hardware blocks (the RHINO platform), to the overall cluster of boards connected to the framework at any given time. They are mainly used for exchanging information between the framework's components about the physical state of each of them and hence enables the framework to determine which operations can be performed.

Of note is the use of the uint32_t and uint16_t integers types as opposed to using unsigned int for some of the elements in the structures. This was done to conserve memory on both the RHINO and the control computer, and to ensure that the information sent across the network will maintain its specific size on either machine regardless of the its platform word size. As well, the *attribute_((packed))* keyword used in the structure to ensure data alignment and to make sure that the structure sent across the network was always the same size by using structure padding.

reg_T Structure This structure represents a register of a running gateway process. The attributes of this structure are obtained from the symbol file of a given BOF process discussed in section 4.4. The reg_T structure can only be populated once a gateway process in successfully executing on the RHINO board. The layout of the structure and the description of each attribute can be seen in Figure 5.18 below.

rhino_T Structure The rhino_T structure was designed to represent the attributes of individual RHINO platforms connected to the RACCMS framework. The structure enabled the framework to keep track of each RHINO and the structure was updated when the status the RHINO it represented changed. The layout of the structure and the description of each attribute can be seen in Figure 5.19 below.

```
struct reg_T{

        char reg_name[32];/*Is a name made of any unique sequence of 31 characters which are null terminated*/
        uint16_t rw_bit;/*used to indicate if reg can be read to or written. Value of 2 if reg is read only and 3 if reg is read/write*/
        uint32_t rh_nm_num;/*Name of RHINO register is located on. Used to identify the board the register belongs to on status table*/
        unsigned int size;/*Indicates how much memory allocation the register has*/

}__attribute__((packed));
```

Figure 5.18: reg_T structure with the description of each element in it made in blue comments

```
//characteristics of a rhinoboard
struct rhino_T{
        uint32_t name_number;/*Unique number used to identify each board on the cluster assigned to board by cluster admin*/
        char rh_ipadd[16]; /*Static IP assigned to board by DHCP Client by DHCP Server. This IP address is located within the local subnet*/
        uint32_t bsy_bit;/*Indicates if RHINO is programmed with gateware application. Can be either 1 for programmed and 0 for board
unprogrammed*/
        uint32_t success_bit;/*Indicates if function call on board has executed successfully or not. Set to "SUCCESS" for execution &
"FAILURE" for error*/
        char error_msg[256];/*String that contains error msg thrown during unsuccessful execution of function*/
        uint16_t data;/*Data contained in the register on RHINO*/
        struct reg_T registers_accessible;/*reg_T structures of registers on running BOF process*/

}__attribute__((packed));
```

Figure 5.19: rhino_T structure with the description of each element in it made in blue comments

rhino_status_table_T Structure This structure is used to represent all RHINO boards in the status table connected to the client application. Its main purpose is to keep track of all the RHINOs on the cluster and maintain their status for all the framework users to view. The rhino_status_table_T structure is available to all RACCMS users for viewing only. However the clusters administrator have the ability to add or remove boards from the system and hence directly impacting the state of the table. It is important to note that it allows the system to meet user requirement [U6] through its actv_bit element. The layout of the structure and the description of each attribute can be seen in Figure 5.20 below.

```
/*defining a structure that holds the Rhino Status Table*/
struct rhino_status_table_T{

        uint32_t rhino_num;/*Unique number assigned to board when added to the cluster by administrator*/
        uint8_t rhino_ip_add[16];/*Static IP assigned to board by DHCP server*/

        uint32_t bsy_bit;/*Indicates if RHINO is programmed with gateware application. Can be either 1 for programmed and 0 for board
unprogrammed*/

        uint32_t actv_bit;/*Used to indicate if client is successfully connect to the server application on RHINO board or if board is
available for use. "BOARD_ACTIVE" means is not available for use and "BOARD_INACTIVE" means the board is available for use */

        uint32_t prog_user_id;/*Keeps track of which user has selected each Board on the cluster*/
        uint32_t sock_fd;/*Used to create the unique 4 tuple TCP network connection between the client and server application*/
        uint32_t success_bit;/*Used to indicate if the last function call executed by the RHINO board was successful or not*/
        uint32_t port_no;/*Used to create the unique 4 tuple TCP network connection between the client and server application*/
};
```

Figure 5.20: rhino_status_table_T structure with the description of each element in it made in blue comments

5.5.2 Control Protocol Structures[F5]

The control protocol determines the way in which both the client and the server application interpret the packets of information they both send and receive. As mentioned in section 3.4.2, the protocol that was used in this framework was a binary based control protocol. As such, all exchanged information between the control computer and the RHINO

board, were wrapped using htons() and ntohs() to ensure that the packets maintained their correct endian values.

Due to the client packet and the server packets containing different information in them, two different packet structures were designed to represent each application. A register packet was also added to the protocol to cater for the list_reg function call that returned a reg_T structure. It is important to note that packets of information exchanged between the client and server application contained no pointers, as these would be referencing the memory location of the sending device which the receiving application would not be able to interpret. A description of each of these structures is given below.

Client Control Protocol Structure The client control protocol structure is generated by the client application sent to the server application through the client transport cp class. Its main purpose is to let the server application know which function to call from its API and to provide the relevant data the function call needs to execute that function. The control packet and its main features are described in Figure 5.21 below.

Packet Field	Size	Description
msg_id	**4 bytes**	Message ID is used to uniquely identify each of the functions in the server application's API running on the RHINO board. The main use of the message id is to let the server application know which state to select in the lookup table and execute the relevant function call.
user_name	**2 bytes**	This entity refers to the user's server name on the framework. It is required in the start_bof_proc () and stop_bof_proc () and list_reg () functions to determine the locations of the .bof and .sym files.
rhino_id	**4 bytes**	Used to ensure that all requests being sent to the server application are for that particular RHINO.
reg_name	**4 bytes**	Indicates the size of memory allocation the register has been given in the gateware design.
data	**2bytes**	The data represents the data that is required to be written to a named register for the reg_write() function call.

Figure 5.21: Image illustrating and describing the client protocol structure

Server Control Protocol Structure The server packet is generated by the server application and is sent to the client in response to a function call. The server packet contains returned information from a given function call. This also includes error messages as well as any data the board generates. The server control packets and its main features are described in Figure 5.22 below.

Packet Field	Size	Description
rhino_nme	4 bytes	Unique number used to identify individual RHINOs on the cluster. It is of unsigned 32bit integer.
ip_add	2 bytes	A static IP Address assigned to the RHINO by the DHCP server on the control computer. This IP Address will be a stored in the framework as a string.
rhino_bsy_bit	4 bytes	Indicates if the RHINO is programmed with a BOF process. The value of the busyBit is "FPGA_PROGRAMMED" with the value of 1 if the RHINO is programmed and "FPGA_UNPROGRAMMED" with the value of 0 if the board is not programmed.
success_bit	4bytes	Indicates if the called function executed on the RHINO has executed successfully. The successBit is set to "SUCCESS" with the value of 1 if the function has executed with no error encountered and "FAILURE" with the value of 0 if the function has encountered an error while execution.
error_msg	32 bytes	A string that contains an error message that has been encountered while executing a function on the RHINO.
data	2 bytes	Contains the data read from a gateware register during the execution of a regRead function call.

Figure 5.22: Diagram representation of server control protocol structure

Register Control Protocol Structure The register control packet was designed specifically for the list_reg() function and is sent only from the server application. The register control packet works in conjunction with the server control packet. It is used to only send the registers characteristic information from the server application. It should be noted that the last element in the register array will contain a null element to indicate there are no more registers to be sent and received to the client application. A description of the register control protocol structure is given below in Figure 5.23.

Packet Field	Size	Description
reg_name	4 bytes	This field is a string representing the name of register created during the gateware design process. The name of the register for this framework is 4 bytes including the string terminator in C programming. The register names need to be unique to each BOF process to avoid clashes between register names when accessing them.
rw_bit	2 bytes	Stands for the read write bit and it Indicates if a particular register is designed to be a read only register or a read and write register. The value of this field is unsigned 16 bit integer. It has a value of 2 if the register is a read only register and 3 if the register is a read and write register.
rhino_nm_num	4 bytes	This is the name of the RHINO the register has been retrieved from.
size	4 bytes	Indicates the size of memory allocation the register has been given in the gateware design.

Figure 5.23: Diagram representation of register control protocol structure

5.5.3 Client Application and API Functions [F4]

The client application is the main access point to the RACCMS framework, and sits on the control computer which connects remotely to a server application that sits on the RHINO's ARM processor. It communicates with the server application using the com-

munication protocol described above and also enables the framework to perform some cluster management functionality. The physical connection used between the server and client application was the 100Mbps Ethernet connection. This because that is the RHINOs main link which enables users to control and monitor and program the board, as mentioned in section 2.1.2.

The control computer API is a library of RACCMS functions that are accessible through the client application. The API functions enable interaction and use of the RHINO boards connected in the cluster. The library can also be included in code, as well as being implemented as part of the cluster application that can be accessed through the raccms client interface class or the raccms GUI. Any code developed using this API library will have to compiled and run on the control computer. The table below describes all the API functions available in control computer.

Name of Function	Description of Function	Return Value	Inputs
num_rhino()	Meets user requirements [U8]. This counts all the RHINOs that are connected and "ACTIVE" in the cluster. The output can be stored in a variable or printed on the screen.	(unsigned int) num_of_rhinos	rhino_status_table_T *rhino_status_tble
list_device_register()	Meets user requirement [U4]. Lists the registers available on a particular RHINO. It stores all the attributes of the registers in an array of reg_T objects. This method requires the RHINO to be programmed with a *".bof"* process.	(struct reg_T) array_of_regs	rhino_T *rhino_device char *bof_process_name char *user_server_id
cluster_status()	Meets user requirement [U8]. Displays the status of the cluster using RHINO_STATUS_TABLE. This table is updated every time this command is executed. Status displayed on GUI.	**GUI output**	rhino_status_table_T *rhino_status_tble
load_bof ()	Meets user requirement [U5]. Enables users to execute a .bof process on selected RHINO. Only one .bof process can run on the board at any given time. NB the .bof process has to be located in the user_server_name's folder on the control computer. The RHINO board needs to be selected by the user before attempting to start a BOF process. The BOF file to be executed needs to be in the user's folder and inside a folder with the same name as the BOF File.	rhino_T rhino_device_ptr	rhino_T *rhino_device char *bof_process_name char *user_server_nm
reg_write()	Meets user requirement [U4]. Allows a user to write to a register on a running gateware design. The RHINO will need to be pre-programmed for this function to work.	rhino_T rhino_device_ptr	rhino_T *rhino_device char *reg_name unsigned int data
reg_read()	Meets user requirement [U4]. Allows a user to read from a pre-specified register on a pre-programmed RHINO.	rhino_T rhino_device_ptr	rhino_T *rhino_device char *reg_name
connect_rhino()	Meets user requirement [U3]. Connects to all the RHINOs using the sockets and ip_address via a file with RHINO ip_addresses and Sockets.	rhino_status_table_T *rhino_status_tble	**none**
rhino_select()	Meets user requirement [U6][U7]. This method allows a user to select a particular RHINO from the cluster and lock it for use from the rest of the users. The framework will not allow any other user to perform any action that requires interaction with that particular board. This method associates the selection of a RHINO uses the program_id so as to uniquely identify the user who is using the board.	rhino_status_table_T *rhino_status_tble	rhino_status_table_T *rhino_status_tble
rhino_release()	Meets use requirement [U6][U7]. Releases a RHINO after a user is done using it. This makes it available to the rest of the Cluster. It uses the program_id to determine if the user has access to the RHINO before it releases it. Once the RHINO is released the active_bit is set back to "BOARD_INACTIVE".	rhino_status_table_T *rhino_status_tble	rhino_status_table_T *rhino_status_tble
connect_specific_rhino()	Meets user requirement [U3]. Connect to a specific rhino given an ip_address and a Socket ID with which to connect.	rhino_status_table_T *rhino_status_tble	char *ip_address int sock_id
disconnect_thino()	Disconnect from a specific rhino given an ip_address and a sock_id	rhino_status_table_T *rhino_status_tble	char *ip_address
kill_bof()	Stops a running ".bof" process on a given RHINO. A ".bof" process has to be running for it to be stopped.	rhino_T *rhino_device	rhino_T *rhino_device char *bof_process_name char *user_server_id
reg_set_bit()	Meets user requirement [U4]. Enables a user to set a specified bit of a named register.	rhino_T *rhino_device	rhino_T *rhino_device char *reg_name unsigned int bit_number
reg_bit_clear()	Meets user requirement [U4]. Enables a user to clear a specified bit of a named register.	rhino_T *rhino_device	rhino_T *rhino_device char *reg_name unsigned int bit_number
reg_bit_pulse()	Meet user requirement [U4]. Enables a user to set-clear and set a specified bit in one operation on a particular register.	rhino_T *rhino_device	rhino_T *rhino_device char *reg_name unsigned int bit_number
toggle_reg_bit()	Meets user requirement [U4]. Enables a user to toggle a specified bit on a named register.	rhino_T *rhino_device	rhino_T *rhino_device char *reg_name unsigned int bit_number

Table 5.2: Description of client application and the main API functions

5.5.4 RHINO ARM Server Application and API Functions [F1]

The server application sits on the RHINO's ARM processor and is the main connection point to the portion of the framework running on the RHINO board. It determines which server function to call and has the protocol packet requests from the client application translated into commands that can be understood by BORPH using the classes that it comprises of. Its main features include the lookup table and the control protocol functions. It is important to note that the POSIX C I/O library was used to execute the read and write related operations to registers on the running BOF process. As well, the data written to and read from the registers was limited to 16 bits (2 bytes long) as this is the maximum size of the FPGA_ARM data bus [53]. Figure 5.24 shows the API functions available in the server application as well as a description of how they work.

Name Of Function	Description of Function	Return Value	Inputs
reg_write ()	Translates a register write call from client application to write to a given register on running .bof process. (Bof file has to be running to perform register write). This function can only be performed once the FPGA has been programmed with a .bof file. The POSIX C I/O write() and open() function were used. The data written to the register is limited to 16bits, this limitation is due to the data size of the FPGA_ARM bus can transfer.	rhino_T rhino_device	rhino_T rhino_device register_name register_data server_state_T *st
reg_read ()	Translates a register read call to from client application to read a given register on running .bof process. (Bof file has to be running to perform register read). The POSIX C I/O read() and open() function were used. The data written to the register is limited to 16bits; this limitation is due to the data size of the FPGA_ARM bus can transfer.	rhino_T rhino_device	rhino_T rhino_device bit_number server_state_T *st
strt_bof_proc()	Starts a .bof process given the bof_process_name. It assumes the bof processes are all located in the RHINO/Server Computers NFS File system. It utilizes the user's server name and the name of the bof process to locate the BOF gateware design on the NFS file system. The BOF process is executed as a background process to enable other interactions with the RHINO's terminal. Its process id is stored in /var/run/bofprocpid which is then accessible to all the control functions that need to utilize the running process ID	rhino_T rhino_device	rhino_T rhino_device bof_process_name server_state_T *st
rhino_list_registers()	This method lists all the Registers on the RHINO. (Bof file has to be running to perform list_register process). It does this by utilizing the gateware design's symbol file. The function call lists the contents of the symbol file, using the user's server name and the BOF process name to access it. The function call uses the opendir ().	reg_T register_object	rhino_T rhino_device bit_number server_state_T *st
stop_bof_proc()	Function used to immediately stop running a BOF process on the RHINO board. The command used to stop the process is: "kill -9 pid", where pid is the process id. The -9 extension is used to perform a SIGKILL command, that stops the process from running immediately	rhino_T rhino_device	rhino_T rhino_device bit_number server_state_T *st
reg_set_bit()	Function used to set a particular bit on a named register of a running BOF process. The function uses the BIT_SET Macro that uses "*number |=1<<x* "	rhino_T rhino_device	rhino_T rhino_device bit_number reg_name server_state_T *st
reg_bit_clear()	Function used to clear a particular bit on a named register of a running BOF process. The function uses the BIT_CLEAR Macro that uses "*number &=~1<<x* "	rhino_T rhino_device	rhino_T rhino_device bit_number reg_name server_state_T *st
reg_toggle_bit()	Function used to toggle a particular bit on a named register of a running BOF process. The function uses the BIT_TOGGLE Macro that uses "*number ^=1<<x* "	rhino_T rhino_device	rhino_T rhino_device reg_name bit_number server_state_T *st
reg_bit_check()	Function used to toggle a particular bit on a named register of a running BOF process. The function uses the BIT_CHECK Macro that uses "*number &=1<<x* "	rhino_T rhino_device	rhino_T rhino_device reg_name bit_number server_state_T *st
reg_test_set()	This function is used to check if a register bit has been set and if it has not been set and if it has not been set it sets that particular bit.	rhino_T rhino_device	rhino_T rhino_device reg_name bit_number server_state_T *st
add_reg_value()	This function increments the value in a given register by a value of one.	rhino_T rhino_device	rhino_T rhino_device reg_name server_state_T *st

Figure 5.24: Diagram describing the Server Applications main API functions

5.6 RACCMS Graphic User Interface [F6]

The RACCMS GUI was implemented to meet functional requirement [F6], which would enable users to interact with the framework functions using a user-friendly interface. The GUI was implemented using GTK+2.0 (GIMP ToolKit) through its C interface. GTK is a library for creating a graphical user interface and is licensed under the LGPL license [12]. All the functions in the framework were accessed through the GTK API functions so as to be output on the GUI. The RACCMS GUI was designed to be an alternative interface to the command line interface and API implementation. Figure 5.25 below shows the how the user interacts with the RACCMS GUI's using the block diagram.

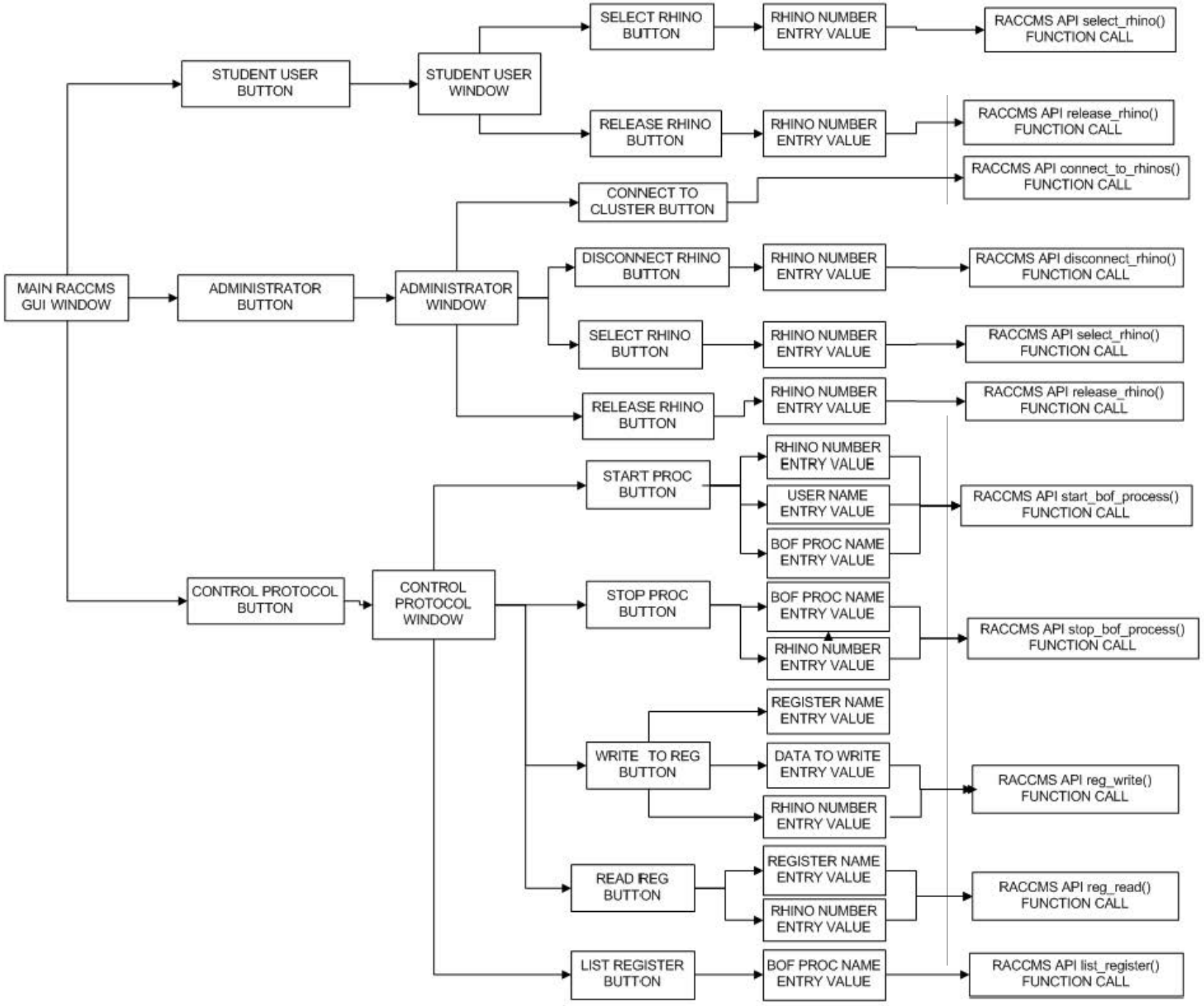


Figure 5.25: RACCMS GUI block diagram

The designed GUI consisted of 4 main interaction windows, namely the main menu window, the administrator's window, the student user's window and lastly the control protocol window. The entire GUI is made up of three classes that implement the four windows and can be seen in Figure 5.26 below.

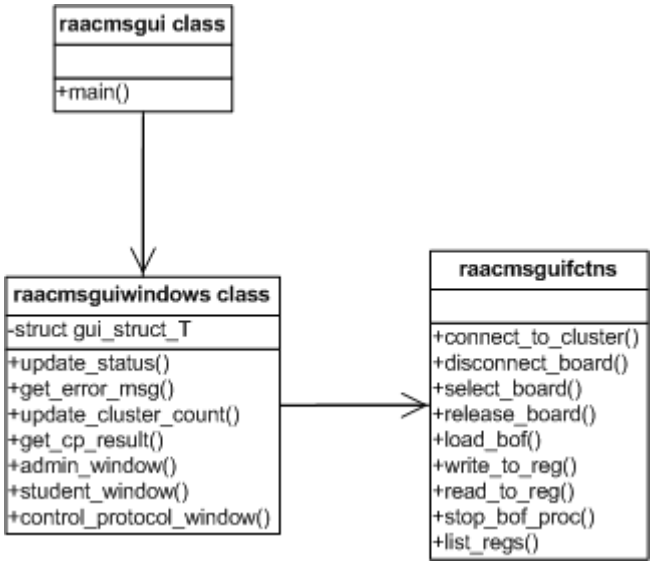


Figure 5.26: RACCMS GUI class diagram

A description of the design of the GUI framework is given below.

5.6.1 Main Menu Window

The main window is the main entry point to the framework, using the GUI. It provides a very brief introduction as to what the framework is and also presents 3 main buttons that lead to the administrator's page, the student's page and the control protocol page respectively. The logo at the top of the main page is the UCT logo, as this particular framework was designed specifically for the University of Cape Town. An image of the RACCMS main page can be seen blow in Figure 5.27.

Figure 5.27: RACCMS GUI main menu window

As can be seen above, the buttons to the other framework windows can be seen at the bottom of the window labeled *"Administrator User"*, *"Student User"* and *"Control Protocol Functions"*.

5.6.2 Administrator User GUI Window

The administrator user page is opened once the user presses the *"Administrator User"* button. This page is meant for specifically the administrator user. It contains all the tasks defined in the use case diagram in the RACCMS Software Design and Implementation chapter. An image of the administrator page can be seen in Figure 5.28 below.

As can be seen in the image above the main features of the Administrator's window include 4 main buttons, the *"CONNECT TO CLUSTER"* which initiates a call to the framework's connect_rhino() function call, the *"DISCONNECT RHINO"* button that is accompanied by an gtk entry box that enables a user to specify which RHINO to disconnect from the cluster using the disconnect_rhino() call. The *"SELECT RHINO"* button, also accompanied by a text gtk entry box for the RHINO number to select, initializes a call to the rhino_select() function call on the framework. Lastly the *"RELEASE RHINO"* button that has a entry text box for the RHINO number to release, calls the rhino_release function from the framework when pressed.

The top right side of the window constantly shows an up to date version of the RHINO status table. This information is retrieved from the framework using the cluster_status() function call and is shown on the GUI using gtk_timeout_add API call which calls and

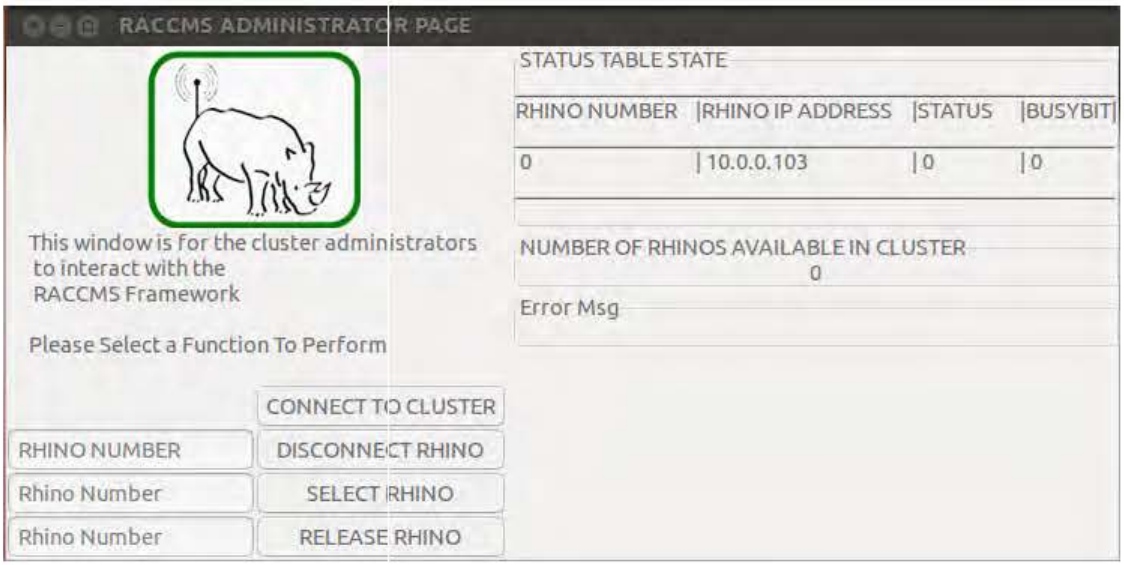


Figure 5.28: RACCMS GUI administrator user window

hence updates the RHINO status table every 1 second. Below the status table is a gtk framebox that counts the number of boards that are active on the cluster through the frameworks count_rhino() function call. The count GUI output, like the status table, is updated every 1 second using the gtk_timeout_add API call. The last framebox will print out on the GUI any error messages that the framework may throw, should it fail to execute any of the functions it is required to.

It should be noted that access to the Administrator's window is to be granted by entering a pre-set password. Should the password not be correct an error message would appear notifying the user that the incorrect password had been provided and prompting the user to retry.

5.6.3 Student User GUI Window

The student user page is opened once the user presses the "Student User" button in the main menu. This page is meant for all other users of the framework besides the administrator of the user. It contains all the functionality specified by the tasks in the student user's use case diagram in the RACCMS Software Design and Implementation chapter. The image of the student user GUI window can be seen below in Figure 5.29.

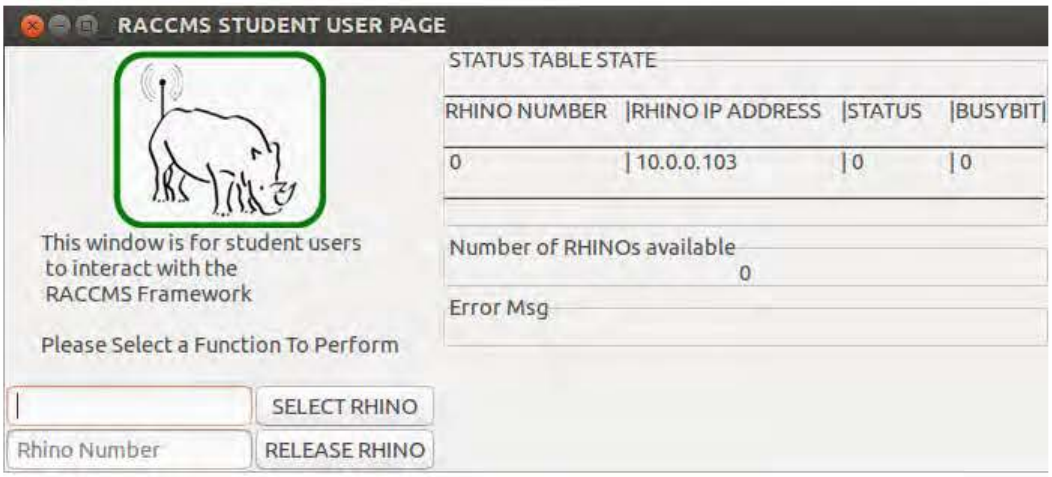


Figure 5.29: RACCMS GUI student user window

As can be seen in the image above the student user window has two main functions the user can perform using the GUI. That is the select RHINO and release RHINO functionality. The *"SELECT RHINO"* button used with the entry text box, initiate a call to the rhino_select() function call on the framework and the *"RELEASE RHINO"* button, that also has a entry text box for the RHINO number calls the rhino_release() function from the framework when pressed.

The student user window also has a status table as well as a RHINO counter that is also kept up to date as with the status table and RHINO counter on the administrator's page. Both of the status tables and RHINO counters are synchronized so that they contain the same information. Should either the administrator or the student user change any element in the table it will reflect almost instantly in both pages. In the case of Figure 5.29, the status table illustrates that there is one RHINO connected to the cluster, but since the status bit is "0" this indicates that the RHINO is not available for use in the cluster which results in the "0" value illustrated in the *"Number RHINOs available"* frame.

5.6.4 Control Protocol GUI Window

This window will enable both administrator and student users to execute the framework's some of control functions. This window is opened when the user presses the *"CONTROL PROTOCOL FUNCTIONS"* button in the main menu window. An image of the control protocol user page from the RACCMS GUI can be seen below in Figure 5.30.

The control protocol user window, as can be seen in Figure 5.30, contains the five of the framework's main control functions that are shown in the Figure 5.3 in the clientraacm-scp.c class. The user is able to enter in the required information for the respective control function and then the appropriate button is pressed. This subsequently then calls the relevant function from the framework's function file.

Figure 5.30: RACCMS control protocol GUI window

To the right of the window there is a control protocol results framework box that prints out the results from each of the protocol function calls. Just below that box there is an error message box that, just like the administrator page and student page, prints all the error messages that may be produced when executing the requested function call on the board or on the control computer.

Chapter 6

TESTS AND RESULTS

This chapter discusses the tests that were performed on the implemented RACCMS framework and documents the results of these tests. It should be noted that the tests will only be described briefly in this chapter as they were discussed in more detail in section 3.5. The tests were split into two major categories, the white box testing which focused on the framework's code design and implementation and the black box testing that aimed to test that the set requirements of the framework were met.

6.1 White Box Testing of Framework Code

The code was tested using white box testing and specifically using the control flow design technique [50]. These tests comprised of automated unit tests, integration and boundary tests all developed in an open source testing framework called CHECK Unit Testing Framework. These tests were performed on both the RACCMS client and server application. The Valgrind framework was used for the memory tests that were performed on the code to ensure that there were no memory leaks or memory corruptions. It should be noted that with white box testing it is not possible to test every single existing condition of the framework's code, however most of paths were tested in either the unit tests or the integration tests.

6.1.1 RACCMS Unit Tests

The RACCMS framework automated tests were designed around the UML flow diagrams as mentioned in section 3.5.1. The aim of these tests was to ensure that developed framework code executed as it was designed to in the flow diagrams. Unit tests were performed for all the functions in the client and server application classes that did not require I/O

or networking operations. The output from the execution of the function calls were compared to the expected output from a particular function based on a set of given inputs. The checks were done using CHECK Unit Testing Framework and the following assertions provided by the testing framework were used for testing the RACCMS framework:

- *ck_assert_ptr_ne/eq*: Used for functions that returned pointers such as the rhino_status_table_T or rhino_T objects to check if the pointer returned as not equal(ne) or equal(eq) to the expected output.
- *ck_assert_uint_ne/eq*: Used for the assertion of function that returned unsigned integer values
- *ck_assert_str_ne/eq*: Used for functions that return outputs that required string comparisons. This assertion test was used mainly for error messages that could be returned by the different function calls.

Each function in the framework that did not involve any I/O or file system operations was tested in this section. Each of the unit test cases were structured as shown in Table 3.3. Each of the tests were executed using the setup and steps specified in test cases. Four examples of the documented test cases for the client application can be seen in Figure 6.1 below. The rest of the documented unit test cases can be found in APPENDIX A.

Test Case	Create table not empty test
Test Setup	RACCMS Client Application running on Control Computer
Unit to Test	create_table()
Steps to Execution	1. Create empty status table, 2. Create a RHINO status table element with port number and IP number 3. Call create table function call with RHINO status table with created elemented added 4. Check if the returned RHINO status table is NULL indicating error 5. Free allocated RHINO status table memory
Assumptions	NONE
Results	Returned pointer was not NULL indicating execution without any of the possible error conditions being executed
Pass/Fail	Pass
Comments	

Test Case	Add RHINO to empty Table test
Test Setup	RACCMS Client Application running on Control Computer
Unit to Test	add_rhino()
Steps to Execution	1. Create rhino_status_table NULL pointer 2. Call the add_rhino() function call with a given RHINO's IP address and port number 3. Check that returned RHINO status table pointer is not NULL indicating error
Assumptions	NONE
Results	Returned pointer was not NULL indicating execution without any of the possible error conditions being executed
Pass/Fail	Pass
Comments	

Test Case	Unsuccessful RHINO search
Test Setup	RACCMS Client Application running on Control Computer
Unit to Test	rhino_search()
Steps to Execution	1. Create RHINO status table 2. Add two RHINOs to the RHINO status table using the add_rhino function call 3. Check that RHINO status table with RHINO elements added is not NULL 4. Call rhino_search() method for RHINO element not in the table 5. Check that function call does not return NULL element
Assumptions	RHINO element being searched for isnot in the RHINO Status Table
Results	RHINO element searched for should be not be found and temp_table element returned should be NULL
Pass/Fail	Pass
Comments	

Test Case	Count RHINOs empty table
Test Setup	RACCMS Client Application running on Control Computer
Unit to Test	This function checks that count_rhino() function call can handle the attempt to count the elements in an empty RHINO status table
Steps to Execution	1. Create empty RHINO status table with a its pointer pointing to NULL 2. Call the count_rhino() function call with the NULL RHINO status table 3. Check if the rhino number returned from the function call is -1 4. Free allocated RHINO Status Table Memory
Assumptions	NONE
Results	The RHINO number returned from the function call was -1 as expected
Pass/Fail	Pass
Comments	

Figure 6.1: Sample of documented unit test cases

In total 35 unit tests were performed for the functions in the client application. Each of the tests were structured to make sure that all possible execution paths the code could take based on a given input were tested. If a given test presented a segmentation fault or the assertion failed the Check Testing Framework would provide a report as seen in Figure 6.2 that provided the percentage of tests that passed as well as how many errors and assertion fails had been encountered and where they were located.

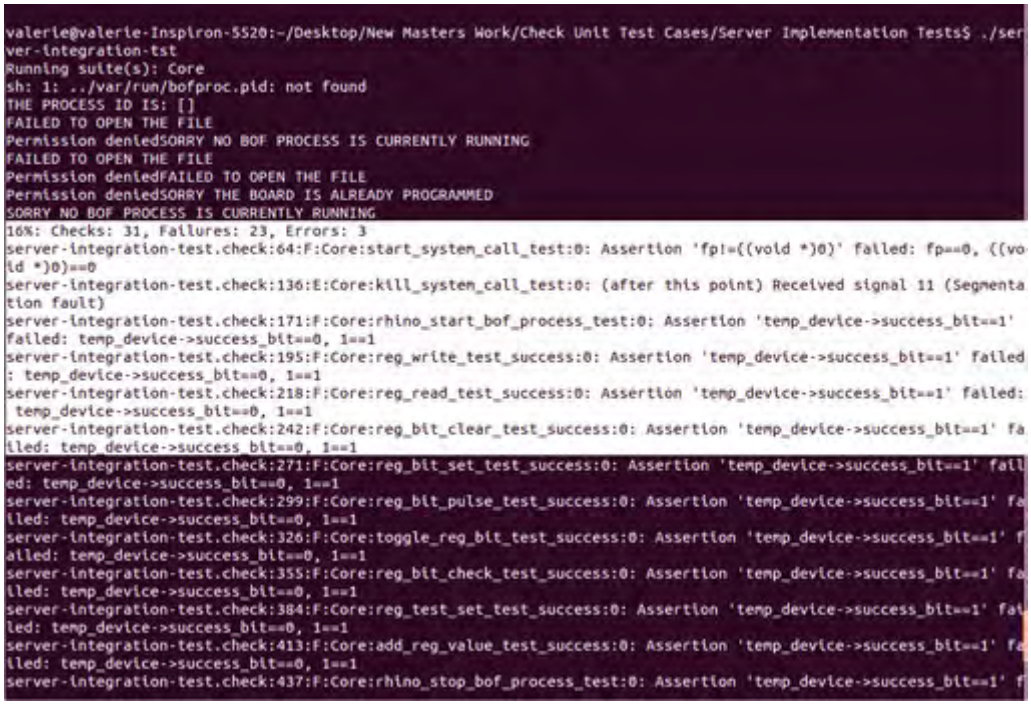


Figure 6.2: Example of failed CHECK Unit Testing framework unit tests

Once the errors and failures were identified the given functions were examined and debugged so as to ensure the expected output was produced. Once all the tests had executed as expected and all assertions passed the results were documented through a screen shot as seen in Figure 6.3 below.

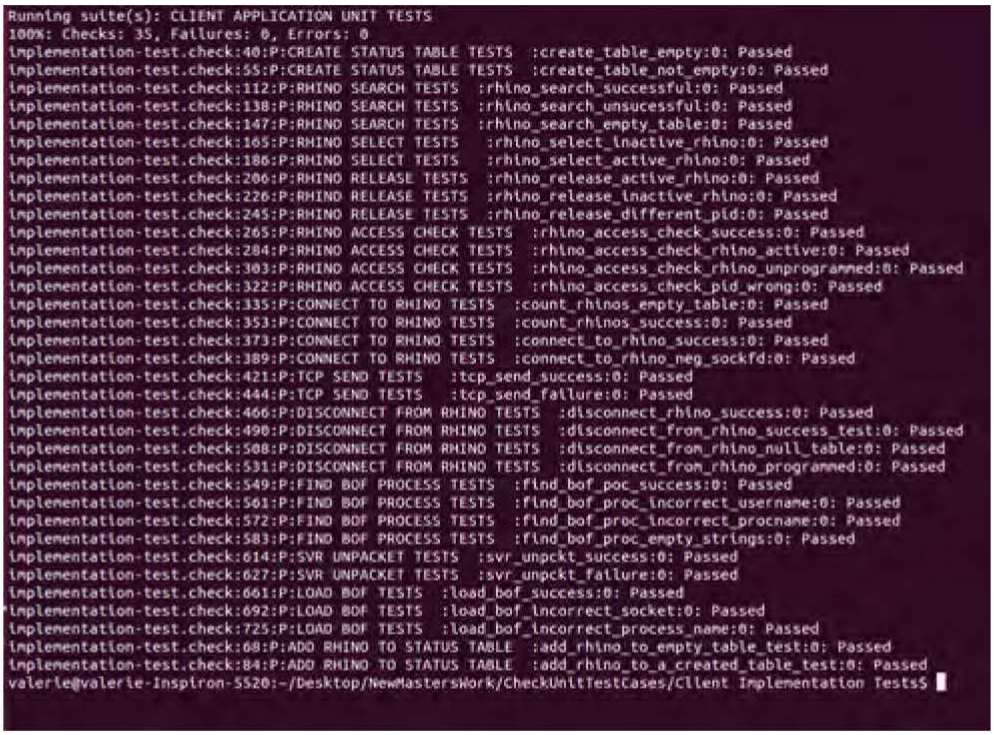


Figure 6.3: Client application unit test case CHECK Unit Testing framework results

As can be seen in Figure 6.3 all 35 of the unit tests for the client application executed as expected and passed with no errors. This is indicated by the *"Passed"* expression at the end of each of the executed tests as well as the 100% indication right at the beginning of the tests which shows the successful completion of the tests run by the testing framework.

For the server application a total of 36 tests were performed on each of the function files. The tests went through the same process as the tests performed for the client application. It should be noted that the server application tests were simulated on the control computer as the execution of CHECK required memory that the RHINO does not currently have (As mentioned in section 4.3.1 the RHINO only has 256MB SDRAM allocated for the ARM processor). This was achieved through the use of symbolic links that would run the "cat" command in the place of a BOF process. The results from execution of the server application unit tests can be seen in Figure 6.4.

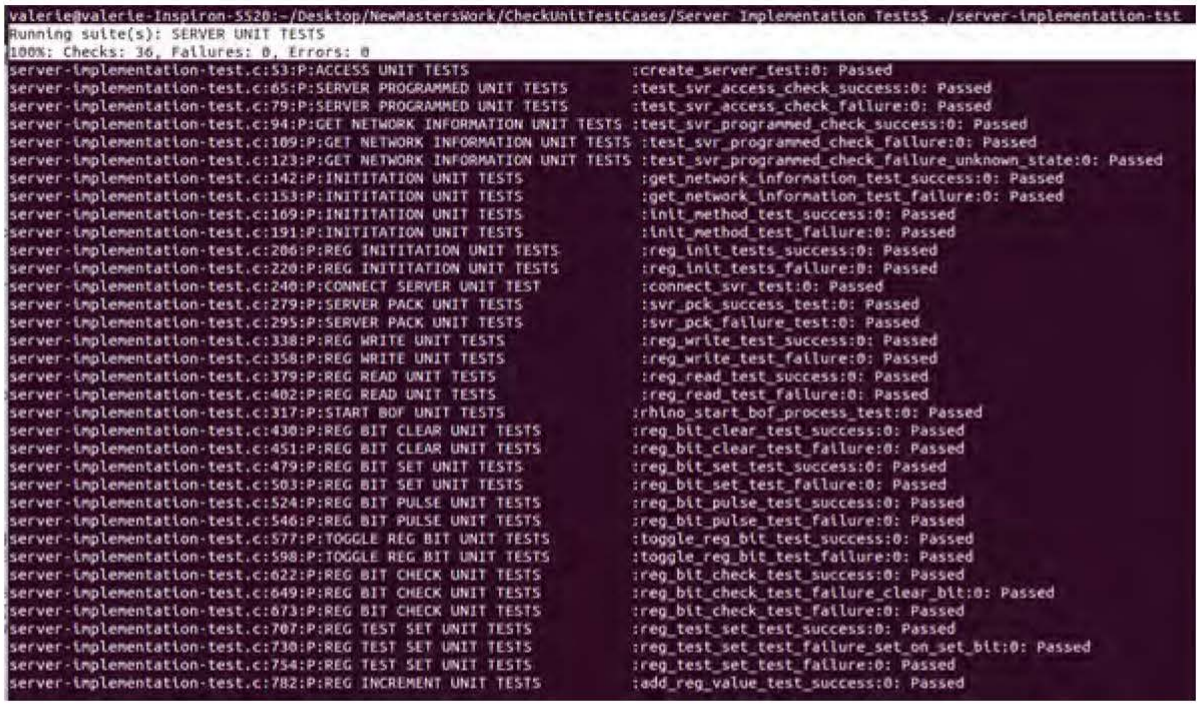


Figure 6.4: Server Application unit test case CHECK Unit Testing framework results

As can be seen in Figure 6.4 all the unit tests developed to test the server application's code implementation based on the flow diagram design executed as expected which is indicated by *"Passed"* at the end of each test line and the 100% highlighted in the image.

6.1.2 RACCMS Integration Tests

As mentioned in section 3.5.1, the integration tests were designed around mainly the RACCMS framework sequence diagrams. The integration tests were designed using the bottom up approach. The integration tests were split into two sets of tests, ones for the client application and ones for the server application. Figure 6.5 illustrates how the integration tests for client application were structured into the different levels of tests. Level 1 functions were tested first and then Level 2 functions were added to the Level 1 tests and so forth.

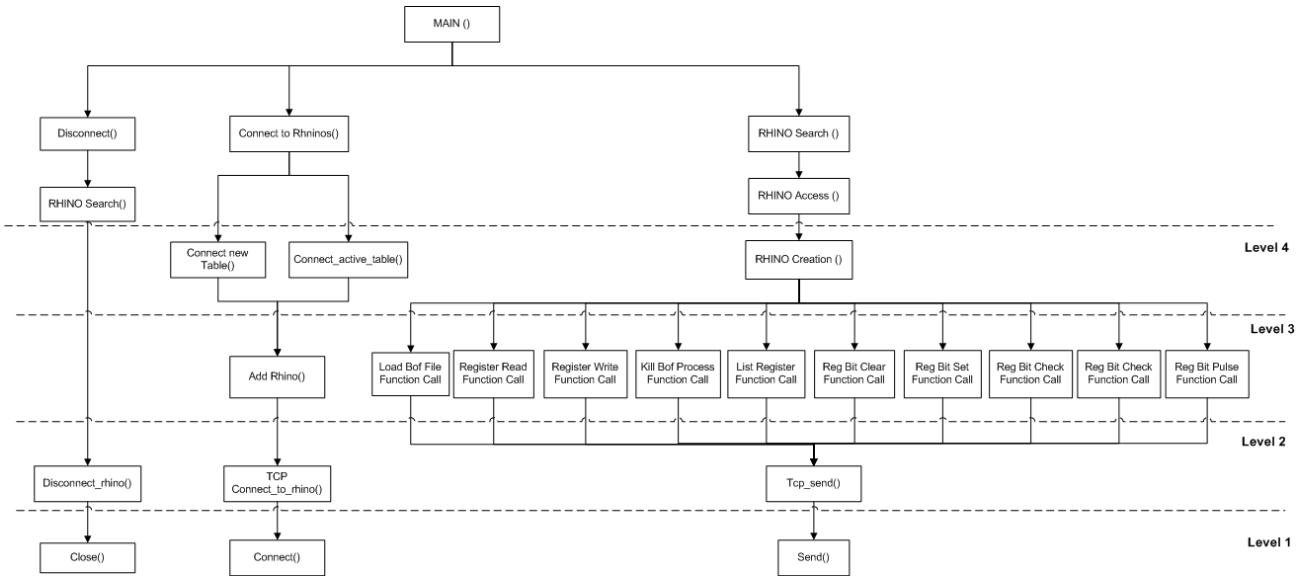


Figure 6.5: Client application integration tests structure

The client application integration tests aimed to test the interfaces between functions that interacted together directly. The lowest level in these tests were the functions that dealt with TCP based communication from the client application to the server application. The client integration tests consisted of a total of 54 checks and the results of these tests can be seen in Figure 6.6.

```
Running suite(s): CLIENT APPLICATION INTEGRATION TESTS
100%: Checks: 54, Failures: 0, Errors: 0
client-intergration-test.check:46:P:LEVEL1 INTEGRATION TESTS:connect_to_rhino_integration_test:0: Passed
client-intergration-test.check:83:P:LEVEL1 INTEGRATION TESTS:tcp_send_integration_test:0: Passed
client-intergration-test.check:115:P:LEVEL1 INTEGRATION TESTS:svr_unpckt_integration_test:0: Passed
client-intergration-test.check:136:P:LEVEL1 INTEGRATION TESTS:disconnect_rhino_integration_test:0: Passed
client-intergration-test.check:166:P:LEVEL2 INTEGRATION TESTS  :load_bof_integration_test:0: Passed
client-intergration-test.check:199:P:LEVEL2 INTEGRATION TESTS  :reg_write_integration_test:0: Passed
client-intergration-test.check:231:P:LEVEL2 INTEGRATION TESTS  :reg_read_integration_test:0: Passed
client-intergration-test.check:261:P:LEVEL2 INTEGRATION TESTS  :reg_bit_clear_integration_test:0: Passed
client-intergration-test.check:291:P:LEVEL2 INTEGRATION TESTS  :reg_bit_set_integration_test:0: Passed
client-intergration-test.check:321:P:LEVEL2 INTEGRATION TESTS  :reg_bit_pulse_integration_test:0: Passed
client-intergration-test.check:352:P:LEVEL2 INTEGRATION TESTS  :toggle_reg_bit_integration_test:0: Passed
client-intergration-test.check:382:P:LEVEL2 INTEGRATION TESTS  :reg_bit_check_integration_test:0: Passed
client-intergration-test.check:412:P:LEVEL2 INTEGRATION TESTS  :reg_test_set_integration_test:0: Passed
client-intergration-test.check:441:P:LEVEL2 INTEGRATION TESTS  :add_reg_value_integration_test:0: Passed
client-intergration-test.check:475:P:LEVEL2 INTEGRATION TESTS  :kill_bof_proc_integration_test:0: Passed
client-intergration-test.check:487:P:LEVEL2 INTEGRATION TESTS  :add_rhino_to_empty_table_test:0: Passed
client-intergration-test.check:512:P:LEVEL3 INTEGRATION TESTS  :rhino_search_successful:0: Passed
client-intergration-test.check:554:P:LEVEL3 INTEGRATION TESTS  :load_bof_integration_test2:0: Passed
client-intergration-test.check:595:P:LEVEL3 INTEGRATION TESTS  :reg_write_integration_test2:0: Passed
client-intergration-test.check:637:P:LEVEL3 INTEGRATION TESTS  :reg_read_integration_test2:0: Passed
client-intergration-test.check:679:P:LEVEL3 INTEGRATION TESTS  :reg_bit_clear_integration_test2:0: Passed
client-intergration-test.check:722:P:LEVEL3 INTEGRATION TESTS  :reg_bit_set_integration_test2:0: Passed
client-intergration-test.check:764:P:LEVEL3 INTEGRATION TESTS  :reg_bit_pulse_integration_test2:0: Passed
client-intergration-test.check:805:P:LEVEL3 INTEGRATION TESTS  :toggle_reg_bit_integration_test2:0: Passed
client-intergration-test.check:847:P:LEVEL3 INTEGRATION TESTS  :reg_bit_check_integration_test2:0: Passed
client-intergration-test.check:889:P:LEVEL3 INTEGRATION TESTS  :reg_test_set_integration_test2:0: Passed
client-intergration-test.check:929:P:LEVEL3 INTEGRATION TESTS  :add_reg_value_integration_test2:0: Passed
client-intergration-test.check:971:P:LEVEL3 INTEGRATION TESTS  :kill_bof_proc_integration_test2:0: Passed
client-intergration-test.check:997:P:LEVEL4 INTEGRATION TESTS  :connect_created_status_table:0: Passed
client-intergration-test.check:1014:P:LEVEL4 INTEGRATION TESTS  :connect_new_status_table_test:0: Passed
client-intergration-test.check:1059:P:LEVEL4 INTEGRATION TESTS  :load_bof_integration_test3:0: Passed
client-intergration-test.check:1100:P:LEVEL4 INTEGRATION TESTS  :reg_write_integration_test3:0: Passed
client-intergration-test.check:1142:P:LEVEL4 INTEGRATION TESTS  :reg_read_integration_test3:0: Passed
client-intergration-test.check:1184:P:LEVEL4 INTEGRATION TESTS  :reg_bit_clear_integration_test3:0: Passed
client-intergration-test.check:1226:P:LEVEL4 INTEGRATION TESTS  :reg_bit_set_integration_test3:0: Passed
client-intergration-test.check:1267:P:LEVEL4 INTEGRATION TESTS  :reg_bit_pulse_integration_test3:0: Passed
client-intergration-test.check:1310:P:LEVEL4 INTEGRATION TESTS  :toggle_reg_bit_integration_test3:0: Passed
client-intergration-test.check:1352:P:LEVEL4 INTEGRATION TESTS  :reg_bit_check_integration_test3:0: Passed
```

Figure 6.6: Client application integration CHECK Unit Testing framework test results

The integration tests for the server application were set up much in the same way as those of the client application and were executed in three levels. The lower levels of the integration tests involved UNIX based system calls that interacted with either the FPGA or the registers on the FPGA through the IOREG mentioned in subsection 4.5.1. The next level of tests that were added to the integration tests were the control protocol related functions that used these system calls. The hierarchal integration test model used for the server application can be seen in Figure 6.7.

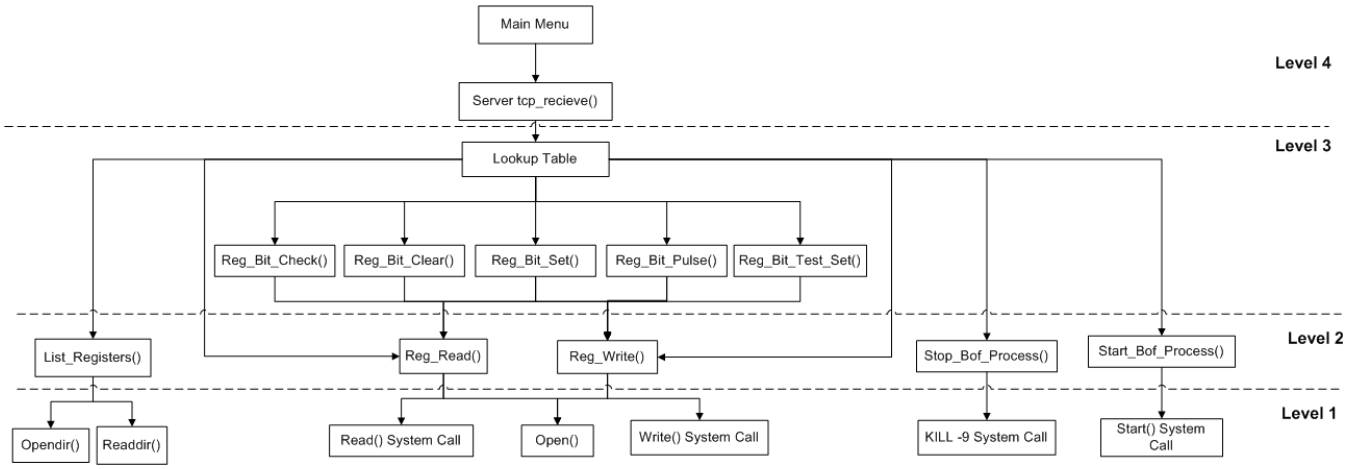


Figure 6.7: Server application integration test structure

The server application integration tests consisted of a total of 36 checks and the results of these tests can be seen in Figure 6.8. All the functions in the server application interacted in the way they were expected to and all assertions passed as indicated by the “Passed” at the end of each test and the “100%” completion indicating that all tests ran with out failures or errors.

```
valerie@valerie-Inspiron-5520:~/Desktop/NewMastersWork/CheckUnitTestCases/Server Implementation Tests$ ./server-implementation-tst
Running suite(s): SERVER UNIT TESTS
100%: Checks: 36, Failures: 0, Errors: 0
server-implementation-test.c:53:P:ACCESS UNIT TESTS                 :create_server_test:0: Passed
server-implementation-test.c:65:P:SERVER PROGRAMMED UNIT TESTS      :test_svr_access_check_success:0: Passed
server-implementation-test.c:79:P:SERVER PROGRAMMED UNIT TESTS      :test_svr_access_check_failure:0: Passed
server-implementation-test.c:94:P:GET NETWORK INFORMATION UNIT TESTS :test_svr_programmed_check_success:0: Passed
server-implementation-test.c:109:P:GET NETWORK INFORMATION UNIT TESTS :test_svr_programmed_check_failure:0: Passed
server-implementation-test.c:123:P:GET NETWORK INFORMATION UNIT TESTS :test_svr_programmed_check_failure_unknown_state:0: Passed
server-implementation-test.c:142:P:INITITATION UNIT TESTS           :get_network_information_test_success:0: Passed
server-implementation-test.c:153:P:INITITATION UNIT TESTS           :get_network_information_test_failure:0: Passed
server-implementation-test.c:169:P:INITITATION UNIT TESTS           :init_method_test_success:0: Passed
server-implementation-test.c:191:P:INITITATION UNIT TESTS           :init_method_test_failure:0: Passed
server-implementation-test.c:206:P:REG INITITATION UNIT TESTS       :reg_init_tests_success:0: Passed
server-implementation-test.c:220:P:REG INITITATION UNIT TESTS       :reg_init_tests_failure:0: Passed
server-implementation-test.c:240:P:CONNECT SERVER UNIT TEST         :connect_svr_test:0: Passed
server-implementation-test.c:279:P:SERVER PACK UNIT TESTS           :svr_pck_success_test:0: Passed
server-implementation-test.c:295:P:SERVER PACK UNIT TESTS           :svr_pck_failure_test:0: Passed
server-implementation-test.c:338:P:REG WRITE UNIT TESTS             :reg_write_test_success:0: Passed
server-implementation-test.c:358:P:REG WRITE UNIT TESTS             :reg_write_test_failure:0: Passed
server-implementation-test.c:379:P:REG READ UNIT TESTS              :reg_read_test_success:0: Passed
server-implementation-test.c:402:P:REG READ UNIT TESTS              :reg_read_test_failure:0: Passed
server-implementation-test.c:317:P:START BOF UNIT TESTS             :rhino_start_bof_process_test:0: Passed
server-implementation-test.c:430:P:REG BIT CLEAR UNIT TESTS         :reg_bit_clear_test_success:0: Passed
server-implementation-test.c:451:P:REG BIT CLEAR UNIT TESTS         :reg_bit_clear_test_failure:0: Passed
server-implementation-test.c:479:P:REG BIT SET UNIT TESTS           :reg_bit_set_test_success:0: Passed
server-implementation-test.c:503:P:REG BIT SET UNIT TESTS           :reg_bit_set_test_failure:0: Passed
server-implementation-test.c:524:P:REG BIT PULSE UNIT TESTS         :reg_bit_pulse_test_success:0: Passed
server-implementation-test.c:546:P:REG BIT PULSE UNIT TESTS         :reg_bit_pulse_test_failure:0: Passed
server-implementation-test.c:577:P:TOGGLE REG BIT UNIT TESTS        :toggle_reg_bit_test_success:0: Passed
server-implementation-test.c:598:P:TOGGLE REG BIT UNIT TESTS        :toggle_reg_bit_test_failure:0: Passed
server-implementation-test.c:622:P:REG BIT CHECK UNIT TESTS         :reg_bit_check_test_success:0: Passed
server-implementation-test.c:649:P:REG BIT CHECK UNIT TESTS         :reg_bit_check_test_failure_clear_bit:0: Passed
server-implementation-test.c:673:P:REG BIT CHECK UNIT TESTS         :reg_bit_check_test_failure:0: Passed
server-implementation-test.c:707:P:REG TEST SET UNIT TESTS          :reg_test_set_test_success:0: Passed
server-implementation-test.c:730:P:REG TEST SET UNIT TESTS          :reg_test_set_test_failure_set_on_set_bit:0: Passed
server-implementation-test.c:754:P:REG TEST SET UNIT TESTS          :reg_test_set_test_failure:0: Passed
server-implementation-test.c:782:P:REG INCREMENT UNIT TESTS         :add_reg_value_test_success:0: Passed
```

Figure 6.8: Server application integration CHECK Unit Testing framework test results

6.1.3 RACCMS Boundary Tests

Boundary tests were performed on the framework functions that involved mainly traversing through the RHINO status table and on the functions that had to access one of the 16 bits in the gateway registers. The tested functions included the rhino_search(), the rhino_release() and rhino_select() functions in the client application managementfctns class and the control protocol functions, reg_bit_clear(), reg_bit_set(), reg_bit_pulse(), toggle_reg_bit, reg_bit_check() and reg_test_set(). The boundary tests were structured to consider all possible states the framework could be in at any given time within and outside the given specified boundaries.

MANAGEMENT FUNCTION BOUNDARY TESTS

The boundary tests for the rhino_search() and rhino_select() functions involved testing the set boundaries of the RHINO status table and attempts to traverse it. The tests were split into six tests, with the RHINO status table having two elements in it. The boundaries of the table was defined as $0 \leq x \leq 1$ and they were tested for each of the two states the RHINOs in the table could be in at any given time, i.e. BOARD_INACTIVE or BOARD_ACTIVE. The tests included RHINO numbers more and less than the given boundary values. The boundary tests executed on these functions can be seen in Table 6.1.

Test Number	RHINO Number	RHINO's Active Bit	Expected Test Result	Test Result
1	-1	BOARD_INACTIVE	Fail	Pass
2	1	BOARD_INACTIVE	Fail	Pass
3	5	BOARD_INACTIVE	Fail	Pass
4	-1	BOARD_ACTIVE	Fail	Pass
5	1	BOARD_ACTIVE	Success	Pass
6	5	BOARD_ACTIVE	Fail	Pass

Table 6.1: Search and select RHINO function call boundary test structure

As can be seen in Table 6.1 only one of each set of boundary tests performed on the two function calls were expected to succeed. This condition was recorded in test number 5, all other tests were expected to fail and print out some form of an error message. All tests executed as expected and the results were documented in Figure 6.9.

The boundary tests for the rhino_release() function in the "managementfctns" class were designed in the same way as the RHINO search and select function calls with the only difference being the test condition that resulted in the only success outcome, which is indicated as test number 2 in Table 6.2 below. All other tests were expected to fail with a handled error by the framework code. The results of these test can be seen in Figure 6.9. All tests executed as expected.

Test Number	RHINO Number	RHINO's Active Bit	Expected Test Result	Test Result
1	-1	BOARD_INACTIVE	Fail	Pass
2	1	BOARD_INACTIVE	Success	Pass
3	5	BOARD_INACTIVE	Fail	Pass
4	-1	BOARD_ACTIVE	Fail	Pass
5	1	BOARD_ACTIVE	Fail	Pass
6	5	BOARD_ACTIVE	Fail	Pass

Table 6.2: Release RHINO function call boundary test structure

RAACMSCP BIT FUNCTION BOUNDARY TESTS

The following boundary tests were performed for the register bit related functions in the raccmscp class. As mentioned in subsection 4.3.5, the Processor-FPGA Bus is only able to transfer 2 bytes of data at any given time. This means that any defined registers were limited to this size and hence resulted in the boundary definition $0 \leq x \leq 15$. A total of 5 boundary tests were performed on each of the functions mentioned above. The setup of the tests can be seen in Table 6.3 below. In these specific boundary tests, three of the five tests were expected to succeed. The tests executed as expected and all boundary related assertion for all the functions passed as can be seen in Figure 6.9.

Test Number	Register Bit Number	Expected Test Results	Test Result
1	x < 0: x =-1	Fail	Pass
2	0	Success	Pass
3	0≤x≤15 : x = 7	Success	Pass
4	15	Success	Pass
5	x>15: x =17	Fail	Pass

Table 6.3: Register bit related function call boundary test structures

Figure 6.9 illustrates the results of the CHECK boundary tests that were executed based on the above mentioned boundary test designs. All 57 boundary related tests passed as can be seen by the *"Passed"* at the end of each test and the "100%" indicating that all tests ran with no assertion failures or errors.

6.1.4 RACCMS State Diagrams Tests

The state diagram tests were designed to test that the pieces of code that involved state transitions executed as expected by the framework. A detailed description of these tests is presented in subsection 3.5.1. Two sets of state tests were performed on the RACCMS framework. One for the server application which involved testing the different states involved in the executing a BOF process, and one for the client application which tested the frameworks ability to switch states using the connect and disconnect functions.

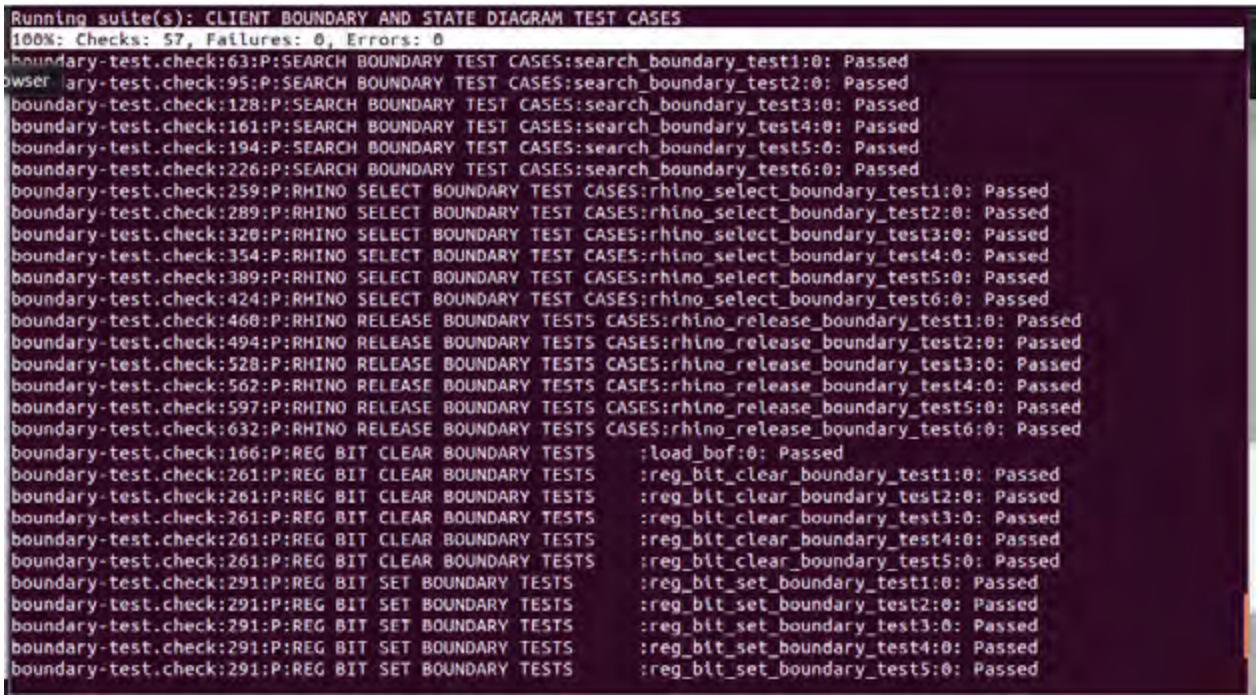


Figure 6.9: Client application boundary CHECK Unit Testing framework test results

The server application's load BOF process state tests were executed in 4 different ways for each of the different transitions the states could be in which are illustrated in Figure 5.11. Two of the tested transitions were expected to result in the framework moving to known states, and the other two transitions were expected to return framework handled error conditions. The tests that were performed can be seen in Table 6.4. Each of the tests executed as designed and expected with the correct state transitions occurring at each step.

Test Number	Current State	Action	Expected State	Test Result
1	FPGA_UNPROGRAMMED/ *Stopped*	Start BOF Process	FPGA_PROGRAMMED/ Executing Running	Pass
2	FPGA_PROGRAMMED/ Executing Running	Start BOF Process	ERROR Condition	Pass
3	FPGA_PROGRAMMED/ Executing Running	Kill BOF Process	FPGA_UNPROGRAMMED/ *Stopped*	Pass
4	FPGA_UNPROGRAMMED/ *Stopped*	Kill BOF Process	ERROR Condition	Pass

Table 6.4: Server application state diagram tests of loading a BOF process

The client application state tests consisted of state changes based on the execution of the connect and disconnect function calls. The state tests involved four tests for each of the transitions between the BOARD_ACTIVE and BOARD_INACTIVE states and were designed around the state diagram shown in Figure 5.17. The executed tests are illustrated in Table 6.5, and the tests passed indicating they executed as expected.

Test Number	Current State	Action	Expected State	Test Result
1	BOARD_INACTIVE	Connect to RHINO	BOARD_ACTIVE	Pass
2	BOARD_ACTIVE	Connect to RHINO	BOARD_ACTIVE	Pass
3	BOARD_ACTIVE	Disconnect spec RHINO	BOARD_INACTIVE	Pass
4	BOARD_INACTIVE	Disconnect spec RHINO	BOARD_INACTIVE	Pass

Table 6.5: RHINO board active bit transition states based on connection and disconnect function call test setup

Figure 6.10 illustrates the output from the execution of the state diagram tests that are described above. As can be seen in the image below, all the tests for both the client and server applications succeed as indicated by the "Passed" statement at the end of each of the executed tests. This shows that all the state transitions used by the framework were implemented as expected.

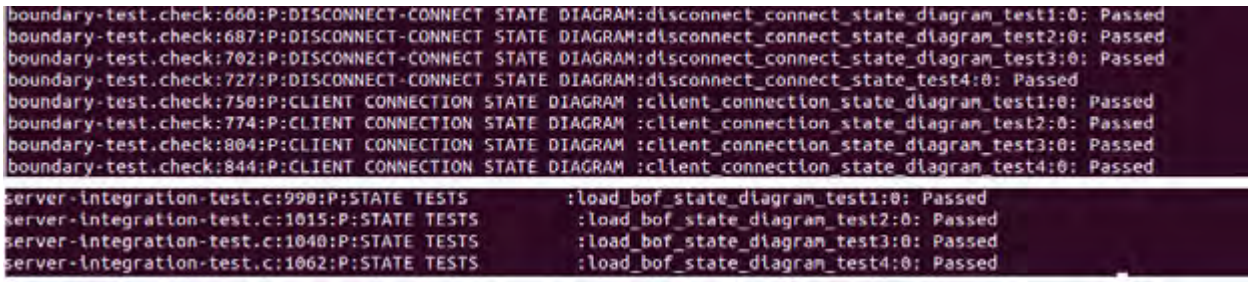


Figure 6.10: Client application state diagram CHECK Unit Testing framework results

6.1.5 Memory Tests

Memory tests were performed on both the client and server application to determine that there were no memory leaks or memory corruptions detected during the execution of the framework. Valgrind was used as the memory testing framework to perform these tests with relevant flags set as mentioned in subsection 3.5.1. It should be noted that the server memory tests were simulated on a PC as Valgrind uses a significant amount of memory for execution which the RHINO board currently lacks as mentioned in subsection 4.3.1.

Figure 6.11: Valgrind errors detected in RACCMS framework

The three main memory related errors that Valgrind was able to identify on the implemented framework can be seen in Figure 6.11 and are described as follows:

1. *Memory loss and leaks*: Indicates all the heap blocks issued using malloc that were not freed after use. In Figure 6.8, six memory heaps were allocated and only one was freed which meant there were some memory leaks in the RACCMS framework.

2. *Syscall param socketcall points to uninitialized bytes*: This error indicated that all direct parameters used in the tcp based send() system call were not addressable or not initialized meaning access to unaddressable space was being attempted and could have resulted in possible memory corruption.

3. *Source and destination overlap in strncpy*: This error indicated that data being copied from one source to a given destination overlap. It should be noted that there is a given POSIX standards that states “that if a copy takes place between objects that overlap, the behavior is undefined”. This indicated that within certain copies in the framework’s code some possible memory corruption could have occurred.

The errors were fixed by initializing each individual client and server packets before they were sent across the network as well as copying data into two separate memory locations

to avoid having the overlapping of data. Lastly all allocated memory was identified and freed.

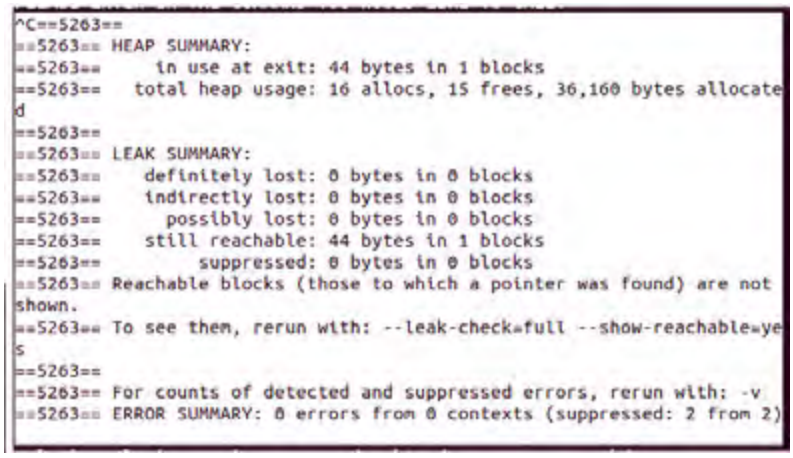


Figure 6.12: RACCMS framework with Valgrind Memory errors fixed

As can be seen in Figure 6.12 when all the errors had been identified and fixed, Valgrind was run again on the framework code and no memory errors or leaks were detected.

6.2 RACCMS Framework Black Box Testing

The framework was tested to ensure that it met its given requirements. These tests were performed on the RACCMS framework as a single unit. The tests were executed in five steps which are discussed in the subsections below.

6.2.1 Network Integrity Tests

These tests were used to verify that the communication between the server and client application occurred as designed. This included tests related to the transmission of the RACCMS control protocol packets over the UCT LAN network without any errors. These tests was performed using TCP Dump with the -X and -v flags set which allowed the viewing of the contents in network packets sent from the client application on the control computer to the server on the RHINO ARM processor.

As can be seen from Figure 6.13 the control protocol data sent across the network in a TCP packet over IP is delivered to the server on the ARM processor with no errors or packet losses encountered. As well it can be clearly seen that the user's name being *"chrval001"* and the name of the BOF file *"rhino_adder"*, sent in the client packet are transfered correctly. This was also noted for the rest of the packets sent between the client and server applications over the network.

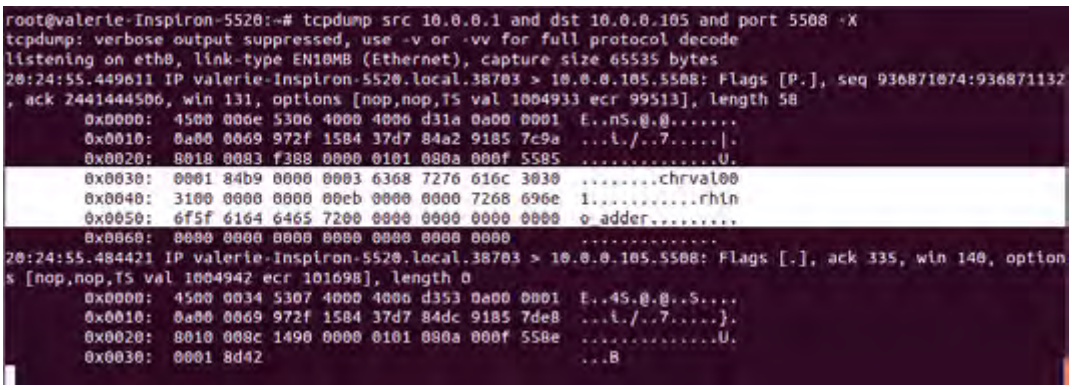


Figure 6.13: Command line image showing packet content for loadBofFile function call sent from control computer to RHINO

6.2.2 Database Tests

Database tests were used to make sure that data received, stored and manipulated by the server application maintained its accuracy and integrity. This was done by checking that the packet data sent by the client application was the same as that received by the server and vise versa. As well, when the server was processing the received packet and executing the requested function call, checks were performed to see that the server did not throw any unknown/unchecked server states.

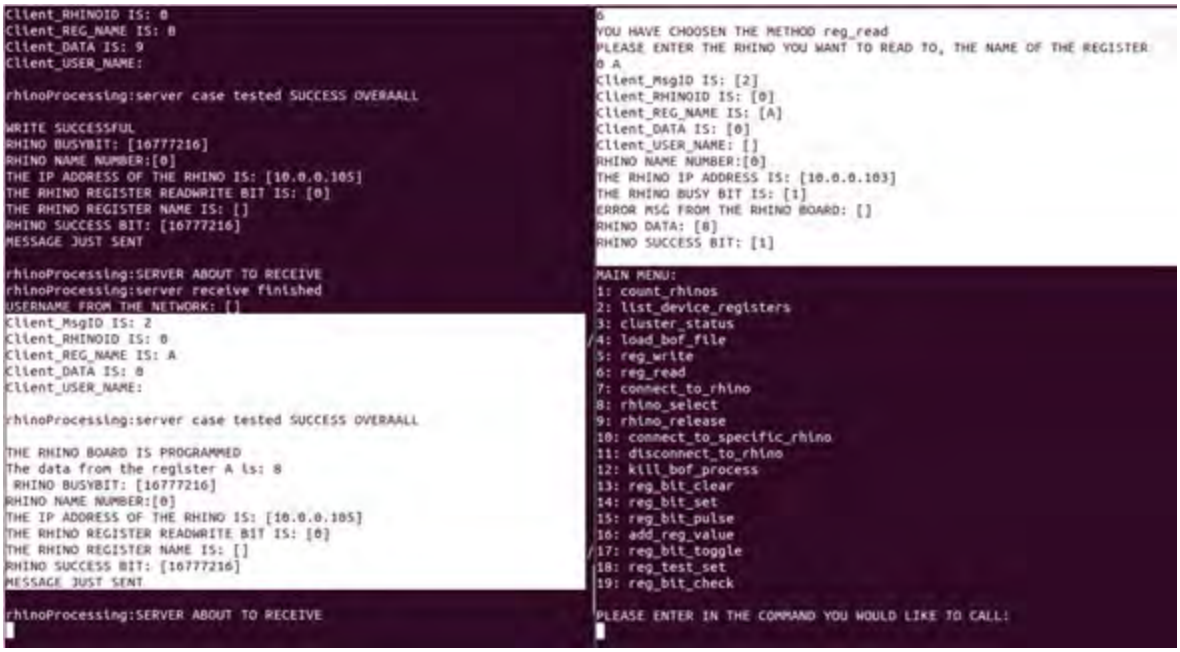


Figure 6.14: Picture illustrating data sent to and from server application is that sent and received from the client respectively

Figure 6.14 shows that the data sent from the client application with a request to read from a register (shown in the screen to the right), is the same as the data from the client message received by the server application (shown in the screen to the left). The same is shown for server data packets generated in the server application sent to the client application. The data does not get corrupted during the sending of packets between the two applications.

6.2.3 Transaction Tests

These tests focus on the time involved in the processing and transmission of data between the server and client applications in the RACCMS framework. These timing tests were design to determine the transmission times of the RACCMS control protocol packets. In this case the transmission times were the times taken to send and receive the three different types of control packets across the UCT LAN based on their different sizes.

An averages of 1000 packets were sent across the network in order obtain the average transmission time. The following control protocol times were taken:

- Time to send a server packet to client application
- Time to send a client packet server application
- Time to send a register packet to client application

The round trip time to send and receive a server packet from the control computer client application to the server on the RHINO ARM and back again was found to take on average 1.071 ms. This means to send the packet one way takes on average 0.536 ms

The round trip time for a client packet was found to take on average 1.101 ms meaning sending the packet on way takes 0.551 ms.

6.2.4 Function Performance Tests

These tests were mainly focused on the functions in the framework that utilized the RACCMS control protocol packets to communicate. These functions included the *reg_read()* and *reg_write()* functions, *reg_bit_clear()*, *reg_bit_set()*, *reg_bit_pulse()*, *toggle_reg_bit()*, *reg_bit_check()* and *reg_test_set()* functions . The execution times were the time it took one of these functions to completely carry out all its designed actions, from the time the client packet was sent to the server and the server packet was received. The functions were executed repeatedly 500 times and each test utilized the same data information when running. As well, the time it would take to manually execute functions that would require multiple operations to the same register using

the network were also presented in the results by adding the times it would take to execute the individual operations. The average of the 500 test times was taken and the results of each of the tests can be seen in Table 6.6.

FUNCTION CALL	REG WRITE	BIT TEST AND SET	REG SET	REG READ	BIT PULSE	BIT CLEAR	BIT CHECK	MANUAL BIT TEST AND SET	MANUAL BIT PULSE
EXECUTION TIMES	5.36605	7.364008	5.73210	5.32172	5.78527	5.89161	5.79959	17.51533	11.53169

Table 6.6: Average execution times for control protocol functions

As can be seen in the table above, all control protocol functions took an average of 5 ms to execute fully with the bit test and set function taking the longest to execute. It can also be seen from the above table that manual tests of the bit_test_set() and those of the reg_bit_pulse function calls would take roughly twice as much time to execute as compared to the RACCMS framework implementations of the same functions. This shows that the RACCMS framework improves the users overhead time the when executing these functional calls.

Figure 6.15: Timing graphs comparing the manual and RACCMS control protocol execution times in milliseconds for the reg_bit_test and set and reg bit pulse function calls.

Figure 6.15 illustrates the performance measurements of the RACCMS framework's execution of the reg_bit_pulse function and the bit_test_set function compared to what it would cost to perform these same operations manually. Both manual tests are illustrated with the red graphs and the RACCMS framework executions are represented by the blue graphs. As can be seen in the graphs all the manual graphs lie above the RACCMS framework execution graphs showing that on average the RAACMS framework takes less time to execute these functions than if the user executed them using multiple operations. As can be seen in the Register Tests and Set Function call timing graph, the minimum time it took the framework to execute this function call was roughly 3 ms which was almost the same with the Reg Bit Pulse timings which took roughly 2 ms to execute.

6.2.5 User Acceptance Test

These tests are used to see if the user requirements specified in section 1.3.2 were met by the designed framework. The tests were executed from the client application side of the framework, however in order for some of these function calls to work they had to in turn call the server API functions on the RHINO. A basic gateway adder was designed to run on the RHINO with three accessible registers, "A", "B" and "OUT", where OUT = A +B. This was the test application used to test these user requirements.

6.2.5.1 Register Read and Write Function Call Tests

This test was aimed to see if user requirement [U3] was met by the framework which specified that a user should be able to read and write to register values while a gateway process is executing.

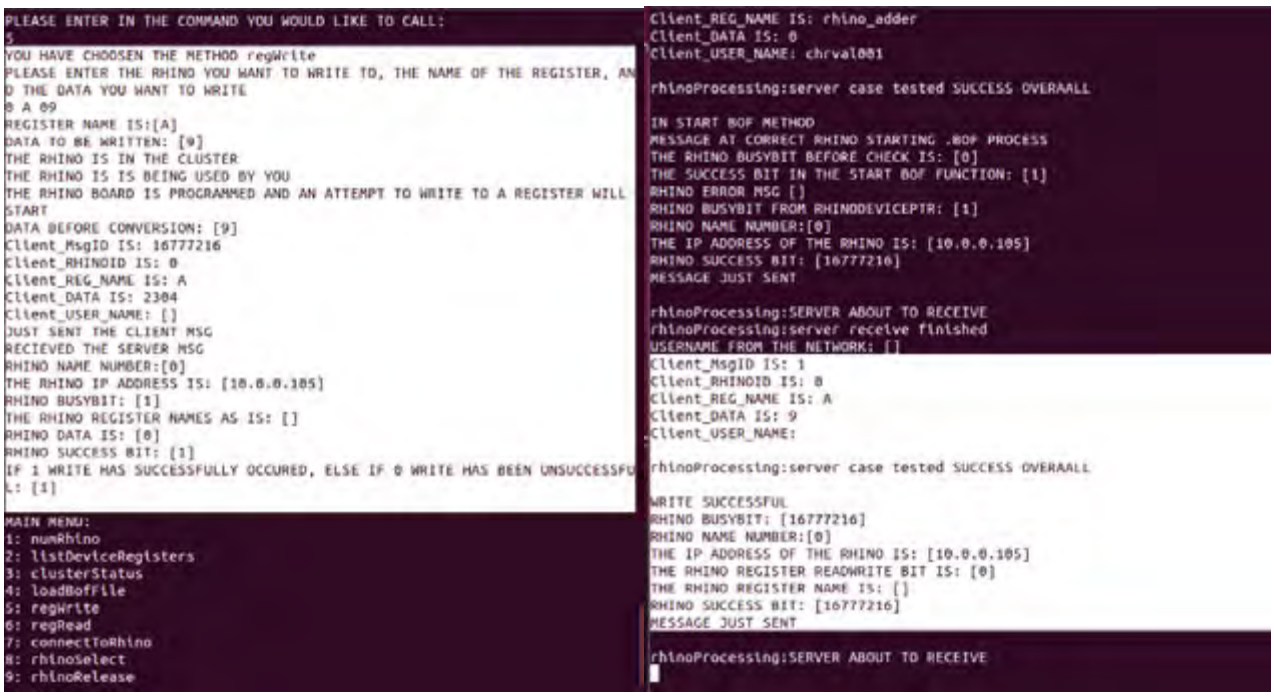


Figure 6.16: Writing to Register 'A' function call from both the client application (on the left) and server application(on the right)

Figure 6.16 shows a user, through the RACCMS framework, writing the number 9 to register A on RHINO 0 successfully. This is indicated by the success_bit returned from the function call being 1 which corresponds to "SUCCESS". Figure 6.17 below shows the data retrieved from the /proc/pid/ioreg/A file using a "od -d" system call which confirms the correct data value written to the register using the framework.

Figure 6.17: Picture showing data in register A from the /proc folder using the 'od -d' system call

Figure 6.18 shows that the framework is able to perform a remote register read from register 'A' on RHINO '0'. This was the same register written to by the framework in Figure 6.16. It can seen in Figure 6.17 that the request for information in register A is successfully sent from the client application to the server application. The server then successfully retrieves this information and sends it back in a server packet to the client application. The data sent is correct as this was the data in the register as was shown in Figure 6.17 above.

Figure 6.18: Reading from Register 'A' function call view from both the client application sides (left side) and server application (right side)

6.2.5.2 Load BOF Function Call Tests

This test was aimed at verifying that user requirement [U4] was met which specified that user should to be able to load configuration settings onto the RHINO's FPGA using the framework. The test was to configure "RHINO 0" with the *"rhino_adder"* BOF process .

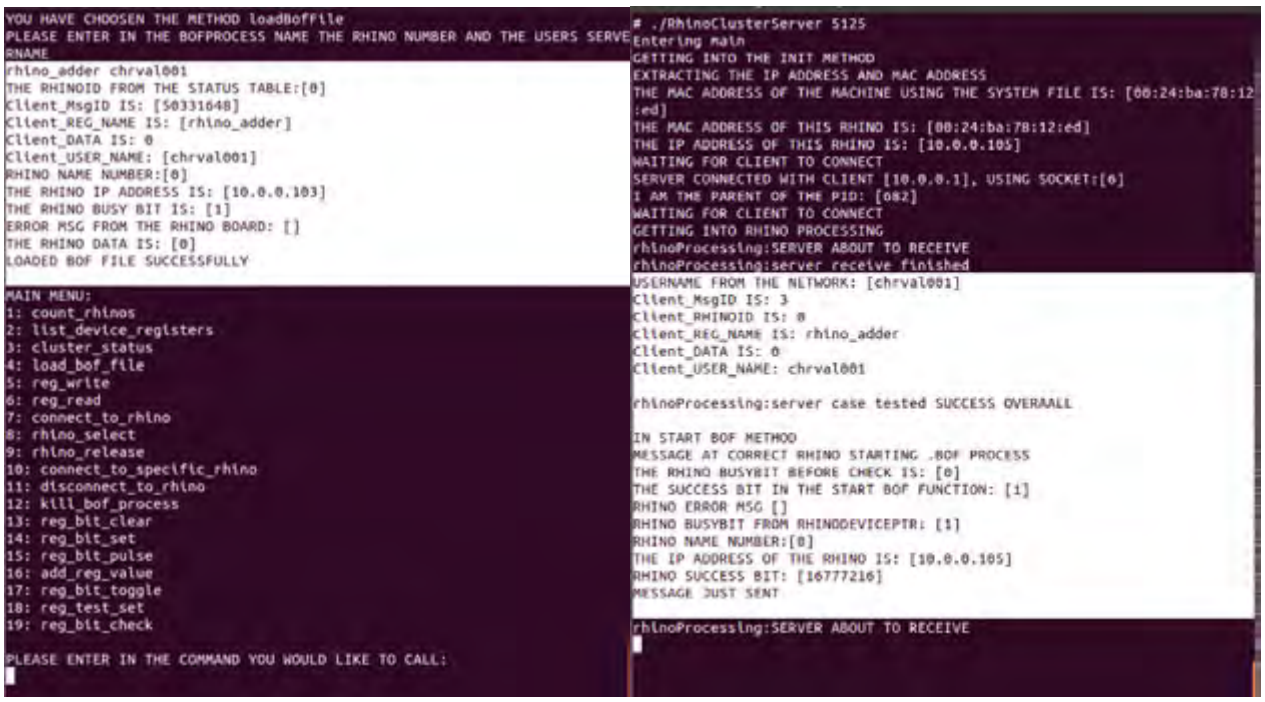


Figure 6.19: Loading a BOF command line output view from both the client application (right side) and server application (left side)

Figure 6.19 shows a user successfully starting a BOF processes called “rhino_adder” with a user ID of “chrval001”, which is confirmed by the success_bit being set to “1” on the client application command line. The message from the client application is successfully received by the server application (on the right of the image) which then configures the FPGA with the named BOF processes and lights up the FPGA configuration LEDs as can be seen in Figure 6.20 below.

Figure 6.20: Picture showing RHINO board being programmed after executing “loadbof” command from RACCMS framework

6.2.5.3 Kill BOF Process Function Call Test

The kill BOF process function was designed to partially fulfill user requirement [U4] by enabling the user to stop the BOF process that they would have loaded onto the RHINO. This test aimed to verify that the kill BOF process function call met this requirement by stopping the *"rhino_adder"* BOF process that was started by the load_bof_process() function call.

Figure 6.21: Killing a BOF process functional call view from command line on client application(left image) and server application (right image)

As can be seen in Figure 6.21, the kill BOF process function call on the RACCMS framework executes correctly. This can been seen by the success_bit sent back from the server application to the client application being set to the number "1" indicating the successful execution of a command on the server application.

6.2.5.4 List BOF Process Registers

This test was used to see if list bof process register function helped fulfill part of user requirement [U7] which was to enable a user to query the status of a particular RHINO. The test was to list the available registers in the *"rhino_adder"* BOF process running on the RHINO.

```
RHINO SUCCESS BIT: [0]
THE RHINO REGISTER RHINONAMENUMBER IS: [0]
THE RHINO REGISTER NAME IS: [A]
THE RHINO REGISTER READWRITEBIT IS: [768]
THE RHINO REGISTER SIZE IS: [2]
THE NEXT REGISTER ON THE BOARD IS
THE RHINO REGISTER RHINONAMENUMBER IS: [0]
THE RHINO REGISTER NAMES IS: [B]
THE RHINO REGISTER READWRITEBIT IS: [3]
THE RHINO REGISTER SIZE IS: [2]
THE NEXT REGISTER ON THE BOARD IS
THE RHINO REGISTER RHINONAMENUMBER IS: [0]
THE RHINO REGISTER NAMES IS: [OUT]
THE RHINO REGISTER READWRITEBIT IS: [3]
THE RHINO REGISTER SIZE IS: [8]
THE NEXT REGISTER ON THE BOARD IS
THE RHINO REGISTER RHINONAMENUMBER IS: [-1]
THE RHINO REGISTER NAMES IS: []
THE RHINO REGISTER READWRITEBIT IS: [0]
THE RHINO REGISTER SIZE IS: [0]
THERE ARE NO MORE REGISTERS TO RECIEVE
FINISHED EXECUTING THE LIST DEVICE REGISTER METHOD
```

Figure 6.22: Picture of list_device_register() function call from the control computer command line

As can be seen in Figure 6.22 the framework is able to print out the correct registers, their read and write bit and lastly their sizes as presented in the symbol file shown in Figure 6.23. The list_device_register function therefore executed successfully.

rhino_adder.sym ×

```
A       3       0x08000000      0x2
B       3       0x08000002      0x2
OUT     3       0x08000000      0x8
```

Figure 6.23: Symbol file defining all the accessible registers in the BOF process

6.3 RACCMS Test Case using a GPMC_test Application

The test case that was used for the case study was a gpmc_test application. It was designed to test that the GPMC interface on the RHINO board was functioning correctly. The gpmc_test application had 4 memory regions allocated to it that represent physical peripherals on the RHINO. These memory regions were BOF process registers that were accessible through the IOREG folder. The physical peripherals these memory regions represented were the RHINO's reg_leds, the reg_fmc, reg_files and the reg_word. Illustrations of framework performing several function calls on the gpmc_test application are presented in the figures below.

The initial test using the test case application, was programming the gpmc_test BOF process on to the RHINO using the framework.

```
PLEASE ENTER IN THE COMMAND YOU WOULD LIKE TO CALL:
4
YOU HAVE CHOOSEN THE METHOD loadBofFile
PLEASE ENTER IN THE BOFPROCESS NAME THE RHINO NUMBER AND THE USERS SER
VERNAME
gpmc_test 0 chrval001
THE BOFPROCESSNAME IS: [gpmc_test]
THE SERVERUSERNAME IS: [chrval001]
RHINO IPADDRESS EXISTS AND IS BEING PROGRAMMED
THE RHINOID FROM THE STATUS TABLE:[0]
Client_MsgID IS: [50331648]
Client_RHINOID IS: [0]
Client_REG_NAME IS: [gpmc_test]
Client_DATA IS: 0
Client_USER_NAME: [chrval001]
RECIEVED THE SERVER MSG
RHINO NAME NUMBER:[0]
THE RHINO IP ADDRESS IS: [10.0.0.103]
THE RHINO BUSY BIT IS: [1]
THE RHINO REGISTER NAMES AS IS: []
ERROR MSG FROM THE RHINO BOARD: []
RHINO BUSYBIT: [1]
RHINO SUCCESS BIT: [1]
THE ELAPSED TIME FOR LOAD_BOF_FILE IS: [30]
IF 1 BOFPROCESS HAS SUCESSFULLY STARTED, ELSE IF 0 BOFPROCESS HAS BOFP
ROCESS HAS NOT STARTED: [1]
```

Figure 6.24: Command line output of RACCMS framework starting gpmc_test application

Figure 6.24 shows the framework successfully uploading a BOF process on the command line output and Figure 6.25 shows the gmpc_test BOF process running on the RHINO board. The orange LEDs are indicative of this.

Figure 6.25: Loading configuring the RHINO with the gpmc_test.bof gateway design illustration from the command line and the RHINO board

The second test was writing to the reg_led register. The reason this register was selected was because the output of this write would be visible through the FPGA and processor user LEDs which are the illustrated as the orange light LEDs on the board.


```
# od -d /proc/739/hw/ioreg/reg_led
0000000     3
0000002
#
```

Figure 6.26: Illustration of Write to Register reg_led while the gpmc_test.bof process is running on the RHINO from command line and RHINO board

A register write using the framework was successful which can be can be seen by the first two FPGA and processor user LEDs being turned on. These lights represents a '11' which is the hexadecimal value of a 3. This can also be seen by the ioreg read of the register reg_led which outputs a value of 3.

```
6
YOU HAVE CHOOSEN THE METHOD regRead
PLEASE ENTER THE RHINO YOU WANT TO READ TO, THE NAME OF THE REGISTER
0 reg_led
REGISTER NAME IS:[reg_led]
THE RHINO IS IN THE CLUSTER
THE RHINO IS IS BEING USED BY YOU
THE RHINO BOARD IS PROGRAMMED AND AN ATTEMPT TO READ A REGISTER WILL S
TART
Client_MsgID IS: 33554432
Client_RHINOID IS: 0
Client_REG_NAME IS: reg_led
Client_DATA IS: 0
Client_USER_NAME: []
JUST SENT THE CLIENT MSG
RECIEVED THE SERVER MSG
RHINO NAME NUMBER:[0]
THE RHINO IP ADDRESS IS: [10.0.0.103]
RHINO BUSYBIT: [1]
RHINO DATA IS: [3]
RHINO SUCCESS BIT: [16777216]
DATA IN REGISTER IS: [3]
```

Figure 6.27: Illustration of a Register Read from reg_led on running gpmc_test.bof process

A read using the framework on the test case application was successful, which is indicated by the value of 3 returned from the functional call as seen in Figure 6.27. This is the same value that can be seen on the RHINO FPGA and processor LEDs as well as in the ioreg/reg_led file.

The last test was to see if the framework was able to retrieve the correct registers from the gpmc_test application. This was done by using the RACCMS list_device_register

functional call. The results of this functional call can be seen in Figure 6.28 below.

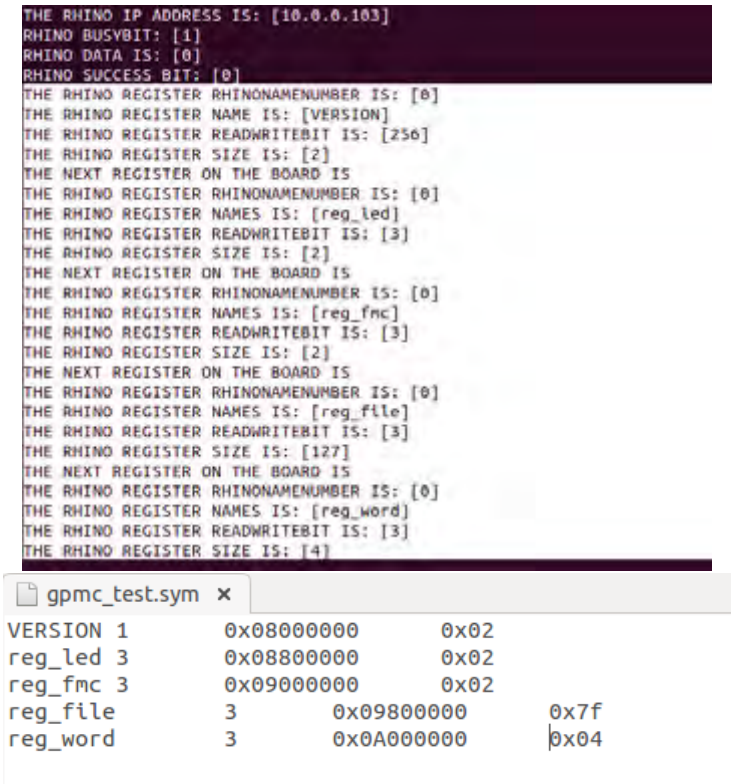


Figure 6.28: Illustration of list of Registers returned from Framework compared to those in the Symbol File

As can be seen in Figure 6.28 (top command line output), the registers were listed correctly with all correct read/write bit values and the correct register sizes as provided by the gpmc_test.sym file, also shown in Figure 6.28 (the bottom image).

6.3.1 Connecting Multiple RHINOs in Cluster

Three RHINOs were connected together through a switch to form a star cluster. The RHINOs were then connected to a laptop that acted as the control computer during the testing of the framework. The setup connection can be seen in Figure 6.29 below.

Figure 6.29: Picture showing physical cluster setup from top view of the cluster as well as a side view of the cluster

The 3 RHINOs were contained in 3 different chassis, a silver, gold and black one as can be seen in Figure 6.29. The framework was easily scalable from one to three nodes on the cluster by simply modifying the RHINO numbers in the individual server applications running on the board and connecting up the new boards to the cluster through the switch.

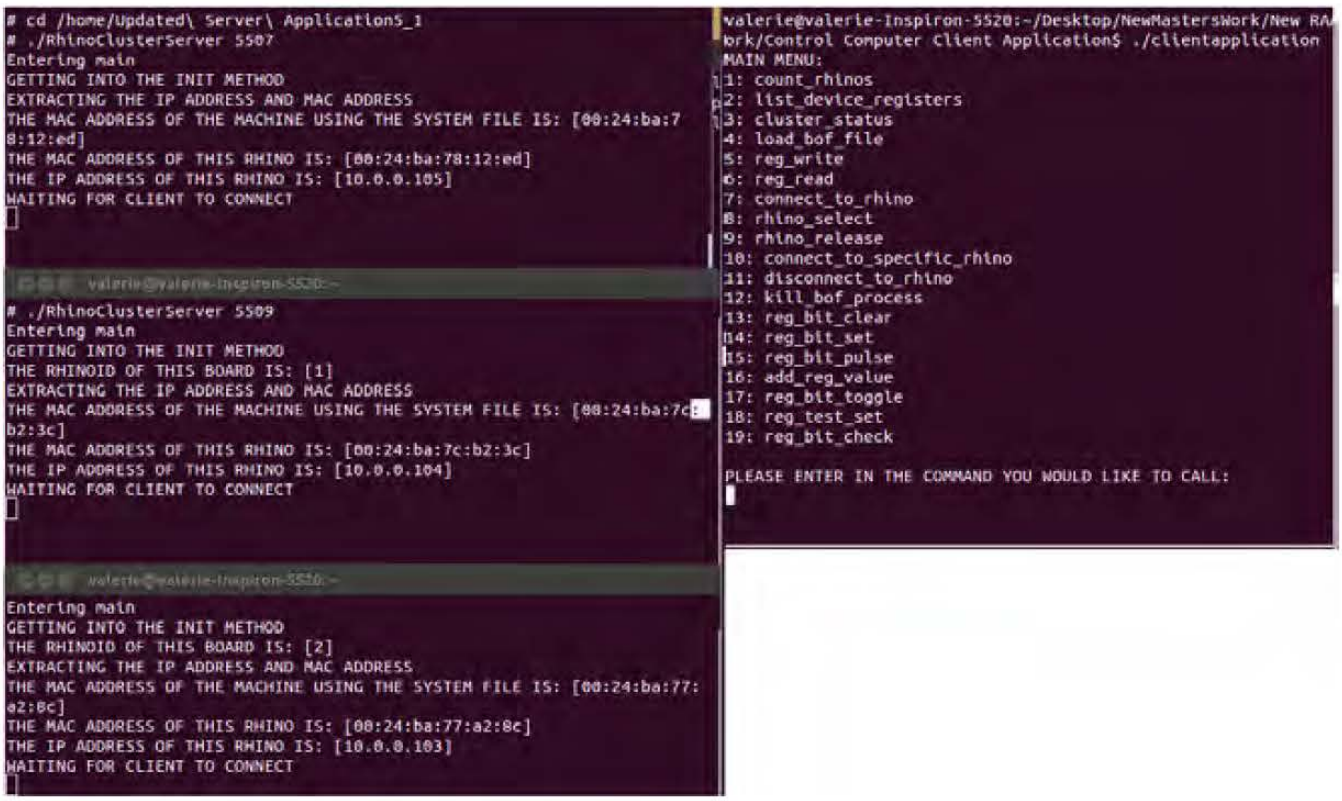


Figure 6.30: Image showing command line output of 3 RHINOs (the 3 command line windows to the left) connected to the Control Computer (the command window the the right of the screen) derived from the setup see in Figure 6.29

It should be noted that each RHINO was assigned its unique IP address as seen in the command windows to the left of Figure 6.30, based on their individual MAC addresses. This shows that the DHCP server was able to assign each RHINO board a static IP addresses successfully based on its MAC address.

6.4 RACCMS Graphical User Interface Tests

This section covers the tests that were executed on the designed RACCMS GUI, explained in section 5.6, as it interacted with the rest of the designed framework. As the GUI utilizes functions that had already been thoroughly tested in the sections above, these tests only aimed to see if the GUI provided an accurate interface to the framework functions. All the framework functions accessible through the GUI were executed and as with the user acceptance tests described above, the same results were observed using the GUI as opposed to the command line interface.

Of note were the tests to see if the GUI would enable access to the administrator's page without the correct password. The framework was designed to only allow the framework's administrator access by using the password *"rhino_admin1"* to access his/her page. As can be seen in Figure 6.31, if an incorrect password was entered in the framework would

open a popup window with an error message indicating the incorrect password had been entered and it then prompted the user to try again. If the correct password was entered the administrator's page would open.

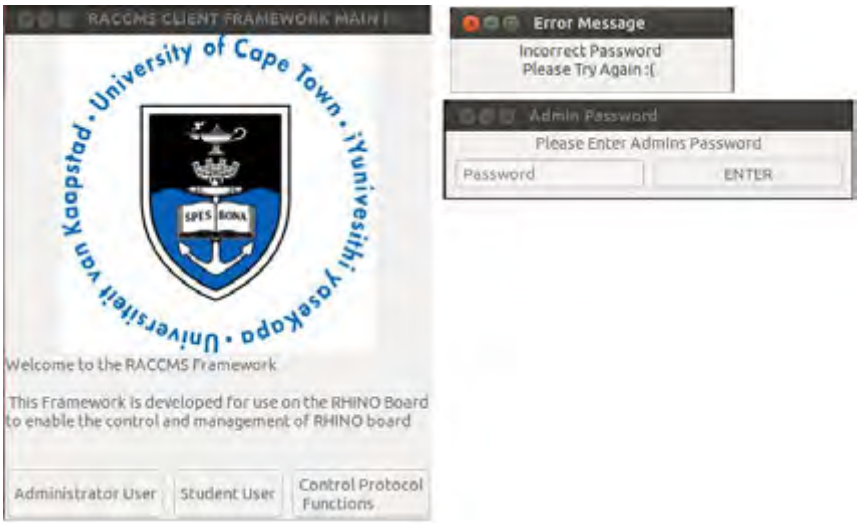


Figure 6.31: GUI illustrating access to administrators page through a password dialog box with incorrect password dialog box showing

The second notable test was to see if the GUI was able to illustrate the correct status of the framework based on the RHINO status table. Figure 6.32 below illustrates the Administrator's page showing the status of the cluster after the administrator has connected to it.

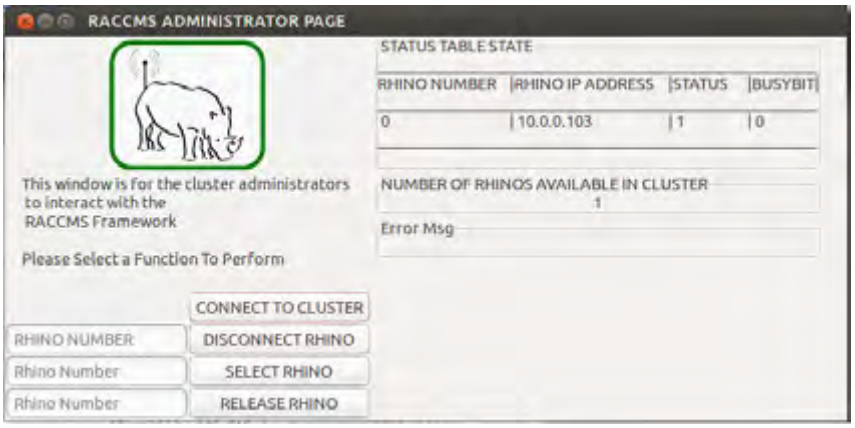


Figure 6.32: GUI illustrates administrator page after "connect to cluster" button is pressed and the RHINO has not yet been selected for use by a user

As can be seen in Figure 6.32, the RACCMS framework has only one RHINO successfully connected to it indicated by the *"1"* in the *"STATUS"* column. As the RHINO has not yet been selected for use the number of RHINOs available in the cluster for use are illustrated by the number "1". Figure 6.33 below goes on to show the new RHINO status updated to indicate that there are no RHINOs for use in the cluster indicated by the identical RHINO status table in both the Administrator's page and the student user's page. The number of RHINOs available in the cluster which was updated to *"0"*.

Figure 6.33: GUI pages illustrating administrator's page and the student users page showing the same RHINO status table

Chapter 7

CONCLUSION AND RECOMMENDATIONS

The aim of this project has been to develop a customized framework that will run specifically on the RHINO board developed at the University of Cape Town as a teaching platform for students to build SDR and Radio Astronomy applications. The designed framework abstracts the complexities involved in using the board and hence increases its accessibility and usability among inexperienced reconfigurable computing RC programmers who many want to utilize the board. The framework also provides a management aspect that allows for multiple RHINO boards to be accessible to users from a remote location at any given time. The framework was the first software application that ran on the RHINO which is still currently in its prototype phase. The work presented in this dissertation has been presented at the 2013 flagship IEEE Africon conference **[16]** and the RACCMS framework has successfully been implemented on the RHINO board.

This chapter presents a summary of the main objectives of the dissertation and the conclusions drawn from the development and implementation of the framework. The results from the tests are discussed and compared to the initial framework requirements set at the start of the project. The chapter ends with possible further directions this project could take that will add on to the usability of the framework.

7.1 Framework Functionality

This section discusses how well the implemented framework function calls met the functional requirements specified at the start of the dissertation. This section utilizes the black box tests that were performed on the framework to determine the extent the functional requirements were met. As well as the overall success of the combined functional units is provided.

7.1.1 [F1] Server-Client Application

The server -client application was implemented successfully for both the single-server single-client application as well as for the multi-server single-client setup. The server-client application managed to ensure both data and network integrity. The server application was able to successfully translate received packets to the relevant server API functional calls. The server-client application successfully enabled users to remotely access the RHINO board for both control and management purposes.

7.1.2 [F3] FPGA-ARM Control and Communication

The implemented BORPH system calls enabled the user to configure a FPGA using the framework, that abstracted this functionality to a single phrase function call. The execution of these API function calls was also successful for all register read and write functionalities. The framework through BORPH successfully allowed the RHINO board to be configured with a designed gateware process given that all the input data was correct.

7.1.3 DHCP Client

Each RHINO, through its DHCP client, was successfully allocated a unique IP address from the DHCP server sitting on the control computer based on its MAC Address . The IP address allocated every time the RHINO connected to the control computer remained constant, showing that the DHCP client on the RHINO's ARM was always successful in being issued the correct IP address.

7.1.4 API Library Functions

Both the designed API libraries for the client and the server application executed as designed on the RHINO board. The functions were developed in such a way that they could be successfully included into user defined code and utilized to build customized applications for the RHINO board. Furthermore all the function tests showed that each of the defined function was implemented and functioned intended. Each function was able to handle error conditions in a graceful manner to avoid catastrophic crashes of the framework.

7.1.5 Control Protocol

A unique control protocol for the RACCMS framework was designed. It consisted of two major packet structures, the client packet structure generated on client application on the

control computer and the server packet generated on the server application sitting on the RHINO ARM processor. The control protocol enabled successful transfer of data using TCP/IP while maintaining data integrity which was shown in the network integrity tests in Figure 6.13.

7.2 Recommendations for future work

The initial design of the framework uses BORPH system calls to perform all necessary register read and write operations on a running BOF process. However these systems calls have been noted to be very costly and the latencies resulting in noticeable performance degradations for applications that make short or random accesses to FPGA resources [52]. A Linux memory mapped[1] device driver has been designed for the ROACH board that could be ported to the RHINO board once it is fully functional. The device driver acts as an interface between the Power PC and the FPGA on the ROACH board [52]. It was developed to reduce current system call latencies which the BORPH OS incurs on the ROACH. The Linux device driver could be ported to the RHINO board and the memory map device driver used as opposed to the BORPH OS system calls to perform the register read and writes, in order to reduce unnecessary overhead.

[1]Memory mapping forms an association between the FPGA and the user process in memory

Appendix A

Source Code

This DVD attachment is included with the dissertation and gives all the source code, project files and documentation not included in the written dissertation.

The folder structure organized in the DVD is as follows:

- Source Code (Contains all the project files relevant to the research
- Electronic copy of the dissertation
- Framework Documented Unit Test Cases

Appendix B

RACCMS Framework Flow Diagrams

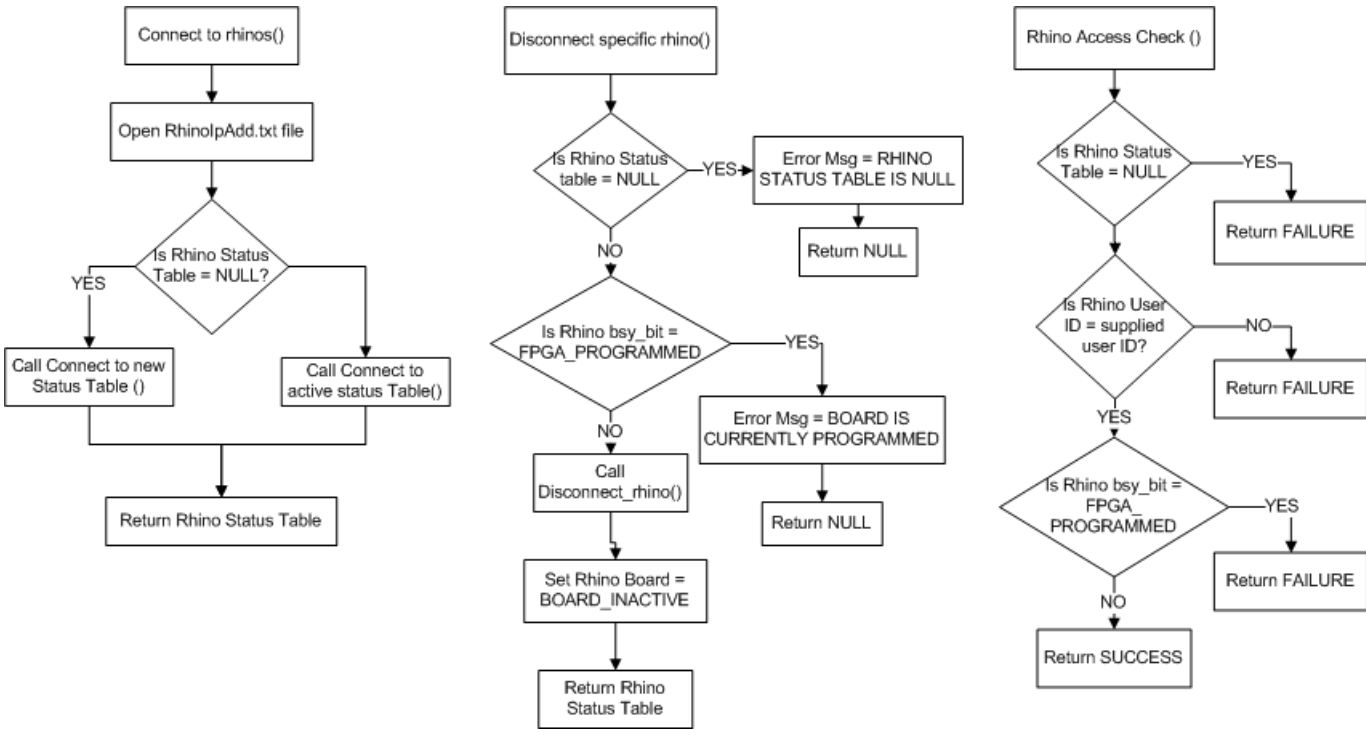


Figure B.1: RACCMS Client Application's Connection and Disconnection and Access Check Flow Diagrams

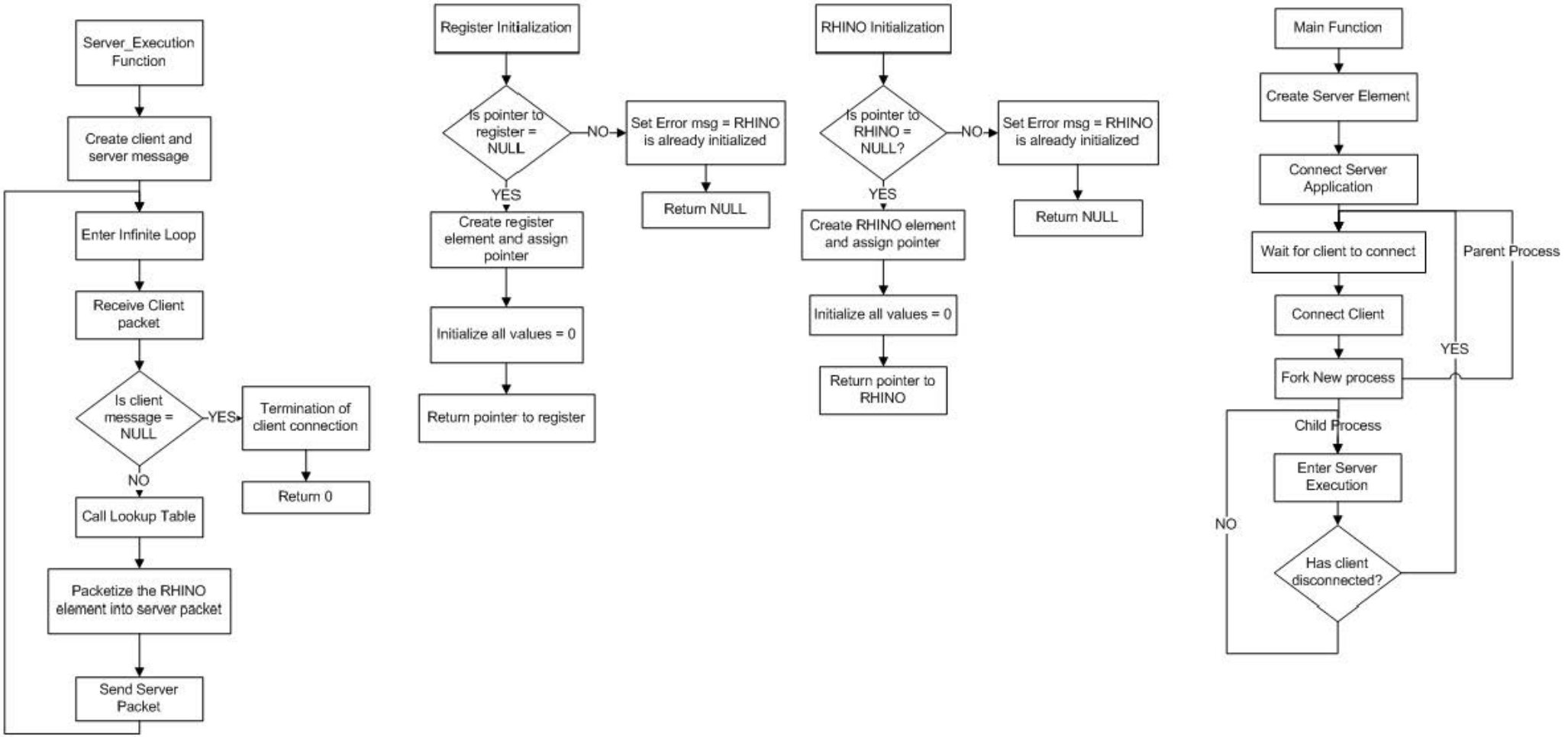


Figure B.2: Server Application Management Function Flow Diagrams

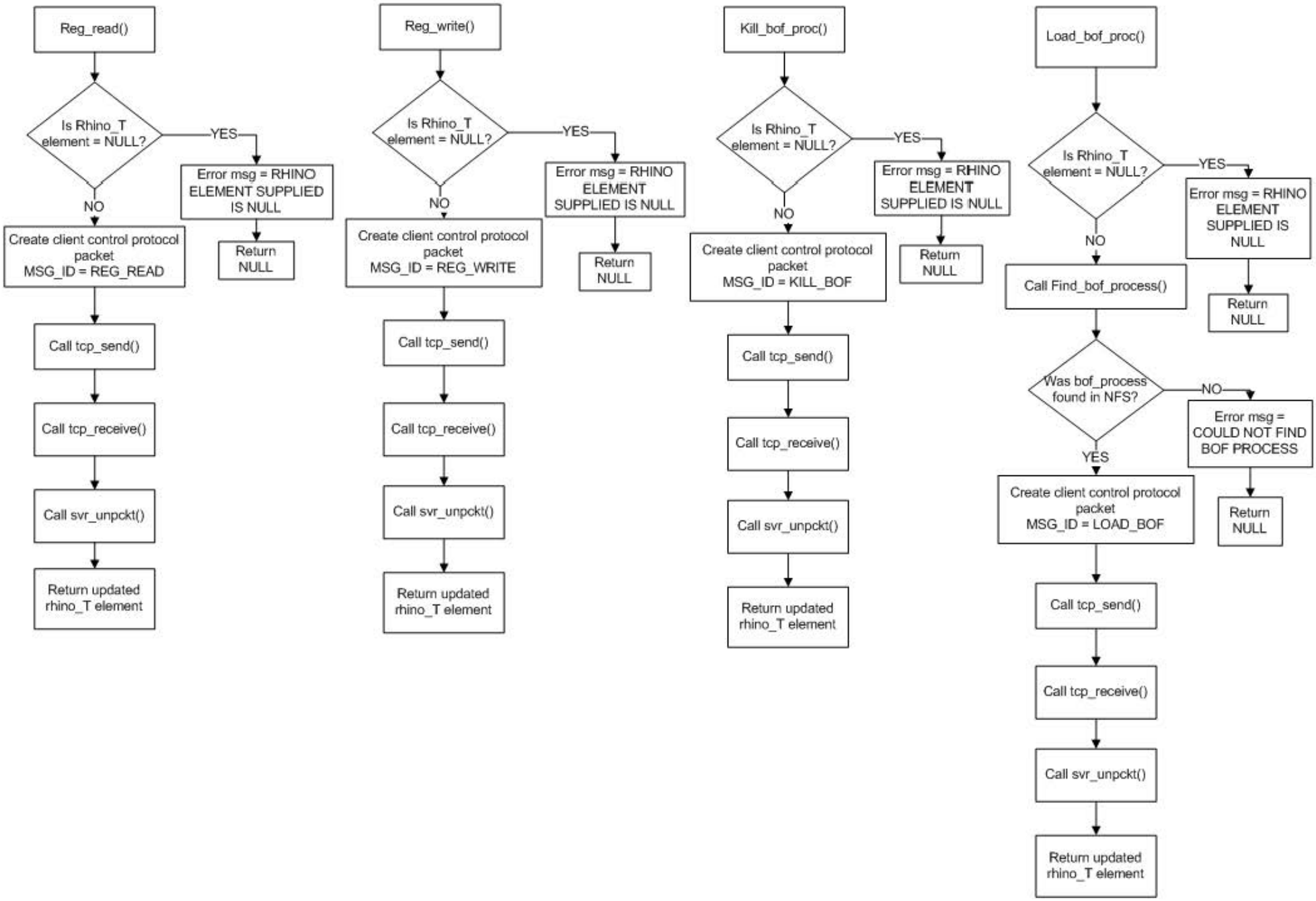
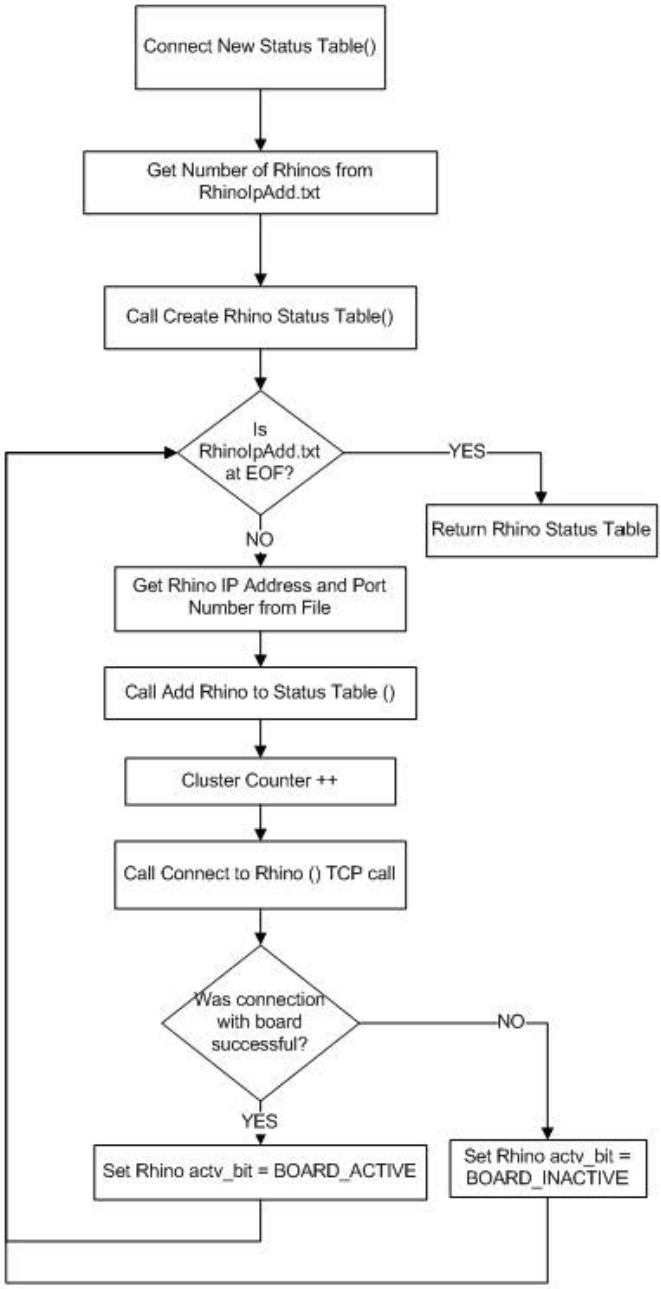


Figure B.3: RACCMS Client Application Control Protocol Function File Flow Diagrams



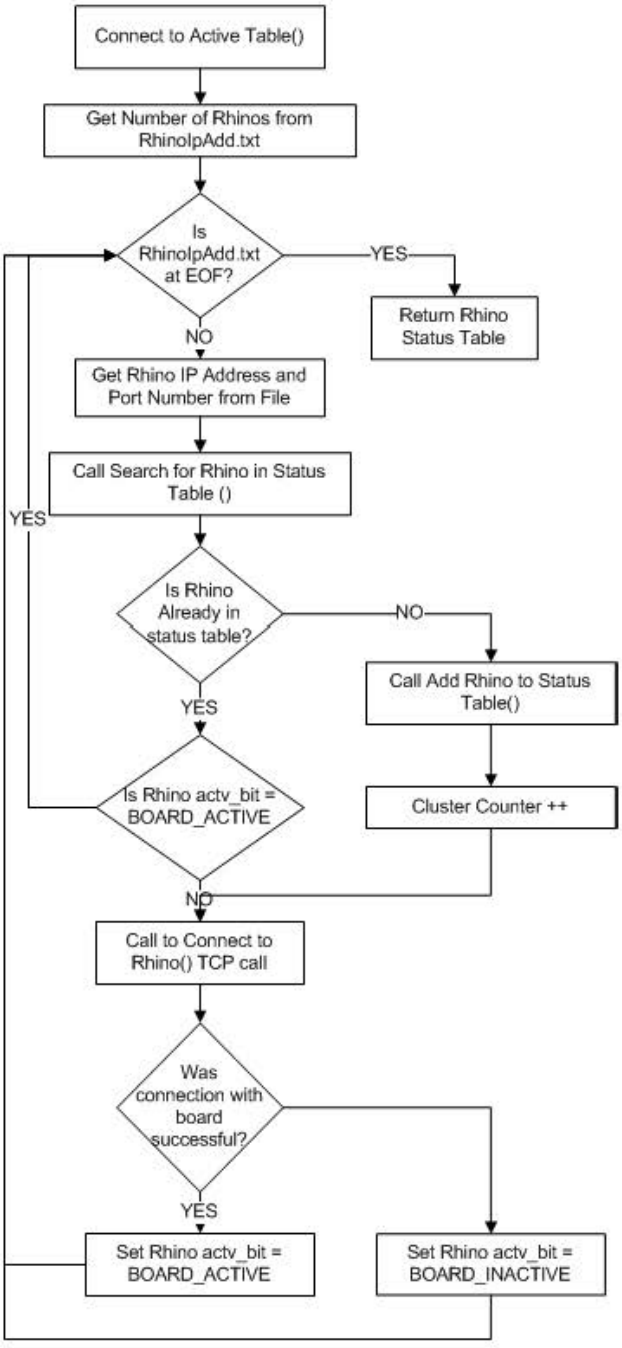


Figure B.4: Client Application Connection Function File Flow Diagrams

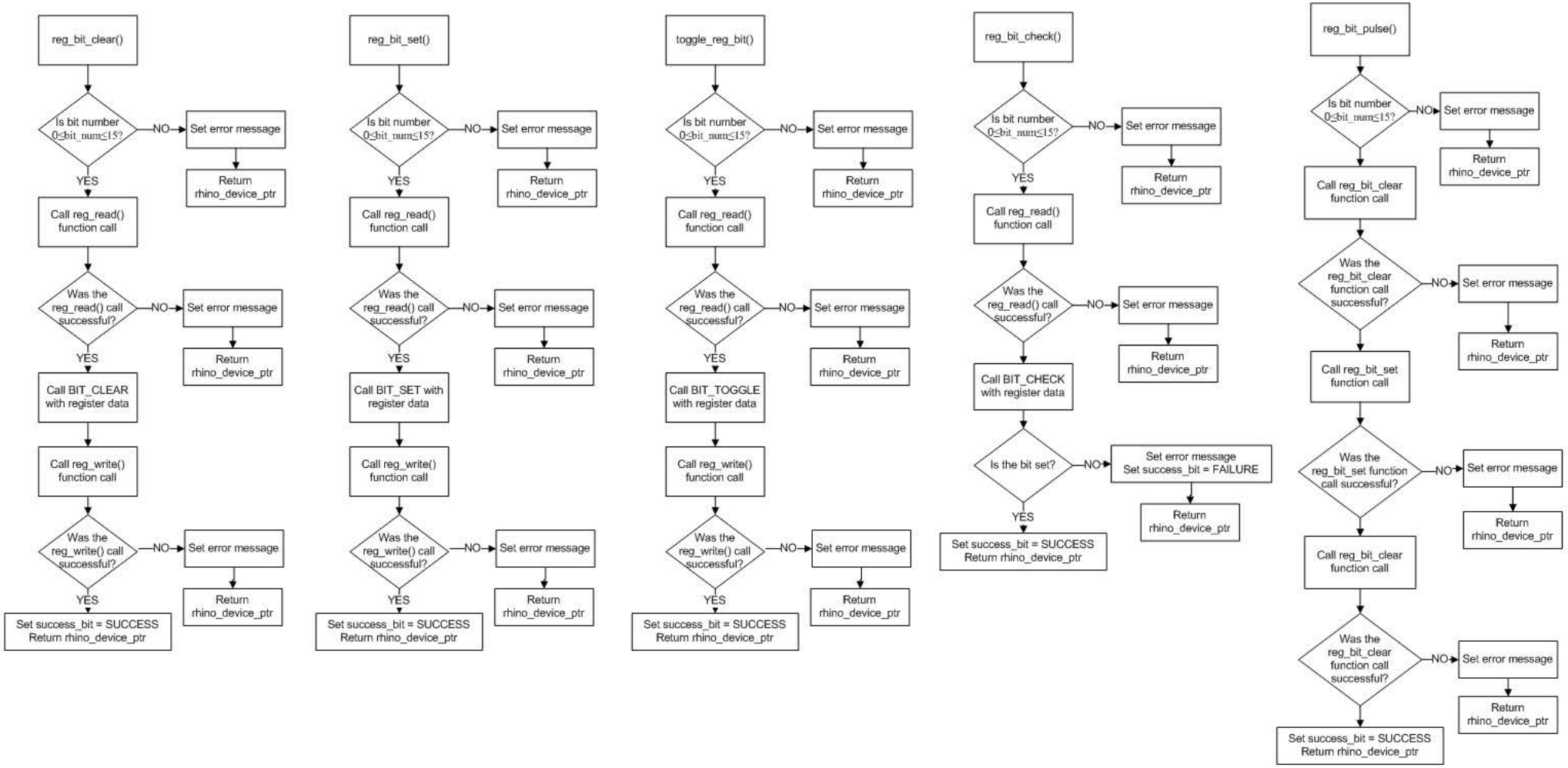


Figure B.5: Flow diagram of the function calls that operate on a single bit of a given register value

Appendix C

RACCMS Framework Sequence Diagrams

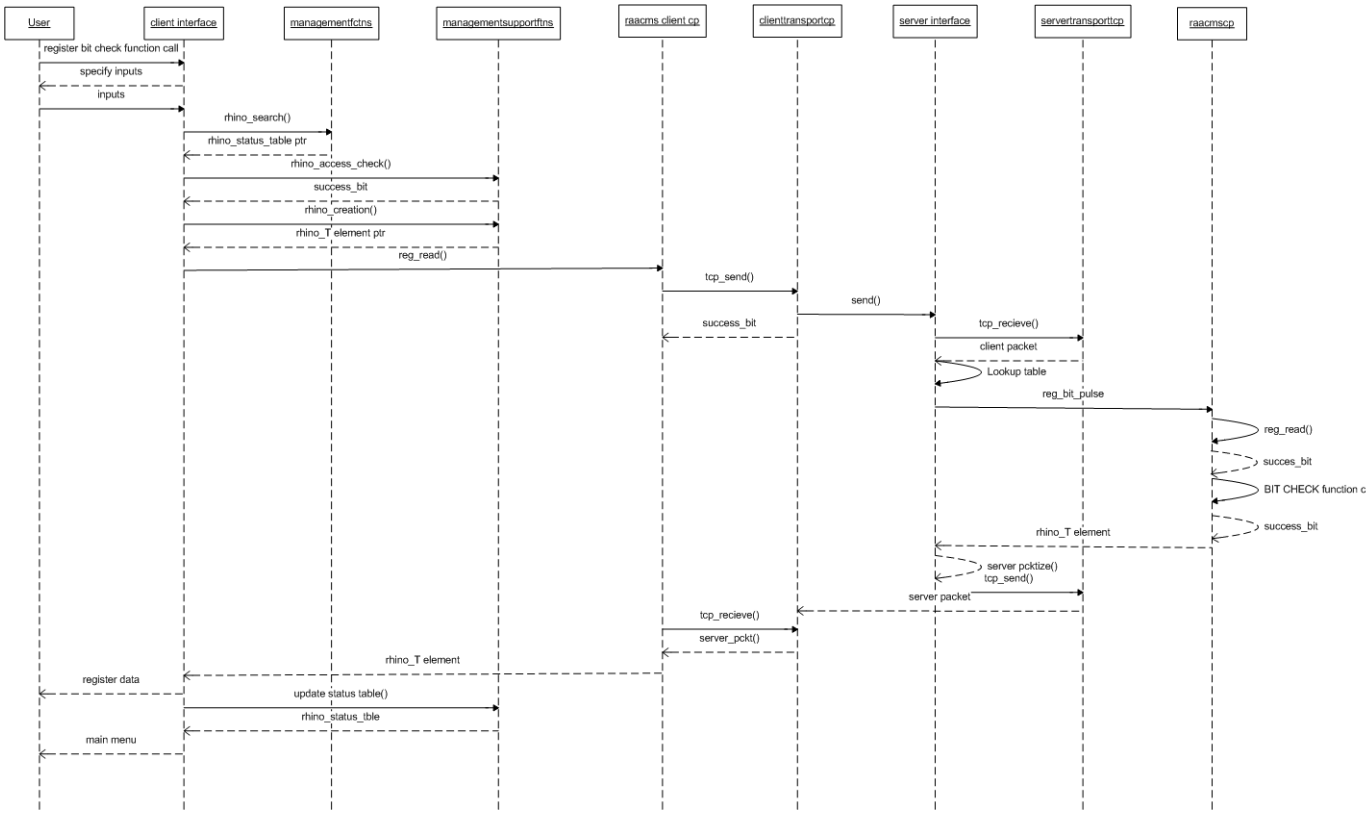


Figure C.1: Illustration of sequence diagram of the register check bit function call

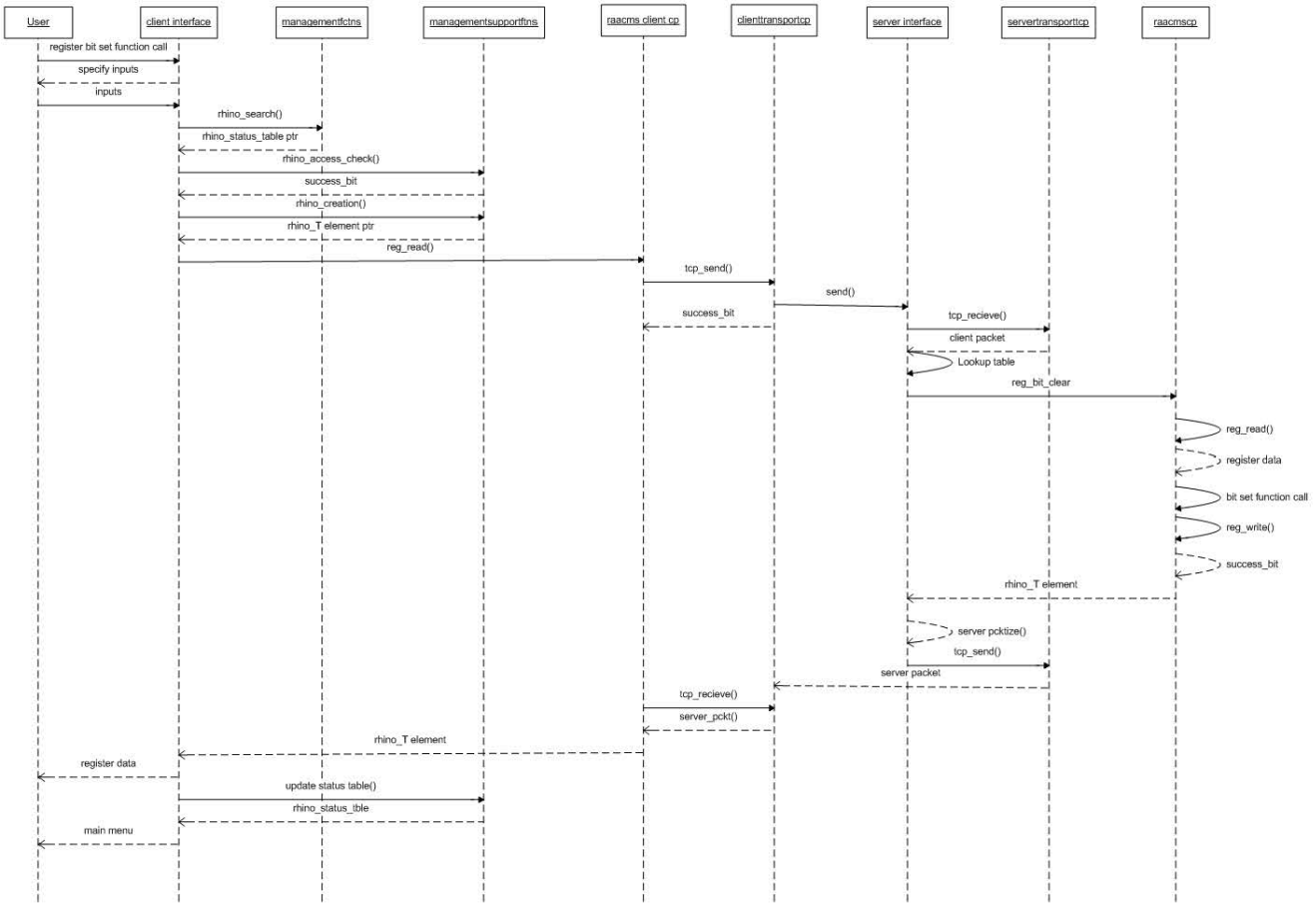


Figure C.2: Illustration of sequence diagram for the register set function call c

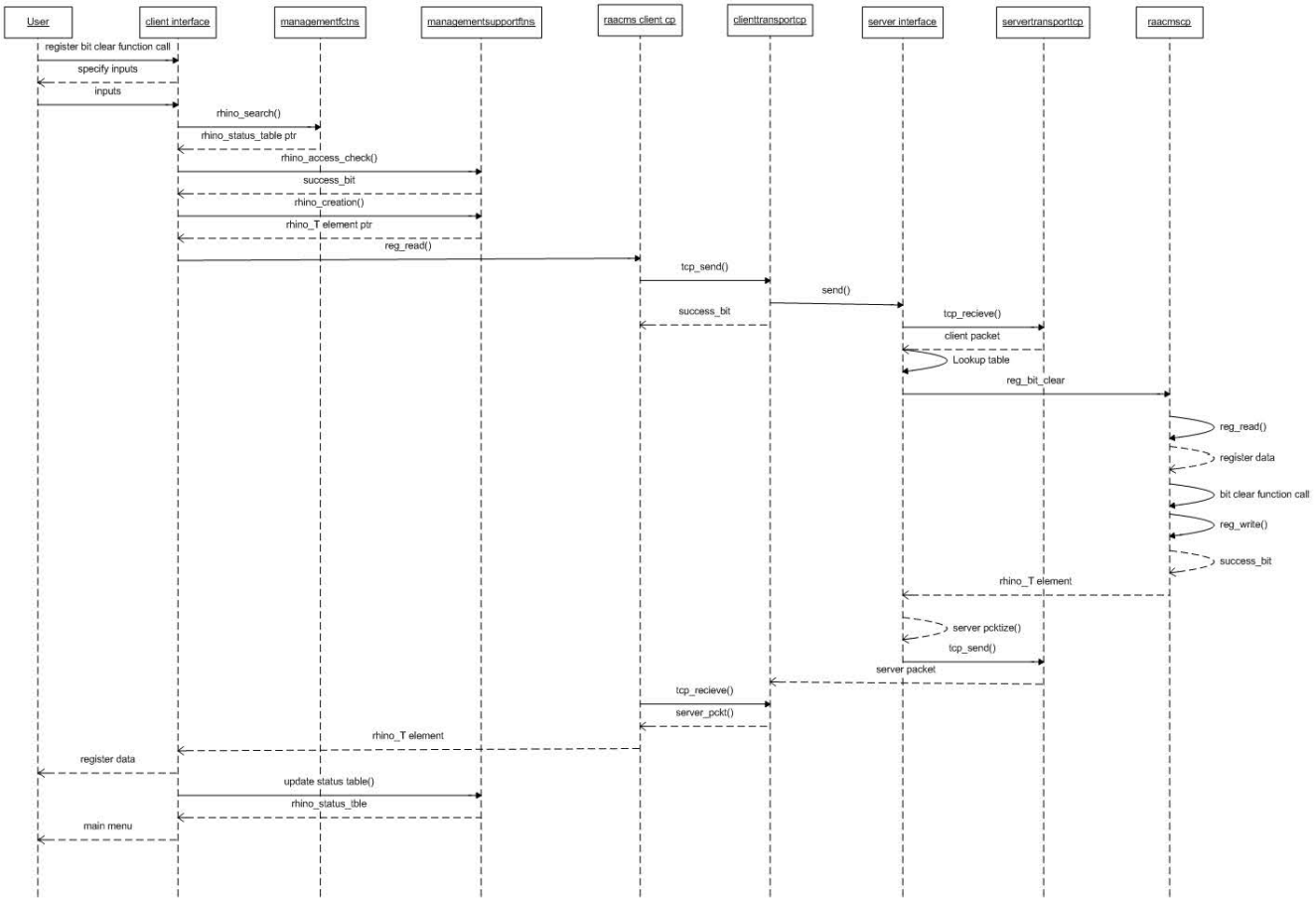


Figure C.3: Illustration of sequence diagram for the register clear function call

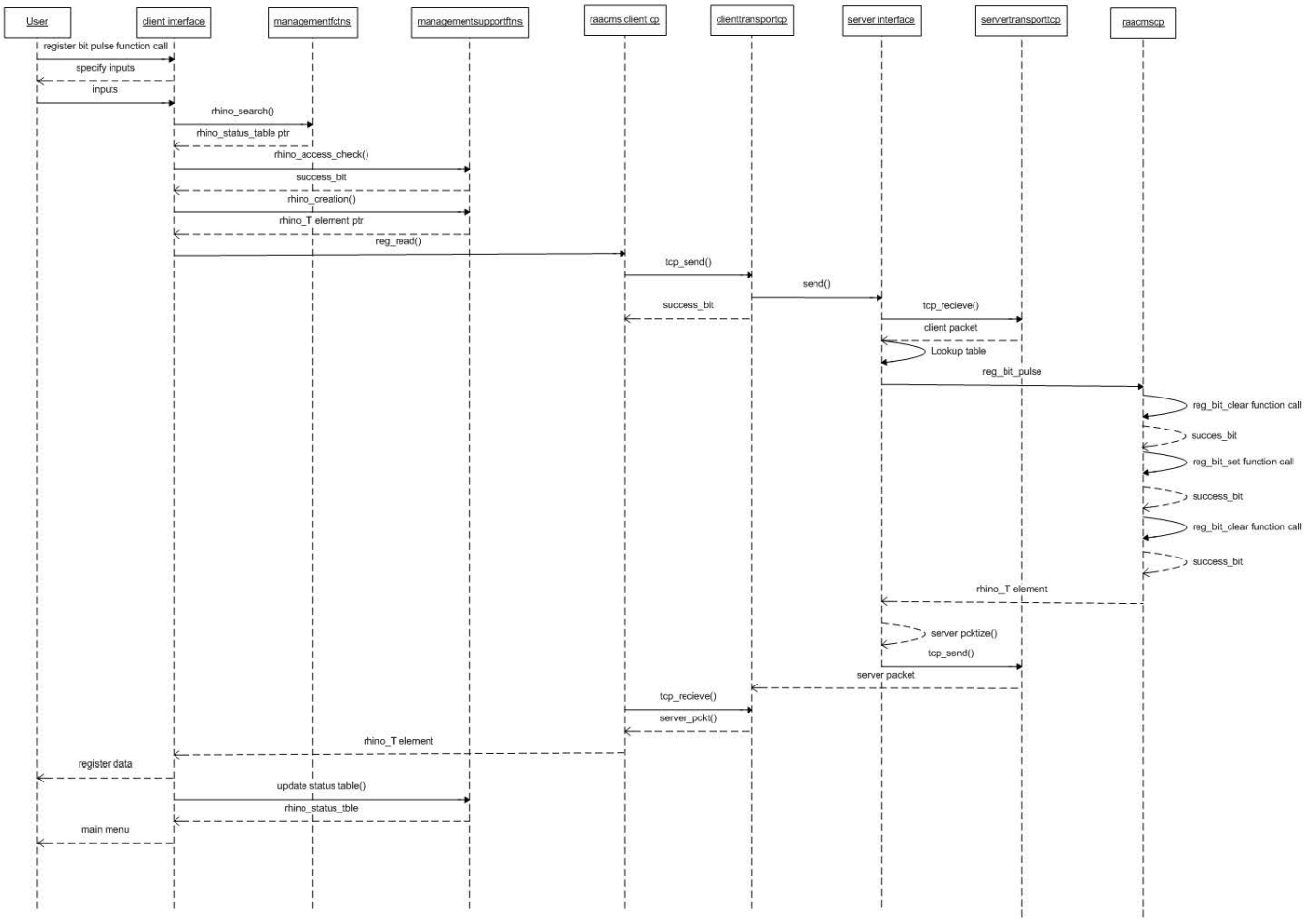


Figure C.4: Illustration of sequence diagram for the register bit pulse function call

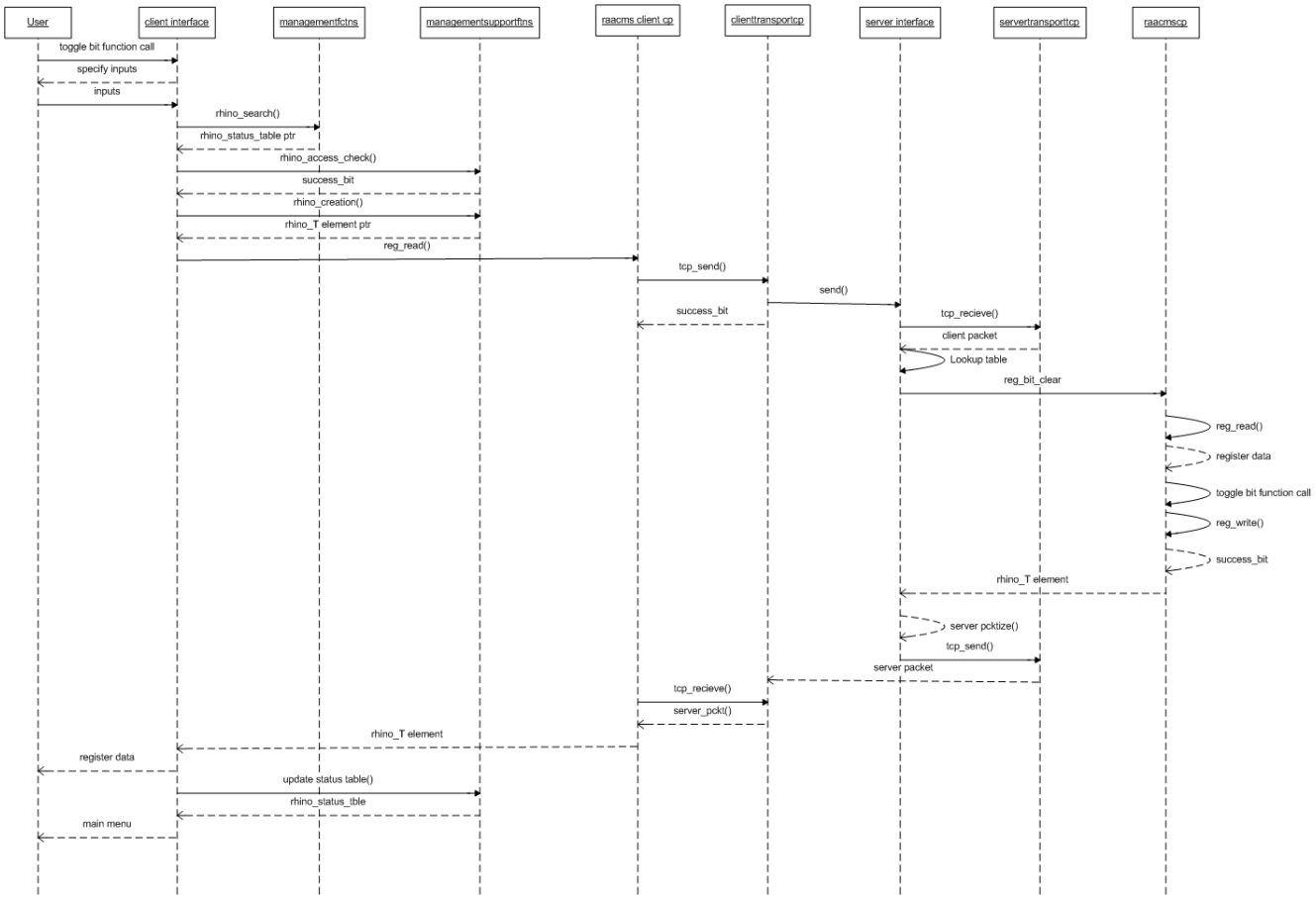


Figure C.5: Illustration of sequence diagram for the toggle register bit function call

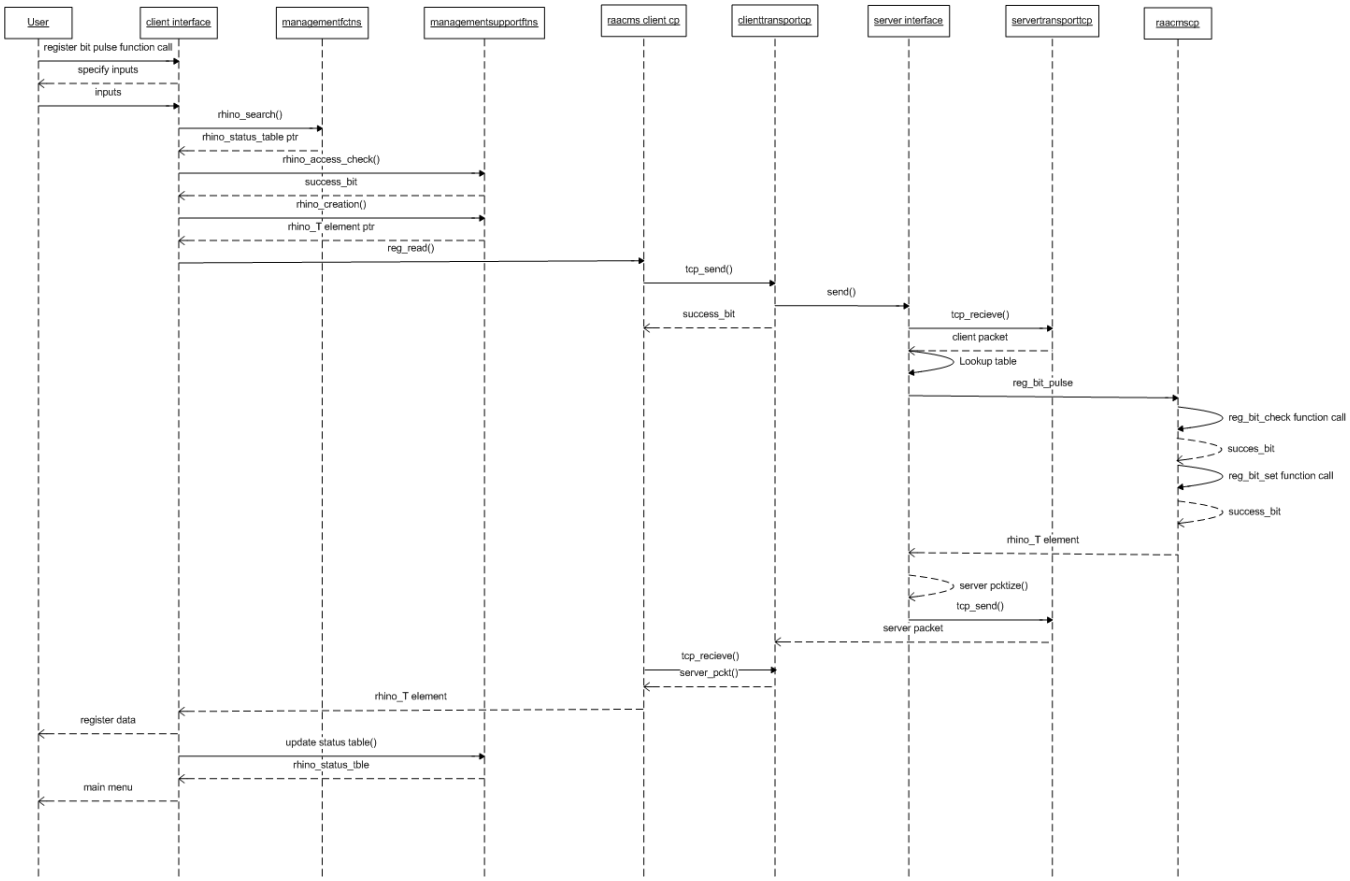


Figure C.6: Illustration of sequence diagram for the register test and set function call

Bibliography

[1] The MPI Message Passing Interface Standard. [Online], http://www.mcs.anl.gov/mpi/standard.html, 1995.

[2] KATCP. [Online], https://casper.berkeley.edu/wiki/KATCP, 2012.

[3] TCPDump. [Online], www.tcpdump.org/tcpdump_man.html, July 2012.

[4] Center for Astronomy Signal Processing and Electronics Research. [Online], https://casper.berkeley.edu, August 2013.

[5] Check Unit Testing Framework for C. [Online], http://www.check.sourceforge.net/doc/check_html/index.html, 2013.

[6] ELDK 5.4 Documentation. [Online], www.denx.de/pub/eldk/5.3/targets/armv7a, 2013.

[7] FPGA Architecture. [Online], www.electronicproducts.com/Digtal_ICs/Standard_and_Programmable_Logic/The_evolution_of_FPGA_coprocessing.aspx, 2013.

[8] Linux Kernal Coding Style. [Online], http://www.kernel.org/doc/Documentation/CodingStyle, 2013.

[9] Port Number Lists. [Online], http://whatismyipaddress.com/port-list, 2013.

[10] ROACH BOARD. [Online], www.ska.ac.za, 2013.

[11] Sockets Tutorial. [Online], www.linuxhowtos.org/C_C++/socket.htm, 2013.

[12] The GTK+ Project. [Online], http://www.gtk.org, 2013.

[13] USRP Board. [Online], www.ettus.com/product/details/UN210-KIT, 2013.

[14] Valgrind Documentation. [Online], http://www.valgrind.org/docs/manual/valgrind_manual.pdf, 2013.

[15] Definition of Computer Management. [Online], www.ask.com/question/definition-of-computer-management, Jan 2014.

[16] *RHINO ARM API cluster Control Managemet System (RAACMS)*, Sep 2013.

[17] M. Saldana, A. Patel, C . Madill, D. Nunes, D. Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow. MPI as an abstraction for software-hardware interaction for HPRCs. In *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, pages 1–10. IEEE, 2008.

[18] A. Langman, B. Hamilton. RHINO GMPC Test. `[GitHubOnline],http://github.com/brandonhamilton/rhino/tree/master/examples/gpmc_test`, 2011.

[19] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Droz, C. de Jesus, D. MacMahon, A. Siemion, D. Wetheimer, and M. Write. A New Approach to Radio Astronomy Singal Processing. In *General Assembly of the International Union of Radio Science*, October 2005.

[20] A Reconfigurable Extension to the Network Interface of Beowulf Clusters. Underwood, Keith D and Sass, Ron and Ligon III, Walter B. In *Cluster*, volume 1, pages 212–221, 2001.

[21] BEEcube. BEE4: All Programmable Rack Servers. `http://www.beecube.com/products/BEE4.asp`, 2013.

[22] Berkely Design Technology Inc. Bdti dsp dictionary, Nov 2013.

[23] Boehm, Barry W. A spiral model of software development and enhancement. *Computer*, 21(5):61 – 72, 1988.

[24] C.Chang and J. Wawrzynek and R. W. Brodersen. "BEE2: A high-end reconfigurable computing system". In *"Proc IEEE Design and Test of Computers"*, pages 114–125, 2005.

[25] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. In *ACM Comput. Surv*, volume 34, pages 171–210, 2002.

[26] D. Roddey. Control Protocol Design. *Charmed Quark, New York*, 2011.

[27] E. Borger and U. Glasser. A Formal Specification of the PVM Architecture, 1994.

[28] Ettus Research. *USRP N200 Series Datasheet*, Nov 2010.

[29] Feng, Tse-yun. A survey of interconnection networks. *Computer*, 14:12 – 27, 1981.

[30] G. Bochmann and C. Sunshine. Formal methods in communication protocol design. *Communications, IEEE Transactions on*, 28(4):624 – 631, 1980.

[31] Gropp W. Lusk E. Goals guiding design: PVM and MPI. In *Cluster Computing Proceeding IEEE International Conference on*, pages 257 – 265, 2002.

[32] G.V. Bochmann, D. Rayner and C.H. West. Some notes on the history of protocol engineering . *Computer Networks*, 54(18):3197–3209, 2010.

[33] H. Kwok-Hay So and H. K.-H. So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. ProQuest, 2007.

[34] H. Kwok-Hay So, R. Brodersen. Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support. In *Field Programmable Logic and Applications 2006 FPL 06 International Conference on*, pages 1–6, August 2006.

[35] H. Kwok-Hay So, R. Brodersen. A unified hardware/software runtime enviroment for FPGA-Based reconfigurable computers using BORPH. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–14, 2008.

[36] H. Zimmermann. OSI reference model –The ISO model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.

[37] I.A. Kalyaev, I.I. Levin, A.I. Dordopulo and L.M. Slasten. FPGA-based reconfigurable computer systems. In *Science and Information Conference (SAI)*, pages 148–155. IEEE, 2013.

[38] J. Davis, C. Thacker, and C. Chang. BEE3: Revitalizing Computer Architecture Research. Technical Report MSR-TR-2009-45, Microsoft Research, 2009.

[39] J.D. Day, H. Zimmermann. The OSI reference model. volume 71, pages 1334–1340. IEEE, 1983.

[40] J.Dongarra, S.W. Otto, M. Snir and Walker. An Introduction to the MPI Standard. `http://www.netlib.org/tennesee/ut-cs-95-274`, January 1995.

[41] P. Jon. RFC 793:Transmission Control Protocol, September 1981. *Status: Standard*, 1981.

[42] K. Thramboulidis and A. Mikroyannidis. Using UML for the Design of Communication Protocls: The TCP case study. In *IEEE International Conference on Software, Telecommunications and Computer Networks, Dubrovnic, Croatia*, 2003.

[43] Khan, Rfiqul, A. Zaman, Javed. A Practical Performance Analysis of Modern Software use in High Performance Computing. *Proceeding of International Journal of Advanced Research in Computer Science and Software Engineering*, 2(3), 2012.

[44] M. Inggs, G. Inggs, A. Langman, S. Scott. Growing horns: Applying the Rhino Software defined radio system to radar. In *Radar Conference (RADAR), 2011 IEEE*, pages 951–955, 2011.

[45] Meredith, Marsha and Carrigan, Teresa and Brockman, James and Cloninger, Timothy and Privoznik, Jaroslav and Williams, Jeffery. Exploring Beowulf Clusters. *Journal of Computing Sciences in Colleges*, 18:268–0284, 2003.

[46] National Semiconductor, AN-1728. *IEEE 1588 Precision Time Protocol Time Synchronization Performance*, October 2007.

[47] Postel Jon. RFC 768: User Datagram Protocol. *User datagram protocol*, pages 1–3, 1980.

[48] R. Droms. Rfc 2131: Dynamic Host Configuration Protocol, March 1997. *Obsoletes RFC1541. Status: DRAFT STANDARD*, 3(1), 1997.

[49] R. Panse, V. Lindenstruth, T. Alt, H. Tilsner and L. Hess. A Hardware Based Cluster Control and Management System. *CHEP2004, CERN, " Interlaken(Suisse)*, 2004.

[50] R. S. Pressman and D. Ince. *Software Engineering: A Practitioner's Approach*, volume 6. McGraw-hill New York, 2005.

[51] W. B. R.Tessier. Reconfigurable Computing for Digital Signal Processing : A Syrvey. In *Journel of VLSI Signal Processing Systems*, volume 28, pages 7–27, 2001.

[52] S. Rajan, and M. Inggs, and M. Welz. Automated gateware discovery using open firmware. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 413–418, Oct 2013.

[53] S. Scott. Reconfigurable Hardware Interface for ComputatioN and RadiO. Master's thesis, University of Cape Town, 2011.

[54] S. Winberg, A. Mishra, B. Raw. RHINO blocks pulse-Doppler radar framework. In *Parallel Distributed and Grid Computing*, volume 23, pages 876–881, India, 2012. IEEE.

[55] S.J. Hussain and A. Ghufran. A comparitive study and analysis of PVM and MPI for parallel and distributed systems. In *Information and Communication Technologies, 2005. ICICT 2005. First Internation Conference on*, pages 183–187. IEEE, 2005.

[56] S.Katz, and S. Winberg, and A. Mishra. Rapid model-based prototyping tool for SDR on a RC platform. In *Parallel Distribution and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*, pages 711–715, 2012.

[57] Software Defined Radio Group, University of Cape Town. RHINO Wiki. `[Online] ,http://www.sdrg.ee.uct.ac.za/rhinowiki/index.php/Main_Page`, 2013.

[58] Texas Instruments Wiki. ARM3715/03 GPMC Subsystem. `processors.wiki.ti.com/index.php/AM315/03_GPMC_Subsystem`, 2013.

[59] The ALICE Collaboration. ALICE-Techinical Design Report of the Trigger, Data Acquisition, High-Level Trigger, and Control System. *CERN/LHCC/2003-062*, 2004.

[60] T.J.Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W.Luk and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. In *Computers and Digital Techniques IEEE Proceeding*, volume 152 of *1320-2387*, pages 193–207, 2005.

[61] T.M. Steinbeck, V. Lindenstruth, H. Tilsner. A Control Software for ALICE High Level Trigger. *Proceedings of the Computing in High Energy Physics*, 2004.

[62] Underwood, Keith D and Sass, Ron and Ligon III, Walter B. A Reconfigurable Extension to the Network Interface of Beowulf Clusters. In *cluster*, volume 1, pages 212–221, 2001.

[63] W. Stevens, R. Wright and R. Gary . *TCP/IP Illustrated: Vol.2: The Implementatoin*, volume 2. Addison-Wesley Professional, 1995.

[64] M. Welz, R. Crida, T. Bennett, S. Cross, R. Crida, L. van de Heever, and M. Marais. *"NRF-KAT7-6.0-IFCE-002-Rev5.pdf"*. "Square Kilometer Array (SKA)", Cape Town, South Africa, 2012.

[65] Wesley New. *Digitiser Master Controller (DMC) -Digitiser Interface Control Document*. Square Kilometer Array South Africa MeerKAT, 1.0 edition, 26 July 2013.

[66] S. Winberg, A. Langman, and S. Scott. "The RHINO platform Charging Towards Innovation and Skills Development in Software Defined Radio". In *"In South African Institute for Computer Scientists and Information Technologists Conference"*, Cape Town, South Africa, 2011.

[67] Xilinx. *XAPP 059 (version 1.1) Gate Count Capacity Metrics for FPGAs*, Feb 1997.

[68] Xilinx. *ISE Design Suite 14: Release Notes, Installation, and Licensing*, March 20 2013.