Distributed Reconfigurable Simulation for Communication Systems

Song Hun Kim

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfilment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Committee:

Dr. William H. Tranter (Chairman)
Dr. Jeffrey H. Reed
Dr. Scott F. Midkiff
Dr. Peter M. Athanas
Dr. C. Patrick Koelling

November 7, 2002

Blacksburg, Virginia

Keywords: Simulation, Reconfigurable Computing, Middleware, Distributed Computing

Distributed Reconfigurable Simulation for Communication Systems

Song Hun Kim

(ABSTRACT)

The simulation of physical-layer communication systems often requires long execution times. This is due to the nature of the Monte Carlo simulation. To obtain a valid result by producing enough errors, the number of bits or symbols being simulated must significantly exceed the inverse of the bit error rate of interest. This often results in hours or even days of execution using a personal computer or a workstation.

Reconfigurable devices can perform certain functions faster than general-purpose processors. In addition, they are more flexible than Application Specific Integrated Circuit (ASIC) devices. This fast yet flexible property of reconfigurable devices can be used for the simulation of communication systems. However, although reconfigurable devices are more flexible than ASIC devices, they are often not compatible with each other. Programs are usually written in hardware description languages such as Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). A program written for one device often cannot be used for another device because these devices all have different architectures, and programs are architecture-specific.

Distributed computing, which is not a new concept, refers to interconnecting a number of computing elements, often heterogeneous, to perform a given task. By applying distributed computing, reconfigurable devices and digital signal processors can be connected to form a

distributed reconfigurable simulator.

In this paper, it is shown that using reconfigurable devices can greatly increase the speed of simulation. A simple physical-layer communication system model has been created using a WildForce board, a reconfigurable device, and the performance is compared to a traditional software simulation of the same system. Using the reconfigurable device, the performance was increased by approximately one hundred times. This demonstrates the possibility of using reconfigurable devices for simulation of physical-layer communication systems.

Also, an middleware architecture for distributed reconfigurable simulation is proposed and implemented. Using the middleware, reconfigurable devices and various computing elements can be integrated. The proposed middleware has several components. The *master* works as the server for the system. An *object* is any device that has computing capability. A *resource* is an algorithm or function implemented for a certain object. An object and its resources are connected to the system through an *agent*. This middleware system is tested with three different objects and six resources, and the performance is analyzed. The results shows that it is possible to interconnect various objects to perform a distributed simulation using reconfigurable devices. Possible future research to enhance the architecture is also discussed.

ACKNOWLEDGEMENT

I would like to thank my advisor, Dr. William H. Tranter, for his support throughout my research. His insightful help and guidance has made this work possible. I would also like to thank Dr. Scott F. Midkiff for his help on networking problems, and thoughtful reviews on numerous papers. I would like to thank Dr. Peter M. Athanas for giving me access to his laboratory, and for providing his help on all of the hardware design problems. I would like to thank Dr. Jeffrey H. Reed, and Dr. C. Patrick Koelling for the support, and being in on my committee. I would like to acknowledge fellow students for their support.

I would like to acknowledge the financial support provided by the National Science Foundation (NSF) Integrative Graduate Education and Research Training (IGERT) program and the Defense Advanced Research Projects Administration (DARPA) Global Mobile Information Systems(GloMo) program.

I would like to thank my mother for being an inspiration. Finally, I would like to thank my wife, Sun Ma, for always being there for me. She has given me strength and kept me focused throughout my research years. Without her patience and love, this work would not have been possible.

List of Acronyms

ACS	Adaptive Computing Systems
API	Application Programming Interface
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
CLB	Configurable Logic Block
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCE	Distributed Computing Environment
DOD	Department Of Defense
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HLA	High Level Architecture
HSC	Hardware-Software Codesign
HTML	Hyper-Text Markup Language
IIOP	Internet Inter-Orb Protocol
IP	Internet Protocol

LUT	Lookup Table
MC	Monte Carlo
MPI	Message Passing Interface
MPP	Massively Parallel Processor
OMG	Object Management Group
ONC	Open Network Computing
PLD	Programmable Logic Device
PN	Pseudorandom Noise
PVM	Parallel Virtual Machine
QPSK	Quadri Phase-Shift Keying
RPC	Remote Procedure Call
SLAAC	System Level Application of Adaptive Computing
SNR	Signal-to-Noise Ratio
SPW	Signal Processing Workstation
SRAM	Static Random Access Memory
TCP	Transport Control Protocol
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WWW	World Wide Web
XDR	eXternal Data Representation

Contents

List of Figures

Chapter 1

Introduction

Traditional simulations of communication systems targeted at performance evaluation, such as finding bit error rate, take too much time due to the excessive number of independent Monte Carlo simulations required. Although today's computers, including desktop personal computers (PCs), can run at 2 giga hertz (GHz) or even faster, the simulation does not run at anywhere near that speed, since the platform, whether PC or workstation, is not specifically designed for the simulation of communication systems. For example, even a very simple program with a moderate level of C or C++ code will generate many times more assembly-level instructions. Each of these low-level instructions takes one or more clock cycles, which leads to excessive runtimes. Also, there is no parallel processing within a normal PC or workstation with one central processing unit (CPU).

The idea behind the distributed reconfigurable simulator is to build a platform where the simulation utilizes other means of computing, such as field programmable devices or digital

signal processors (DSPs). A field programmable device is a set of configurable interconnected logic units. Generally, each of these logic blocks contains lookup tables, multiplexers, and flip-flops to perform various combinations of functions. Depending on how it is configured, a logic block can perform different functions such as being a part of a multiplier, accumulator, shift register, etc.

These diverse computing elements can be interconnected to form an integrated simulation platform. A form of specialized middleware is designed to enable the interconnection in a distributed environment. This middleware consists of a *master* and a number of *agents*. The master is a server for the overall system and performs several additional functions. The functions include accepting new agents into the system, processing information for the user interface, mapping and configuring the system for user-specified simulation, etc. An agent is associated with an object (a computing element) and a number of resources (algorithms implemented in the particular object). An agent connects to the master and become part of the distributed reconfigurable simulator.

Chapter 2 covers the background of this research. First, a discussion of simulation of physical-layer communication systems is presented, followed by an overview of reconfigurable technologies. An introduction to related architectures and standards follows. CORBA and Jini are popular middleware architectures. High Level Architecture (HLA) is a method of distributed simulation developed by the Department of Defense mainly for military simulations. Remote Procedure Call (RPC), Grid Computing, Parallel Virtual Machine (PVM), and Message Passing Interface (MPI) are methods for distributed and/or parallel computing. Adaptive Computing Systems Application Programming Interface (ACS/API) can be used for programming multiple reconfigurable devices with a single host program.

As a demonstration of this idea, a simple communication system model is implemented

using a WildForce board. A WildForce board has five Xilinx 4062XL Field Programmable Gate Arrays (FPGA). The feasibility of reconfigurable simulation will be shown using this proof-of-concept implementation. Also, it will be shown that there is a need for middleware in order to interconnect various reconfigurable devices and other processing elements such as DSPs. A detailed description of the implementation is given in Chapter 3.

Using middleware, it is possible to create a distributed reconfigurable simulation platform that can perform a simulation of the physical-layer communication system more efficiently, either by introducing higher fidelity to the simulation or by increasing the speed of the overall simulation. Chapter 4 discusses the details of the middleware for distributed reconfigurable simulation.

Possible future research is discussed in Chapter 5. First, a vision of a future implementation of the distributed reconfigurable simulation platform is presented. Based on this vision, several detailed research issues are presented including web-based user interface, support for multiple users, and real-time exchange of resources.

1.1 Contributions

The general methodology for simulation of physical-layer communication systems has not been changed much. Simulations are often written in high-level programming languages such as C or C++, or are modeled using simulation software such as Matlab or Signal Processing Worksystems (SPW). In this paper, a new direction for the simulation of communication systems is presented. Reconfigurable technologies provide a flexible and fast computing environment. It is shown in this dissertation that it is possible to increase the speed of

simulation of physical-layer communication systems using reconfigurable technology.

There are numerous distributed computing methods for interconnecting various objects. A new middleware is designed and presented in this paper to interconnect reconfigurable devices and other computing elements such as programmable digital signal processors. This middleware architecture has two distinctive features. First, the algorithms or resources are treated separately from the computing elements or objects. This feature introduces advantages such as resource exchange between objects that enables the overall simulation platform to be more flexible. Secondly, the middleware can map the system based on the user's desired simulation configuration. This feature provides abstraction to the user that enables a simplified simulation environment for the user. This dissertation presents a framework of this distributed and reconfigurable simulation environment.

Chapter 2

Background

2.1 Simulation of Communication Systems

With the increasing complexity of communication systems, computer-aided simulation plays an important role in the development of these systems. Since many systems operate in environments in which the channel cannot be described using the simple additive white Gaussian noise (AWGN) model, design and analysis of these systems is difficult. This is especially true for the current wireless personal communication system and, of course, for future generations of systems. [TrK94]

A primary purpose of communication system simulation is to measure the performance of a given system design. Although mathematical analysis of a system would be the ideal solution for measuring or predicting the performance of the system, it is not possible to analyze

the complete system in most cases. This is why simulation is important. The signal-to-noise ratio (SNR) and bit error rate (BER) estimate are the most common metrics for performance measurement of physical-layer communication systems. For analog communication systems, SNR estimation is used. For digital communication systems, BER estimation is the primary performance measure.

For digital communication systems, the BER simulation is often done using the Monte Carlo (MC) method. First, a model of the system under study is developed. The simulation then processes a number of bits or symbols through this model. Obviously, the simulation can only run a finite number of bits or symbols. The question is how long should the simulation run in order to obtain a meaningful result. This is usually determined by using a confidence interval:

$$Pr[P_{\mathsf{a}} < P_{\mathsf{e}} < P_{\mathsf{b}}] > \alpha \text{ for all } P_{\mathsf{e}}$$

This can be interpreted as an indication that the true BER P_{e} is within P_{a} and P_{b} with α confidence. With the MC method, the general rule is that 100 errors will result in the estimation of BER within a factor of 1.3 of the true P_{e}. Although this sounds simple, obtaining 100 errors can require a considerable investment of time and computing power for a complex system with a low BER. For example, simulation of a system with BER of 10^{-7} requires running 10^{9} samples. In some cases, the MC method is not suitable for that reason. In addition, since P_{e} is not known, P_{a} and P_{b} cannot be computed exactly for a given α, but must be estimated.

To avoid the shortcomings of the Monte Carlo method, several alternate methods have been developed. Semi-analytic simulation works well when the system is linear. Importance sampling biases the noise to intentionally generate more errors, hence reducing the overall

execution time for generating the same number of errors, but determining an appropriate biasing function is often difficult. Tail extrapolation methods provide a curve fit to the BER. A short tutorial on the simulation of communication systems can be found in [TrK94]. A more comprehensive and detailed discussion is in [JeB92].

As explained above, the main problem with Monte Carlo simulation is the amount of time it requires to obtain a statistically reliable result. In this paper, a new approach to the standard single-CPU Monte Carlo method is discussed. It will be shown that by using today's field programmable technology, distributed Monte Carlo simulations can run at a much higher speed. This makes the Monte Carlo approach a feasible solution for a complex system with a low bit error rate.

2.2 Reconfigurable Computing

Configurable hardware systems have been around since the 1980s. They feature reconfigurable logic devices such as field programmable logic arrays (FPGA) and programmable logic devices (PLD). An FPGA logic block usually has one or more lookup tables (LUT) and a number of flip-flops to store the data. The major difference is that the PLDs usually implement logic in the sum of product form, which may be inefficient for representing all signals. Also, the FPGAs usually have smaller logic block sizes, which better utilizes the available resource.

Reconfigurable logic devices can be categorized based on their configurability characteristics. First, *configurable logic* is a general term used for devices that can only be programmed once. Second, *reconfigurable logic* refers to devices that can be customized many times.

These devices often use EPROM, EEPROM, or Flash technology. Devices that use SRAM can be categorized as *dynamically reconfigurable logic* devices.

From an architectural point of view, reconfigurable devices can be categorized into three groups. First, *attached processor systems* are connected to other computers, such as workstations and personal computers, through external ports or internal slots such as a parallel port or PCI slot. They usually perform functions that are not handled well by the general processor of the computer. Also, development boards and evaluation boards are in this category. Second, reconfigurable devices can be used as a *coprocessor* [FaW96]. The function of a reconfigurable coprocessor can be changed based on the application need. The third category is *special-purpose machines*, which can often be booted and perform functions without any workstation or personal computer attached to them [Mi98].

Reconfigurable devices feature flexibility of software and high speed based on parallel execution, as in hardware. Figure 2.1 shows the role of reconfigurable devices compared to microprocessors and hardware [Ha01]. This feature of reconfigurable devices can be used for various applications. Rapid prototyping and emulation is a popular application. Also, high speed data handling and control, image and signal processing are some other applications. Field programmable technology has been advancing at a remarkable rate in terms of both speed and the available number of gates, and this trend is expected to continue [Al99]. This ensures that using reconfigurable computing for the simulation of communication systems is a viable option.

Programming a field programmable device usually involves several steps. First, the design must be created. This step can be accomplished through the use of hardware description languages such as Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) or Verlog [ArG93]. A high-level block diagram of a system can be created

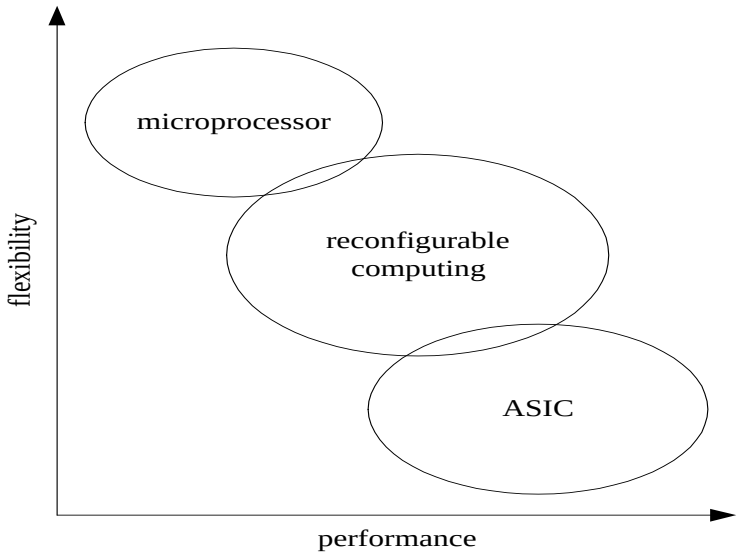


Figure 2.1: Role of reconfigurable devices.

using a graphical program such as the one available in the Signal Processing Worksystem (SPW) [Sp02]. The corresponding hardware description language code can be automatically created. An obvious advantage of this method is the fact that it does not require the knowledge of a hardware description language. A disadvantage is that it may not be as easy to optimize the system since the generated code usually contains extra information which may not be directly related to the system being designed. Once the hardware description language is created, either manually or automatically, the resulting code is then used to create a downloadable bit file. This step involves several operations and different development software depending on the target device.

2.3 Parallel and Distributed Computing

Parallel and distributed computing refers to distributing the execution of a program over multiple computers. Parallel computing usually refers to execution of a program over multiprocessor computers with multiple central processing units (CPUs), with frequent interaction. Distributed computing refers to execution of a program over loosely connected computers with less frequent interaction. The major benefit of parallel and distributed computing is that the execution time can be reduced by executing the program concurrently by subdividing the program [Fu01]. In this section an overview of parallel and distributed computing is presented, with emphasis on parallel and distributed simulation.

2.3.1 Remote Procedure Call (RPC)

When designing a distributed program there are two approaches. First, the programmer can design message format and syntax first, and design the client and server based on the message format designed. This approach is called communication-oriented design. The second approach is application-oriented design. With this approach, the programmer designs the application program as if it was to run on a single computer. Then, this program is divided into pieces to run on multiple computers. Remote procedure call (RPC) uses the application-oriented design. Figure 2.2 illustrates the concept of RPC.

There are many different forms of RPCs. Sun Microsystems has defined *Open Network Computing (ONC) RPC*, which has been widely accepted in the industry [CoS97a]. The Open Software Foundation Distributed Computing Environment (OSF/DCE) platform uses RPC for communication [KhH95]. Microsoft also developed its version of RPC for programming

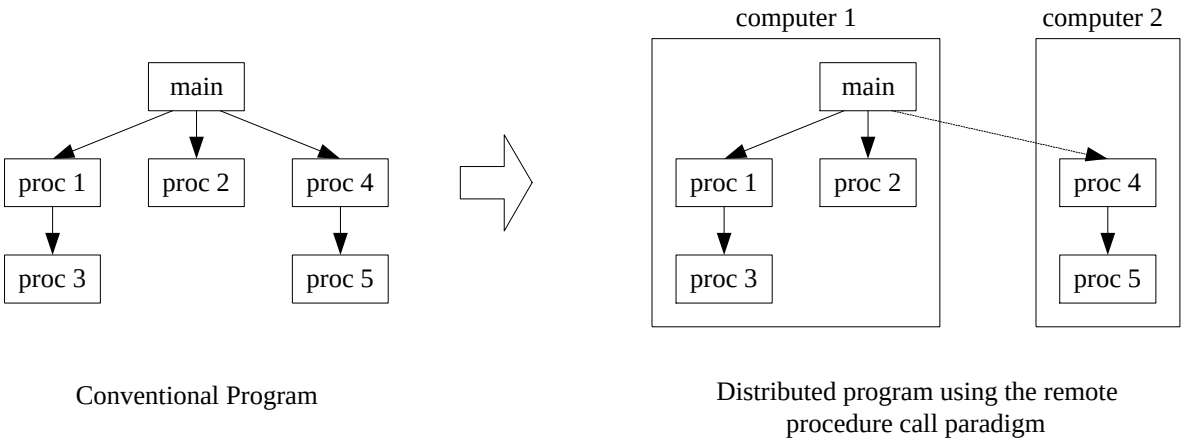


Figure 2.2: Concept of remote procedure call.

that is compatible with DCE RPC, and also offers extensions that are not compatible with DCE RPC [Mi02]. *Rpcgen* is the RPC program generator, which reads a specification file as input and generates C code based on the specification. Using rpcgen simplifies the creation of a distributed application, and enables programmers to avoid unnecessary programming.

2.3.2 Parallel Virtual Machine (PVM) and Message Passing Interface (MPI)

PVM and MPI are both sets of application programming interface (API) functions developed for distributed and parallel computing. They are considered the most popular methods for parallel computer programming.

PVM was developed in late 1980s as a framework for heterogeneous distributed computing

by Vaidy Sunderam and Al Geist. The central of PVM is the *virtual machine*, which is a set of heterogeneous machines connected by a network that appears as a single large parallel machine to the user. PVM uses simple message-passing constructs to exchange data between parallel tasks [GeK96].

MPI was developed in the mid 1990s by a group of high-performance computing experts. MPI was developed as a standard message-passing specification for massively parallel processors (MPPs).

Because PVM was originally designed for heterogeneous distributed computing and MPI was designed mainly for MPPs, they offer different advantages and disadvantages. MPI offers more point-to-point and collective communication options than PVM. PVM is better when applications run on a distributed network of heterogeneous machines, and offers good interoperability between different hosts [FaL00]. A project to merge features of PVM and MPI has been started by the University of Tennessee, Oak Ridge National Laboratory, and the Innovative Computing Laboratory in 1996 [Pv02].

2.3.3 Grid Computing

Grid computing is a relatively new form of distributed computing. The *Grid* refers to the interconnection of numerous computational resources over the internet. Grid computing implies the idea of accessing these resources to solve problems requiring a large number of processing cycles and a huge amount of data. Even though grid computing offers many benefits such as enabling collaboration of distributed teams and maximizing resource utilization, there are concerns like security [Ge02]. NASA's Information Power Grid (IPG) is a great example of grid computing. The objective of IPG is to interconnect computing resources

across multiple NASA sites for applications such as the simulation of space shuttle design [De02a].

More information on grid computing can be found in numerous websites dedicated to grid computing such as [Gr02a] and [Gr02b].

2.3.4 High Level Architecture

High Level Architecture (HLA) is a technical standard for Department of Defense (DOD) simulations [BuJ98]. Unlike middleware such as CORBA or Jini, HLA is specifically designed for distributed simulation. An HLA compliant simulation forms a federation that consists of a federation object model (FOM), a set of federates, and the run-time infrastructure (RTI). A FOM holds the information needed for a particular federation. Federates include simulation models and utilities for simulations. The RTI is the distributed operating system for the federation. The HLA has three main components: rules, interface specifications, and object model templates. The HLA rules are federation and federate rules describing responsibilities of simulations. The HLA interface specification is for federates to communicate with RTI. And, the HLA object model templates are used to create HLA compliant simulations.

2.4 Middleware

The idea of middleware originally comes from distributed computing systems. Because large computer networks such as the Internet are heterogeneous, it is necessary to provide

a method to interconnect various components. Middleware provides such a method [Vi97]. The challenge of middleware is to provide flexibility while maintaining high performance [Vi02]. Following are examples of middleware.

2.4.1 CORBA

Common Object Request Broker Architecture (CORBA) is a widely recognized middleware standard developed by the Object Management Group (OMG). Figure 2.3 shows the architecture of CORBA. The Object Request Broker interconnects various services and applications in the system. Various applications are connected through interfaces built for particular applications. The common facilities provide objects useful for connecting various components. Some objects are useful for a particular domain, such as telecommunications, finance, and healthcare. These objects are defined in the domain interfaces. The object services are objects for performing the basic functions required for interconnecting various components to build distributed systems. An example of an object service is the naming service for providing reference to objects. OMG has a website for CORBA [Co02], and comprehensive introductions to OMG/CORBA can be found in [Vi97], [Em97a], and [Em97b]. Also, there is a real-time CORBA for real-time applications [ScK00] and mobile applications [ChP00].

2.4.2 Jini

Jini™ is a relatively new middleware developed by Sun Microsystems. Jini fully utilizes the inherent platform independence of the Java language. The goal of Jini is to enable a flexible distributed computing environment. Because Jini is based on a specific language, it is not

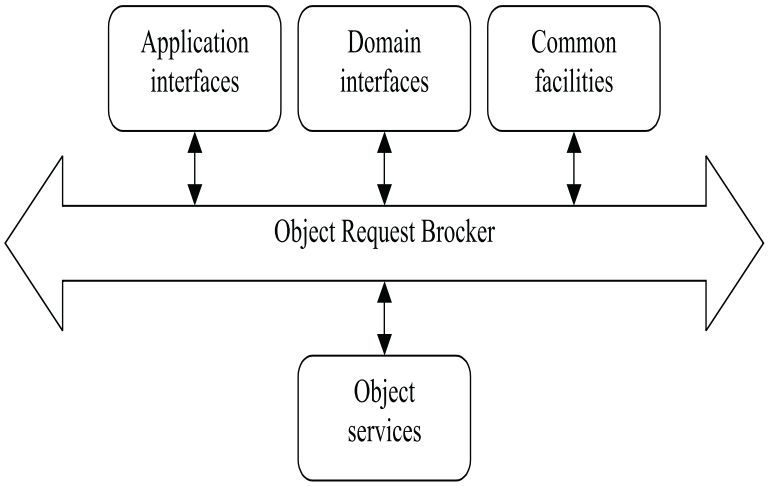


Figure 2.3: Architecture of CORBA

suitable for legacy systems. An overview of Jini can be found in [Ar99] and [Ma00]. Also, Sun Microsystems has a website for Jini [Ji02a].

2.5 Adaptive Computing System (ACS) API

The Adaptive Computing System API, also known as System Level Application of Adaptive Computing (SLAAC) API, has been developed to simplify programming of multiple reconfigurable boards without having to be concerned about the networking aspect of the system [JoS99]. ACS API requires a MPI-based network for inter-node communication. ACS API allows the user to control multiple reconfigurable boards with a single host program through API functions. More information on ACS API can be found in [Ya00]. Also, the API specification and source code of ACS API can be found at the website [Ac02].

2.6 Summary

In this chapter, the background of the research presented in this paper was discussed. First, an overview of the simulation of physical-layer communication systems was presented. Because of the nature of the Monte Carlo simulation, the simulation of physical-layer communication systems often requires lengthy runtimes. Second, the reconfigurable technology was discussed. The reconfigurable devices can be more flexible than ASIC devices, and faster than microprocessors. Finally, various distributed computing architectures and methods were presented. In Chapter 3, it will be shown that the reconfigurable technology can enhance the simulation of physical-layer communication systems. In Chapter 4, it will be shown that the reconfigurable technology can be integrated with other computing elements such as digital signal processors and general-purpose microprocessors by interconnecting them in a distributed environment.

Chapter 3

Design and Implementation of a Reconfigurable Simulator

In Chapter 2, it was discussed that reconfigurable technology can be faster than microprocessors. In this chapter, the use of reconfigurable technology for simulation of physical-layer communication systems is discussed. First, the design process of the reconfigurable simulation is presented. Because development of a reconfigurable simulation platform requires hardware programming and software programming, the overall design process is relatively complicated compared to traditional software simulations. Following the design process overview is a detailed discussion of the implementation. The reconfigurable simulator is implemented in a WildForce board. Finally, the performance of the reconfigurable simulator is shown, and the summary follows.

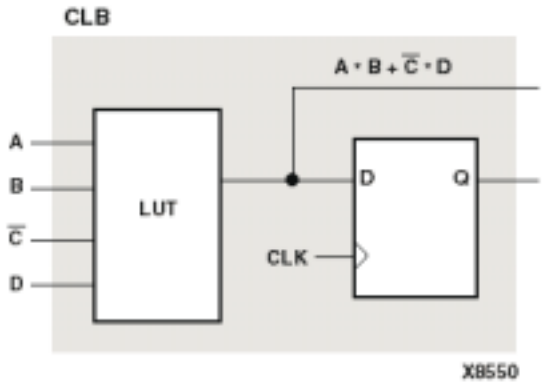


Figure 3.1: Configuration of a CLB.

The reconfigurable simulator is implemented using a development board based on Xilinx FPGAs. Xilinx is one of the major FPGA manufacturers. A Xilinx FPGA has base logic blocks called Configurable Logic Blocks (CLBs) [Xi02]. Figure 3.1 shows the configuration of a typical CLB. A set of CLBs is arranged to form an array of interconnected CLBs. The resulting array is connected to a set of input/output (I/O) blocks to enable input and output operations.

The core of the distributed reconfigurable simulation is the use of reconfigurable computing hardware. Since certain algorithms execute much faster when implemented using a Field Programmable Gate Arrays (FPGA) rather than a general purpose processor, the overall performance can be greatly enhanced. This chapter discusses a reconfigurable simulator that has been developed as a proof of concept model. The detailed design process and problems are discussed.

3.1 Design Process

Compared to the traditional software simulator, the design process of a hardware based reconfigurable simulator involves a complicated decision-making process. To make the simulator independent of the platform, any architecture-specific resource should not be used. Accomplishing this is a difficult task. For example, a lookup table uses external memory, which is platform dependent, and prevents the creation of a simulator that is platform independent. Also, because it is hardware, traditional algorithms designed for software simulations are not always applicable. This section discusses the design process in detail.

The development process starts with design specifications. Once the design is made, the source code can be developed and debugged. Two separate sets of source code are needed. One set is the hardware code for the actual hardware description, usually written in a hardware description language such as VHDL or Verlog, and the other set is for the host program, usually written in a high-level language such as C or C++. The host program is needed for the user control of the hardware residing inside the FPGAs. Figure 3.2 shows the development process in terms of the tools used and the associated file extensions. The details of the process can be different depending on the target platform. This particular diagram illustrates the design process using a WildForce FPGA Board as a target board, and VHDL as the development language. A brief summary of each block is given below.

3.1.1 WorkView Office

WorkView Office is used for simulation of the VHDL code [Wo02]. By analyzing the source code, errors can be identified. This step is similar to the debugging process of the general

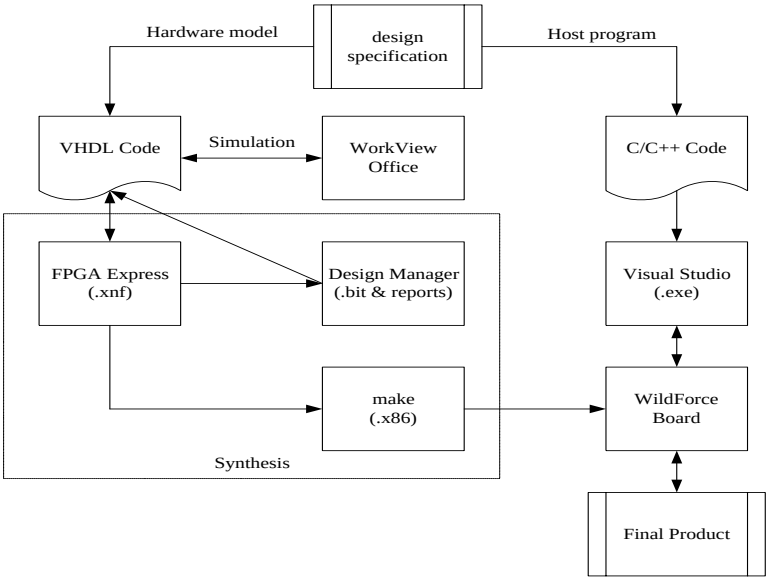


Figure 3.2: Development process of a reconfigurable hardware application uUsing a WildForce board.

software development process. Once the code is analyzed, it can be simulated by using SpeedWave, the simulation platform of the WorkView Office package. By using the timing diagram, the behavior of the hardware can be evaluated, which supports the debugging process.

Although the simulation can serve as a debugging tool, the VHDL code that can be analyzed is not necessarily synthesizable. This is due to the nature of the VHDL language, which was initially was developed as a description language, and later used for synthesis [ArG93]. Therefore, there are certain commands working within a simulation that will result in an error if synthesis is attempted. This means that even if a code is correctly functioning within the simulation environment, it may not pass the first stage of the synthesis process due to grammatical and structural differences between the analyzable code and the synthesizable code. For example, if a user writes

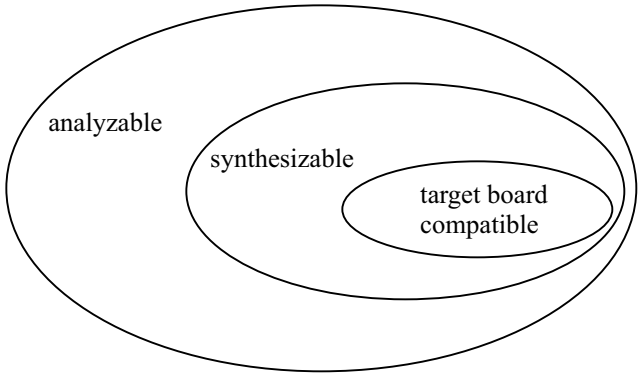


Figure 3.3: Source code limitations.

```
A <= '0' after 10 ns;
```

as a way of setting the value of A to 0, simulation tools such as WorkView Office will recognize this as a valid command. However, because *after 10 ns* cannot be synthesized, this line of code will result in an error if synthesis is attempted.

Since each development board has a different structure, such as different I/O port specifications and a different number of bits for address lines, there are certain sets of variables that must be used for each board which have no meaning to other boards. Therefore, it is important to write the code with these limitations in mind. The diagram in Figure 3.3 illustrates this concept.

3.1.2 FPGA Express

FPGA Express is the first step of the synthesis using the VHDL code as the input file [Sy02a]. The output file of FPGA Express is a netlist, which has an .xnf extension. This is where the

target FPGA is decided.

A similar program called Synplify [Sy02b] is used instead of the FPGA Express for other development boards such as the SLAAC board. If the target development board includes support for a specific program, then it is obviously better to use that program.

3.1.3 Design Manager

Design Manager can be used to perform the *place and route* operation required to create the downloadable bit file. Or, it can be used to generate the quick report which gives a very good timing performance measurement. By examining the reports after finishing the process, a relatively accurate timing performance such as maximum delay can be examined. Also, the space requirement can be obtained at this step.

3.1.4 Make

Although the Design Manager generates the .bit file, which is downloadable to the actual FPGAs, the WildForce requires the .x86 format. This is why a separate process of "make" is needed. This process executes a series of programs to generate the final format of this rather long sequence of the development process, the .x86 file. This file can be downloaded into the FPGAs in the WildForce by calling the appropriate subroutine from the host program.

3.1.5 Visual Studio

Microsoft's Visual Studio is a widely used high-level language development platform. Here, C or C++ code written as the host program can be compiled. The output will be a .exe file. By running this host program, the .x86 file will be loaded into the FPGAs.

There are a number of built-in functions for the host programs. These functions are needed to properly interact with the actual hardware. Because these built-in functions are not general C or C++ functions, some experience is required before writing a reasonable host program. Also, the fact that each development board has its own set of functions makes development even harder.

3.1.6 Hardware-Software Codesign

The development process presented above is similar to that of the hardware-software codesign (HSC) development process. HSC is used for creating embedded systems with shorter development time, along with better integration of hardware and software components. However, the above development process is not targeted toward dividing tasks over hardware and software. The software side is only for developing a host program, which is necessary to interact with the hardware. Any design or model resides in the hardware, not the software. This differentiates the above development process from HSC. A comprehensive overview on HSC can be found in [Gu02], [PuF92], and [ScR98].

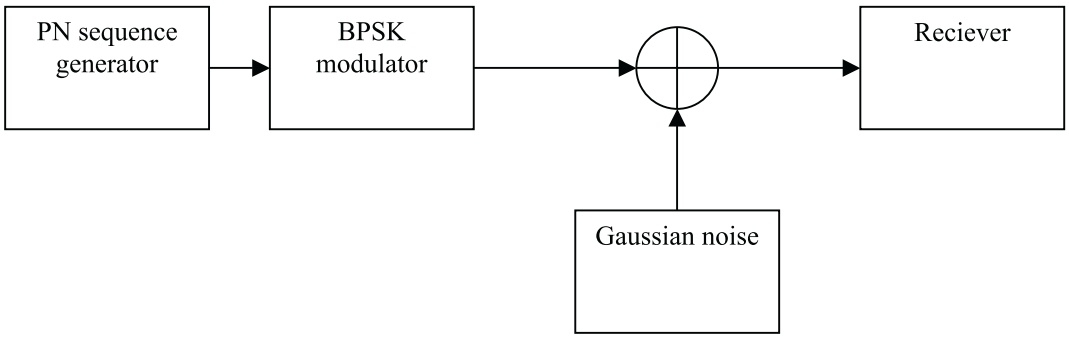


Figure 3.4: Block diagram of a simple communication system implemented in hardware.

3.2 Implementation of a Proof-of-Concept Model

3.2.1 Overview of the System

As an example to show the advantage of using a reconfigurable computing device for a simulation of communication systems, a simple communication system is implemented. Figure 3.4 illustrates the system.

The input data is generated using a pseudorandom noise (PN) sequence generator. This data sequence is then modulated using binary phase shift keying (BPSK) to represent the transmitted signal. The received signal is the sum of the transmitted signal and Gaussian noise. This signal is then demodulated and compared to the transmitted signal to calculate the bit error rate (BER). The following sections discuss the implementation of each function in detail.

31(sign)	30-16 (integer part)	15-0 (fraction part)

Figure 3.5: Structure of the number representation.

3.2.2 Number System

Simulations of communication systems involve various parameters with different ranges of values. A traditional software simulation uses a CPU and its built-in number system, which is usually an IEEE-standard floating-point representation. However, this built-in representation does not exist for a reconfigurable simulator.

Because of the unavailability of any sort of built-in number system for handling real numbers within FPGAs or PLDs, a number system had to be built specifically for this project. Traditional floating-point operations tend to require more space and execute slower than fixed operations. This complicates the design and implementation of the system. Therefore, a fixed-point representation is used to represent numbers.

A WildForce board was the first platform on which the reconfigurable simulator was realized. Because the external memory attached to the FPGAs in WildForce has a word length of 32 bits, a 32-bit fixed-point representation is a reasonable choice for the number system, where the first bit is the sign bit, the next 15 bits is the integer part, and the last 16 bits is the fractional part of the number. Two's complement representation is used because the IEEE standard implementation in VHDL uses two's complement arithmetic. This format is shown in Figure 3.5.

With this system, the range of numbers that can be represented is:

Decimal	32 bit representation
1	0000 0000 0000 0001 0000 0000 0000 0000
.5	0000 0000 0000 0000 1000 0000 0000 0000
2^{-16}	0000 0000 0000 0000 0000 0000 0000 0001
	0000 0000 0000 0011 0010 0100 0011 1111

Figure 3.6: Examples of number representation.

$$-2^{15} \text{ to } 2^{15} - 2^{-16}$$

The smallest magnitude that can be represented is 2^{-16}. Some examples of numbers represented in this number system are shown in Figure 3.6.

Although the above number system is the primary method for representing numbers in this system, it certainly does not restrict any other representation as needed in another part of the system. An example of this exception is the uniform random variable generator. Since the range of a uniform random variable is generally 0 to 1, there is no need for any larger magnitude or sign bit. So, in this case, the whole 32-bit structure represents a fraction between 0 and 1, and the smallest number is 2^{-32} not 2^{-16}.

3.2.3 Implementation of Each Functional Block

Most of the functional blocks in the reconfigurable simulator are implemented as portable components to prevent any platform dependency. However, because it is specifically designed for each board, the interconnection among FPGAs or PLDs is not portable. With this exception, which has to be rewritten as needed, the whole design can be implemented on any platform.

Some of the functional blocks, such as the PN sequence generator, are fairly easy to implement since the structure of the function does not require extensive mathematical operations. However, there are several blocks that do require more complicated structures, such as a Gaussian noise generator based on the Box-Muller algorithm [JeB92].

PN Sequence Generator

The PN sequence generator is a straightforward module that has a simple hardware implementation. Utilizing a simple shift-register with primitive polynomial as the basis of feedback, a PN sequence with the maximal length of 2^{n-1} can be obtained with a shift-register having 2^n stages. Figure 3.7 is the block diagram of the PN sequence generator. Here, `reg_length` is a 4-bit selector that can be used to select the length of the shift register, and, therefore, the maximal length of the sequence. The `init_state` is a 16-bit input which is used to provide the initial state of the shift register. Although it is a 16-bit input, it may need just some of the bits depending on the `reg_length`. Once enabled and reset, this module will reconnect the internal lines to form a primitive polynomial with the order given in the `reg_length` line, and it will start generating a PN sequence.

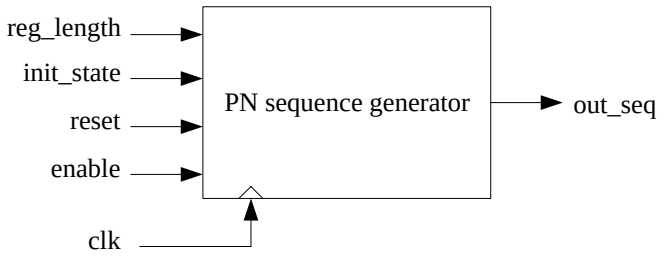


Figure 3.7: Block diagram of the PN sequence generator.

Gaussian Noise Generator

There are a number of ways to generate a normal distribution [PaC96]. However, when it is to be implemented using reconfigurable hardware, there are only a couple of viable choices. This is because of the indefinite loops within the algorithms. The backbone of the reconfigurable simulator is that several functions can be performed at the same time. This is achieved using a master clock that synchronizes the input and output of each function. Therefore, if there are any indefinite loops, such as a while loop in the C language, the overall system cannot be synchronized. Among all available algorithms there are two popular choices without the indefinite loops. These are based on the Box-Muller algorithm and the Central Limit Theorem.

The Box-Muller algorithm [JeB92] states that if U_1 and U_2 are uniformly distributed random variables between 0 and 1,

$$X = \sqrt{-2 \ln U_1} \cos 2\pi U_2 \tag{3.1}$$

$$Y = \sqrt{-2 \ln U_1} \sin 2\pi U_2 \tag{3.2}$$

gives two zero-mean unit variance Gaussian random variables X and Y. The advantage of the Box-Muller algorithm is that it is computationally simple and models the tails well.

The Central Limit Theorem is another way of generating a Gaussian random variable. Let X_i be uniformly distributed on [0, 1] where

$$X = \frac{B}{N}\sum_{i=1}^{N}(X_i - \frac{1}{2}) \tag{3.3}$$

The Central Limit Theorem says that X will tend towards a Gaussian distribution with mean 0 and variance $\frac{B^2}{12N}$ as N gets large. Although it is simple to implement N independent random variable generators, the disadvantage is that the tails of distribution are not Gaussian but are truncated for finite N.

When a communication system is simulated, the tail of a Gaussian distribution has more importance than the rest of the distribution. Therefore, it is necessary to use an algorithm, which models the tail of the distribution with a high level of fidelity. This obviously makes the Box-Muller algorithm a viable candidate.

For the implementation of the Box-Muller algorithm, sine, cosine, square root, and natural log function along with generating U1 and U2 are operations that can be done easily in general-purpose software by calling built-in mathematical library functions. However, these functions have to be implemented one-by-one using hardware.

Algorithms should not include any division except what can be done with binary shift operations, because the hardware implementation of division often requires too much resource. Therefore, implementing the functions using algorithms is not an easy task. For example,

the sine and cosine function can be approximated using the Taylor Series. However, the Taylor Series includes division operations other than binary shifting, which may not be best for hardware implementation.

For the lookup table method, the problem is the size of the lookup table and the placement of it. Because the number of available gates are limited, it is not desirable to have large tables inside of the FPGA or PLD. The external memory can be a good choice for the lookup table. However, the system might need to read two or more entries simultaneously. For the Box-Muller algorithm, sine, cosine, logarithm, and square root values have to be read to generate a single random variable. If each of these functions has to get the value from the same memory, it will take four clock cycles to generate a single Gaussian random variable, while the PN sequence can be generated at every clock cycle. This is certainly not a desirable structure.

To avoid having too many lookup tables, functions are not implemented separately. Instead, the $\sqrt{-2\ln()}$ is implemented as a single function. Also, the sin() and cos() functions are implemented in one lookup table by splitting the available 32-bit word into two 16-bit words. Figure 3.8 shows the block diagram of the Gaussian noise generator. This module has reset and enable lines which can be used to control the output of the noise generator. If the enable line is set, two Gaussian random variables will be generated every clock cycle, and will be available at `X_out` and `Y_out`.

Although the Central Limit Theorem does not provide a good model for the tail of a Gaussian distribution, the effect of truncation can be minimized by increasing the number of constituent random variables sufficiently. Figure 3.9 demonstrates this concept. As is clearly seen, increasing the number of random variables can greatly enhance the quality of the resulting Gaussian approximation. The main reason for not using the Central Limit

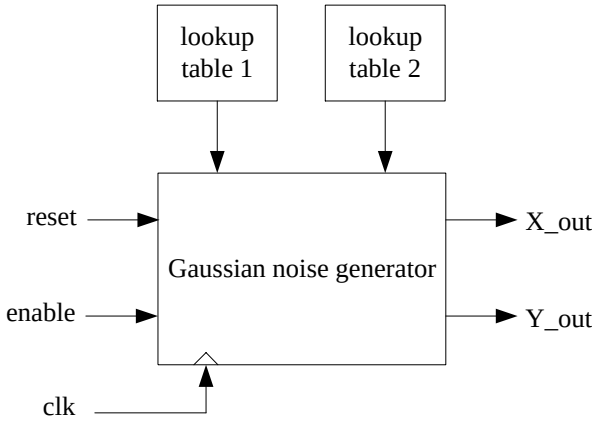


Figure 3.8: Block diagram of the Gaussian noise generator.

Theorem to generate a Gaussian distribution is the excessive number of independent random variables needed. More random variables means more computing time in general-purpose processors, and this results in slower simulation overall. This problem does not exist in reconfigurable computing, since all of the random variables can be generated simultaneously. The number of random variables is only limited by the available hardware space. Since the size of reconfigurable hardware keeps increasing rapidly, implementing a Gaussian noise generator using the Central Limit Theorem can be a viable solution.

In Figure 3.10, several different distributions using the Central Limit Theorem with various numbers of random variables are shown. Each distribution is normalized to have unit variance. The performance of each distribution is compared with the analytical result by comparing the value of the Q-function for the range of 0 to 10. The Q-function is defined by

$$Q(x) = \int_{x}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} dy \tag{3.4}$$

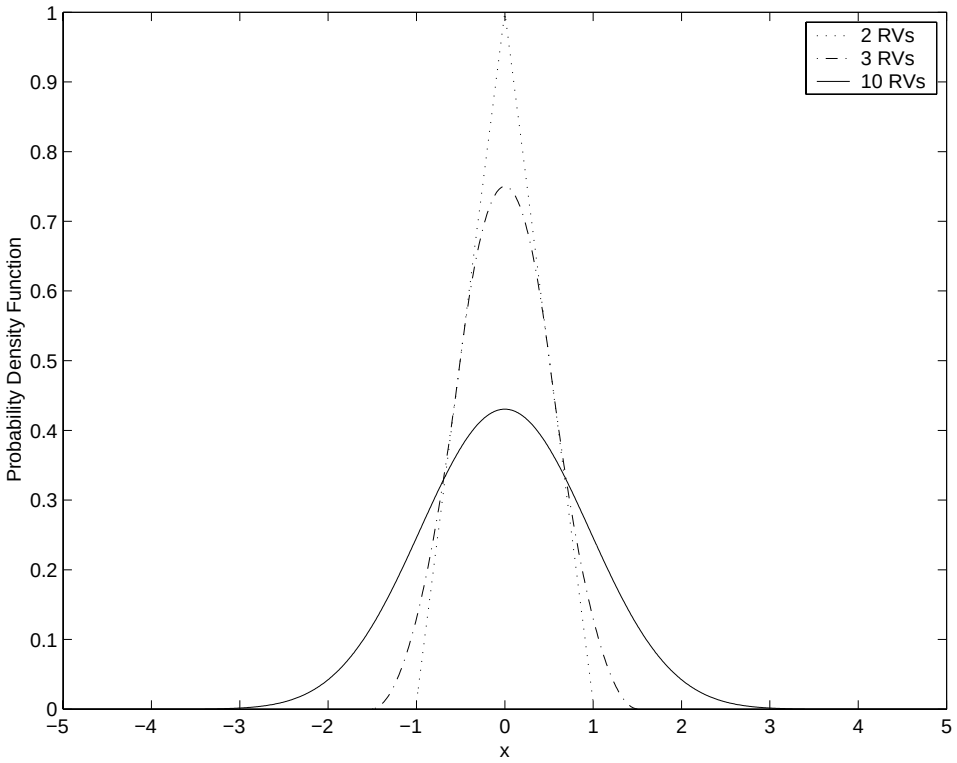


Figure 3.9: Generating a Gaussian distribution using the Central Limit Theorem.

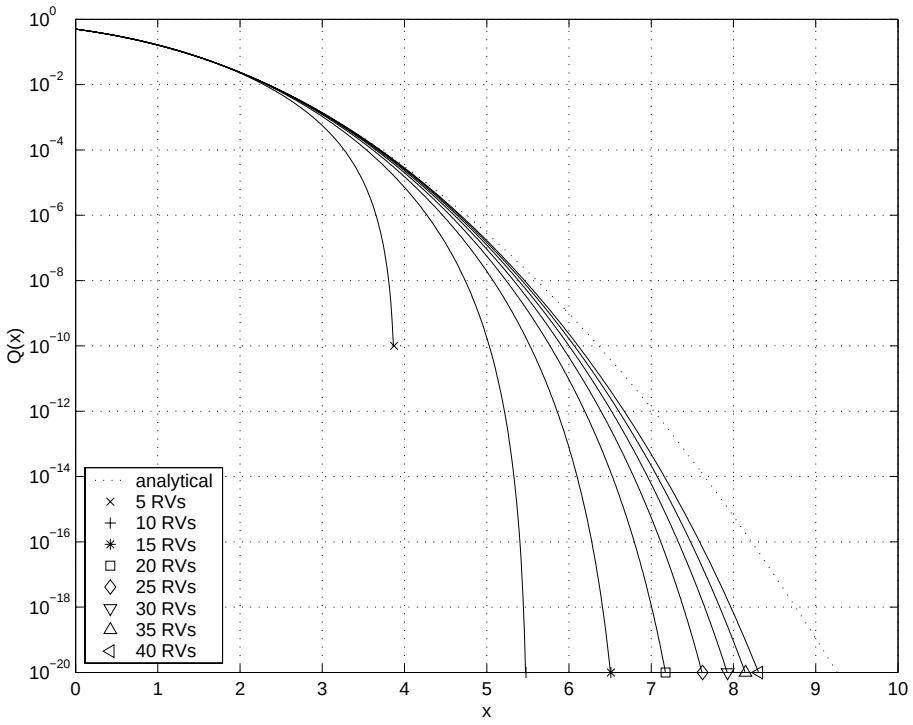


Figure 3.10: Estimation of Q-functions using the CLT with various number of random variables (RVs).

As is clearly seen, more random variables result in better performance. For example, with 5 random variables the resulting distribution can only approximate the Q-function relatively accurately up to somewhere around Q(3) or 10^{-3}, where the distribution with 40 random variables can approximate the Q-function up to around Q(5) or Q(6) before it diverges. This result shows that the performance of the Central Limit Theorem and the resulting Gaussian distribution can be improved as the number of random variables increases. The exact number of random variables to be used can be determined by the desired level of accuracy, tolerance, and the available hardware resources.

Modulator

With all the available variations of modulation schemes, creating a universal model to simulate the modulation is an impossible task. For this example, a Binary Phase Shift Keying (BPSK) system is implemented.

For BPSK, the modulated signal *s(t)* is defined as

$$s(t) = \begin{cases} A\cos(2\pi f_c t + \theta) & \text{if source data} = 1 \\ A\cos(2\pi f_c t + \pi + \theta) & \text{if source data} = 0 \end{cases} \tag{3.5}$$

where A is the amplitude, f_c is carrier frequency, and θ is the phase offset, which can be zero.

In Figure 3.11, the `mod_sel` line is a 2-bit selector with two reserved selections for any future implementation of different modulation schemes. If BPSK is selected using the selector

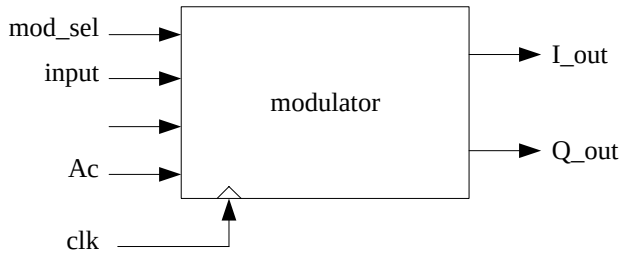


Figure 3.11: Block diagram of the modulator.

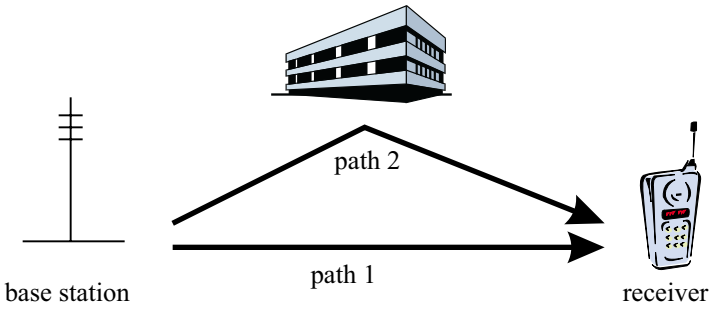


Figure 3.12: Multipath noise model.

bits, then the output will be generated each clock cycle. `I_out` and `Q_out` represent the I-channel and Q-channel data, respectively.

Multipath Noise Generator

Modeling of the multipath channel noise is important when simulating a wireless communication system. Without a confined channel such as a telephone line or fiber optic cable, the multipath noise is unavoidable. Figure 3.12 illustrates a two-ray multipath channel. Here, path 1 and path 2 obviously take different times to reach the receiver. Also, the power of

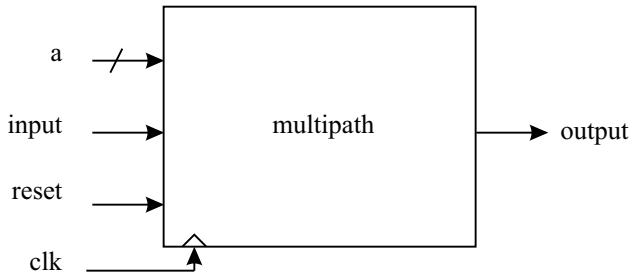


Figure 3.13: Multipath noise generator.

each signal may be different. Then the received signal can be expressed as

$$y(t) = a_1 x(t - \tau_1) + a_2 x(t - \tau_2) \tag{3.6}$$

For a *n-ray* multipath channel, the general expression for the received signal is

$$y(t) = \sum_{n=1}^{N} a_n x(t - \tau_n) \tag{3.7}$$

This model is implemented as in Figure 3.13. Internally this block has a 20 stage shift register, where each stage represents a bit or chip depending on the system. The input line a is a 20 words coefficients line. Once the reset is active, the coefficients are associated with the shift register to form a multi-ray multipath channel noise. Because the basic operation required for generating multipath noise is multiplication, the number of stages has to be carefully chosen to make sure that the required system gates do not exceed the number of available gates.

3.2.4 Implementation in the WildForce Architecture

Architecture of the WildForce board

The platform of the system implementation is a WildForce board, which has a set of FPGAs with additional functionality. A WildForce board is an extension card with a PCI interface. It has five Xilinx 4062XL FPGAs, and each of these 4062XL FPGAs has 2,304 Configurable Logic Blocks (CLB), which are the main building blocks for the system. And, there is an external memory with size of 256K 32-bit words associated with each of the FPGAs. A host program can be written in C or C++ to enable interaction between the host PC and the hardware. Some of the main functions of the host program include the configuration of the FPGAs by downloading the bit images, and memory read/write. The five FPGAs are connected in a 36-bit systolic bus, with some additional links having smaller width and special purposes such as 2-bit wide handshake bus. In Figure 3.14, a simplified diagram of a WildForce board is shown. The FPGAs are called processing elements (PEs), and are numbered from 0 to 4. The first PE functions as a control unit with some special bus, and is, therefore, called the control processing element, or CPE0. There are more units that are not in this simplified diagram, including a crossbar that can be used to broadcast a signal to all processing elements, or to interconnect any two particular PEs to serve as a separate bus. This is how the I-channel and Q-channel can be realized. The I-channel is represented using the systolic bus, and the Q-channel can be represented using the crossbar as needed.

The following is an example of an unavoidable dependency. In Figure 3.14, interconnecting signals have unique names. For example, the 36-bit buses between two PEs are referred as `PE_Left` or `PE_Right` depending on which way the data is going. The 2-bit bus is called `CPE_PEX_Bus` or `PEX_CPE_Bus`, where X is the PE number. Also, each PE has memory

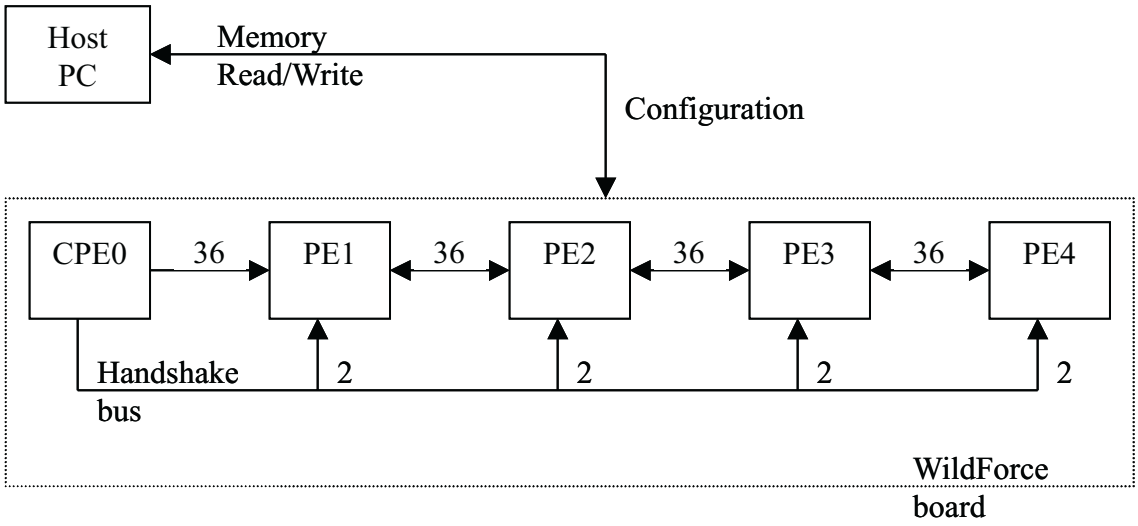


Figure 3.14: Simplified block diagram of the WildForce board.

attached to it using 22-bit address lines.

Here, no matter how well-structured the internal modules are, they must be interconnected using the given bus structure of the WildForce board. Also, the memory being used for the lookup tables have different sizes depending on the platform, which also results in a different number of address lines.

Implementation

Because the data flow within the processing elements is limited by the width of the bus and the number of different buses available, it is important to design the system so that the data can move without any critical bottlenecks. Since each data word is 32 bits long as described earlier, only one value can be passed through the 36-bit wide systolic bus at any given clock cycle. With this in mind, the BPSK system being simulated is implemented as

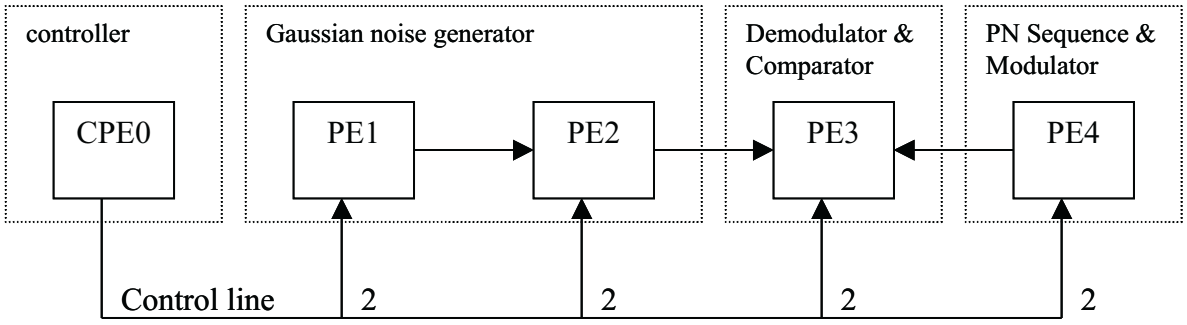


Figure 3.15: Implementation of the BPSK system in the WildForce board.

in Figure 3.15.

The CPE0 receives information from the host PC, such as how many samples to run, and using this information, it controls other PEs using the control line as an on-off switch. Here, the Gaussian random variable generator takes up two processing elements. This is for the lookup table purpose. Since there are more than one lookup tables to be read as discussed earlier, two separate memory locations will be used to enable simultaneous memory read for two different lookup tables. PE4 acts as a transmitter, while PE3 performs the receiver functions.

3.2.5 Results

Since the main purpose of building a hardware level simulator is to increase the speed of the simulation, it is critical to compare the speed of the reconfigurable simulator to that of a traditional software simulator. Each data point in Figure 3.16 is generated using enough independent Monte Carlo simulations to produce 100 errors. Of course, the number of samples can be changed easily by providing the desired number of samples from the host

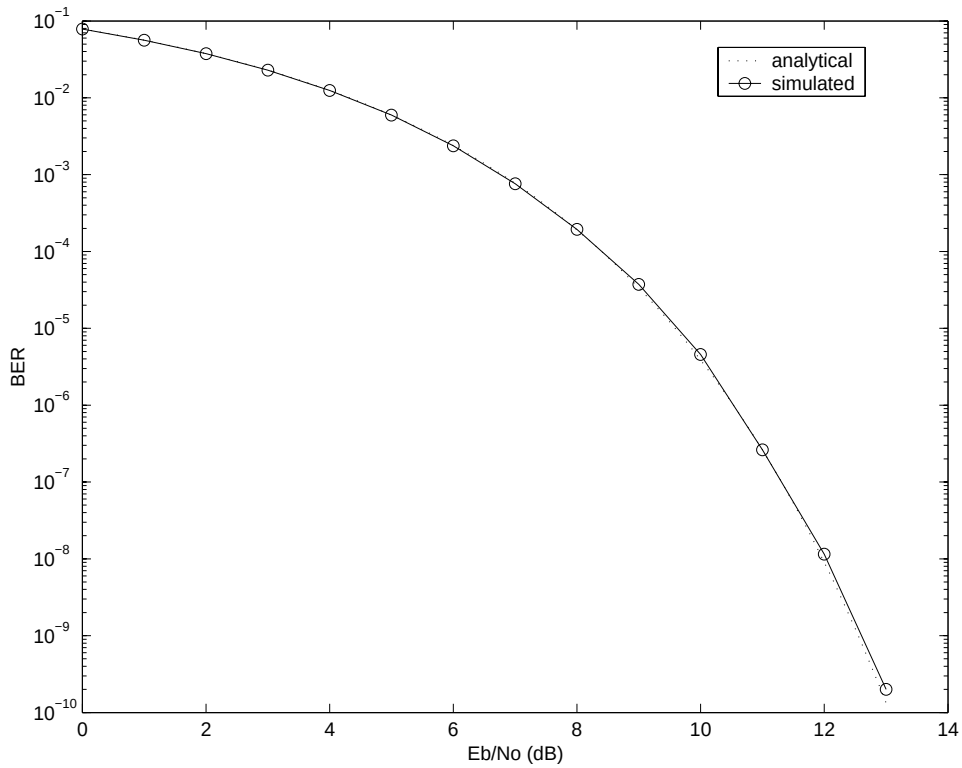


Figure 3.16: Performance of the hardware-based simulator.

program. The entire simulation runs at 5 MHz, or 5 million samples per second. In other words, to generate data points with 1 billion independently generated samples, it takes less than four minutes. There is a fixed overhead such as download time for the bit files and a few clock cycles of propagation at the beginning. However, compared to the actual simulation time, this fixed overhead is negligible. A compatible model has been developed entirely in software using C. When it was tested, the time required to perform the same task was about 100 times longer. As is clearly seen in Figure 3.16, the accuracy of the simulation is also remarkable. This simple example proves that the reconfigurable simulation can be fast and accurate, if implemented correctly.

3.2.6 Platform Dependency

With some exceptions, traditional software simulation programs can be easily ported to different types of platforms. For example, a C or C++ program that was originally written on a PC can be compiled and run on a UNIX workstation without much difficulty, if standard library routines are used. Unlike software programs, a hardware-based simulator is platform dependent. Each and every platform has its own unique interconnection schemes including the number of connections, width of the bus, timing, and attached peripherals. Therefore, to have portability, or to change the platform of a hardware-based simulator easily, the whole system should depend on the platform architecture as little as possible.

One of the biggest reasons for changing the platform of implementation is the evolution to better and faster devices. The core of the implementation is an FPGA or a PLD, and the performance of these programmable devices and the development boards using these devices gets better and better with time. Therefore, it is desirable to have a system that can be ported to a different platform to prevent the system from rapidly becoming obsolete.

Although most of the system can be implemented to be independent of the target platform, there are some unavoidable dependencies. One such example is the lookup table which utilizes the memory, such as the Gaussian noise generator discussed earlier.

To avoid using platform dependent external memory, the first attempt was to implement the lookup table within the FPGA. By doing so, the memory dependency can be removed. However, the number of available gates is limited and, therefore, the size of the lookup table is limited. Each of the five FPGAs in a WildForce Board has 2,304 CLBs, where each CLB has 2 flip-flops. This means 4608 bits of storage area or 144 32-bit words are available for a lookup table.

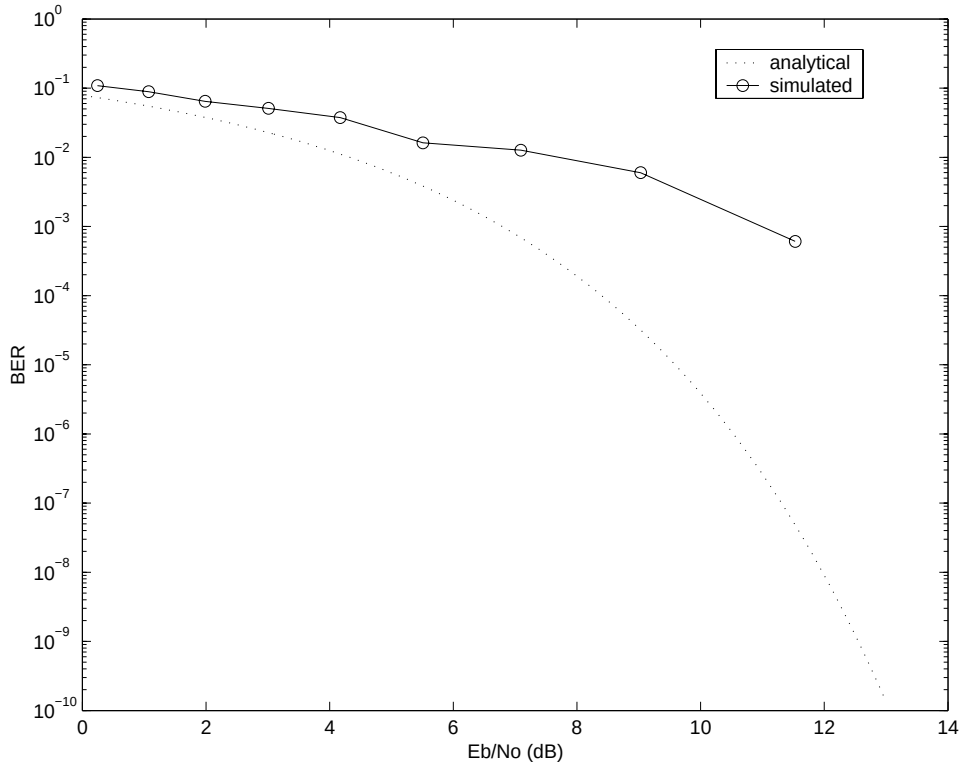


Figure 3.17: Performance of the simulator with LUT within the FPGA.

Figure 3.17 shows the result of running the BER simulation with the lookup table inside the FPGAs. Here, the resulting curve has the general shape that indicates the relationship between the BER and Eb/No. However, compared to the analytic BER curve, or the result in Figure 3.16 with 128K lookup table, we see that it is far from accurate. This is due to the inaccurate Gaussian noise generated by small lookup tables. This illustrates the platform dependency of implementing a simulation in reconfigurable hardware.

3.3 Summary

As shown above, reconfigurable computing offers a great possibility for enhancing the speed of simulations of physical-layer communication systems. By creating the desired system to be simulated using hardware description languages, the performance of the overall simulation can be improved.

One problem with using reconfigurable computing for the simulation of communication systems is that the designs are often device-specific, because virtually all reconfigurable devices have different architectures. This incompatibility may prevent the utilization of reconfigurable technology for simulation of communication systems, because porting a design from one platform to another may require excessive development time compared to traditional software programs. There are two solutions to this problem. First is to create device-independent designs. However, as shown in the previous section, platform dependency is sometimes unavoidable. The second solution is to create a method of connecting different types of devices. Chapter 4 will discuss this solution.

Chapter 4

Middleware for Distributed Reconfigurable Simulation

4.1 Overview

As discussed in the previous chapter, a reconfigurable hardware-based simulation can be useful for many applications. However, a systematic method for interconnecting various computational components to form a complex simulator is needed. The concept of middleware can be applied to define a method for accomplishing this. By using a systematic simulation architecture, a number of different reconfigurable components and the implemented algorithms can be connected easily.

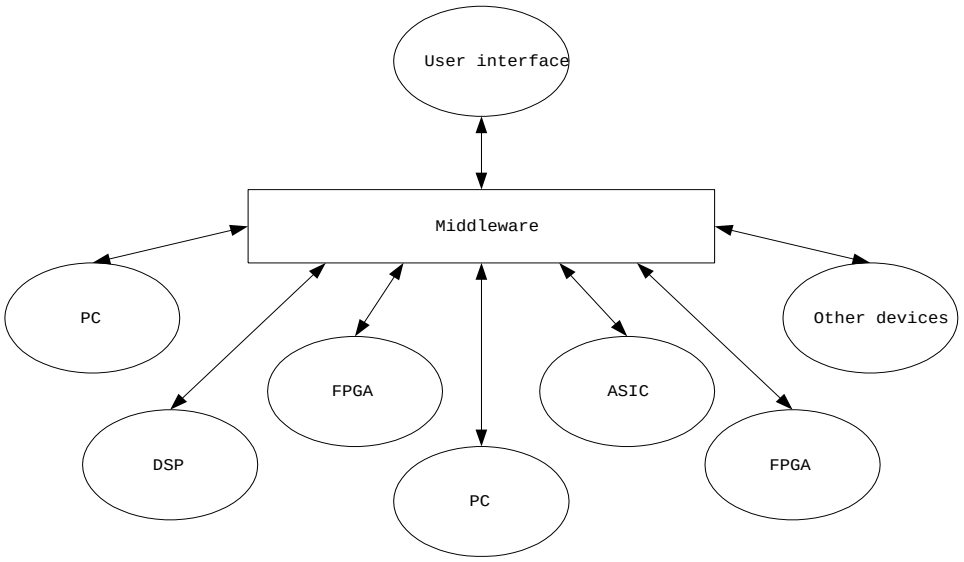


Figure 4.1: Middleware and interconnection.

As shown in Figure 4.1, the middleware interacts with the simulation. The user-defined simulation is interpreted by the middleware and the middleware determines which processing elements are best suited for realizing the simulation. Also, the user can test an actual hardware prototype by connecting it with the rest of the simulation. The middleware may control several machines including PCs, workstations, FPGAs, and programmable DSP platforms through Transport Control Protocol/Internet Protocol (TCP/IP).

A variety of reasons exist for distributing the computational burden associated with a simulation across a number of components as shown in Figure 4.1. The two most important reasons, however, are to reduce the time required to execute a simulation and to provide for "hardware in the loop" simulations so that the hardware can be tested and validated. As shown in Figure 4.1, the middleware provides the communication link between these various components. The middleware must be developed so that the overall computational burden is not significantly increased by the middleware. In addition, the middleware must be as simple as possible. In addition, it must be easily expandable.

4.1.1 Low Computational Burden

Various processing resources are incorporated to increase the simulation speed. Since Monte Carlo is the most common simulation methodology to be targeted, very long simulation runtimes are frequently encountered. A distributed simulation may reduce the time required to execute a simulation by allowing one to develop a simulation architecture closely tied to the system being simulated. This allows the use of parallel processing, fast computational engines where appropriate (e.g. FPGAs), and hardware in the loop. Speed should also be a top priority when designing middleware, so that the increased computational burden associated with the middleware does not significantly add to the overall time required to execute the simulation. This implies that the hardware bit files must be pre-compiled, since the generation of bit files can take several hours. It is also desirable to minimize the interaction between processing elements through TCP/IP. If each bit or symbol is processed and passed to another processing element, this excessive number of transmissions over TCP/IP may become a computational bottleneck in the system if the transmission time is longer than the processing time of each bit or symbol. This will cause the whole simulation to be block or packet based instead of bit or symbol based. This potential bottleneck can be avoided by grouping a number of bits or symbols into a block and transporting them as a single entity between various processing elements.

4.1.2 Simplicity

The main reason for providing this middleware is to enable a simple and systematic interconnection of the resources available for executing the simulation program. The user interface should be as simple as possible.

4.1.3 Expandability

Since new communication systems and algorithms are constantly being developed, it is important to develop simulation models for newer systems and integrate these with existing models. Incorporating newer processing elements such as faster reconfigurable devices must be possible, also. The whole distributed reconfigurable simulator can quickly become obsolete unless it can be expanded to meet future requirements. One must be able to accomplish these tasks with ease.

4.1.4 Scalability

User-desired systems may require a different number of computing elements to perform the simulation. In order to accommodate different simulation needs, the system must be scalable. It must be possible to connect additional computing elements with ease.

4.2 Architecture of the Middleware

Figure 4.2 shows a detailed architecture of the middleware for the distributed simulation platform. The core of this architecture is the *middleware master*. The master provides the interface for the user simulation. When the user creates a simulation model, it is interpreted by the master, and is mapped to appropriate available *objects*. Each object has a *middleware agent* attached to it, that provides an identical access method for various objects. Depending on the type of object, there may be *resources* associated with the object. For example, an

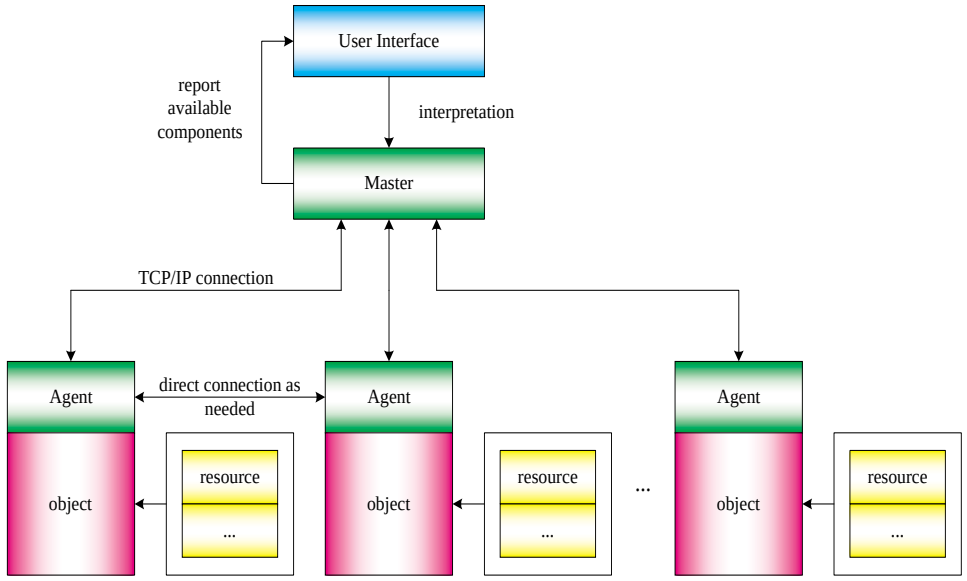


Figure 4.2: Architecture of the middleware.

FPGA object has bitfiles as resources. These resources can be added to as needed, or transferred from one object to another. There will be a TCP/IP connection between the master and each agent for control data and application data. The following sections discuss the architecture in more detail. First, each part of the system is discussed. Then, the simulation steps using the distributed reconfigurable simulator are discussed, and a simple example is given.

4.2.1 Middleware Master

The middleware master is the core of the system and acts as a server for the entire system. The master must be running before any of the agents can make connections, as in server-client models. The master will accept incoming connections from agents, and collect the data provided by the agents regarding the associated objects and resources. The master will

then present this data to the user through a graphical user interface. The user can select algorithms from the provided choices to form the system to be simulated. Once the user makes the selection, the master sends commands to the agents to form the simulation model for the system to be simulated.

4.2.2 Middleware Agent

The middleware agent mainly provides a connection point between the associated object and the middleware master. The agent first collects the information on the object and resources, then sends this data to the master. Once the master sends user configuration data back to the agent, the agent loads the necessary resource to the object, and makes connections to other agents as needed. The agent may collect data during the simulation if requested by the user.

4.2.3 Objects and Resources

An object can be any element with processing capability. The resource provides the necessary data to make the object perform the various required functions. For example, a reconfigurable board can be an object, and the downloadable bit files can be resources for this object. A general purpose CPU can be an object with executable files as resources. A single executable file can also be an object. As long as the agent can fully understand the object and the format of the resources, there is no restriction on what can or cannot be an object or a resource.

4.2.4 User Interface

The user interface can use the data passed by the middleware master to provide necessary information to the user. As the master determines what objects and resources are available, the user interface is dynamically created in real-time with this information. Using this information, the user can create a system to be simulated. The user can also specify the number of bits or symbols to be simulated.

4.2.5 Preprocessing

The preprocessing stage includes four tasks:

1. service registration process,
2. reporting the available objects and resources to the user,
3. letting the user define a system to be simulated, and
4. mapping the system to be simulated to appropriate objects and resources.

The following discusses each of these tasks in more detail.

Service Registration

When the user initiates the simulation process, the middleware master communicates with each agent to determine what object and what resources are available. The master runs

as a server and waits for the agents to connect. The TCP/IP connection should be made through a predefined port. Although the object itself might physically be in the system, the middleware has to know where it is and what it is. The agent provides this information, which is the service registration process. For example, if an FPGA board is present in the system, the middleware must know what type of board is present. Also, the middleware must know what kinds of resources (functions or algorithms) are available with the board. Although the user does not need to know what type of objects are within the system, the middleware needs to use this information to provide an even more versatile simulation environment by allowing objects of the same type to share resources as needed.

Reporting to the User

When the master is aware of the location and capabilities of the attached objects and resources, it will dynamically create a library of components that can be used to create the system to be simulated. The data for this dynamic feedback is obtained from the agents during the service registration process. If a function or algorithm is needed but is not available, the user must create it and have it attached to the system as an object. This is accomplished by using the middleware agent. Once it is created in this way, it can be used by other users as part of the library.

Also, the user can get the information on what type of data can be obtained from each function or algorithm and what parameters can be modified. This information must be passed to the middleware master by each agent since this information is different depending on how the algorithm is implemented.

System Definition

Once the information on available resources is obtained, the user can build the system to be simulated. When the user completes this task, the middleware master then maps the system.

Mapping

Mapping the user-defined system to the available objects and resources is one of the key tasks of the middleware master. The master determines which of the objects and resources are needed and maps the user's system to the appropriate objects and resources.

Since it is possible to have more than one instance of the same object, the middleware should be able to transfer a resource from one object to another. If the simulation requires the output of a function implemented in one object as the input of a function implemented in another object, the master should then provide a direct connection between the two objects instead of having the data pass through the master. This reduces the overall computing the master has to handle and, therefore, enhances the performance. Once this step is completed, execution of the simulation can begin.

4.2.6 Running a Simulation

Once the middleware determines how to partition the user-defined system into blocks to use the available objects and resources, the simulation can begin. During the simulation, the

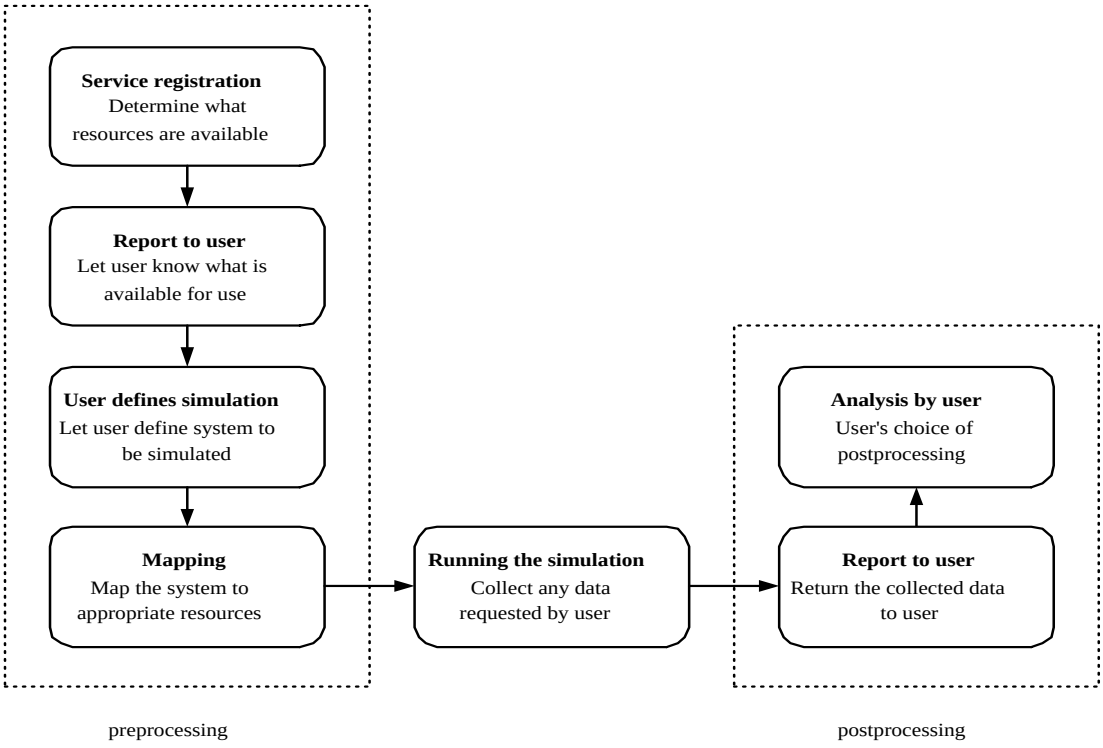


Figure 4.3: Steps of a simulation using middleware.

master's main job is to

- collect the data user needs for postprocessing,
- synchronize each component, and
- monitor the simulation to make sure the simulation is running smoothly.

Figure 4.3 shows the simulation stages graphically.

Collecting the Data

The whole purpose of the simulation is to collect the necessary data required to analyze the performance of the system under study. The user determines the data to be collected, and the middleware collects the required data during the simulation.

Synchronization

Since the simulation of communication systems at the physical layer is clock-driven, synchronization of the simulation is a critical issue. The middleware must synchronize all objects and resources, and determine when data is valid, when data is needed, etc.

Monitoring the Simulation

Monitoring is necessary because there can be problems during the simulation such as software errors, loss of connection, and power failure. If an agent with an active part of the simulation loses the connection, the whole simulation must restart.

4.2.7 Postprocessing

During the simulation, the user-desired data, such as the original PN sequence or noise samples, is collected at the local machines where the agents and associated object are running. When the simulation is complete, the necessary data is passed back to the user. The type

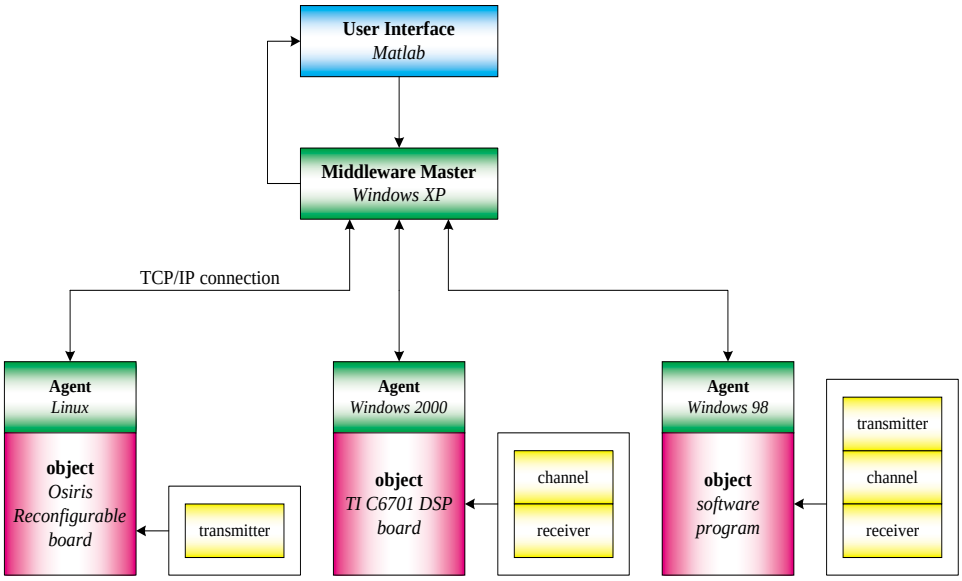


Figure 4.4: Implementation of the middleware system.

of data to be collected can be specified by the user during the preprocessing stage. Only the data given by the objects and resources and defined in the preprocessor can be collected through the middleware agents. If the user wishes to obtain data not provided by the objects, the user must create an object or resource as needed.

4.3 Implementation

A proof-of-concept model has been developed to demonstrate the functionality of the middleware system. This model includes a working middleware with three different types of objects and resources. Figure 4.4 shows the implementation of this system. There are four computers involved in the system. One computer runs the master and the user interface, while the other three computers run agents with different types of objects and resources.

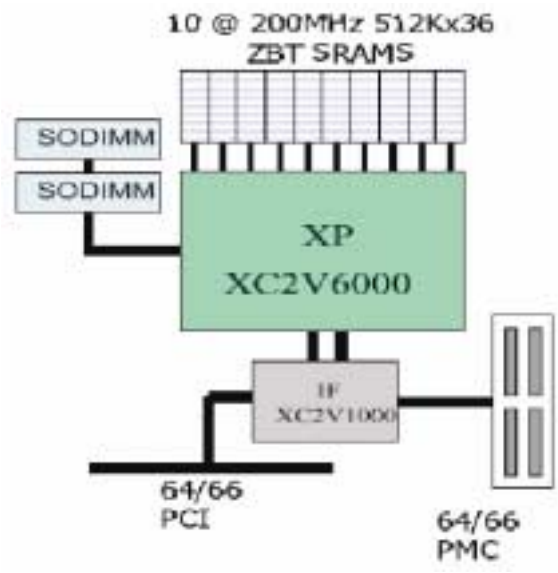


Figure 4.5: Osiris board architecture [Os02].

4.3.1 Components

There are three objects in this system. The first object is an Osiris FPGA reconfigurable board, which is shown in Figure 4.5 [Os02]. An Osiris board is faster than a WildForce board, and has more available gates than a WildForce board. An Osiris board consists of two Xilinx Virtex-II FPGAs, XP and IF. XP is the user-programmable Virtex-II XC2V6000 chip, and IF is the data I/O chip. The machine hosting this board runs the Linux operating system. The only available resource for this object is the transmitter.

Next, a Texas Instrument TMS320C6701 Digital Signal Processor Evaluation Board is connected [TI02]. The C6701 DSP is a floating-point processor, which runs on 120, 150, or 167 MHz clock, and is capable of processing eight 32-bit instructions per cycle. Figure 4.6 shows the functional block diagram and block diagram of the CPU of the C6701 processor. The host for the board runs the Windows 2000 operating system. There are two resources

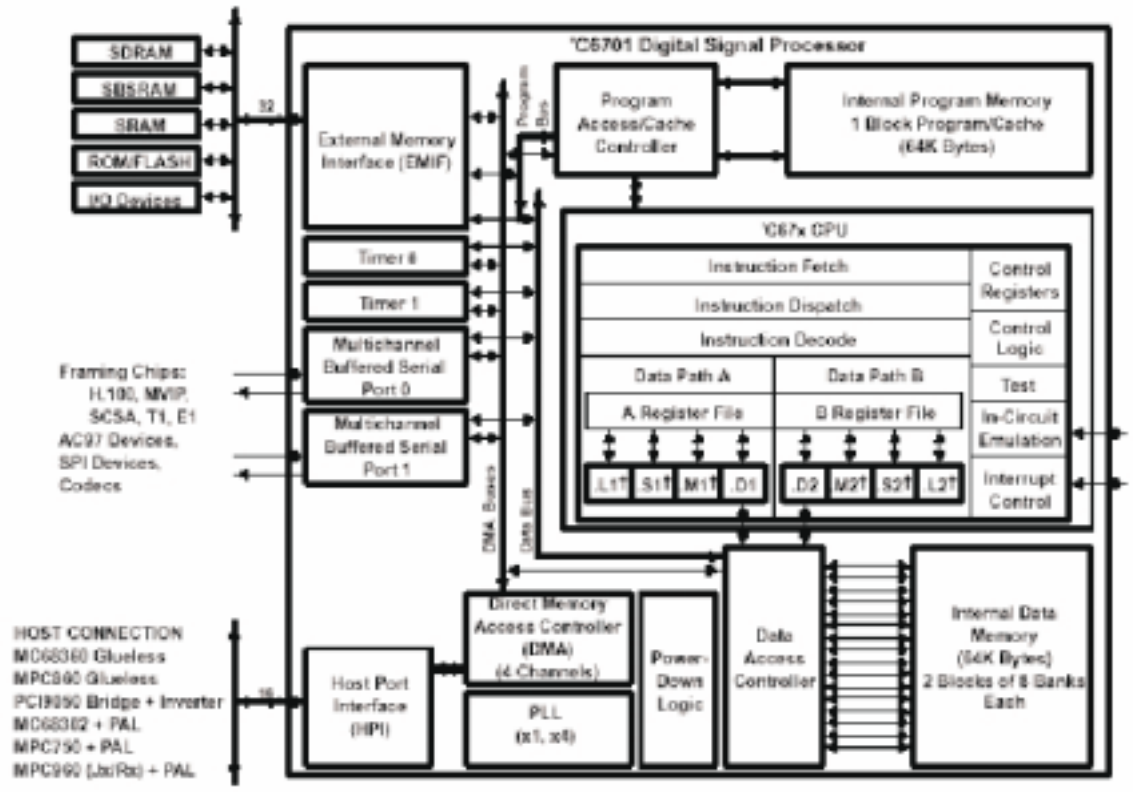


Figure 4.6: Functional block and CPU diagram of C6701 processor [TI02].

for this object. A receiver and a Gaussian channel model are implemented using this object.

Finally, software programs running in a general-purpose processor running under Windows 98 are used as the third object. There are three system components implemented within this object. They are transmitter, Gaussian noise channel, and receiver. Figure 4.7 summarizes the objects and resources used in the system.

Object	Available resources	OS
Osiris reconfigurable board	transmitter	Debian Linux
TI C6701 DSP	Gaussian channel receiver	Windows 2000
software	transmitter Gaussian channel receiver	Windows 98

Figure 4.7: Objects and resources used in the system.

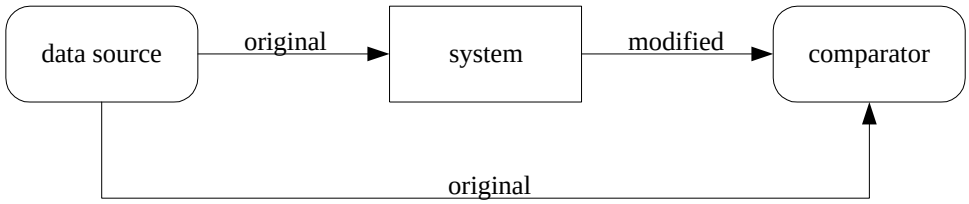


Figure 4.8: Simplified block diagram of simulation method.

4.3.2 Flow of Simulation

Figure 4.8 is a simplified block diagram illustrating a common method for simulation of communication systems. The data generated from the source passes through the system, and is compared with the unmodified original data. This requires a direct connection between the data source and the data sink, or the comparator.

To enable this direct connection, the data being passed between agents contains two sets of data as shown in Figure 4.9. An agent modifies the data in a set called *modified*, while

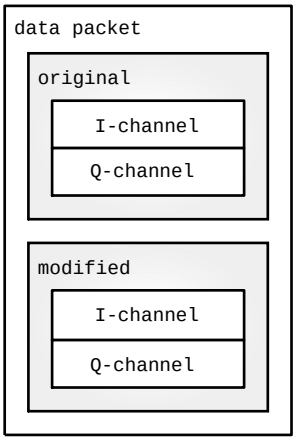


Figure 4.9: Structure of the data being passed between agents.

leaving the set of data called *original* as it is. This way, the original set of data can be passed throughout the system along with the modified data. This makes comparison of data easier at the receiver. Each set has two floating point numbers; one for I-channel data and the other for Q-channel data. Figure 4.10 illustrates this concept. Here, the value of the original data set remains the same through out the system, while the value of the modified data set changes from *a* to *a'* and *a"* as it passes through the system. This structure allows the user to compare the original data to any modified data within the system under study.

The master must run before any agents run, since the master works as a server for the rest of the system. Once the master is running, the agents can be started with the IP address of the master as an argument. If no address is specified, it is assumed that the master and the agent are running on the same machine. Figure 4.11 shows a screen shot of an agent during execution. An agent will read the information on available resources from a text file called *agent.txt* located in the same directory. The format of this file is shown in Figure 4.12 and an example is shown in Figure 4.13. In this example file, there are two resources for the object. The first resource is a receiver algorithm, and the second resource

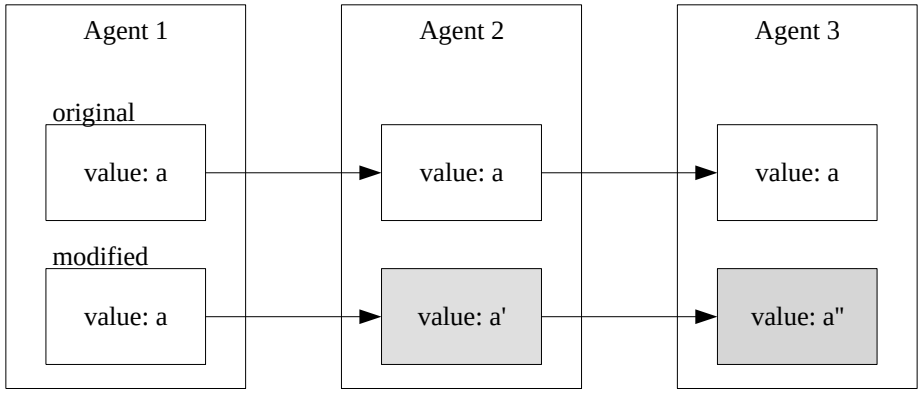


Figure 4.10: Using a two-part data structure to pass original value.

is an algorithm to implement a Gaussian noise generator. Information on the object is not included in this file. Because the agents are object-specific, the object information is stored within the agent program. By storing the resource information in a separate text file, it can be easily updated as the available resources change. Once the file is correctly read and the information is obtained, the agent will attempt to connect to the master.

The master can be started with a timeout period as an argument or a default timeout period of 5 seconds. The master will wait for 5 seconds or a specified timeout period for an agent to connect. Once an agent connects, the timeout period resets and the master goes back to the wait status. If there are no incoming agent connections for the timeout period, the master moves to the next stage, which provides the user with the information gathered from the agents.

Once all agents are connected, the user interface can be started. The user interface has been implemented using Matlab's graphical user interface functions such as `uicontrol` [Ma02b]. Since Matlab is an excellent tool for postprocessing tasks such as plotting, it is a logical choice for the user interface. Integrating the user interface with the master in a single

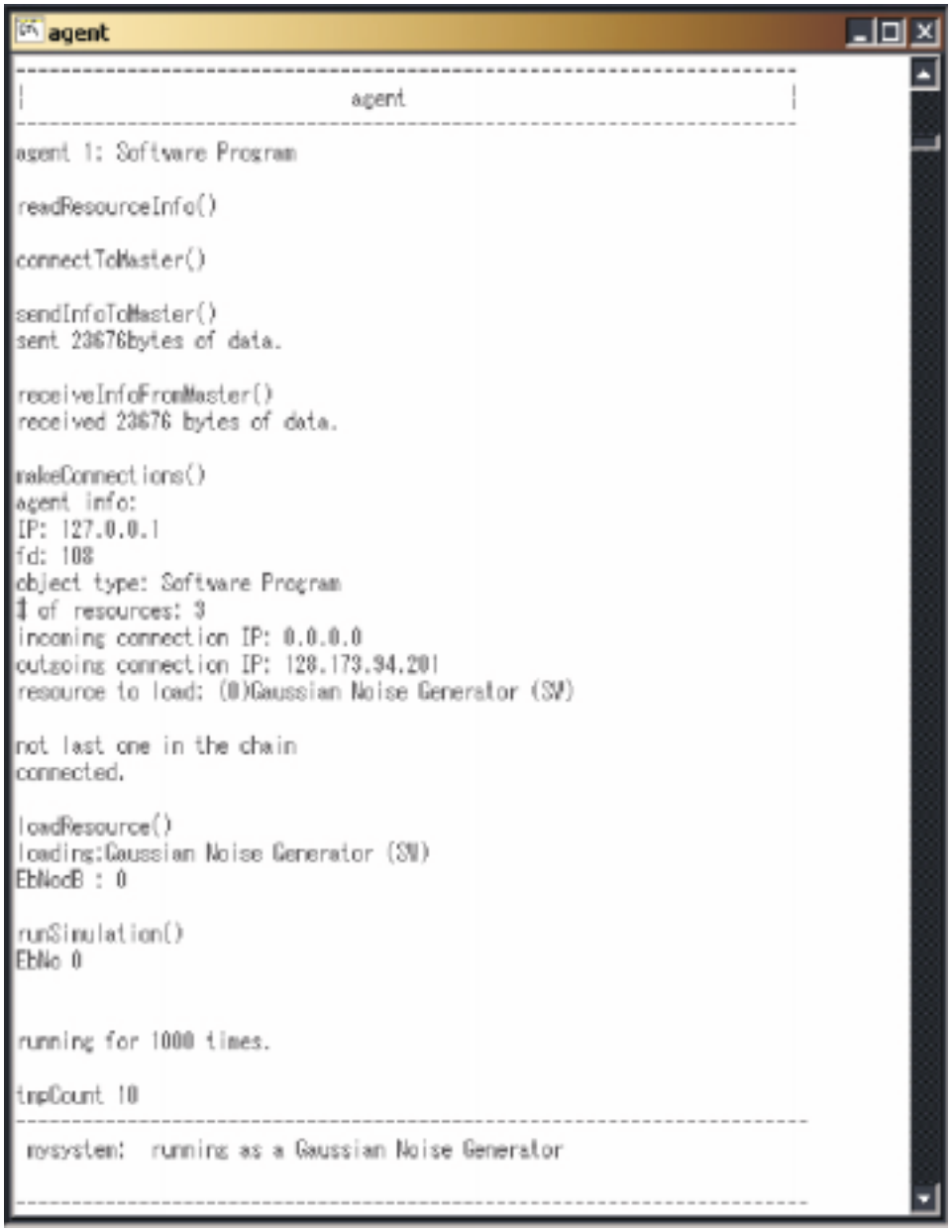


Figure 4.11: Running an agent.

```
file name of the resource #1
delay of the resource #1
name of the algorithm of the resource #1
full description of the resource #1
input parameter and default value listing of the resource #1
name of the output of the resource #1
...
...
...
file name of the resource #N
delay of the resource #N
name of the algorithm of the resource #N
full description of the resource #N
input parameter and default value listing of the resource #N
name of the output of the resource #N
```

Figure 4.12: Format of the agent.txt file.

```
receiver.out
1
receiver (DSP)
This is the description of the receiver
input1 0
output
gauss.out
1
Gaussian noise generator (DSP)
A Gaussian noise generator with EbNo as an argument.  EbNo:  desired Eb/No in dB
EbNo 1
noise
```

Figure 4.13: An example of agent.txt file.

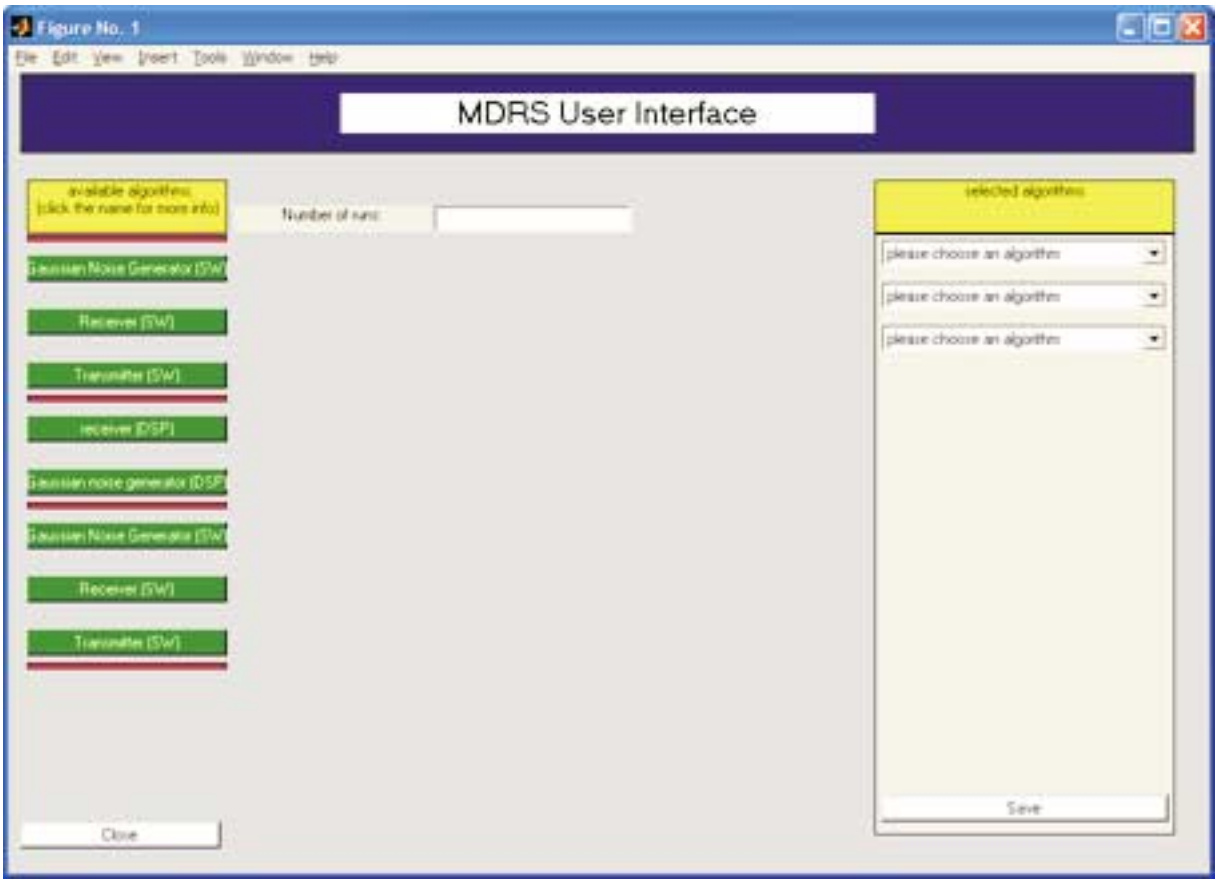


Figure 4.14: User interface implemented in Matlab.

program is another possible option. Figure 4.14 shows a screen shot of the user interface. The content of this user interface window is dynamically created based on the information the master has collected from the various agents. Here, the information on available resources are listed on the left side of the window. The number of simulation runs can be specified by the user by entering the desired number of runs in the box in the middle of the window. The right side of the window has drop-boxes for the user to choose the resources to be loaded. There are three boxes, since there are three agents connected to the master. The user can use these drop-boxes from top to bottom to select desired algorithms. The order of selection in these drop-boxes becomes the order of the algorithms in the simulation.

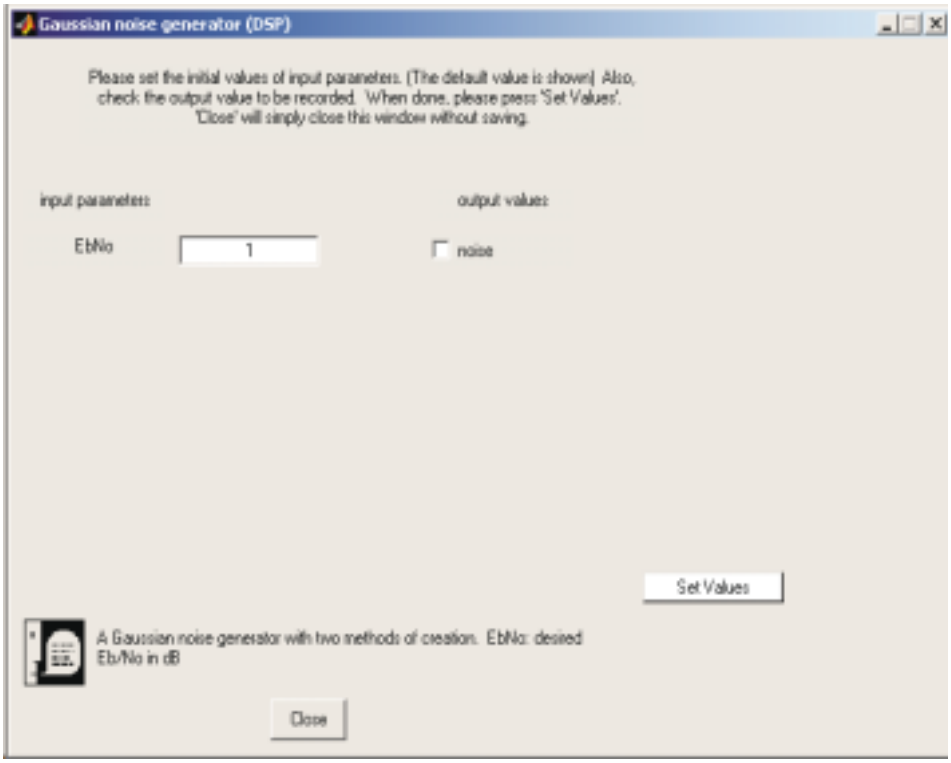


Figure 4.15: Input output dialog box for a resource.

When the user clicks on the name of a resource, a dialog box starts with input and output information along with a description on the resource, as shown in Figure 4.15. Here, the user can enter the initial required values of the parameters of the resource. If the user does not enter any values, then the default values are used. The default values are read by the agents from the agent.txt files. Also, the user can specify if any given output of the resource is to be saved for use by the postprocessor. By checking the box next to the output variable name, the data are recorded during the simulation. This enables the user to look at any data within the simulation as needed.

Once the user completes the process of selecting the algorithms to be simulated, and enters initial values for resource parameters, the master will process this information and

send the appropriate commands to the agents. Each agent will load the resources needed for the given simulation, make connections to other agents as necessary, and run the simulation. Figure 4.16 shows the timeline of the events throughout the simulation process. This timeline represents a situation in which agent 1 connects to agent 2, and agent 2 connects to agent 3. This means that the user has chosen a resource in agent 1 to be the first algorithm of the simulation, agent 2 to be the second, and agent 3 to be the last agent of the simulation.

4.3.3 Configuration

As an example, a simple communication system, which is shown in Figure 4.17 is implemented. Two different configurations have been tested. First, the DSP board runs the channel function, while the general software program runs the receiver function. Then, the DSP board will run the receiver, while the channel is running in a software program. The Osiris board is chosen to run the transmitter function in both cases. Figure 4.18 summarizes the configurations.

The first example results in the configuration shown in Figure 4.19. Once the selection is made through the user interface, the information needed for the agents to configure is passed to the agents from the master through the TCP/IP connection. At this point, each agent knows the resource to load, which agent will connect to it, and which agent it should connect to. These functions are performed as soon as the configuration data is received from the middleware master. If the user does not select any resource of a particular object for a given simulation, then the agent will not perform any of these functions.

Once the configuration is finished, the simulation begins. After processing for a user-specified number of symbols, the simulation terminates. Any data the user requested is

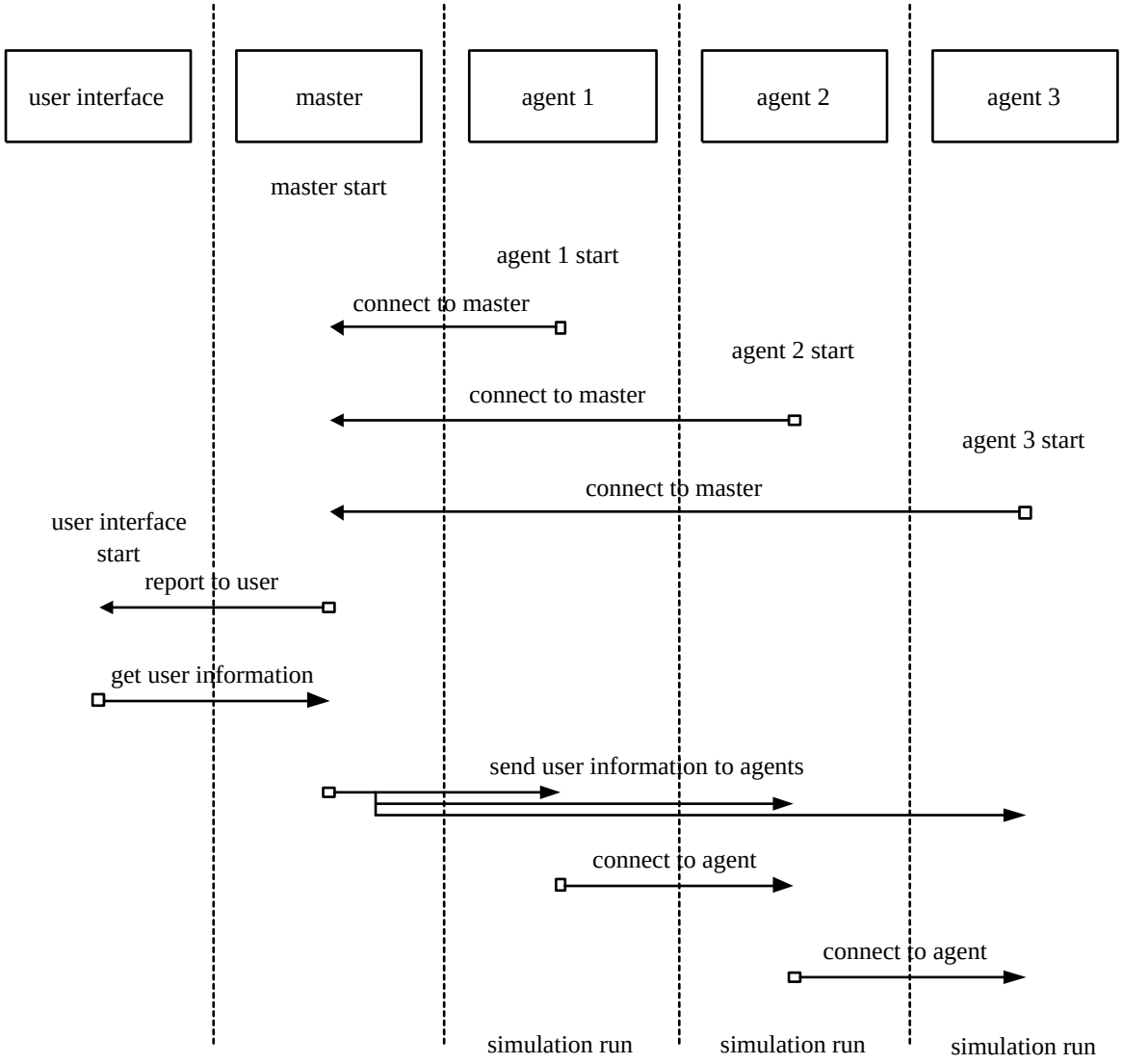


Figure 4.16: Timeline of the simulation using middleware system.

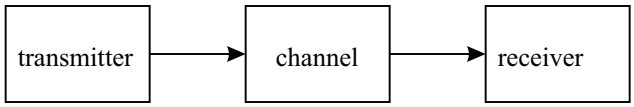


Figure 4.17: A simplified model of communication system.

	Transmitter	Channel	Receiver
Configuration 1	Osiris reconfigurable board	TI C6701 DSP	software
Configuration 2	Osiris reconfigurable board	software	TI C6701 DSP

Figure 4.18: Two different configurations used for the simulation.

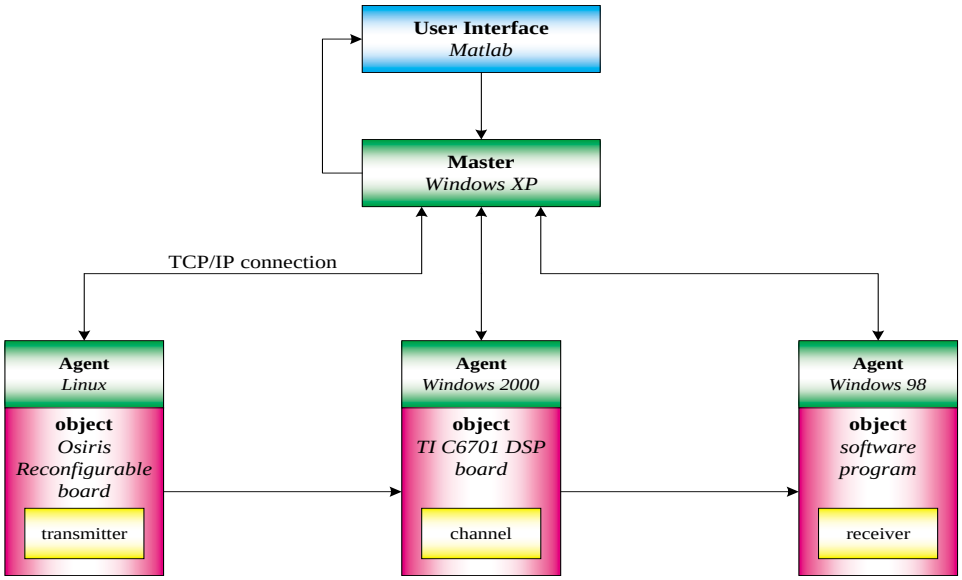


Figure 4.19: Configuration of the middleware system running a simple simulation.

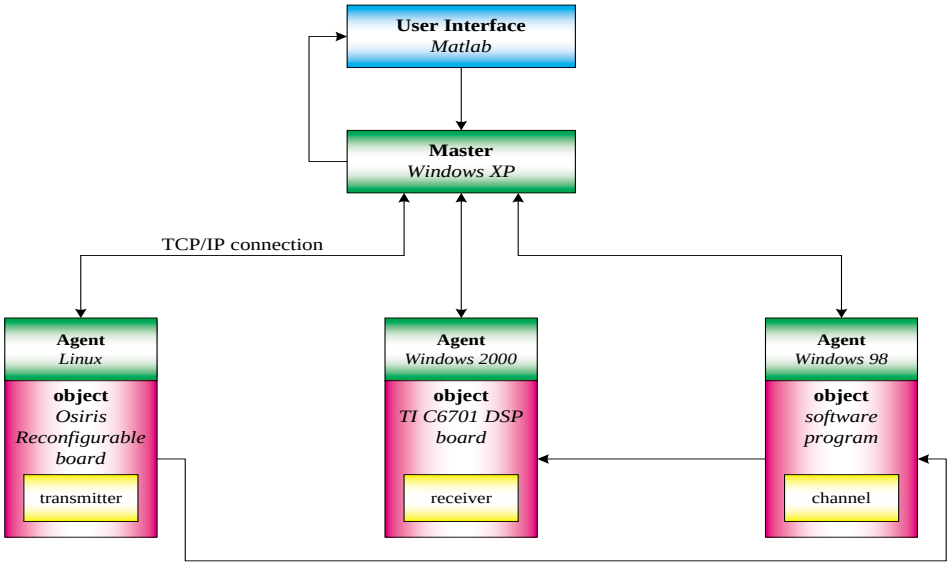


Figure 4.20: Different configuration of the middleware system running a simple simulation.

saved during the simulation for postprocessing.

The second configuration, where the DSP runs the receiver function, is shown in Figure 4.20. Again, the connections between agents are made automatically by agents once the configuration data is received from the master.

4.3.4 Analysis

The result of these simulations is shown in Figure 4.21. Because the DSP and the software program both used exactly the same algorithm for the noise and the receiver, the resulting data are identical. Six different values of E_b/N_0 have been tested, ranging from 0 dB to 5 dB in steps of 1 dB. The simulation clearly runs as expected.

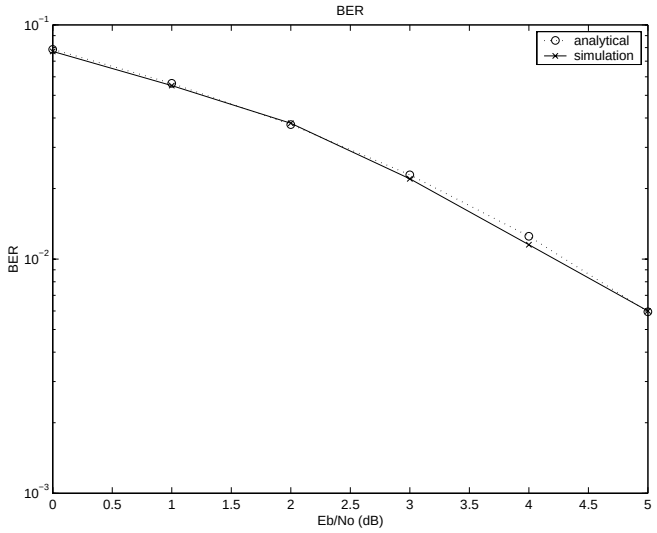


Figure 4.21: Result of the simulation using middleware system.

Figure 4.22 summarizes the runtime of each resource used in the system when running individually. For example, if the C6701 DSP board is running the Gaussian Channel model without any transmitter or receiver connected to it, it takes 58 seconds to run 1,000 samples. It is clearly shown that the number of runs and the required runtime are linearly proportional.

Figure 4.23 summarizes the performance of the resources when they are running together in a simulation. For both configurations, the runtime for each resource remains the same when the simulation runs 1,000 times. However, when the simulation is executed 10,000 times, the Osiris board takes significantly longer compared to the runtime required when executed individually. This is due to the inter-agent TCP/IP communication. Because the TCP buffer has a finite size, the first object, the Osiris board, cannot transmit any more data when the TCP buffer is full. Even though the Osiris board can execute its resource at a faster speed, it has to wait until it can send the data to the adjacent agent.

resource	runtime (seconds)		
	100 runs	1,000 runs	10,000 runs
Transmitter	2	19	184

a. Runtime of resource implemented in Osiris board.

resource	runtime (seconds)		
	100 runs	1,000 runs	10,000 runs
Channel	6	58	573
Receiver	4	38	378

b. Runtime of resource implemented in TI C6701 DSP board.

resource	runtime (seconds)		
	100 runs	1,000 runs	10,000 runs
Transmitter	5	49	481
Channel	5	48	476
Receiver	5	48	473

c. Runtime of resource implemented in software.

Figure 4.22: Individual runtime of resources.

# of runs	runtime of resources (seconds)		
	Osiris board Transmitter	C6701 DSP Channel	software Receiver
1,000	20	58	59
10,000	489	575	576

a. runtime of simulation for configuration #1.

# of runs	runtime of resources (seconds)		
	Osiris board Transmitter	software Channel	C6701 DSP Receiver
1,000	20	47	49
10,000	409	474	476

b. runtime of simulation for configuration #2.

Figure 4.23: Runtime of resources running in a simulation.

Inter-agent TCP/IP Communication

Figure 4.24 through Figure 4.26 illustrates the effect of inter-agent TCP/IP communication. Here, *t2* represents the longest processing time, or the slowest resource in the system. And, *t1* and *t3* are processing times of other resources in the system. *d1* and *d2* represent the transmission time of the data from one resource to the next.

In Figure 4.24, a resource in the middle of the simulation takes the longest amount of time. This results in the TCP buffer being full for the resource which has to send data to it. The first resource must wait until the second, slower resource drains the data from the TCP buffer. Clearly, the second resource, which is the slowest resource, becomes a computational bottleneck.

In Figure 4.25 the first resource takes the longest to process data. The subsequent resources run faster than the first one and, therefore, process the data before the next data is available. This results in the gap shown in the Figure 4.25, which is idle time for the resource.

In Figure 4.26, the last resource is the slowest. This resource eventually makes the TCP buffer full and this full buffer may propagate backwards. It is clearly seen that the overall simulation time is the same as the runtime of the slowest resource of the given simulation, plus the minor fixed overhead of filling and draining the pipe. In other words, no matter how the resources are configured, there is no significant delay added to the system, except the initial filling of the pipe and the draining of the pipe at the end of the simulation.

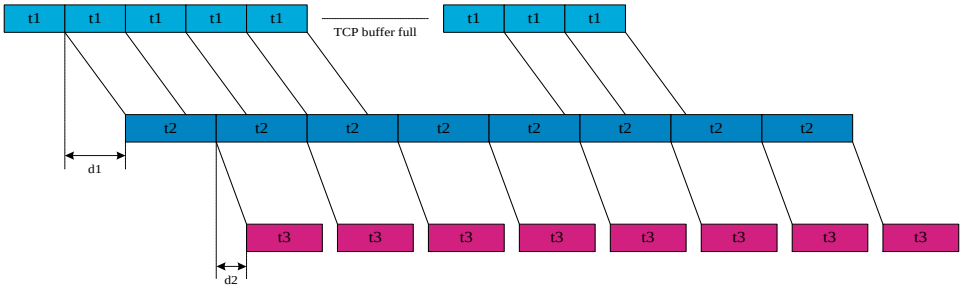


Figure 4.24: Timing scenario 1: second resource takes longest.

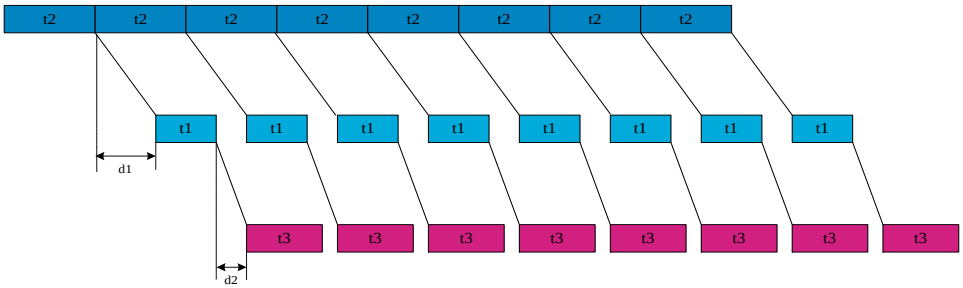


Figure 4.25: Timing scenario 2: first resource takes longest.

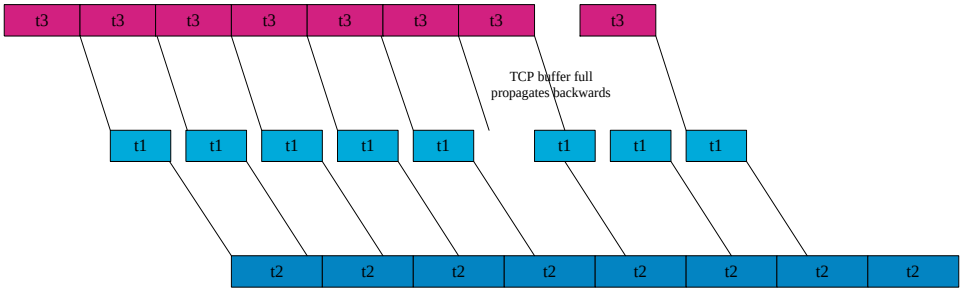


Figure 4.26: Timing scenario 3: last resource takes longest.

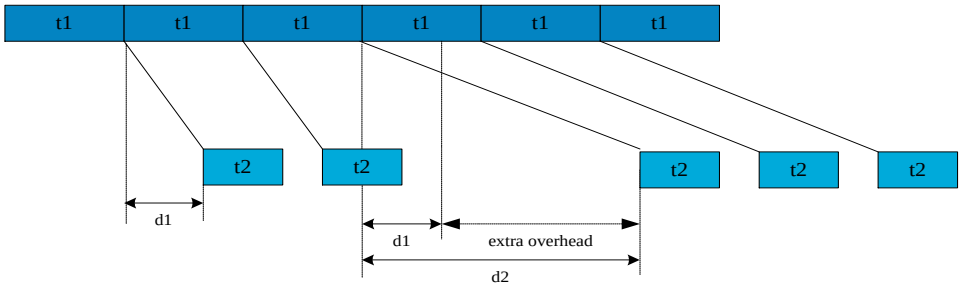


Figure 4.27: Effect of unstable or congested network.

Unstable or Congested Network

In Figure 4.24 through Figure 4.26, the inter-agent TCP/IP communication time, represented in *d1* and *d2*, does not have a significant impact to the performance of the system. In simulations of physical-layer communication systems, the overall number of symbols being simulated far exceeds the number of resources involved. Therefore, this delay becomes negligible. However, if the network is not stable or if the network experiences severe congestion, the inter-agent communication may add extra overhead to the overall execution time. This is illustrated in Figure 4.27. Here, *d1* is the propagation time between the two agents. If this propagation time changes to *d2*, which is longer, then there is extra overhead added. However, this overhead happens only once and is insignificant when the overall simulation runs for millions or billions of samples.

Processing Agent Functions

Another possible computational bottleneck is the time required for processing agent functions. Since the agent has to send and received data during the simulation, it requires some

processing time from the local CPU. If this processing takes longer than the time the resource takes to process the data, it becomes a bottleneck. There are several variables involved in this issue such as the clock speed of the local CPU, other processes sharing the same CPU, and the number of cycles it takes for the agent to process necessary functions. The delay time of the given resource is also involved. For example, if the resource runs at 10 MHz, and the local host machine has a CPU running at 2 GHz or 2000 MHz, the agent should finish the necessary functions such as send and receive functions in roughly 200 cycles. Of course, the CPU is time-shared with other takes such as operating system functions or any background processes, and this is uncontrollable.

Initial Configuration Time

The initial configuration time can vary depending on the components of the system. Also, the user may spend certain amount of time when creating the system to be simulated using the user interface. Time required for these tasks are not relatively fixed and small compared to the overall simulation time. Therefore, the initial configuration time can be ignored.

Communication Between Agent and Object

There are numerous ways of communication between an agent and its associated object. For example, a reconfigurable board may use FIFO or memory read/write to communicate with the agent. A TI C6701 DSP board may use RTDX (Real Time Data eXchange) to talk to the agent [TI02]. A software algorithm can be implemented as a separate program and talk to the agent through various methods such as a socket connection, pipe, or file. Or, the agent may have the software program integrated. These variety of communication methods

between an agent and its associated object have different communication times. Because the agent must talk to the object periodically throughout the simulation to process simulation data, any excessive communication time between agent and its associated object can become computational bottleneck to the overall system.

Summary

In this section, the performance of the distributed reconfigurable simulation system using middleware has been presented. A number of performance variables have been identified. There are variables that do not have significant impact, such as the initial configuration time, inter-agent TCP/IP communication time, and number of simulation runs. Also, there are variables that need careful attention such as the processing time for agent functions and communication time between an agent and its associated object.

When compared to a single device, system-on-chip, approach of simulation, the distributed simulation certainly has variables that can prevent optimal performance of the devices in the system. However, the distributed simulation has several advantages such as the ability to integrate existing simulation modules with new modules, the ability to perform co-simulation with other simulation tools, and the ability to perform "hardware in the loop" simulation. The following section discusses these advantages in more detail.

4.4 Application

The flexibility provided by the distributed simulator using middleware allows it to be used for a variety of purposes. It can be a standalone simulator to simulate a complete system under study. In addition, it can perform part of a simulation within a larger simulation as needed. Also, as mentioned, it can be used for "hardware in the loop" design and validation. In this section some of the possible applications are presented.

4.4.1 Standalone Simulator

By implementing the complete system using available objects and resources, a distributed simulator for the system can be created. A complete communication system mainly consists of three basic components as shown in Figure 4.17. With the flexibility of reconfigurable distributed computing, the implementation of different systems having different components can be done with relative ease. Several system components can be implemented into a single object such as an FPGA chip.

The input data source can be an arbitrary sequence created by the PN sequence generator or it can be a specific sequence of bits if the goal of the simulation is to test a coding algorithm. This specific sequence can be implemented within the FPGA if the sequence is short enough to fit or it can be implemented using a general purpose processor using a hard drive as storage space.

With the greatly increased throughput of the system, the reconfigurable simulator can be used as a standalone simulator, which replaces a conventional software simulation.

4.4.2 Providing Channel Models for Receiver Testing

In some cases, it may not be necessary to implement the whole system. A software radio has been developed at Virginia Tech that uses the reconfigurable device as the platform [Sr99]. Testing of the performance of this receiver can be done in several different ways. However, because the receiver is developed in an FPGA, using a reconfigurable simulator is the perfect choice for receiver testing and performance verification. For example, the receiver can be connected to the middleware system by attaching an agent to it. This agent will work as a bridge between the rest of the simulation and the receiver by providing appropriate I/O data back and forth.

4.4.3 Hardware-Software Co-simulation with OPNET

OPNET™ modeler is a commercial tool developed by OPNET Technologies primarily for network simulation [Op02a]. However, it does not have the capability to perform high-fidelity physical layer simulation at fast speeds. By incorporating a reconfigurable simulator, the overall fidelity of the network simulation can be improved while maintaining reasonable runtimes.

The interconnection between an OPNET simulation and a reconfigurable simulator, on which the physical layer is simulated, can be accomplished using the TCP/IP protocol. The OPNET simulation can be implemented on a PC or a workstation. The link-level simulation, running on the distributed simulator using middleware, can transfer the necessary data to the OPNET simulation using the TCP/IP connection. Figure 4.28 shows this configuration.

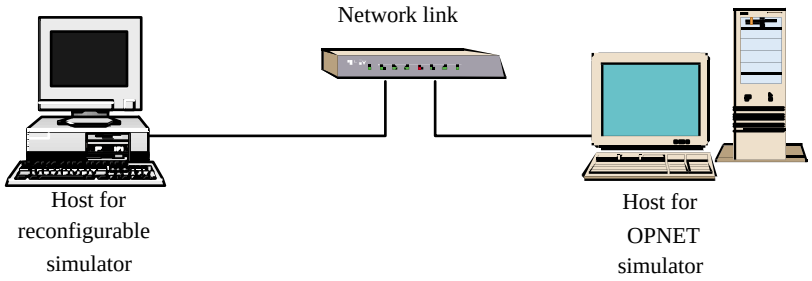


Figure 4.28: Configuration of the OPNET simulation running with a reconfigurable simulator.

An example has been created in which the overall network being simulated has four nodes. A single link is simulated using the reconfigurable simulator. Figure 4.29 is a screen display of the window running the OPNET simulation with the reconfigurable simulator communicating with it using a TCP/IP protocol [Sm00]. When the jammer introduces noise to the communication link, the link-level simulation reflects this change and the corresponding bit error rate is changed. The bit error rate displayed in the smaller window titled "Simulation status" is obtained from the link-level simulator and sent to the OPNET through TCP/IP network.

OPNET Technologies has recently developed an API for co-simulation [Op02b]. The purpose of this API is to avoid recreating existing software and integrating OPNET into existing environments. This API is to be included in version 9.0 of OPNET. It might be worthwhile to use this built-in API for integration of the reconfigurable simulation and OPNET simulation in the future.

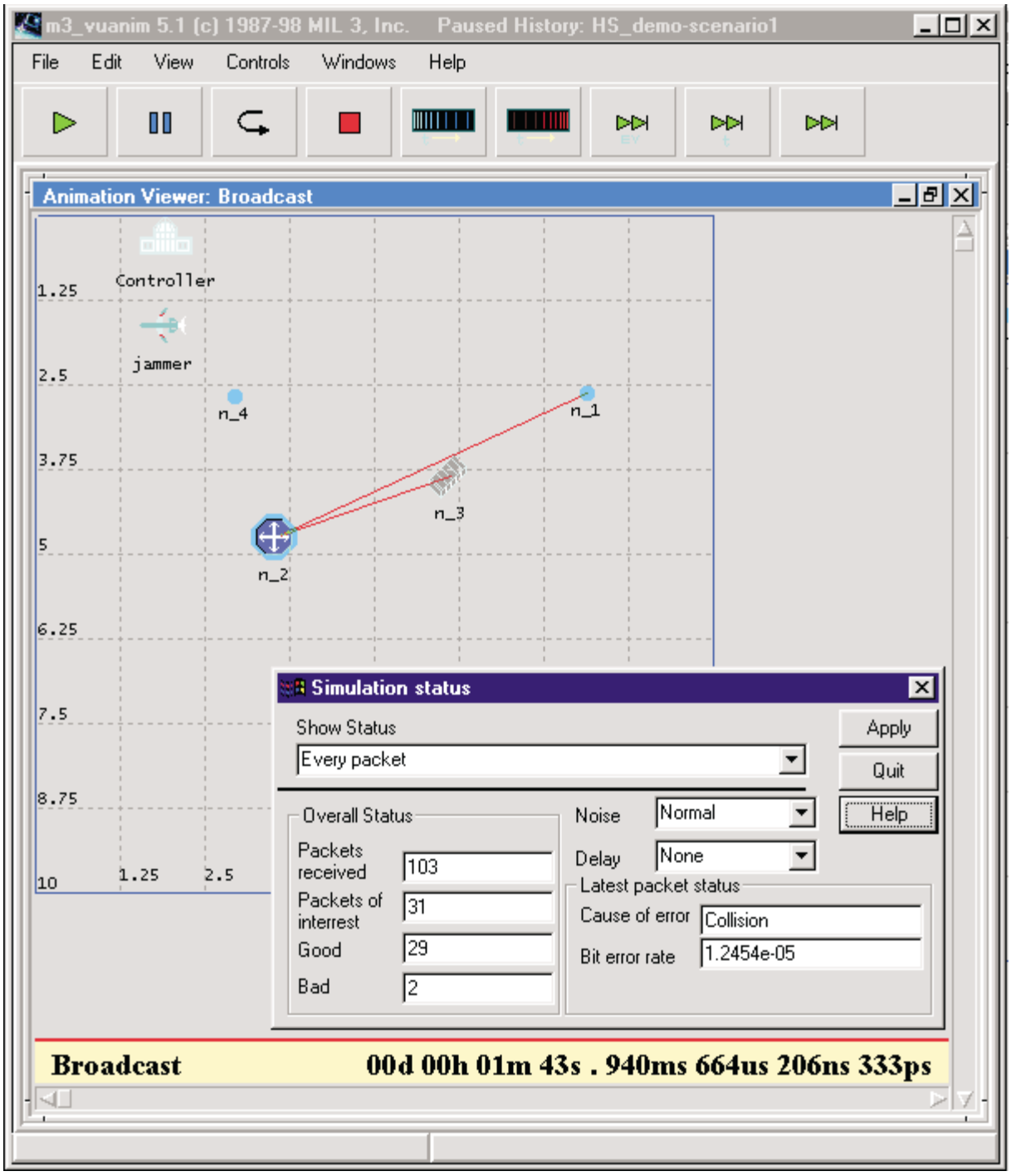


Figure 4.29: Screen shot of the OPNET simulation running with hardware simulator.

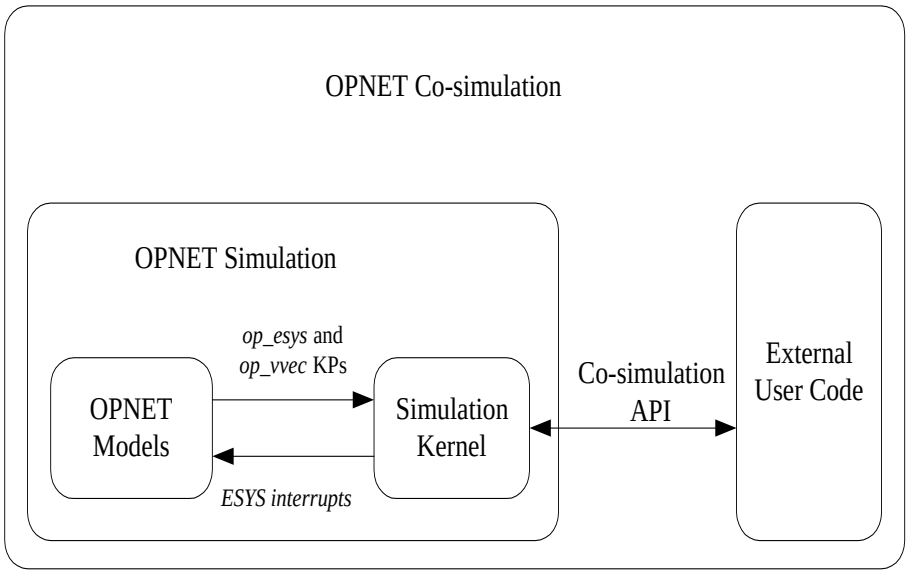


Figure 4.30: OPNET co-simulation architecture [Op02b].

4.5 Comparison with Other Methods

In Chapter 2, a brief overview of various distributed and parallel computing architectures and middleware was provided. In this section, the proposed middleware for distributed reconfigurable simulation for communication systems is compared with these other methods.

MPI and PVM are sets of API functions for distributed programming [FaL00]. Programmers can use these functions for writing distributed programs. However, the proposed middleware system is not targeted for programming in a distributed environment. The target of the middleware system is to provide a simple and flexible simulation environment. From this aspect, MPI and PVM may be used for implementing the inter-agent communication functions, but may not replace the middleware system. Applying MPI and PVM for the middleware system is discussed in Chapter 5.

There are two major differences between the proposed middleware for a distributed reconfigurable simulator and other distributed computing methods such as CORBA, Jini, and HLA. First, the proposed middleware is based on the *objects* and associated *resources*, not just *objects*. This makes resource exchange possible, which is one of the key advantages of the proposed system. Second, the *middleware master* can map appropriate algorithms as the user desires. This provides an environment for easier simulations.

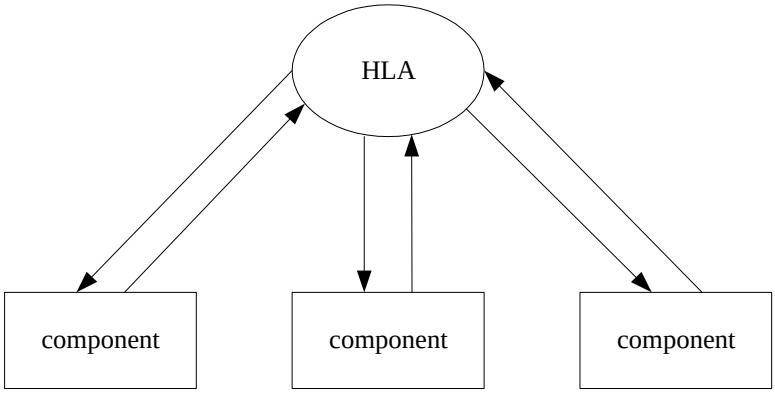
CORBA and Jini are different from HLA and the middleware proposed in this paper in terms of the intended applications. CORBA and Jini are general-purpose standards which are not necessarily for distributed simulations, while HLA and the proposed middleware are targeting simulations. Jini primarily works with Java-based systems while the other standards are language independent.

CORBA requires using interface definition language (IDL) for creating interfaces for objects [Em97a]. The IDL program is then compiled into code in one of the supported languages such as C or C++. In other words, if CORBA is to be used for the distributed reconfigurable simulation, an extra process is required for developing the agent programs. The core of the CORBA system is the object request broker (ORB), which is similar to the master in the proposed middleware system. The ORB is conceptually one server entity for client objects. However, the ORB actually consists of software residing both in the server and client machines. Although this is transparent to the programs, it implies that every object in a CORBA application must have ORB software installed within the same machine. This characteristic of CORBA restricts the user interface of the proposed middleware system. In the proposed middleware system, there is little restriction to where the user can access the system and initiate a simulation. A web-based user interface is discussed in Chapter 5, which enables the user to run simulations simply by connecting to the middleware master through a web page. This feature is not realizable when using CORBA, since the user interface must

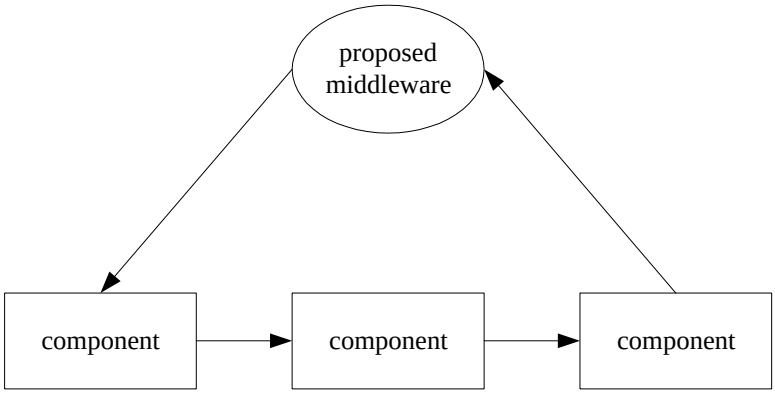
have ORB software installed with it.

Because the proposed middleware has a specific goal of simulation of communication systems, it has advantages over other methods. For example, HLA provides no direct communication between objects. Because HLA was developed by the Department of Defense mainly for interactive military simulations, it requires centralized control throughout the simulation. However, most of the communication system simulation is one-directional. The data is generated from the source and is processed at the transmitter. The transmitter output signal is transmitted through the given channel and is received by the receiver, which processes the signal to obtain the data transmitted. This one-directional property can be better realized by providing direct connections between adjacent components. In HLA, the centralized object manager must handle every component of the simulation. In the proposed middleware, the local connections can lower the computing requirements of the master. Figure 4.31 shows the effect of having local connections. To send data from transmitter to receiver, the HLA requires four transmissions, while it only takes two transmissions when the proposed middleware is used. This results in twice more inter-object communication time and extra overhead of processing the data, which may reduce the performance of the overall system significantly.

Figure 4.32 graphically shows the difference between various methods. There are numerous methods for general-purpose distributed and parallel computing. HLA has the specific goal of distributed simulation, primarily interactive military simulations. SLAAC API can be used for programming multiple reconfigurable devices. The work presented in this paper can combine all three areas.



a. High Level Architecture



b. Proposed Middleware

Figure 4.31: Comparison of inter-object communication method between HLA and the proposed middleware.

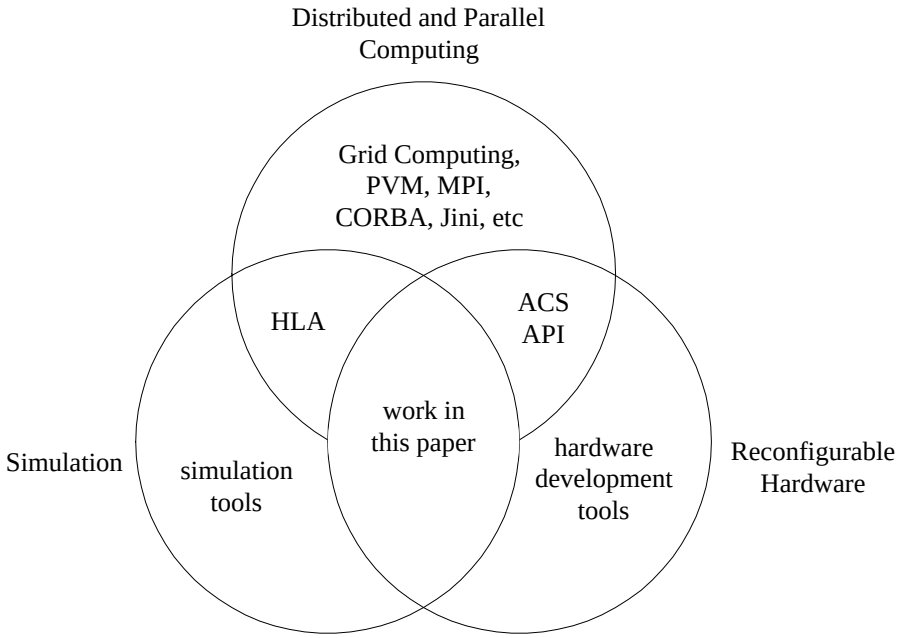


Figure 4.32: Comparison of the proposed middleware and other methods.

4.6 Summary

The middleware for distributed reconfigurable simulations provides several advantages. It enables various computing elements to work together to perform a simulation. It can run a co-simulation with other simulation tools such as the OPNET. Also, it can perform "hardware in the loop" simulation to provide a test bench for hardwares. It can be used to integrate existing simulation modules with new modules. Also, the user can initiate the simulation with ease through the user interface provided by the middleware system.

Because the system is distributed, extra computation and communication is needed throughout the system. Each agent and its associated object must send and receive data to each other. Agents within a simulation must form a communication link using TCP/IP.

Depending on the network characteristic, this communication link may introduce overhead. Also, the agent must share the computational resource with other processes running on the same machine. This uncontrollable overhead may prevent optimal performance of the system. These computational overheads and extra communication certainly adds overhead to the overall system performance. However, the advantages of distributing the system as discussed above can compensate this disadvantage.

Chapter 5

Possible Future Research

In previous chapters, it has been shown that reconfigurable technology can greatly enhance the simulation of communication systems, while the middleware concept can provide a flexible environment for incorporating various processing elements. In this chapter, possible future research is discussed.

5.1 Vision

This paper has presented a framework for distributed reconfigurable simulation. This framework can be expanded to form a versatile and flexible platform for the simulation of communication systems. Multiple users can access the simulator through the Internet and can run simulations concurrently. By creating a web-based user interface, the user can run the

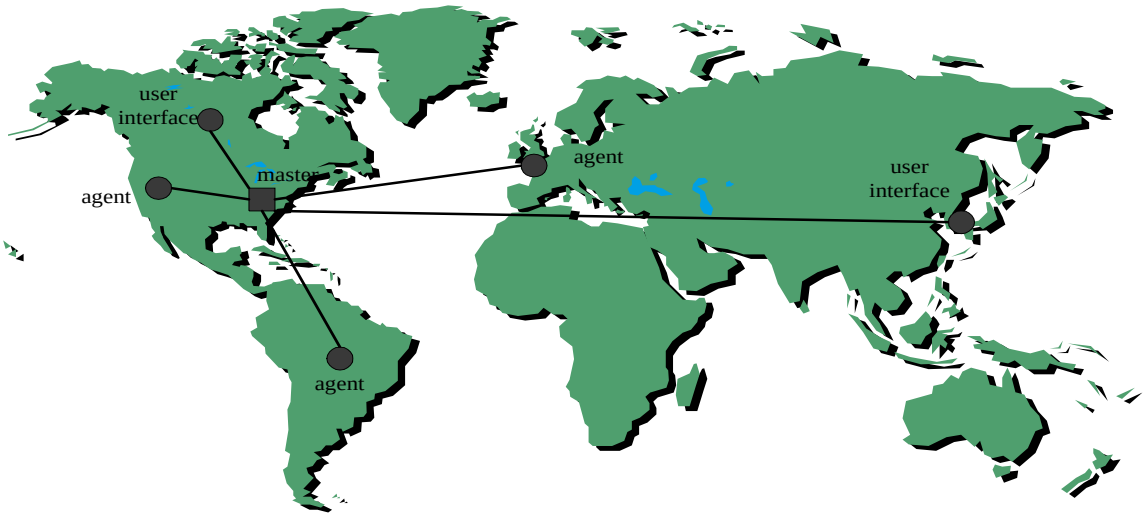


Figure 5.1: Vision: Distributed Reconfigurable Simulator.

simulation from anywhere with a generic user interface. With network bandwidth growing at a remarkable speed [Ma02a], the master and agents can truly be distributed. Figure 5.1 shows the possible future of the distributed reconfigurable simulator. The following sections discuss this vision in more detail.

5.2 Web-based User Interface

The World Wide Web (WWW) is the most popular and widely available Internet application. By creating a web-based user interface, the user can access a distributed reconfigurable simulation platform from anywhere in the world with access to the internet and a web browser. The computer running the middleware master can interact with the user as HyperText Transfer Protocol (HTTP) server. Obviously, the user must know the IP address or

hostname of the computer running the master.

When the user connects to the computer by entering the IP address or hostname in a web browser, information about the available resources can be viewed. The user can enter information on the type of simulation that is to be simulated through a graphical user interface built as a Java applet. This user interface can be similar to Matlab's Simulink, where the desired system is created using block diagrams. By interconnecting the block diagrams, the user can create the system to be simulated. This information can be sent back to the master and the simulation can be performed. Once the simulation is successfully finished, the user can use same web-based user interface for postprocessing, such as creating a plot of bit error rate. And, of course, the user can print results to a local printer or save data to a disk for future use.

5.3 Support for Multiple Users

The framework presented in this paper only supports a single simulation user at any given time. This makes sense for a limited number of objects and resources. However, if the number of objects and the associated resources increase, it might be desirable to support multiple concurrent users.

As the first user selects a system to be simulated, the master can remove the user-chosen object and resources from available object and resource list. When the second user connects to the master, the master then only reports the remaining available objects and resources to the user as being available, and whatever is being used by the first user as not available. If the second user configures a system to be simulated using the available resources only, then both

the first user's simulation and the second user's simulation can be executed concurrently.

The master can also tell the second user when the first simulation is expected to be completed. This expected completion time of a simulation can be calculated simply by multiplying the longest delay of the resources and the number of simulation runs chosen by the first user. Although this estimate is not accurate because of overhead of inter-agent communications, it can give the second user a general idea about the wait time. Based on the expected wait time, the second user can choose to wait for a particular resource to be available. This action puts the second user's simulation in the queue for any resource being used by the first user. As soon as the first user's simulation finishes, the master can configure the second user's simulation using the required resources.

5.4 Resource Exchange Between Agents

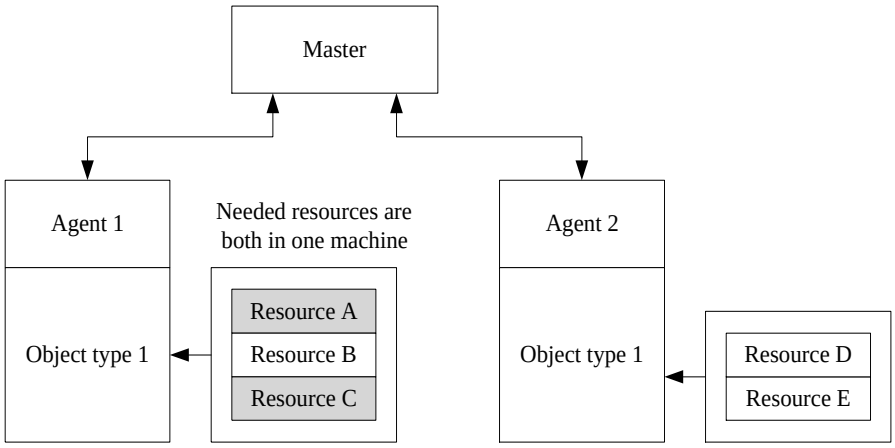
As the simulation platform grows larger, it is possible to have more than one instance of an object type. Each agent can have its own set of resources when connecting to the master initially. When an agent connects, the master should check to see if there is any other agent with same object type. Then the master can tell the agent, which has been already in the system to send any resource to the newly connected agent. Also, any resource the newly connected agent has can be sent to the existing agent if the existing agent does not already have that resource.

There can be situations where the user configures a system to be simulated using two resources for a type of object, where both of the resources are located on a same machine. If the user initiates the simulation before the resources are distributed among the same type of

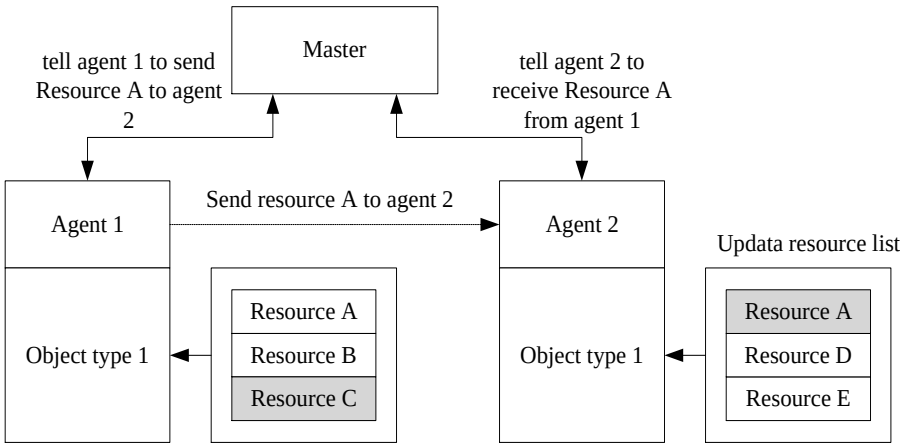
objects, the resource has to be sent from one agent to another in real time. If this happens, the master has to tell the agent with both resources to send one of the resources to another agent. Figure 5.2 illustrates this situation. Here, agent 1 with object type 1 has resources A, B, and C, while agent 2 has resources D and E for the same object type 1. If the user configures a system to be simulated including both resource A and resource C, the master tells agent 1 to send resource A (or resource C) to agent 2.

To enable resource exchange, it is necessary to standardize the communication method or the I/O format between the agent and the resources. Two possible options are shown in Figure 5.3. The current implementation without the capability for resource exchange has custom agent programs with embedded host programs. Because the host program is integrated with the agent, different types of resource I/O can be easily handled by the host program. However, this implementation does not permit resource exchange. For resource exchange to work, it is necessary to have a uniform agent for a given object type. In other words, adding extra resources to the available resource list should not require rewriting of the agent program, since this prevents automatic real-time resource exchange.

The first possible solution is to standardize the I/O format between the resource and the host program. A common method of communication between a reconfigurable board and a host program is through a First In First Out (FIFO) and memory read/write. For a given reconfigurable board, a programmer chooses the best suited communication method between the host program and the hardware. For different algorithms or different implementations, the programmer may use a different I/O method. For example, for one algorithm the programmer may use FIFO for I/O, while using memory for another algorithm. This results in a need for different host programs for each implementation to handle FIFO or the memory read/write as necessary. By standardizing this I/O format or the method of communication between the host program and the hardware, a single host program can be used for differ-



a. Before agent 1 sends the resource A to agent 2



b. After agent 1 sends the resource A to agent 2

Figure 5.2: Real-time resource exchange.

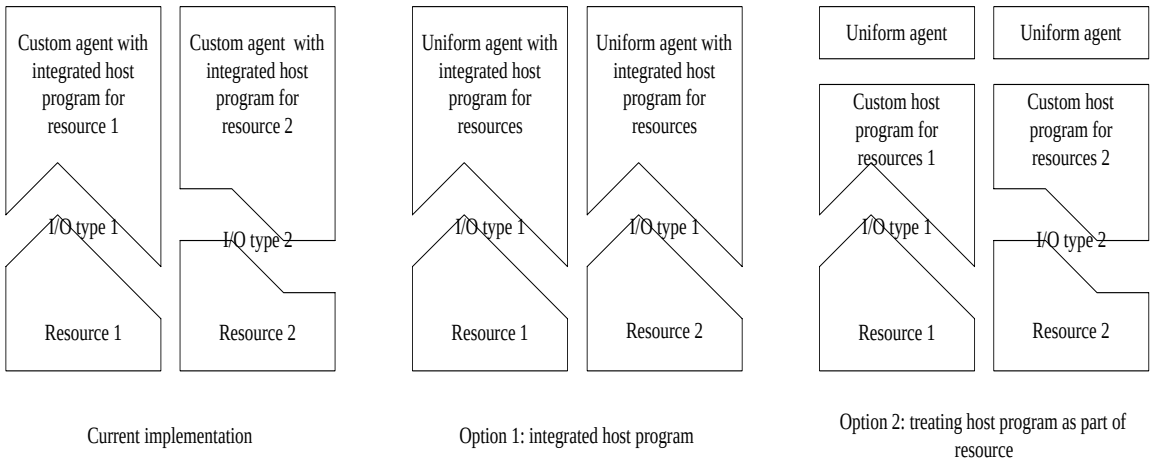


Figure 5.3: Standardizing I/O to enable resource exchange.

ent implementations. However, this implies restriction on hardware programming, and may result in less-than-optimum design for a given algorithm.

A second possible solution is to treat the host program as part of the resource. The resource now consists of the actual bitfile for the hardware and the host program written for the bitfile. Although this method enables use of a uniform agent program, this separates the agent from the host program and results in an extra communication link between the agent and the host. Also, the resource exchange now means transferring not only the hardware bitfiles, but also the host program, which increases the overall time required for resource exchange.

As seen, both methods have disadvantages such as restricting programming freedom and increasing communication. By combining these two methods as shown in Figure 5.4, it is possible to form a better solution. Although the communication between the host program and the hardware bitfile can vary, there are usually only a limited number of choices such as FIFO and memory. Therefore, it is possible to create the host program with different

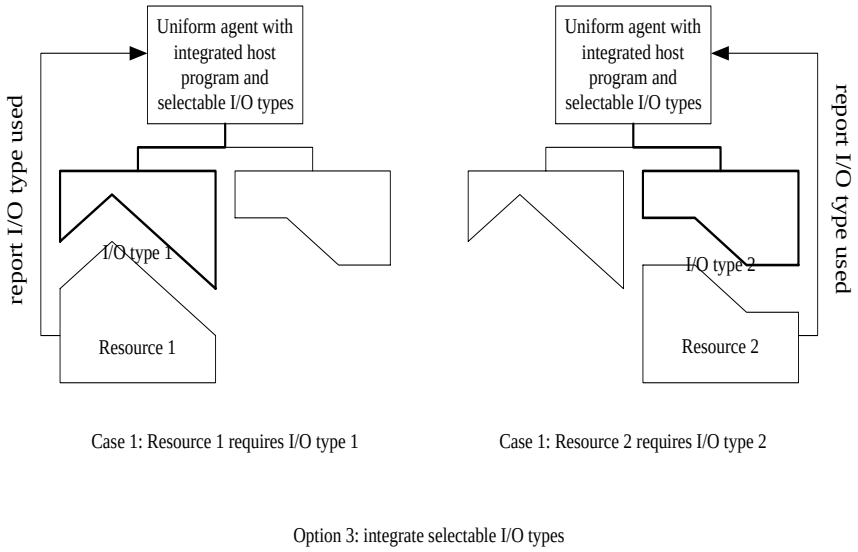


Figure 5.4: Hybrid method of standardizing I/O to enable resource exchange.

variations of the I/O methods built-in. This enables the integration of the agent and the host program while giving programmers the freedom to choose the I/O method. When the agent reads the resource information at startup, information on the I/O method is also read. And, when a resource is chosen for a simulation, the agent uses the appropriate I/O method for the given resource based on the information read at the startup. For example, the agent in Figure 5.4 is integrated with the host program with two different types of I/O methods for communication with the hardware resource. In case 1, resource 1 requires I/O type 1 and, therefore, the agent chooses I/O type 1 from its available built-in I/O methods. In case 2, resource 2 needs I/O type 2 to communicate with the host program, so the agent connects to the resource using I/O type 2. This hybrid method overcomes the disadvantages of the two methods described above by enabling a real-time change in the I/O interface.

5.5 Unique Identification of Objects

To perform resource exchange, the master must be able to distinguish different object types. When two agents have identical objects associated with them, the master must know that they are the same and that the resources can be shared. This can be accomplished by giving a unique identification (ID) number to each type of known object. By carefully assigning the numbers to each object type, the compatibility can be determined by the master using these ID numbers.

5.6 Using ACS API, MPI, and PVM

It is possible to have more than one reconfigurable board as one functional unit as in the Tower Of Power (TOP) [To02]. Then, the programmer can use the ACS API to program the multiple boards as one unit. In this case, this unit must be treated as one object, even though it consists of multiple boards. This can be an advantage if an algorithm needs to span multiple boards. However, clustering multiple boards as one unit prevents resource exchange and, therefore, may not be a desirable feature.

As discussed in Chapter 2 and Chapter 4, MPI and PVM are API functions for distributed heterogeneous programming. Since the ACS API uses MPI and the proposed middleware system is based on the heterogeneously distributed environment, applying MPI or PVM to the middleware system may be an option.

5.7 Data Format

Since the target application of the proposed middleware system is simulation of physical-layer communication systems, the data format for inter-agent communication should consider the needs of physical-layer communication systems. The middleware implementation shown in previous chapter features a data structure with two sets of two floating point numbers, one set for original data and another set for modified data, where each set consists of I-channel and Q-channel data. A *de facto* standard for data representation in client-server programming is called eXternal Data Representation (XDR) by Sun Microsystems [CoS97a]. An investigation on the application of XDR to the distributed reconfigurable simulator might be a good topic for future research.

5.8 Summary

In this chapter, a vision on the distributed reconfigurable simulator and a number of topics for possible future research has been presented. The distributed reconfigurable simulator has a potential of becoming a large-scale fast simulator with support for multiple concurrent simulations, where users can initiate the simulation from anywhere through a web-based user interface.

Chapter 6

Conclusion

This dissertation introduced the possibility of a new method of simulation using reconfigurable hardware and middleware for distributed simulation. A simple communication system was implemented to demonstrate the performance of communication system simulation using reconfigurable hardware. In Chapter 3, it has been shown that it is possible to implement various communication system components in reconfigurable hardware and run the simulation faster than traditional software simulations.

Another advantage of the reconfigurable hardware simulator is that it is more closely tied to a software radio implementation. As FPGAs replace or support the role of DSP within a software radio, the distinction between actual implementation and simulation will become smaller. This will clearly improve the quality and credibility of the simulation as it becomes more of a prototype than just a plain simulator.

Although reconfigurable hardware can be used for certain implementations of communication systems, there is a need for general-purpose processors and programmable DSPs. For example, a general-purpose processor is certainly better than reconfigurable hardware for implementing a user interface, and programmable DSPs generally perform sum-of-product (multiply-accumulate) operations faster than other processors. These various processing elements can be interconnected by using middleware.

Middleware can provide flexibility and expandability to the simulation platform while optimizing the performance of the overall simulation by mapping the required functions to appropriate processing elements. The functionality of middleware has been shown by interconnecting three different types of computing elements to form a distributed reconfigurable simulation platform. A communication system with a transmitter, channel, and a receiver has been successfully simulated using this simulation platform.

It has been identified that there are several performance variables in the middleware system. First, there are less significant variables such as number of simulation runs, network propagation time, and initial configuration time. Then, there are other variables that can affect the performance of the simulation, such as the communication time between an agent and its associated object, and processing time of the agent functions. However, there are several advantages of using a distributed environment. It is possible to interconnect various computing elements with ease. Legacy simulation modules can be integrated with new simulation modules. Hardware-software co-simulation using other simulation tools such as OPNET is possible. Also, it is possible to run a "hardware in the loop" simulation to test the functionality of modules such as software radio by providing channel models in hardware.

Possible future research to enhance the system has been discussed, which includes web-based user interface, multiple user support, real-time resource exchange, and so on. The

distributed reconfigurable simulation of communication systems has great potential, and this paper has presented this potential.

Bibliography

[Ac02] *http://www.ccm.ece.vt.edu/acs_api,* Web site of Adaptive Computing System Application Programming Interface.

[Al99] P. Alfke, "The Future of Field-Programmable Gate Arrays," *5th Workshop on Electronics for LHC Experiments,* 1999.

[Al02] *http://www.altera.com,* Web site of Altera Corporation.

[An98] *WildForce manual,* Annapolis Microsystems Inc., 1998

[Ar99] Ken Arnold, "The Jini Architecture: Dynamic Services in a Flexible Network," *Proceedings of 36th Design Automation Conference,* pp. 157-162, 1999.

[ArG93] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL,* Prentice Hall, 1993.

[Ba00] P.M. Baggenstoss, "A modified baum-welch algorithm for hidden Markov models with multiple observation spaces," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* pp. 717-720, 2000.

[Bo96] P. Bonner, *Network Programming with Windows Sockets,* Prentice Hall, 1996.

[BuJ98] A. Buss and L Jackson, "Distributed simulation modeling: a comparison of HLA, CORBA, and RMI," *Winter Simulation Conference Proceedings,* pp. 819-825, 1998.

[ChM99] P. Chaudhury, W. Mohr, and S. Onoe, "The 3GPP Proposal for IMT-2000," *IEEE Communications Magazine,* vol. 37, no. 12, pp. 72-81, December 1999.

[ChP00] F. G. Chatzipapadopoulos, M. K. Perdikeas, and I. S. Venieris, "Mobile Agent and CORBA Technologies in the Broadband Intelligent Network," *IEEE Communications Magazine,* vol. 38, no. 6, pp. 116-124, June 2000.

[Co02] *http://www.omg.org/gettingstarted/corbafaq.htm,* Web site for OMG Corba Basics.

[CoS97a] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, Volume 3, Linux/POSIX Sockets Version,* Prentice Hall, 1997.

[CoS97b] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP, Volume 3, Windows Sockets Version,* Prentice Hall, 1997.

[CuH99] Mark Cummings and Shinichiro Haruyama, "FPGA in the software radio," *IEEE Communications Magazine,* vol. 37, no. 2, pp. 109-112, February 1999.

[Da93] R. Davis, *Windows Network Programming,* Addison-Wesley, 1993.

[De02a] R. Dettmer, "Unlocking the power of the Grid," *IEE Review,* vol. 48 no. 3, pp. 9-12, May 2002.

[De02b] *http://www.opengroup.org/dce,* Web site of Open Group Distributed Computing Environment.

[Du95] A. Dumas, *Programming WinSock,* Sams Publishing, 1995.

[Em97a] Wolfgang Emmerich, "An Overview of OMG/CORBA," *IEE Colloquium on Distributed Objects - Technology and Application,* pp. 1-6(Digest No: 1997/332), 1997.

[Em97b] Wolfgang Emmerich, "An Introduction to OMG/CORBA," *Proceedings of the 1997 International Conference on Software Engineering,* pp. 641-642, 1997.

[FaL00] G. Fadlallah, M. Lavoie, and L.-A. Dessaint, "Parallel Computing Environments and Methods," *International Conference on Parallel Computing in Electrical Engineering,* pp. 2-7, 2000.

[FaW96] B. K. Fawcett, and J. Watson, "Reconfigurable processing with field programmable gate arrays," *Proceedings of 1996 International Conference on Application Specific Systems, Architectures and Processors,* pp. 293-302, 1996.

[Fu01] R. M. Fujimoto, "Parallel and Distributed Simulation Systems," *Proceedings of the 2001 Winter Simulation Conference,* vol. 1, pp. 147-157, 2001.

[Ge02] W. Gentzsch, "Grid Computing, a Vendor's Vision," *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid,* pp. 290-295, 2002.

[GeK96] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, *PVM and MPI: a Comparison of Features,* Calculateurs Paralleles, vol. 8, no. 2, 1996.

[Gr02a] *http://dsonline.computer.org/gc,* Web site of IEEE Distributed Systems: Grid Computing.

[Gr02b] *http://www.gridcomputing.com,* Web site of Grid Computing Info Center.

[Gu02] P. Gupta, "Hardware-software codesign," *IEEE Potentials,*, vol. 20, no. 5, pp. 31-32, December 2001-January 2002

[Ha01] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," *Proceedings of Design, Automation and Test in Europe,* pp. 642-649, 2001.

[HjN98] H. Hjalmarsson, B. Ninness, "Fast, non-iterative estimation of hidden Markov models," *IEEE International Conference on Acoustics, Speech and Signal Processing,* pp. 2253-2256, vol. 4, 1998.

[HuW00] J. F. Huber, D. Weiler, and H. Brand, "UMTS, the mobile multimedia vision for IMT 2000: a focus on standardization," *IEEE Communications Magazine,* vol. 38, no. 9, pp. 129-136, September 2000.

[JeB92] M. C. Jeruchim, P. Balaban, and K. S. Shanmugan, *Simulation of Communication Systems,* Plenum Press, New York, 1992.

[Ji02a] *http://www.sun.com/jini,* Web site for Jini Network Technology.

[Ji02b] *http://www.javacoffeebreak.com/books/extracts/jini/Corba.html,* CORBA and Jini.

[JoS99] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for distributed adaptive computing systems," *Proceedings of Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 220-230, 1999.

[KhH95] A. M. Khandker, P. Honeyman, T. J. Teorey, "Performance of DCE RPC," *Second International Workshop on Services in Distributed and Networked Environments,* pp. 2-10, 1995.

[KoC02] G. Koutsogiannakis and J.M. Chang, "Java distributed object models: an alternative to Corba?," *IT Professional,* pp. 41-47, vol. 4, no. 3, May-June 2002.

[Ma00] Qusay H. Mahmoud, "Using Jini for High-Performance network Computing," *Proceedings of International Conference on Parallel Computing in Electrical Engineering,* 2000.

[Ma02a] E. P. Markatos, "Speeding up TCP/IP: faster processors are not enough," *21st IEEE International Conference on Performance, Computing, and Communications,* pp. 342-345, 2002.

[Ma02b] *http://www.mathworks.com/access/helpdesk/help/techdoc/ref/uicontrol.shtml,* Web site describing a Matlab command *uicontrol*.

[Mi02] *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/comext_3aw3.asp,* Web site for Microsoft RPC.

[Mi98] T. Miyazaki, "Reconfigurable systems: a survey," *Proceedings of 1998 Design Automation Conference,* pp. 447-452, 1998.

[Na00] S. Narayanan, "Web-based modeling and simulation," *Proceedings of 2000 Winter Simulation Conference,* vol. 1, pp. 60-62, 2000.

[Op02a] *http://www.opnet.com,* OPNET Technologies homepage.

[Op02b] "Session 1532 Interfacing Multiple Simulators Using OPNET Modeler's Cosimulation API," OPNETWORK 2002, 2002.

[Os02] *http://www.ccm.ece.vt.edu/ wworek/doc/Osiris_VHDL_Guide.pdf,* Osiris Board Architecture and VHDL Guide.

[PaC96] Jagdish K. Patel and Campbell B. Read, *Handbook of the normal distribution,* Marcel Dekker, New York, 1996.

[PaR98] N. Padovan, M. Ryan, and L. Godara, "An Overview of Third Generation Mobile Communications Systems: IMT-2000," *IEEE Region 10 International Conference on Global Connectivity in Energy, Computer, Communication and Control,* pp. 360-364 vol. 2, 1998.

[PrT92] Press, Teukolsky, Vetterling, and Flannery, *Numerical Recipes in C: The Art of Scientific Computer,* Cambridge University Press, 1992.

[PuF92] M. K. Purvis and D. W. Franke, "An Overview of Hardware/Software Codesign," *Proceedings of IEEE International Symposium on Circuits and Systems,* vol. 6, pp. 2665-2668, 1992

[Pv02] *http://www.netlib.org/mpi/pvmpi/pvmpi.html,* Web site of PVMPI: PVM integrating MPI applications.

[QuS96] B. Quinn and D. Shute, *Windows Sockets Network Programming,* Addison-Wesley, 1996.

[ReG99] M. Resch, E. Gabriel, and A. Geiger, "An Approach for MPI Based Metacomputing," *The Eighth International Symposium on High Performance Distributed Computing,* pp. 333-334, 1999.

[ScK00] D. C. Schmidt, and F. Kuhns, "An Overview of the Real-Time CORBA Specification," *Computer,* vol. 33, no. 6, pp. 56 -63, June 2000.

[ScR98] S. Schulz, J. W. Rozenblit, M. Mrva, and K. Buchenriede, "Model-Based Codesign," *Computer,* vol. 31, no. 8, pp. 60-67, Aug. 1998

[Si93] A. K. Sinha, *Network Programming in Windows NT,* Addison-Wesley, 1993.

[SkD95] A. Skjellum, N.E. Doss, K. Viswanathan, A. Chowdappa, and P.V. Bangalore, "Extending the Message Passing Interface (MPI)," *Scalable Parallel Libraries Conference,* pp. 106-118, 1994.

[Sm00] N. Smavatkul, "Range Adaptive Protocols for Wireless Multi-Hop Networks," Ph.D. Dissertation, Virginia Polytechnic Institute and State University, 2000.

[Sp02] *http://www.cadence.com/products/spw.html,* Signal Processing Worksystem homepage.

[Sr99] S. Srikanteswara, "Design and Implementation of a Soft Radio Architecture for Reconfigurable Platforms," Ph.D. Dissertation, Virginia Polytechnic Institute and State University, 1999.

[Sy02a] *http://www.synopsys.com/products/fpga/fpga_solution.html,* Synopsis FPGA solutions homepage.

[Sy02b] *http://www.synplicity.com/products/synplifypro/index.html,* Synplify homepage.

[TI02] *http://www-s.ti.com/sc/ds/tms320c6701.pdf,* Texas Instrument TMS320C6701 Digital Signal Processor.

[To02] *http://www.ccm.ece.vt.edu,* Tower of Power homepage.

[TrK94] W. H. Tranter and K. L. Kosbar, "Simulation of Communication Systems," *IEEE Communications Magazine*, vol. 32, no. 7, pp. 26-35, July 1994.

[TrS02] W. H. Tranter, K. Sam Shanmugan, T.R. Rappaport, and K. L. Kosbar, *Computer-Aided Design and Analysis of Communication Systems with Application to Wireless Systems,* Prentice Hall, 2002 (anticipated publication date).

[Vi97] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Commnucations magazine,* vol. 35, no. 2, pp. 46-55, February 1997.

[Vi02] S. Vinoski, "Where is is middleware," *IEEE Internet Computing,* vol. 6, no. 2, pp. 83-85, March-April, 2002.

[Wi95] S. B. Wicker, *Error Control Systems for Digital Communication and Storage,* Prentice Hall, 1995.

[Wo02] *http://www.prenhall.com/workview,* Workview Office homepage.

[Xi02] *http://www.xilinx.com,* Web site of Xilinx Corporation.

[Ya00] K. Yao, "Implementing an Application Programming Interface for Distributed Adaptive Computing Systems," Master's Thesis, Virginia Polytechnic Institute and State University, 2000.

VITA

Song Hun Kim was born and raised in Seoul, Korea. He moved to the United States of America with his mother in 1989, and graduated from Konawaena High School in Kealakekua, Hawaii, where he won first place in the Hawaii District Computer Programming Contest. He received the Bachelor of Science in computer and systems engineering from Rensselaer Polytechnic Institute in May, 1995, where he graduated first rank in his class with *Summa Cum Laude* honors. He received the Master of Science in electrical engineering from Virginia Tech in May, 1997, and continued his graduate work to pursue a Ph.D. degree. He is currently affiliated with the Mobile and Portable Radio Research Group (MPRG), and is a fellow funded by the National Science Foundations Integrative Graduate Education and Research Training (IGERT) program.

He has worked as an intern at numerous companies including the Wireless Product Group of Lucent Technologies, the R&D Laboratory of DACOM Communications, and the Communication R&D Center of Samsung Electronics. He has also copyrighted a mathematical graphing software package.

Publication

S. H. Kim, W. H. Tranter, and S. F. Midkiff, "Design of Middleware for Distributed Reconfigurable Simulation for Communication Systems," *The 35th Annual Simulation Symposium,* pp.253-258, April 2002.