Väinö Helminen

Distributed Discrete Time Network Simulator

Master of Science Thesis

Subject approved by Department Council
October 13th, 2004
Supervisors: Prof. Tommi Mikkonen
Prof. Timo D. Hämäläinen

Preface

I would like to thank prof. Timo D. Hämäläinen for guidance and my colleagues for their help. Especially Jouni Riihimäki gave valuable advices. Special thanks to my wife Elina for her support.

Tampere, April 25th, 2010

Väinö Helminen

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
Faculty of Computing and Electrical Engineering
Department of Computer System
Helminen, Väinö: Distributed Discrete Time Network Simulator
Master of Science Thesis, 50 pages
Supervisors: Prof. Tommi Mikkonen and Prof. Timo D. Hämäläinen
Funding: Nokia Mobile Phones
June, 2010
Keywords: Parallel and Distributed Simulation, System Design, Verification

The Discrete Time Network Simulator (DTNS) is a System-on-Chip (SoC) simulator developed at Tampere University of Technology. It is used to analyze interconnection architectures and systems built around them. The abstraction level is between the conventional hardware simulators, such as Mentor Graphics ModelSim, and algorithm level simulators, such as Synopsys System Studio.

DNTS makes it possible to get cycle-accurate information about the communication while other high-level tools often lack timing completely. This is because DTNS is a time-driven simulator where the simulated time advances in fixed increments of half a clock cycle and the system bus is always simulated at that level of detail.

The simulator itself is programmed in C and the system design is described in C or C++ programming language with detail level from high level functional code to almost hardware description language level code. The accuracy of the simulation increases as the model is refined. However, this also makes the simulation times longer. The goal of this thesis work was to develop a distributed version of DTNS as a remedy.

The emphasis on the system bus made it a natural point of partitioning. The simulator was split to a central core process and separate processes for system component models which can then be executed in parallel. Processes communicate any writes to the system bus to the core process which then accumulates them and communicates changes to all other

processes. At the same time all processes synchronize with the core process after every simulation step.

For communication between processes executed on different computers, it was originally given as a premise that Common Object Request Broker Architecture (CORBA) should be used. However, the amount of overhead was found to be too significant and another implementation that communicated directly over the TCP/IP protocol was created, too.

For performance testing a statistical model of a H.263 video encoder was used. The model was instrumented to mimic different complexity levels with artificial delays. Multiple simulations was then executed using both communication implementations while varying the delay gradually from very high level and fast model to very complex and slow. Also, the number of computers was varied from one to three.

The measured wall clock times of these simulations clearly show the high overhead of CORBA in comparison to TCP/IP. Both implementation were able to speed-up the simulation as the models became slower. The performance of the TCP/IP implementation seems rather impressive.

The distribution method of DTNS was also used to distribute a commercial simulator, ModelSim. The system consisted of two to eight TUTWLAN terminal and this was distributed up to eight simulators executed in parallel with signals passed between them using TCP/IP protocol. The simulation times show that this method is capable of significant speed-up even in real world simulations if the system model is in fine enough detail.

In conclusion the distribution of DTNS is not very useful in real life as the models are unlikely to be slow enough to see any speed-up. This new version of DTNS is, however, also capable of parallel execution on a single computer with, for example, a multi-core processor and without network overhead simulation times can be improved noticeably even for higher level models.

Tiivistelmä

TAMPEREEN TEKNILLINEN YLIOPISTO
Tieto- ja sähkötekniikan tiedekunta
Tietokonetekniikan laitos
Helminen, Väinö: Distributed Discrete Time Network Simulator
Diplomityö, 50 sivua
Tarkastajat: Prof. Tommi Mikkonen ja Prof. Timo D. Hämäläinen
Rahoittajat: Nokia Mobile Phones
Kesäkuu 2010
Avainsanat: Rinnakkainen ja hajautettu simulaatio, järjestelmäsuunnittelu, varmennus

Discrete Time Network Simulator (DTNS) on System-on-Chip (SoC) -simulaattori, joka on kehitetty Tampereen teknillisessä yliopistossa. Se on tarkoitettu yhteysarkkitehtuurien ja niiden ympärille rakennettujen järjestelmien analysointiin. Sen abstraktiotaso on tavanomaisten laitteistosimulaattorien, kuten Mentor Graphics ModelSim, ja algoritmitason simulaattorien, kuten Synopsys System Studio, väliltä.

DTNS:lla on mahdollista saada ajoitustietoja järjestelmäväylällä tapahtuvasta liikennöinnistä kellojakson tarkkuudella, kun taas useilla muilla korkean tason työkaluilla ei saada ollenkaan ajoitustietoja. Tämä johtuu siitä, että DTNS on aikaan perustuva simulaattori, jossa simuloitu aika etenee vakiomittaisina, puolen kellojakson hyppäyksinä ja väylä simuloidaan aina tällä tarkkuudella.

Simulaattori on ohjelmoitu C-kielellä ja mallinnettava järjestelmä voidaan kuvata joko C tai C++ -kielillä. Kuvaus voidaan tehdä hyvin korkealla tasolla tai jopa lähes laitteistonkuvauskielen tasolla. Simulaation tarkkuus tietysti paranee mitä alhaisemman tason kuvausta käytetään. Tämä kuitenkin hidastaa simulaatiota. Tämän työn tarkoitus olikin nopeuttaa simulaatiota luomalla DTNS:stä hajautettu versio.

Painopiste järjestelmäväylän simuloinnissa määritti luonnollisen osiointikohdan hajautusta ajatellen. Simulaattori hajotettiin ydinprosessiin ja komponenttimalleja suorittaviin prosesseihin. Komponenttiprosessit kertovat ydinprosessille mitä ne haluavat kirjoittaa väylälle ja ydinprosessi yhdistää saamansa tiedot ja välittää muutokset muille prosesseille.

Samalla kaikki prosessit synkronoidaan ydinprosessin kanssa jokaisen simulaatioaskeleen jälkeen.

Eri koneilla suoritettavien prosessien väliseen kommunikointiin oli alun perin tehtävänannossa määritelty käytettäväksi Common Object Request Broker Architecturea (CORBA). Sen aiheuttama rasite todettiin kuitenkin niin huomattavaksi, että hajautus toteutettiin myös käyttämällä suoraan TCP/IP-protokollaa.

Suorituskyvyn mittaamista varten käytettiin tilastollisesti H.263-videonpakkausta mallintavaa järjestelmää, johon lisättiin mahdollisuus lisätä keinotekoista viivettä eri abstraktiotasojen mallintamiseksi. Tästä suoritettiin useita simulaatioita käyttäen molempia hajautustoteutuksia. Simulaatioissa vaihdeltiin asteittain sekä keinotekoista viivettä nopeasta hyvin korkean tason kuvauksesta hitaaseen hyvin matalan tason kuvaukseen, että käytettyjen tietokoneiden määrää yhdestä kolmeen.

Mitatut suoritusajat osoittavat selvästi CORBA:n suuren rasitteen TCP/IP:hen verrattuna. Molemmilla toteutuksilla saatiin silti simulaatiota nopeutettua, kun mallien suoritusajat kasvoivat. TCP/IP-toteutuksen suorituskyky näyttää melko vaikuttavalta.

DTNS:n hajautustapaa käytettiin myös kaupallisen laitteistosimulaattori ModelSimin hajauttamiseen. Järjestelmä koostui kahdesta kahdeksaan TUTWLAN-päätettä ja nämä hajautettiin jopa kahdeksaan eri simulaattoriin, joita suoritettiin rinnakkain TCP/IP-protokollan välittäessä signaaleita niiden välillä. Simulaatio osoitti, että käytetyllä hajautustavalla voidaan saada huomattava nopeutus jopa todellisissa simulaatioissa kunhan mallinnus on tehty tarpeeksi tarkasti.

Yhteenvetona voidaan todeta, että hajautettu DTNS ei ole kuitenkaan käytännössä kovin hyödyllinen, koska mallit eivät todennäköisesti ole riittävän hitaita. Uudella DTNS:lla on kuitenkin mahdollista suorittaa simulaatio rinnakkaistettuna yhdellä, esimerkiksi moniydinprosessorilla varustetulla, tietokoneella ja ilman verkon rasitetta simulaatioajat nopeutuvat huomattavasti jopa korkeamman tason malleilla.

Table of Contents

List of Abbreviations

API	Application Programming Interface
BEEP	Blocks Extensible Exchange Protocol
C	C Programming Language
C++	C++ Programming Language
CORBA	Common Object Request Broker Architecture
DTNS	Discrete Time Network Simulator
FIFO	First In First Out
FLI	Foreign Language Interface
FTP	File Transfer Protocol
HDL	Hardware Description Language
HIBI	Heterogeneous IP Block Interconnection
HTTP	Hypertext Transfer Protocol
IDL	Interface Description Language
IIOP	Internet InterORB Protocol
ILU	Xerox Inter-Language Unification
IP	Internet Protocol
IP	Intellectual Property
JNI	Java Native Interface
JVM	Java Virtual Machine
LAN	Local Area Network
NOW	Network Of Workstations
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
PBD	Platform-Based Design
PPP	Point-to-Point Protocol
RLE	Run Length Encoding
RPC	Remote Procedure Call

SMTP	Simple Mail Transfer Protocol
SoC	System-on-Chip
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol over Internet Protocol
UDP	User Datagram Protocol
WWW	World Wide Web
XDR	eXternal Data Representation

1 Introduction

The functionality of a hardware design is usually confirmed with simulations. Finding and fixing possible design errors and bottlenecks is easier, quicker, and also cheaper at an early design phase. Therefore, there is a need for a simulation tool that can be used as early as possible in the design flow. Previously there has been a lack of tools for hardware simulator that can give exact information about the system in an early design phase. The Discrete Time Network Simulator (DTNS) [1] was developed at Tampere University of Technology to fill this need. With DTNS it is possible to get cycle-accurate information about the communication between system component even with very high-level models.

However, with more detailed simulation models the simulation times of a complex system tend to increase to an intolerable level. The goal of this thesis work was to develop a distributed version of the DTNS to remedy the problems of overlong simulation times. The idea is to find tasks that can be executed in any order, or, indeed, in parallel, without altering the result. Then distributing these tasks to multiple computers to be executed in parallel and, thus, reduce the overall wall-clock time needed for the simulation.

When the work was started on this thesis, it was given as a premise that Common Object Request Broker Architecture (CORBA) should be used for inter-process communication. However, when working with CORBA version the amount of overhead involved with it was noticed. Therefore, it was decided to use an alternative communication method, also. To reduce the communication overhead near to minimum, TCP/IP was used directly. Later the results achieved with both of these version was compared.

The distribution method developed in this thesis for DTNS was also used to distribute a commercial Hardware Description Language (HDL) simulator, ModelSim. A system modeled in VHDL is partitioned to suitable size parts which are simulated on different computers. The simulated parts communicate using TCP/IP with the aid of a simple Foreign Language Interface (FLI) block.

The structure of this thesis is following. Parallel and distributed computing and problems involved are discussed in Chapter 2. Also specific problems and possible solutions of parallel simulation are discussed.

In Chapter 3 some communication methods available in the target environment for this thesis work are introduced. The levels of abstraction and the overhead associated with each method are discussed and compared.

The DTNS is introduced in Chapter 4. It is explained how the simulator works and how it is build. Also the hardware design flow with DTNS is discussed.

The distribution of DTNS is explained in Chapter 5. It is shown how the simulator was parallelized and how the distribution is handled using the CORBA remote object abstraction and with a lower level TCP/IP byte stream abstraction. Finally it is shown how the distribution method developed for DTNS was applied to a commercial VHDL simulator.

Chapter 6 describes the test cases used for evaluation and shows the results. Two test cases for DTNS were used. One is more theoretical and a second is a real-life simulation of a video codec. Also test cases for the distributed VHDL simulation and the results are presented here.

The thesis is concluded in Chapter 7 with the conclusion that the simulator was successful and it would be beneficial to use DTNS in the future.

2 Distributed Computing

Despite the fast development in processor speeds, a single processor is still not fast enough for major computation within reasonable time. The amount of data may also be so huge that the memory available in a single computer is not enough. However, these massive computations must be carried out and they can be done with distribution, where several processors are computing at the same time for a common goal. Another reason for distribution is to maximize the availability of services and data.

2.1 Background

Traditionally heavy computing must have been done with expensive parallel computers (also called as supercomputers) that can have thousands of processors [2, 3]. Load balancing is a major concern because synchronization causes the faster processes to waste processing time. Time is either spend waiting for a slower process to progress, or rolling back to a previous state as a result of receiving a message too late, and therefore, doing some of the processing all over again. In traditional parallel computers, communication overhead is usually fairly small because inter-process communication can be handled using shared memory, which is just like ordinary memory but can be accessed by more than one processor.

However, the expenses related to parallel computers often render them out of reach. As a remedy, there has been a lot of interest in networks of workstations [4]. A network of workstations (NOW) has significantly lower cost because ordinary desktop PCs can be used to build them. Modern PCs, connected with a high-speed network, are capable of serious computation with performance comparable to super-computers with only a fraction of the cost. However, the network connecting the PCs is fairly slow compared to shared memory. This makes the communication overhead worse by an order of magnitude compared to parallel computers. Therefore, successful distribution requires more sophisticated distributed algorithms, which can tolerate communication delays better.

For less serious computation the same workstations used in ordinary office work during the daytime can be used for distributed computation during the night, which further lowers the cost because no additional hardware is required. With this approach the possible heterogeneity of the environment can cause problems and must be dealt with. For example workstations can be running different operating systems, they most likely are of different speeds or have completely different architectures. All this makes the distribution even more difficult.

Despite the difficulties there are an increasing number of reports of successful NOW computation projects. For example, the Google Internet search engine is estimated to be powered by hundreds of thousands of Linux servers. Also many of the big Hollywood movies have had their special effects rendered by farms of cheap PCs, for example, the Lord of the Rings trilogy. The well-known SETI@home project [5] is an example of successful use idle processor cycles of desktop PCs for scientific computation. There are currently over 5,000,000 participants and it is in fact already the largest computation ever done (measured in floating point operations), and it is still going on.

2.2 Partitioning

Partitioning means the way a complex task is divided for the available computers or processors in case of a multiprocessor computer. The workload of each partition and their communication needs must be taken into account. To fully take advantage of the available hardware the workload of each partition should match the performance of the computer.

Or, in case of equal computers, the partitions should have equal workloads, too. The frequency of communication and size of messages needed between partitions can also be a determining factor. Synchronization needs of possible partitions also affects the frequency of communication needed.

In ray tracing the pixel values of the final image can be calculated without knowing anything about the surrounding pixels. Therefore, partitioning is a simple case of giving each processor a certain amount of pixels to render and then combine these pixels to form the final image. With animation each computer can be given a range of frames it should render and the frames can be combined later to complete the animation. Because different frames can be rendered independently, there is no need to synchronize after every frame to combine pixels.

All processors do not necessarily carry out a similar task like in the ray-tracing example. Instead it is common that every processor has a different task and probably there are few different tasks. Tasks can be quite different and have varying needs for local and shared memory, for example.

However, in general, the distribution is not an easy task to do. In fact, developing distributed algorithms requires particular expertise, and sometimes the computation of an algorithm can be impractical to implement, although theoretically possible. The main reason for this is that the communication overhead will become larger than the gain from parallel computation.

2.3 Communication

When designing distributed applications one must choose a networking model by which the different processes of the application communicate. The basic models to choose from are server-client model [6], peer-to-peer model [7] and master-slave model.

Traditionally distributed applications have been designed to follow the server-client model. In the server-client model there are one or more servers that usually do nothing but wait requests from the clients and act upon them, and then there are clients who ask

the server to do them services. The World Wide Web (WWW) is an example of the server-client model where web servers offer the content and browsers request the offered content based on the actions of the users.

In server-client model all communication is always between a server and a client and it is initiated by the client. Clients never communicate directly with other clients. In larger applications servers can, however, act as clients to other servers. It is possible that one of the servers becomes a bottleneck for the whole systems performance.

Lately there has been a lot of discussion about the peer-to-peer model, mostly related to the illegal file sharing over the Internet. There is also a lot of interest to this model in the scientific world as there are no servers to cause bottlenecks. In peer-to-peer model all participants are equal and can communicate with whomever they need to. Processes can of course be grouped to neighborhoods or layers where communications between different neighborhoods are handled by only a few processes and others can only communicate within their own neighborhood.

In master-slave model, after the master-slave relationship is established, one master controls the communication of one or more slaves. The master-slave model is more common in hardware design and rarely seen in software.

In practice, the need for performance and availability can cause the server-client model to resemble the peer-to-peer model and vice versa because of the difficulty of distributing algorithms. The servers might often have to communicate with each other, for example a distributed database must do this to keep the different copies of the data consistent. In the peer-to-peer model some process can become responsible of synchronizing the whole execution and, therefore, act very much like a server. However, it is usual that the synchronizing of similar tasks is also distributed so that the process doing it changes over time. This leads to a more fault tolerant overall system.

The physical layer used for communication can be any network or point-to-point link available. Ethernet networks are common and cheap network used today, which are used, for example, to build the local area networks (LAN) in most, if not all, offices. Larger distributed applications can use the whole Internet, and therefore, any IP compatible

network. IP protocol is often used even with LANs. In chapter 3 IP and some higher level protocols are look into.

2.4 Distributed Simulation

Physical behavior is usually simulated using a discrete event or discrete time model with the former being more common. The difference is that in discrete event simulation events can happen at any point in simulated time and simulated time jumps from one event to the next in order. On the other hand, in discrete time simulation time advances in constant discrete step and simulation events are aligned those steps. Both methodologies have been used in distributed simulations. Distributed discrete event simulations are presented in [8, 9] and a distributed discrete time simulation in presented is this work.

The correctness of distributed simulation can be ensured by using either conservative or optimistic evaluation [10]. The conservative approach will let the simulation advance only when it is absolutely certain that no causality violation will occur. Whereas, the optimistic approach allows parts of the simulation to proceed on their own. If a causality violation is later detected, the simulation is rolled back to a known good state before the conflict. After that, the simulation is proceeded in a way that avoids the problem.

3 Communication Methods

There are many ways for computer programs to communicate over a network. In this Chapter the following widely available and used methods for inter-process communication, TCP/IP, UDP/IP, RPC, shared file system, CORBA, and Java RMI, are presented.

All the presented methods are available for workstations with any common networking capable operating system. But not all these methods are usable for embedded systems because the overhead they impose is too high. The basic low-level protocol for all of these is the Internet Protocol (IP) [11].

This thesis concentrates on IP networking because it is most widely used. Other low level protocols are, for example, AppleTalk and IPX. In a workstation environment the network utilized is nowadays usually Ethernet. Other possible networks could be LocalTalk and Token Ring. There are several versions of the Ethernet which differ by their speed and have different physical cabling and small differences in the protocol. Hardware is, however, usually downward compatible, if the physical layer has not changed. For example, the gigabit Ethernet interfaces, which has started to become commonplace in PC, are compatible with 100 and 10 megabit Ethernets, but of course, then operate at the lower speed.

The focus will be on the functionality and any security implications are ignored. In general applications made with any of these methods can be made reasonably secure with

proper care, or additional layers such as using the Secure Socket Layer (SSL), even in untrusted networks. The SSL can be used to add authentication and encryption to most TCP/IP protocols, but secure versions of common protocols are readily available. All the presented methods are available to most common operating systems of desktop PCs.

3.1 Sockets

The simplest form of inter-process communication in an IP network is realized with sockets, which are the application programming interface (API) for Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) protocols. Sockets API also provide access to UNIX sockets, which can be used within a single computer for inter-process communication.

3.1.1 Background

The socket API was originally introduced in the BSD UNIX [12] but the growth of the Internet has made clones of it available at practically every network-capable computer. There are some rather small differences, however, between different operating systems, which makes portability an issue. The language is not a liming factor as sockets can be used with programming languages ranging from UNIX shell scripts to Java, and the differences between languages are often surprisingly small.

If different computer architectures or operating systems will be used, their differences must be taken into account. This situation can be handled by specifying the data formats in detail. It is important to specify the byte ordering and word size for binary data and the used character set for text-based communication, for example. Programmers must be very careful with these things, too, if the code planned to be portable.

A socket is identified by a socket descriptor that can be used like a regular file descriptor on a UNIX system. A socket (or file) descriptor is basically an integer that identifies a socket (or file) when making system calls. On non-UNIX systems the API for using sockets is mostly the same but socket descriptors are not necessarily compatible with file descriptors.

TCP and UDP provide only a mean to transport raw data between applications. Usually at least some small application specific protocol on top of them is required. In a more complex application, the custom made protocol would became so complicated that it is safer to use one of the more advanced methods for inter-process communication just to ensure that the communication layer actually works correctly and also to save development time.

For identifying processes, both TCP and UDP have the concept of ports. This is needed because an IP address only identifies the network interface of a computer. The port itself is an unsigned 16-bit integer and the numbers are assigned independently for both protocols.

On UNIX-like systems port numbers from 0 to 1023 are reserved for system services executed with administrator permissions, the rest are available for everyone to use. A port is allowed to be used by only one process at a time, and after a port is released, or closed, there is a timeout before it can be re-opened to reduce the change that the new process is confused with the previous one. Reserved port numbers are assigned by Internet Assigned Numbers Authority [13].

A port is reserved with the API by opening a socket. The actual port number is not usually important, but the server port must be one that all possible clients know, or can find out from some kind of name service. If the port is not specified when a socket is opened, it can later be requested from the operating system using the API.

The layers needed for TCP and UDP are shown in Figure 1. Ethernet is used as an example here because it is widely used in office LANs today. Other possible lowest level protocol could be Point-to-Point Protocol (PPP), which is used with serial lines and modems.

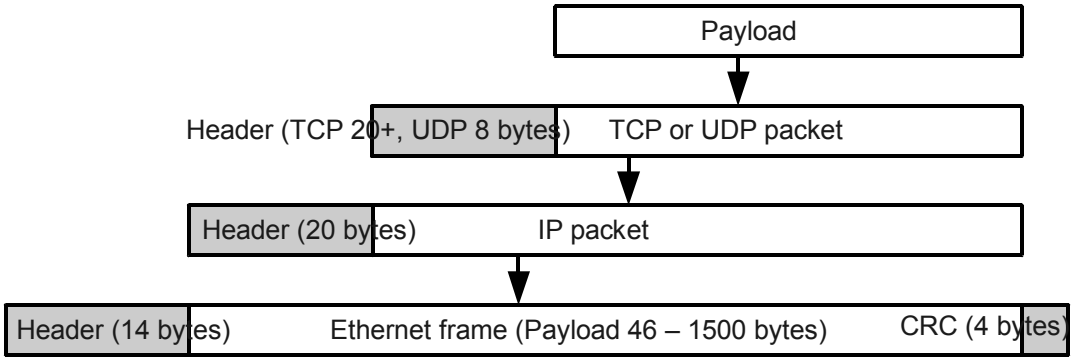

Figure 1. *Protocol layers for TCP and UDP over IP in an Ethernet LAN.*

3.1.2 TCP/IP

Transmission Control Protocol (TCP) [14] is the standard protocol that is used for most of the data transmissions over the Internet. It provides a two-way byte stream between two processes running on computers connected by an IP network, or on the same computer. Any data written by a process to the socket can be read by the process on the other end of the connection. In more detail, any data sent is guaranteed to eventually arrive to the receiver intact. No data will be lost and bytes arrive in the same order they were sent, unless the connection is broken before transmission is complete. There is, however, no guarantee on how long the transmission takes.

Behind the scenes TCP handles segmentation of the data and recovery from network errors, such as packet lost, duplicate packets, and packets arriving in the wrong order. Of course if the network is broken between the computers, no more data can be transmitted and the connection is closed. In such a case it might not be known how much of the data sent was received.

TCP/IP connections always has the concepts of a servers and clients. Nevertheless, this does not prevent its use as a lower level protocol under a peer-to-peer protocol. An example case of opening a TCP/IP connection is shown in Figure 2. A server opens a port for anyone to connect to, and a client opens a connection to a known server port. After a connection is made, there is no further difference between the server and the client. However, the server can accept more than one connection to the same server port.

The most complicated part of using TCP sockets is making connections. Every connection has two ends, the client and the server. Opening them is quite different. When opening the server one must first open a server socket that reserves a server port from the host computer to which the clients can then connect. After that, server waits for connections to the server port and opens a new socket for accepted connection. The server socket can be kept open for later connections from the same or other clients.

The client opens a connection by simply creating a new socket and connecting it to the server by giving its IP address and server port. If one wants the client to take the connection from a particular port, the socket can be bound to a port. However, this is not usually necessary. The server is often specified by a human readable name instead of an

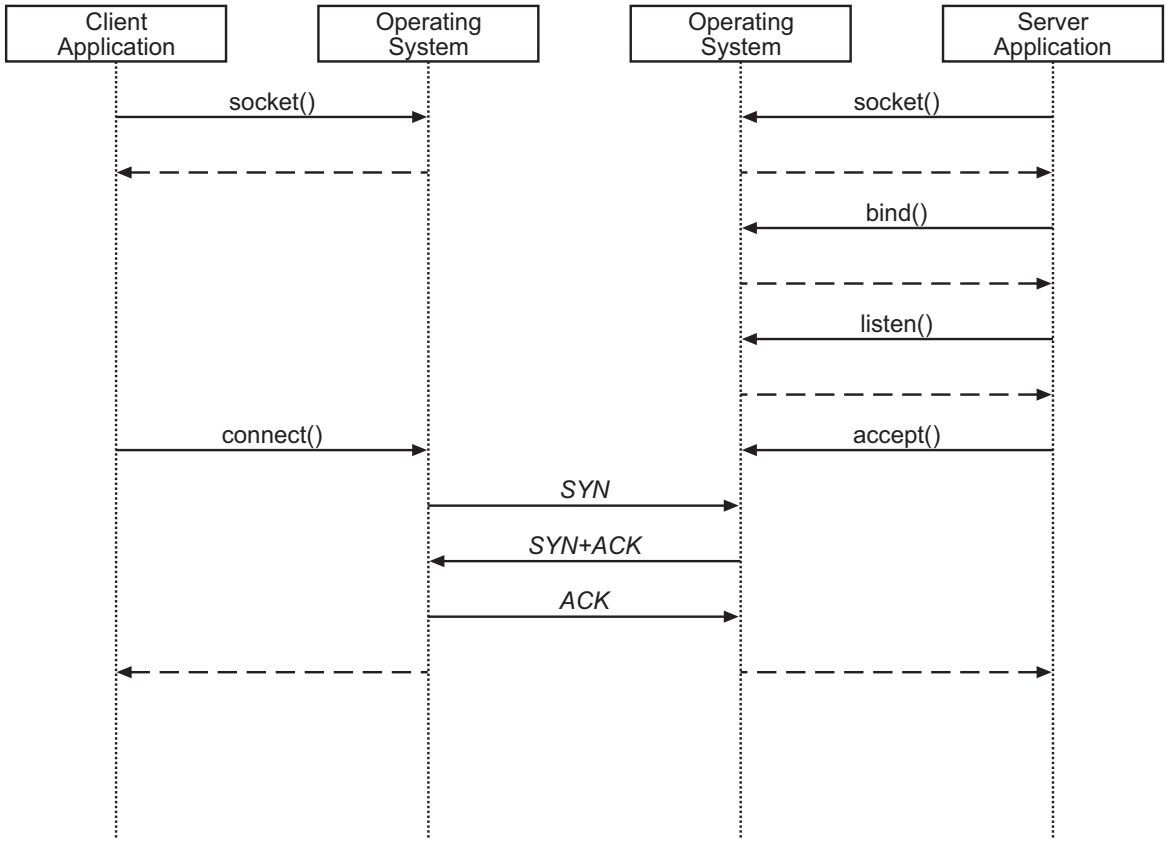

Figure 2. *Opening a TCP/IP connection.*

IP address. Therefore, one must first contact a name server to get the actual IP address of the server. This is usually a service of the operating system and can be done with only a few system calls. If one must connect to the name server himself, like the operating system does, the IP address of the name server must be known beforehand.

3.1.3 UDP/IP

User Datagram Protocol (UDP) [15] is another widely used protocol on the Internet. UDP allows processes to transmit single datagrams, or messages. Unlike TCP, the transmission is not guaranteed. Therefore, messages can be lost or duplicated, or they can arrive in the wrong order.

The length of the messages is limited by the maximum size of Internet Protocol (IP) packets. The maximum length of a message in an UDP packet is limited by the maximum size of an IP packet which is 64 kibibytes and IP header uses 20 bytes and UDP header 8 bytes of it. If messages are longer than the maximum length, the application must know how to break it into pieces and recombine it explicitly.

The IP layer can split large IP packets to smaller fragments if the maximum packet size for a network link is smaller than the transmitted packets. If all fragments arrive to the destination they are combined by the IP protocol and the upper level UDP protocol never sees this. But if some fragments are lost, the whole packet will be, too, and bandwidth has been wasted. For this reason, it is often desirable to not use a large UDP message size that it would cause fragmentation at the IP level.

The gain acquired from using UDP is that the amount of overhead is reduced compared to TCP. Firstly, with UDP there is no need to open a connection, which saves a few packets sent across the network. Secondly, acknowledgments are optional and must be sent explicitly by the application if needed. Therefore, the sending of acknowledgments can be optimized for the particular application, or omitted completely. With some application it does not matter if few packets are lost. Thirdly the UDP header is smaller, because UDP has fewer responsibilities than TCP.

UDP has one major advantage compared to TCP: it supports broadcasting and multicasting messages. Broadcast messages are received by every host in a network (with some restrictions). Multicast messages are only received by the hosts that have subscribed to that particular multicast group.

When using UDP the communication overhead can be reduced, compared to TCP, by careful design but it is paid with more program code to handle the communication. UDP is often used in applications such as video streaming.

3.2 Sun RPC

Remote Procedure Call (RPC) [16] provides an abstraction that allows a client to call a server with a simple function call within program code. RPC takes care of all the network traffic and data representation problems related to different computer architectures such as differences between big-endian and little-endian, and 32-bit and 64-bit, architectures. If the programmer wants more control over the authentication and other details, RPC gets more complicated to use.

The interfaces of remote procedures are defined using *eXternal Data Representation language* (XDR). C language stubs are generated automatically (by a utility called *rpcgen)* from XDR definition. A stub contains all the code required for the communication but the actual implementation of the procedures and function must of course be completed. For the client side there is no additional coding needed. There is no need to concern about the networking between the client and the server; using a remote procedure is as simple as calling a normal function. Exposing the functionality of an existing C code is done by writing a suitable XDR definition and filling the stubs with simple calls to normal functions.

RPC protocol is an open standard designed by Sun Microsystems. Implementations can be found on most UNIX like systems and even some non-UNIX systems. Although the protocol can be implemented with any programming language, the programming interface is often for C language. For on-wire communication RPC uses either TCP/IP or UDP/IP protocols (shown in Figure 3) and it can therefore be used across the whole Internet and can be implemented on practically every modern computer. For example, Microsoft Windows use a version of RPC for communication between different parts of the operating system.

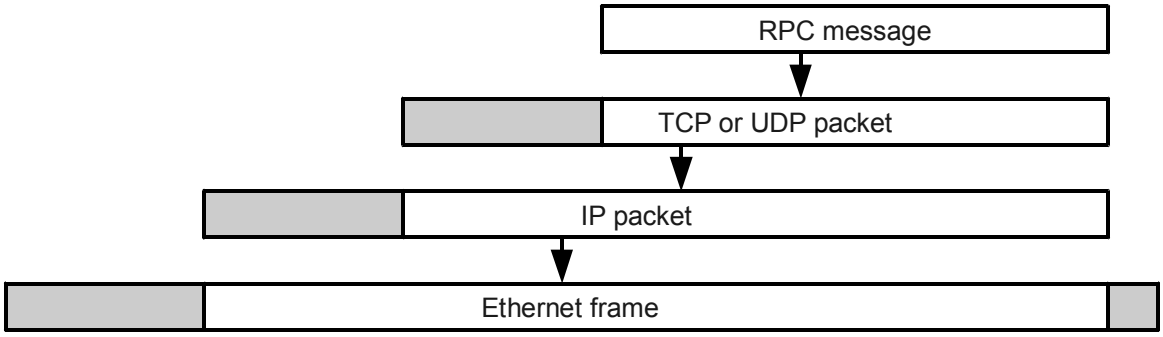

Figure 3. *Protocol stack needed for sending RPC messages.*

3.3 Shared File System

A shared file system, such as Sun Microsystems Network File System (NFS) or Microsoft Windows File Sharing, can also be used for inter-process communication on networked computers. Using a shared file system for sharing information between processes is much like using ordinary local files from the programmers point of view and it is, therefore, one of the easiest methods for a programmer to adopt.

The situation is however complicated by the use of many computers and networks. Firstly, shared file systems cannot be really used across low bandwidth or high latency networks. That is because they were designed to be used in an office environment where computers are connected to a high-speed local area network. This is also reflected by the fact that older shared file systems have little or no security features.

Secondly, network problems or server crashes can cause the distributed files to become temporarily available. This can either cause the operating system halt the process accessing the files until the network connection is re-established or file access can simple fail at any point. The results depend on the file system chosen and its configuration.

Thirdly, file locking, which is often used with processes running on a single computer and accessing same files simultaneously, might not work as well, or at all, with shared file systems. This, too, depends on the particular setup and its functionality must be verified and application requirements documented before it can be relied upon. Even the lack of file locking can be worked around, but obviously it will not be as efficient. Some applications, however, might not need locking and are unaffected by the this.

Lastly, different operating systems have support for different shared file system. This is only a problem if application is indented to be used in a mixed environment. Additional software, such as Samba [17], exists to allow usage of Windows File Sharing on UNIX systems. Samba is an independent free implementation of Windows File Sharing protocol and allows non-Microsoft systems to work as as file servers for Windows clients. There are commercial NFS implementations available for most widely used operating system since MS-DOS including Microsoft Windows. A common shared file system might have already been set up for ordinary file sharing in an office environment. Otherwise, it might be easier and cheaper to use some other method of communication for a distributed application.

Shared file systems often builds up on top of existing protocols. As an example the protocols working underneath NFS are shown in Figure 4. NFS clients use RPC calls to communicate with the server. RPC calls are packed into TCP or UDP packets for transmission. Both TCP and UDP packets are packed into IP datagrams. Finally IP datagrams are transmitted within Ethernet frames.

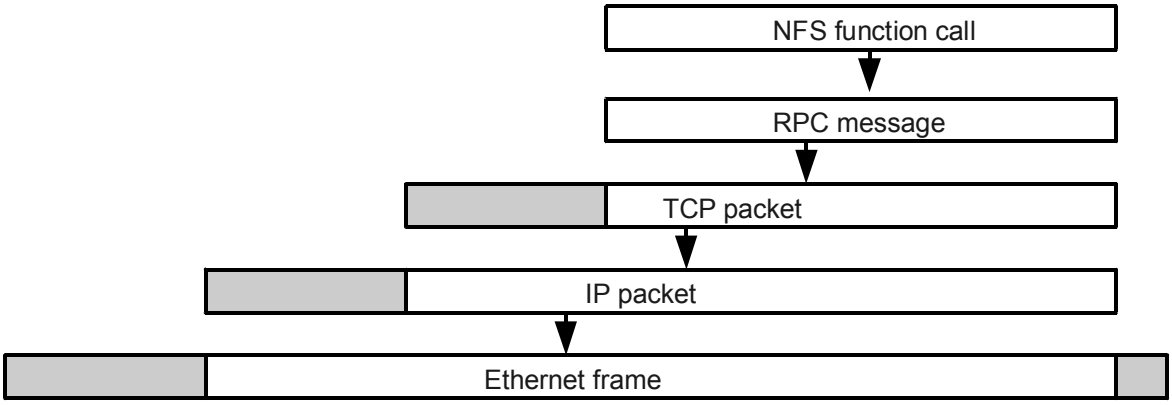

Figure 4. *Protocol stack when using NFS over Ethernet LAN*

3.4 CORBA

The Common Object Request Broker Architecture (CORBA) is an open standard by Object Management Group (OMG) [18]. The standard defines an object oriented communication protocol and it is platform, operating system, and programming language independent. There are several implementations of the standard from different vendors. And most importantly the different implementations can be used together. An object implemented with one CORBA implementation can be invocated by application using a different implementation. A CORBA implementation provides all the program code for remote object creation and invocation, thus making remote objects as easy to use as an ordinary local object.

The public interfaces for objects are defined using Interface Description Language (OMG IDL). IDL mappings for widely used languages such as C, C++, Java, Ada and Python are standardized. Non-standardized mappings for some other languages are also available. In reality, however, even the standardized language mappings for different vendor products have some differences. Especially the application code required to connect to remote objects is not portable. The code using the objects when already connected, on the other hand, is quite consistent between implementations.

The interface specification is mapped to the implementation language using an IDL compiler that comes with the selected CORBA implementation. The IDL compiler provides skeleton code, that handles the actual communication, for both server and client. The server skeleton must be completed with the actual functionality. The client skeleton provides a ready-to-use interface for the remote object that can be used pretty much like

any other object within the application. Some implementations for dynamic, non-compiled, languages allow IDL files to be compiled and used to access remote objects even at run-time.

The communication protocol, Internet InterORB Protocol (IIOP) is strictly specified by the CORBA standard, thus enabling applications compiled with different vendors implementations to use remote objects from each other without any problems. ORBs are allowed to use other protocols besides IIOP. Especially when invoking local object through the ORB it would be unnecessary to use network protocols. The protocol stack with IIOP on an Ethernet network is shown in Figure 5. As can be seen from the figure IIOP operates directly above TCP/IP protocol.

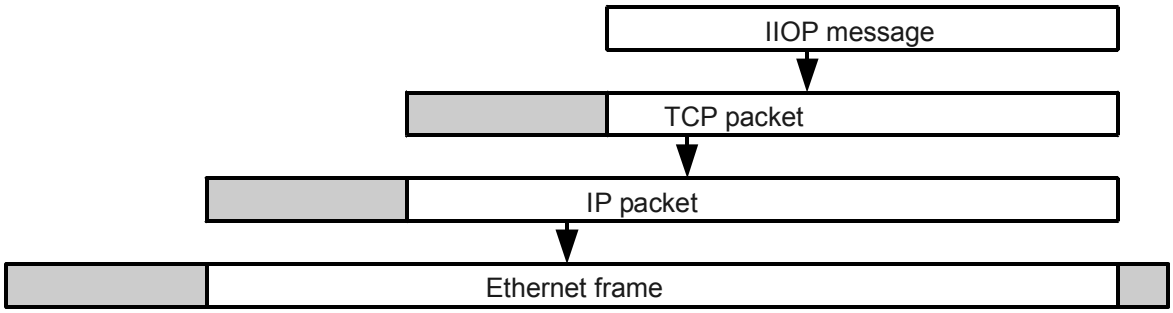

Figure 5. *CORBA protocol stack.*

3.5 Java RMI

Java programming language [19] developed by Sun Microsystems provides its own method for inter-process communication, the Java Remote Method Invocation (RMI) [20]. RMI provides a remote object abstraction similar to CORBA which is quite natural since Java is an object oriented programming language. RMI is part of the Java Platform standard and, therefore, it is included in every Java Virtual Machine (JVM), and available everywhere Java is. An advantage of RMI compared to CORBA is that the public interfaces for remote objects are defined using Java language, therefore, there is no need to learn a special interface description language.

Although RMI only works with Java, non-Java services can be made available by making a Java wrapper for them, for example, by using Java Native Interface (JNI). JNI is another part of the standard Java platform, which allows parts of an application to be

implemented using C programming languages, or another language which can be linked with C code.

RMI is very flexible with data that can be passed to, and returned from, remote objects. Method parameters and return values can be of any basic data type and any class that implements the Serializable interface. The Serializeable interface is a guarantee that an object of a class implementing it can store its internal state to a byte stream and later restore the object as it was. A remote object interface does not even have to be specific about the class of an passed object, instead the values can be defined to be just the general Object type. The Object class is the abstract root class in Java and all classes are derived from it.

By using an abstract type for passed object executable code can be transferred from one computer to another. If the object implements a known interface, the actual implementation will be dynamically loaded at runtime by the JVM. The class file can be a local file or it can be downloaded with Hyper Text Transfer Protocol (HTTP) or File Transfer Protocol (FTP). This allows runtime modification on the function of both the client or the server and moving processing where it makes sense for best overall system performance and usability, which is very hard to achieve with other distribution methods.

The protocols Java uses when making a remote invocation with RMI are shown in Figure 6. The use of HTTP in remote invocations is optional and is only used by Java for passing firewalls if necessary. If there is no need for such a workaround, RMI works directly over TCP/IP.

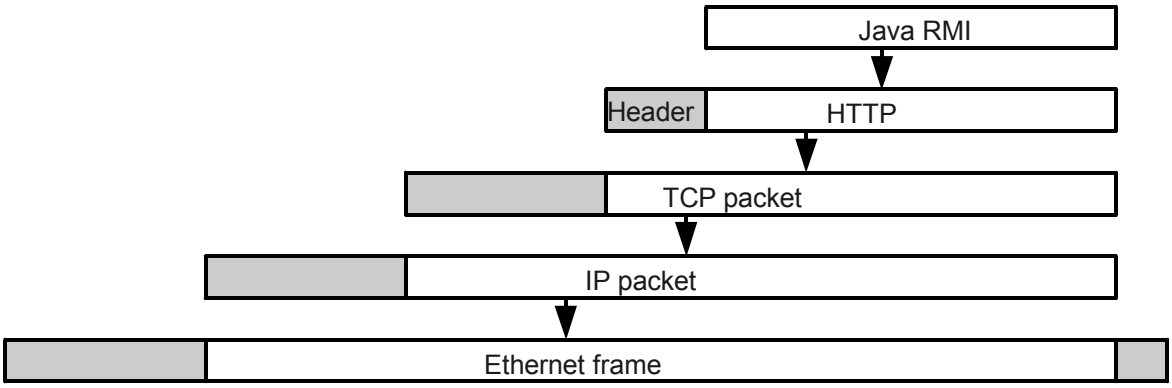

Figure 6. *One possible protocol stack for RMI invocation.*

3.6 Summary

A comparison of the methods described above is shown in Table 1. They can be available as a separate middleware, that sits between the application and underlying operating system (OS) to extend the readily available system services, or can be integrated into the operating system itself. Most of them are readily available on many UNIX systems.

The popularity of the Internet has made IP the most widely used communication protocol. All the methods described here are usually used over an IP network. The more complex protocols also built onto the simpler ones.

Table 1: *Comparison of communication methods.*

	Abstraction	***Overhead***	***Operating System Support***	***Programming Language Support***	***Ease of use***
TCP/IP	Stream of bytes	Small	Included with most operating systems	Any language	Socket programming, similar to low level file access
UDP/IP	Packets of bytes	Small	Included with most operating systems	Any language	Socket programming, similar to low level file access
Sun RPC	Remote function calling	Medium	UNIX systems, Microsoft Windows has a variation	C, C++	Write a remote definition and use like ordinary functions
Network File System	Shared files	Medium	NFS on UNIX systems, Microsoft Windows	Any language	Same as local files with maybe more locking needed
CORBA	Remote objects	High	Open source and commercial libraries available to most systems	Most languages	Write remote object definition and use in programs just like ordinary objects
Java RMI	Remote objects	High	Support in Java Runtime	Java	Almost like ordinary Java objects

The downside is that every layer adds more overhead to the communication. Packing and unpacking the packets at every layer consumes time and memory at the communicating hosts, and the headers for every layers can add to a significant amount of data to be transmitted. Therefore it should be carefully thought out what level of abstraction and performance is needed for any particular application as it is impossible to have both.

Using one of the existing protocols, such as HTTP or Simple Mail Transfer Protocol (SMTP), and building the new application on top of them can be easier than defining a completely new protocol. A newer standard, The Blocks Extensible Exchange Protocol (BEEP) [21], has been developed to ease designing new custom protocols. BEEP does all the non-application-specific tasks related to communication, including encryption, and communication between processes using different protocol versions, all implemented in a way that has been proved to work.

Obviously the more complex systems provide more functionality with the expense of possibly more overhead. In many cases this overhead is negligible considering the ease of implementation abstractions such as the object paradigm. Systems like CORBA provide ready made tools for load balancing and at the same time makes the system scalable. With the rapid development times needed today there is just no reason to spend time reimplementing and testing all these features. Instead, custom build communication systems are only made when absolutely needed, or the needs are fairly simple and no extra overhead is wanted.

4 Discrete Time Network Simulator

The Discrete Time Network Simulator (DTNS) is a simulation tool developed at the Institute of Digital and Computer Systems at Tampere University of Technology. DTNS is aimed to be used at the high-level phases in the design flow of platform-based complex digital systems [1]. DTNS can be used to analyze interconnection architectures and systems built around them such as System-on-Chip (SoC) designs.

4.1 Overview

DTNS is a time-driven simulator, which means that during the simulation the time advances in fixed increments. Time-driven simulation is inherently not as accurate as event-driven simulation but the approximation is accurate enough for majority of logical simulations with a global clock. The time-driven approach was chosen because it makes the simulation kernel less complicated and it is also faster to execute. The extra complexity of event-driven simulation would not even be much of use because DTNS is intended to be used at the higher abstraction levels where simulation results are approximations at best. The simulated time advances in time steps of half a clock cycle and this resolution is sufficient to allow events happening at both rising and falling edges of the system clock.

The motivation for DTNS was to reduce the gap between specification and the first implementation that could be simulated and to lift the design process of complex systems to a higher abstraction level. Because of the large size of SoC designs, the design process focuses on communication. Therefore, it is ideal to have detailed communications model even though the model otherwise is still at a high level.

The system design is described in C or C++ programming language and can then be verified by simulation. Especially the C++ language has such high level that fairly complex functionality can be described with just few lines of code compared to the normally used hardware description languages such as VHDL [22] and Verilog [23]. Of course, this benefit is lost when the system model is refined to a more detailed version.

DTNS also decreases the simulation time compared to existing hardware description language (HDL) simulators. However, it is still a slow process to simulate functionality of accurate multiprocessor design. Therefore, it is possible to distribute DTNS simulation employing multiple computers connected with a network. The speed-up with distribution is only achieved when the models are described in detailed level. Otherwise, the overhead caused by the communication slows down the simulation enough to negate the benefit.

The objective of DTNS was to produce as much timing information as possible early in the design process using a high abstraction level model. Therefore DTNS was designed to always use detailed communication model with exact timing information, i.e. it is suitable for communication based design. DTNS is aimed to be easy to use without any knowledge of special purpose HDLs and even with limited knowledge of systems functionality at hardware level.

The abstraction level of DTNS is between the conventional hardware simulators, such as Mentor Graphics ModelSim [24], and algorithm level simulators, such as Synopsys System Studio [25]. With DTNS, the first simulation can be executed at an earlier phase of the design flow than with other tools and thus important information about the systems functionality is gained at the beginning of the architectural design.

Unlike some of the other simulators, DTNS does not make abstractions of the interconnections of functional blocks but the system bus traffic is always simulated accurately even when inaccurate high level agent models are used. As a result the

simulation produces always accurate information about the sufficiency of the bus capacity.

Agents are the functional blocks that are connected together by the system bus. Each agent is an independent block that performs a well defined function or functions. An agent can be an accelerator, general purpose processor, or something in between.

The functionality and abstraction level of DTNS is conceptually very close to the well-known Ptolemy II [26] simulation tool and SystemC [27] simulation kernel. However, since the C/C++ is still very widely used in algorithm design, the use of Java-based Ptolemy II would require extra work.

Current tools lack the timing information, even if it is available at the original specification, and the communication information is usually ignored at the higher levels of abstraction. The lack of this important information has caused the architectural design to be done with less information that is available at the lower levels. This has been the primary concern in the design of DTNS and thus the interconnection is always simulated accurately. The overall accuracy of high-level simulations is increased and more useful information is gained at a very high abstraction level with reasonable simulation time.

DTNS is implemented in C programming language. The structure of DTNS is modular as presented in Figure 7. Different interconnection models and agents can be used fairly easily by replacing the module describing the functionality of the bus or an agent. The modules are explained in more detail in Section 4.3.

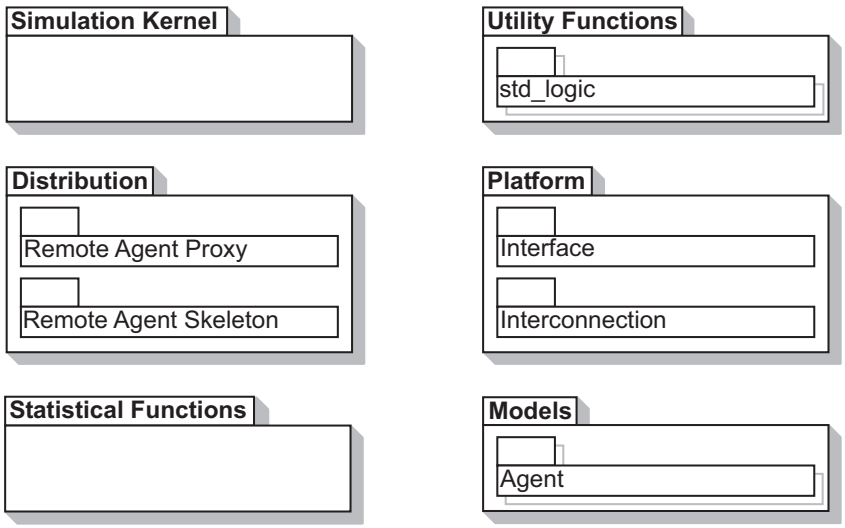

Figure 7. *DTNS modules.*

The typical area for DTNS is a SoC design where the blocks are connected by a system bus, which takes care of all the communication in the system. The interconnection and agents are modeled and then a simulation is run to determine the performance of the system.

4.2 Design Flow with DTNS

DTNS was designed to support Platform-Based Design (PBD) [28]. The suggested design flow that takes full advantage of DTNS is shown in Figure 8. DTNS can be used at the four top levels, from executable specification to cycle-approximation level (or register transfer level, RTL), after that a vendor specific tool, such as Mentor Graphics Seamless co-verification tool, must be used. One possible classification of abstraction levels is shown in Table 2.

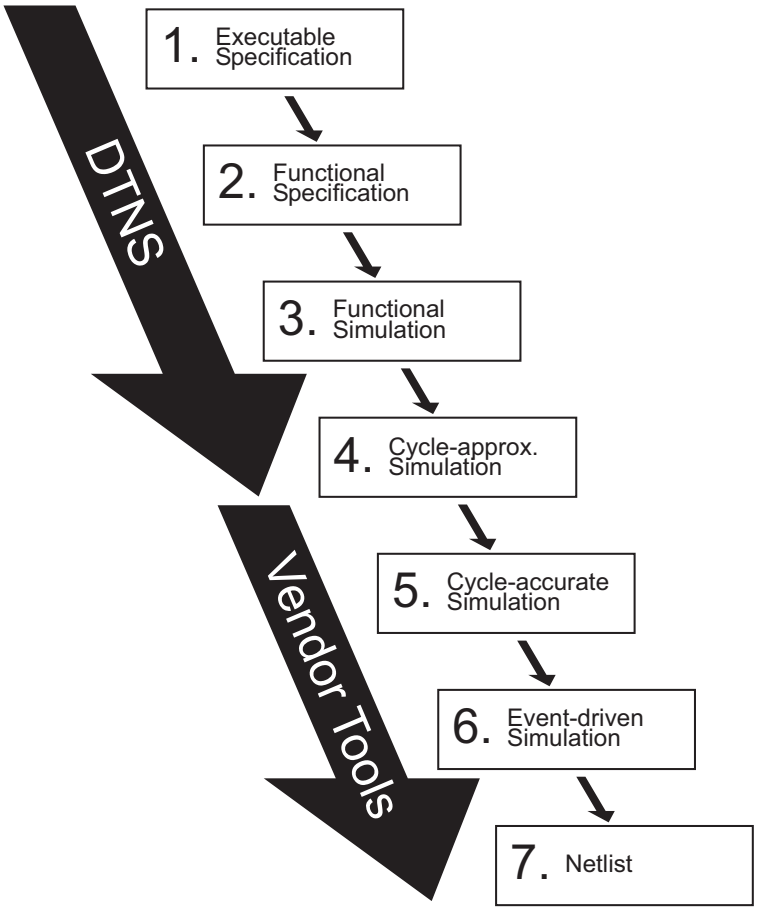

Figure 8. *Design flow when using DTNS.*

Table 2. *Simulation levels and abstraction levels.*

Simulator Type	***Abstraction***	***Typical Use***	***Design Flow***	***Tool / Language***
Statistical	Input: statistical distributions	General purpose architectural exploration	Feasibility analysis	Spread sheet
	Data transfers: abstract channels			
	Data processing: queuing systems			
Data Flow	Input: known data flow	Embedded architectural exploration	Feasibility analysis	SDL
	Data transfers: channels / protocol			
	Data processing: queuing systems			
Algorithm Level	Input: test data	System specification design and verification	Algorithm selection	C/C++
	Data transfers: channels / protocol			
	Data processing: algorithm level code			
Instruction Level (Functional Level)	Input: test data	HW/SW partitioning and performance estimation	Processor selection, SW development	C/C++ / High level HDL
	Data transfers: channels / protocol			
	Data processing: processor cores / algorithms			
Architectural Level (Behavioral Level)	Input: test data	HW/SW logical design and verification	Behavioral synthesis, IP selection, SW development	Behavioral HDL
	Data transfers: protocol			
	Data processing: processor cores / behavioral HDL			
Register Transfer Level	Input: test data	HW logical design and verification	Logic synthesis, SW verification	RTL HDL
	Data transfers: protocol / cycle-accurate			
	Data processing: RTL HDL			

Simulator Type	*Abstraction*	*Typical Use*	*Design Flow*	*Tool / Language*
Switch Level	Input: test data	HW electronic design and verification	Automatic layout, placement, route	PLD design languages
	Data transfers: protocol / phase-accurate			
	Data processing: switch level transistors			
Circuit Level (Transistor Level)	Input: test data	Final HW electronic design and verification	Physical design	PLD design languages
	Data transfers: electrically accurate			
	Data processing: circuit models			

System design is an iterative process, and if simulation results at some level are found not to correspond the specification, the design at the earlier phases need to be corrected. It should be noticed that DTNS can be used as early as the executable specification to simulate the system, whereas traditional simulators can be used only from the behavioral specification. Thus, with DTNS important information is achieved for the architectural design.

The systems design starts from a natural language specification (not shown in the figure.) An executable specification (phase 1) of the system is written in C or C++ language. The specification can be at a very high abstraction level at this stage. The executable specification is then simulated with DTNS and some information about the systems functionality is acquired.

Based on the information gained from the simulation the accuracy of the description is refined to a functional (or behavioral) specification (phase 2). The exact timing information about the functional blocks can be ignored at this level, but the platform information and interfaces are accurate. With the simulation of the functional specification (phase 3) detailed information about the transactions in the system is achieved.

This information is used to write a more detailed description of the system. At the next simulation the platform is simulated cycle-accurately and the connected blocks are cycle-approximated (phase 4) and even more detailed information about the system is gained.

The system can be simulated between the design phases using agents from different abstraction levels because the tool remains the same. This allows the implementer to verify the agents one by one at every step in the way and find bugs earlier and more easily from a smaller amount of code.

DTNS is not currently capable of doing event-driven simulation, thus, for the event-driven simulation (phase 6) the code must be translated from C/C++ to the HDL of choice and the simulations must be done with a vendor specific tool. The cycle-accurate models of agents could be simulated with DTNS (phase 5), but currently it is not reasonable to do it for the whole system, because the HDL version must be done anyway.

Unfortunately, currently the error-prone translation must be done by hand which can cause problems at this stage, but there are ways to make it automatic. One possibility is to make a compiler from C to, for example, VHDL, which could be done if a fairly strict coding style is used when writing the C code. Another possibility would be to use SystemC for modeling.

4.3 Structure of DTNS

In DTNS, the signals on the system bus are modeled using a bit type that can have nine different values instead of the usual two, similar to the IEEE std_logic type in VHDL. The possible values are logic one *(1)* and zero *(0)*, weak one *(H)* and zero *(L)*, uninitialized *(U)*, conflict *(X)*, weak conflict *(W)*, high impedance *(Z)* and don't care *(-)*. With the extra values it is possible to identify situations where more than one agent is writing to the same signal or someone is trying to read a floating signal. The type and conversion functions between it and normal C types are available for the agents to use as well.

Because of the distribution there are two types of agents. Local agents are executed in the same process with the simulation kernel. Remote agents, on the other hand, are executed in separate process and are compiled into distinct binaries. In fact, the only difference between these is in the way they are compiled and the same source code can be used for both.

The modules of distributed DTNS are shown in Figure 7. The main modules are *Simulation Kernel, Utility Functions, Distribution, Statistical Functions, Platform* and *Block Models.* These main modules contain sub-modules but only some of them are shown in the figure.

The *Simulation Kernel* module contains the main function for the DTNS simulator. The code for the actual simulation kernel and code that interprets the command line arguments and the configuration file is also there. In addition the module contains a simple agent that reads an initialization vector from a file and writes it to the bus. After the file has been written to the simulated system bus the agent does not do anything. It is used to initialize the simulated interconnection blocks instead of using simulation of a processor, which would do the work in the actual implementation. It could also be used to input the initial data for the simulation, but this is better done in separate agents that reads the data from a file and also writes the output to another file.

The *Utility Functions* module contains all sorts of miscellaneous functions. One important sub-module is the std_logic type and the conversion functions, like conversion between an integer and std_logic vector. In another sub-module are functions for handling threads. There are also functions for acquiring parameters defined in the configuration file, which is read when the program starts, and functions for printing debug and error messages.

The *Distribution module* contains the code that has something to do with inter-process communication. The *Remote Agent Proxy* sub-module is the glue between the simulation kernel and a remote agent. The kernel sees it as an ordinary local agent and handles it in the same way. Behind the scenes the proxy makes the remote procedure call for the remote agent skeleton that contains the actual agent model.

The *Remote Agent Skeleton* is basically the main function for the remote agent binary. It handles almost all the same tasks that the simulation kernel has, for example

configuration file reading, agent initialization and calling the agent model. The agent model sees no difference between the simulation kernel and the *Remote Agent Skeleton*. However, the skeleton does not advance the simulated system clock by itself, but depends on the simulation kernel to do that.

The *Statistical Functions* module contains functions that are used to calculate statistical properties of the system bus and the interconnection state of a block, and writes the data to files. Examples of statistics are bus efficiency, rate of FIFO fullness and amount of transferred data.

The *Platform* module has a very important role. The platform architecture, including the bus signals (the *Interface* sub-module) and the interconnection block (the *Interconnection* sub-module) are described there. Unfortunately it is very difficult, or perhaps impossible, to come up with a universal programming interface for all possible platforms and, therefore, a lot of code has dependencies of at least the bus signals. Major changes are needed all over the program code if the simulated platform is changed in a way that affects its interface. Besides the code for the interconnection block, there is not that much functional program code there but only interface definitions.

Finally the *Models* module contains the agent models. The agent can be specific for an application or they can be reusable. The agent models here form the DTNS library of ready-to-use models for common blocks, such as memory. A typical simulation consists of the simulation kernel, platform specification and usually more than one agent, depending on the applications complexity.

Pseudo code for the simulation kernel is shown in Figure 9. There are slight differences with this non-distributed version and the distributed version presented in the next chapter. The kernel handles the ticking of the global clock and resolving of the signal values on the system bus. It can also write current bus signals to a file for later analysis if it is enabled and it can read a test vector from a file and feed it to the system bus to initialize the agents. The platform specifies the signals on the system bus and provides the interconnection model. The lower level versions of the agents are modeled in a VHDL-like coding style that, for example, takes into account the clock edges and real timing.

```
initialize bus signals
initialize agents

clock := 0
while clock < CLOCKS do

     for each agent do
          transmit current bus values to agent
          run agent
          receive agent's interconnection values
     loop

     resolve interconnection for next half-cycle

     toggle interconnection clock signal
     if clock signal = 1 then
          clock := clock + 1
loop
```

Figure 9. *Pseudo code for the simulation kernel.*

4.4 Simulation Flow

The simulation flow for non-distributed simulation is shown in Figure 10. When the simulator is started, the simulator reads the configuration file. It defines at least the number of clock cycles to be simulated and the file names of the output files. The configuration file can also contain agent specific parameters, which the agent code can access using utility functions. After that, the command line parameters are parsed and the values defined are merged to the in-memory configuration. The last initialization step is to initialize the bus signals and call initialization functions for each agent.

After initialization the main simulation loop is started. It is executed until the predefined number of clock cycles has been reached. In the main simulation loop the current bus values are passed to every agent and the agent modeling function is called. The agent provides new interconnection values for the next clock cycle. After all agents have been executed, the interconnection values from all the agents are combined and the bus signals for the next cycle are resolved. Finally the system clock is advanced by half a clock cycle.

During a simulation data about the simulated system is collected and written to files. The simulation kernel writes the bus signals to an output file, the interconnection model writes statistics about the status of the input and output FIFOs for every agent, and the agents can write whatever data is seen necessary.

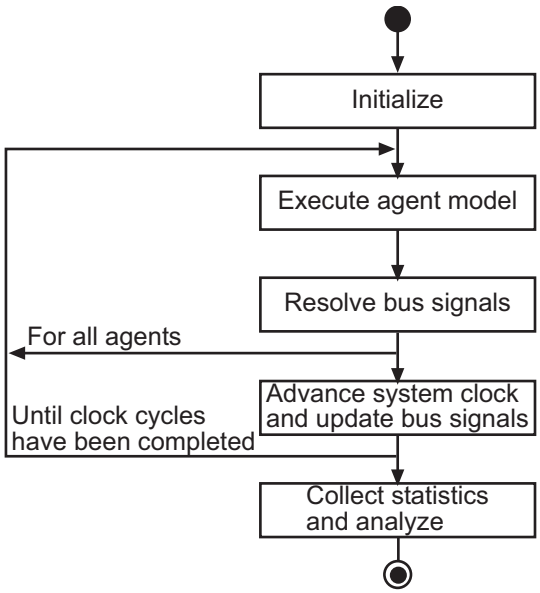

Figure 10. *Non-distributed simulation flow.*

After the simulation had ended output files are analyzed. The file with bus signals can be used to determine whether bus signaling was correct. For example, verify that only one interconnection block wrote to the bus at any one time. Also the utilization and throughput of the simulated system bus can be calculated from this data. The correctness of the whole simulated system can be determined from the output files written by the agents. For example, a video decoder can write the decoded video frames to files which can be viewed and verified to be correct.

Since the signal values are usually much easier for human beings to look at as a graphical presentation, instead of just text, a tool was needed to visualize the bus, and other similar, signals. At first Mentor Graphics ModelSim was used for this task but it is unnecessarily heavy tool for such a simple use. Therefore, a much lighter graphical signal viewer, called Waveform, was developed during this work to be used with DTNS.

Waveform can read the simple ASCII file format written by DTNS. It understands and can view all the nine possible values a std_logic bit can have in the simulator. In addition to viewing the single bit signals, it is possible to combine bits to form words and view their values in hexadecimal and give bits and words meaningful names. For cross platform support Waveform was written in Java and uses the Swing toolkit for the graphical user interface. Waveform is shown viewing the HIBI bus signals from a DTNS simulation in Figure 11.

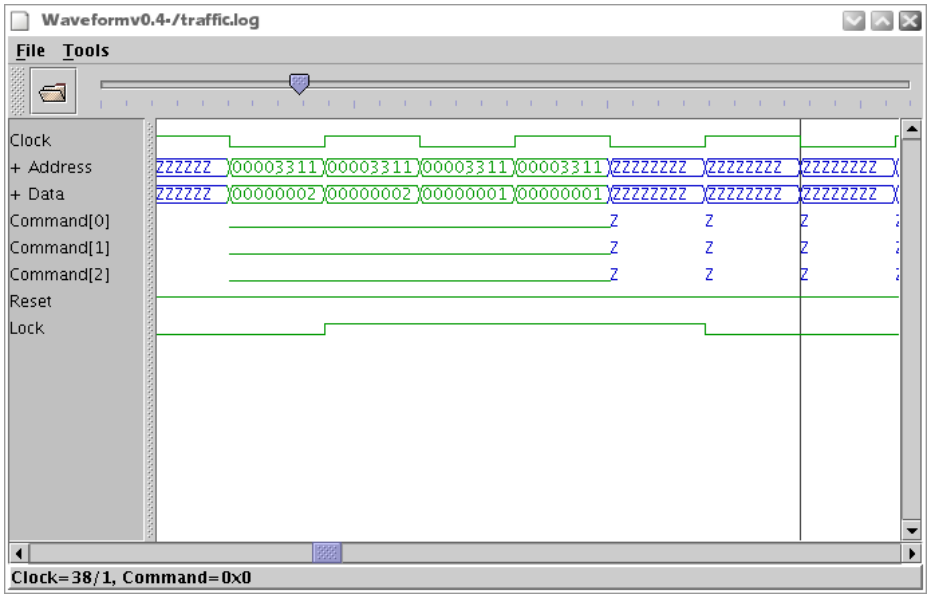

Figure 11. *Waveform viewer developed in this work*

5 Distributed Discrete Time Network Simulator

The main goal of this thesis was to distribute the existing DTNS. The distributed version targeted to be executed on network of workstations (NOW) environment to achieve better performance while keeping the cost of hardware down. In this chapter the distribution is described in detail.

5.1 Background

The non-distributed version of DTNS was designed without taking distribution into account and different modules had many dependencies. For example, a lot of data was stored in global memory, which cannot be done in a distributed implementation. It was, however, justified in a non-distributed program, because global data does reduce the need for data copying and, therefore, improves performance, and allowed easier collection of statistics.

In a distributed program, the shared memory needed for global data would have to be simulated with message passing. A simpler solution was to remove all global data and design another way for the processed to communicate. At the same time a lot more modular structure was gained.

In DTNS, the agents, or functional blocks, are separate entities and are only connected by the system bus, which is, therefore, their only communication channel. The signal values

on the system bus are also the only required shared data that all agents must know. The bus is a natural point that could be used to divide the simulation to processes. Basically, every agent is running in a process of its own and everything they would write to the bus is transmitted over the network to other agents for reading. The agents would, however, have to form a fully connected net for this kind of communication. Therefore, a central process, the simulation kernel, is needed. With the central node the network will be star-shaped and the amount of communication is linear to the number of agents in the system, which scales much better than a fully connected net.

The simulation flow for distributed simulation is shown in Figure 12. With distribution all the agent models can be executed simultaneously if enough computers are available. Therefore, the task of transmitting and receiving bus signals from agents was needed in the simulation kernel.

The pseudo code for the kernel is shown in Figure 13, and it is similar to the non-distributed version in Figure 9. The distribution is conservative, i.e. all processes are synchronized at every step on the way. This happens as a side effect of transmitting the bus signals, because agents cannot proceed to process the next clock cycle before they have received the signals on the system bus.

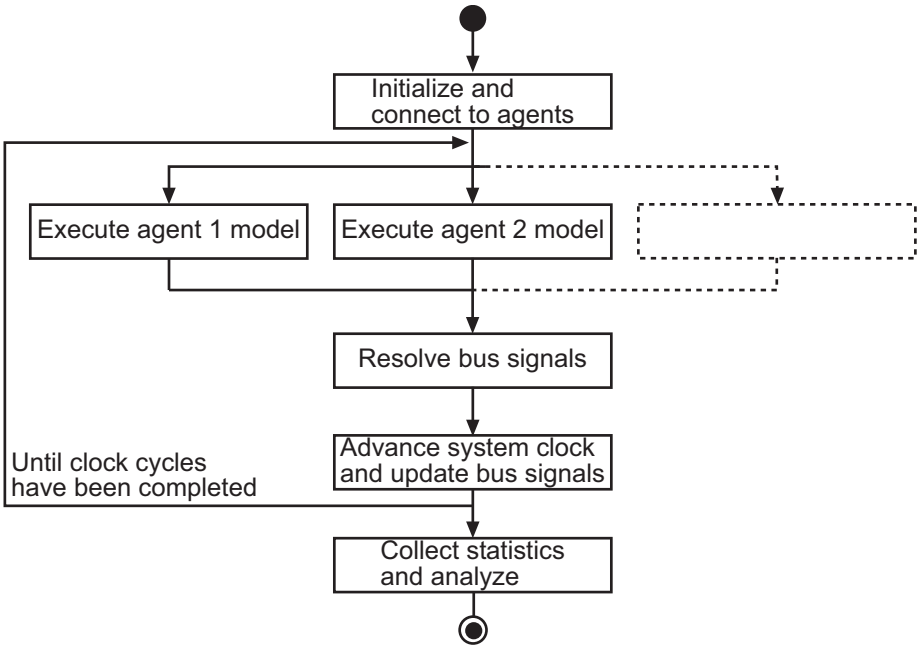

Figure 12. *Distributed simulation flow.*

```
for each agent do
     connect to agent
loop

initialize bus signals

clock := 0
while clock < CLOCKS do

     for each agent do
          transmit current bus values to agent
     loop

     {wait for agents}

     for each agent do
          receive agent's interconnection values
     loop

     resolve interconnection for next half-cycle

     toggle interconnection clock signal
     if clock signal = 1 then
          clock := clock + 1
loop

for each agent do
     close connection to agent
loop
```

Figure 13. *Pseudo code for the distributed simulation kernel*

Firstly upon start-up the simulation kernel connects to all agents, which must have been started earlier and must be waiting for a connection from the kernel. The addresses required to find the remote agents is conveniently located in the configuration file, which was read as the first step of initialization. After that the initialization continues as in the case of the non-distributed version described in Section 4.4. After initializations the main simulation loop is started.

Every half a clock cycle the kernel transmits the current signal values on the simulated system bus to all agents, which also informs the agents that they are allowed to simulate forward half a clock cycle. This transmission can be seen as a request for the agent to perform its task. Then the kernel begins to wait for the agents to start sending back their own interconnection values, which is the response for the kernel's request.

The responses are handled in the order that they arrive, which is not necessarily the same order the requests were sent. This reordering allows optimization of the resolve function,

which in fact performs the same operation for interconnection values from each agent, and the order of these operations is not important. Therefore, this resolving can be done partially while waiting for the other agents to respond. The partitioning of the resolve function on the other hand shortens the amount of time the agents have to wait for the kernel between every cycle.

Finally, after the wanted amount of clock cycles have been simulated, the kernel closes the connections to agents, which in turn informs the agents to shutdown themselves. Before shutting down, agents can execute an analyze function and write the results to files. The analyze function can be implemented to produce, for example, information about the performance of the agent.

In the distributed version a couple lines of pseudo code have been moved from the kernel to the main function of an agent. That is shown in Figure 14, which describes the general functionality of a remote agent. The kernel does not need to know anything about the agent except how to call the modeling function. It follows that agent initialization belongs to the process executing the agent and not the kernel.

When the execution of a remote agent is started, it reads a configuration file and parses command line parameters similarly to the main program. The initialization procedure includes the initialization of the actual agent and connecting to the simulation kernel. During simulation the agent waits for current bus signals from the kernel, executes the actual agent code, and sends back the signals the agent has written to the bus. This loop is repeated until the connection to the kernel is closed, which can be assumed to mean that the simulation has ended. Before the process exits, the agent calls a clean-up function if necessary.

```
wait connection from kernel

initialize agent

while connected do
     receive current bus from kernel

     run agent code

     transmit agent's interconnection values to kernel
loop
```

Figure 14. *Pseudo code for main function of a remote agent*

To reduce the overhead related to the inter-process communication over a network, there exists an alternative agent main function that can actually handle several agent models running within the same process. The additional code that is needed for this multi-agent process is quite similar to what is in the simulation kernel for calling multiple agents.

The agent models are called in a loop and their combined interconnection values are resolved. Therefore, the interconnection values must be transmitted between the agent process and the kernel only once for a group of agents instead of once for each agent.

This improves the performance when the number of available computers is less than the number of agents, or the agents are so simple that the communication overhead dominates the execution time. The extra code, of course, increases overhead, so it should be only used when multiple agents are executed on a single computer.

The call-paths for both local and remote agents are shown in Figure 15. Network lies between *Remote Agent Proxy* and *Remote Agent Skeleton*. The remote object invocation is hidden inside the *Distribution* module and, therefore, only it needs to know how the

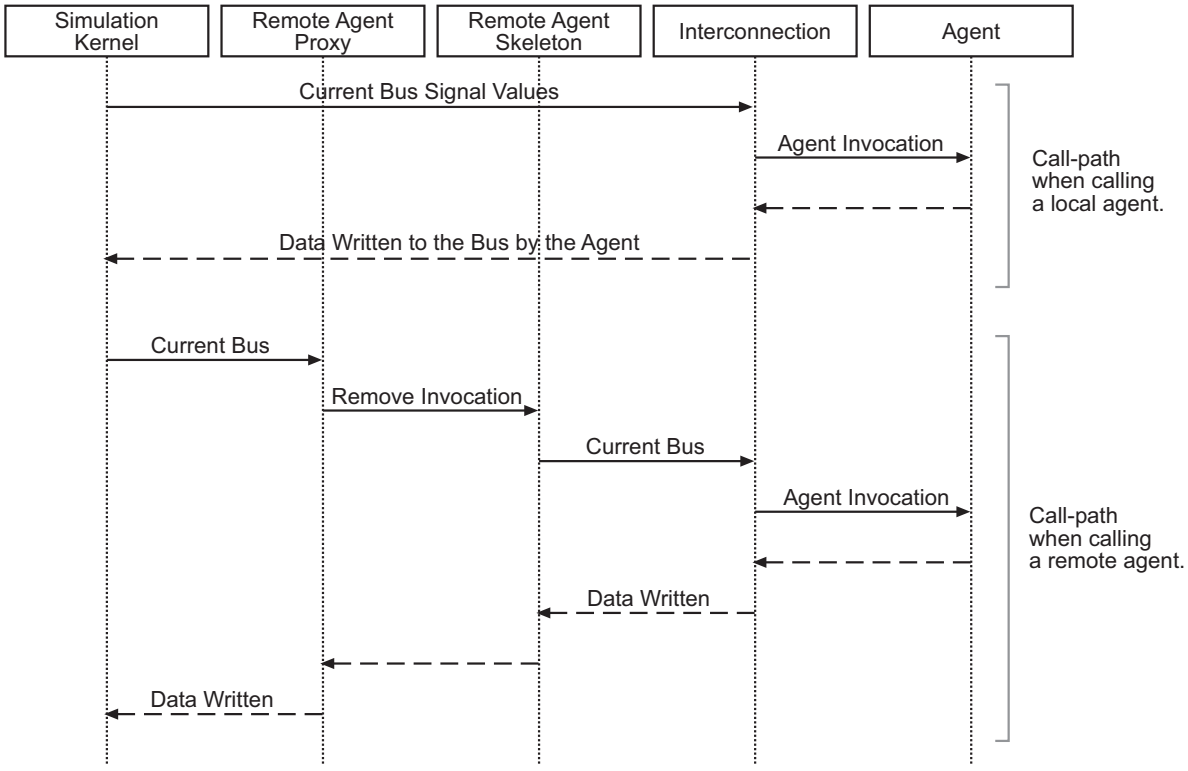

Figure 15. *Call-paths for local and remote agents*

invocation happens. The interface of the *Distribution* module is flexible enough to allow several different implementations for remote invocations.

5.2 Distribution with CORBA

The Xerox Inter-Language Unification (ILU) [29] was chosen as the CORBA implementation. ILU was a natural choice from the freely available CORBA implementations because it was the only one, which supports multi-threaded programs and has C language mapping. The only feature that is missing is the capability to start a server at a specific TCP port, which is available in, for example, MICO [30], an open source standard compliant CORBA implementation. This was not, however, a necessary feature and implementation was possible without it.

Using CORBA is convenient because the ILU handles all the system specific tasks related to the communication over a network and, with the exception of threads, all DTNS code is portable.

Threads are hidden behind a minimalist interface without priorities or other fancy features. Only a few simple utility functions to handle threads and semaphores are needed. Therefore, the requirement for threads does not make porting DTNS too difficult. Most UNIX systems have POSIX threads, which were supported first because the development computer was running Linux. The port for Microsoft Windows, which uses completely different API for threads, has been also implemented.

The need for threads came from the fact that calling remote objects with CORBA is synchronous. To gain anything from the distribution, different agents must be executed in parallel. Therefore, agents, which are remote objects, must be invocated simultaneously from separate threads.

This could have been circumvented by using two invocations to the CORBA objects instead of one, one to transmit current bus signals to an agent and one to receive the response from an agent. However, this approach would have wasted time because the order in which the responses were fetch would have been fixed and would not adapt to the

execution speed of the agents at a particular moment in simulation. With the use of threading DTNS now can also parallelize the execution of agents when run on an multiprocessor computer with minimal overhead.

The CORBA implementation has some unexpected overhead. For example, an enumeration type takes always 32 bits when sent across a network. The std_logic type is an enumeration with 9 distinct values, which could be represented with 4 bits. To reduce the amount of overhead the distribution was also implemented in another way. This implementation uses TCP/IP, the transport layer used by CORBA, directly and is discussed next in more detail.

5.3 Distribution with TCP/IP

In the sockets implementation, the CORBA remote objects are replaced with agent programs that open a TCP/IP server socket. Next the main program containing the simulation kernel connects to these agent programs. The agents still act as a server and the kernel is a client for all of them like in the CORBA implementation.

Unfortunately the use of sockets removed the advantages gained from CORBA. The network is not hidden behind a common API but utility functions providing access to the sockets must be ported to every target platform. At least most UNIX systems are alike which should reduce the amount of work.

Because the need of communication between DTNS processes is very simple and has a static form, also the protocol used on top of TCP/IP is very simple. Every bit of the simulated bus is converted to a descriptive 8-bit ASCII character and the characters are combined into a single string, which is then send or received over the connection. ASCII presentation was chosen because handling octets is easier than four bit nibbles. The possibility of an odd number of nibbles also causes a problem because data is always transmitted in octets with TCP/IP.

The client, the main program, sends such a string first and then waits a similar string from the server, the agent, and the server does the same operations in a reversed order. There is

no need for any extra headers except for the ones that come with TCP. The conversion used for transmission of the data is much simpler than the marshaling performed by CORBA. Every std_logic bit is presented by a single ASCII character in the TCP stream. Depending on the value the character used is one of *0* (zero), *1* (one), - (dash), *U*, *X*, *Z*, *W*, *L* or *H*.

A simple compression was developed for the socket version of distributed DTNS to even further reduce the amount of transmitted data. The utilized compression scheme is similar to run length encoding (RLE).

In ordinary RLE, a long string of repeating byte is replaced with three bytes. First comes an escape byte which tells that a repetition follows. Second byte is interpreted as an integer value telling the number of times the data byte repeats in the original stream. The third byte is the actual repeating byte or data. If the byte value, that is used as the escape byte, is present in the original data stream, it must be escaped in the same way even if it does not repeat.

The implemented compression, however, has some differences compared to usual RLE scheme. Firstly the compressed data remains in a human readable ASCII format. Secondly the repeating strings can be arbitrary character sequences instead of only one byte. Finally all data is encoded as repetitions even if the string is present only once in the original data.

To enhance the compression rate only changes between simulation cycles are transmitted. This is achieved by first comparing the current bus signals to the ones from previous cycle and marking everything that have not changed equal. No-change is transmitted as an ASCII equals sign (=). As a result, a long string of non-changing signal values can be compressed to one repetition of one character, no matter what the actual signal values are.

Due to the use of a text format, the repeat count cannot be represented with a single byte. Instead it is encoded as a text presentation of a positive decimal number. This also gives the repeat count, at least in theory, an infinite range. In practice, the small amount of data transmitted between simulation cycles keep the repeat counts quite low.

Because characters *1* and *0* can appear in both the repeat count and the following data, special characters are needed to separate the two. A colon *(:)* is used to terminate the

repeat count and a semicolon *(;)* terminates the string of data. This use of colon and semicolon replaces the escape byte in ordinary RLE. The same character could be used for both but using different characters increases readability. Despite that only a small part of the ASCII character set is in use.

To avoid unnecessary overhead two optimizations are used. Firstly the repeat count is implicitly one if it is omitted, but the colon must always be present. Secondly the semicolon can be omitted at the end of the data, because it would be redundant.

An example of uncompressed bus traffic is shown in Figure 16. For readability a header line and some white space for field separation was added. The same data compressed with the above scheme is shown in Figure 17. This sample data has been reduced to about 18% of the original size. A simple binary encoding, instead of ASCII, would halve the compressed size. For comparison, *gzip*, a commonly used compression utility, packs the same data to about 6%.

```
Clk Address                          Data                                 Cmd Rst Lock
0   ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ ZZZ 1   0
1   ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ ZZZ 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000111101 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000111101 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000111100 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000111100 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000111011 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000111011 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000111010 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000111010 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000111001 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000111001 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000111000 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000111000 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000110111 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000110111 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000110110 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000110110 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000110101 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000110101 000 1   0
0   00000000000000000000010001001000010 00000000000000000000000000000110100 000 1   0
1   00000000000000000000010001001000010 00000000000000000000000000000110100 000 1   0
```

Figure 16. *Example of uncompressed bus traffic.*

```
:0;67:Z;:10
:1;69:=
20:0;2:1000100;25:0;4:1;:01;3:0;:==
:1;69:=
:0;63:=;:0;5:=
:1;69:=
:0;61:=;:011;5:=
:1;69:=
:0;63:=;:0;5:=
:1;69:=
:0;62:=;:01;5:=
:1;69:=
:0;63:=;:0;5:=
:1;69:=
:0;60:=;:0;3:1;5:=
:1;69:=
:0;63:=;:0;5:=
:1;69:=
:0;62:=;:01;5:=
:1;69:=
:0;63:=;:0;5:=
:1;69:=
```

Figure 17. *Example of compressed bus traffic.*

5.4 Distribution of a VHDL Simulator

The same method of distribution was also used to distribute Mentor Graphics ModelSim, a commercial VHDL simulator. A distribution bridge was implemented to connect to separate task. The bridge is a reusable component which is in no way depended on the simulated application.

In Figure 18, a distributed simulation of two separate tasks is shown. The simulation processes can be run on a multiprocessor computer or on two distinct computers connected by a IP network. By adding more connecting bridges the application can be distributed to an arbitrary number of processes. As the connections are peer-to-peer type, more than one bridge can be used in a single simulator to allow connecting more than two simulators together.

The distribution bridge passes the signal values written to its input port to the output port of the bridge it is connected running in an other simulator process, and vice versa. The

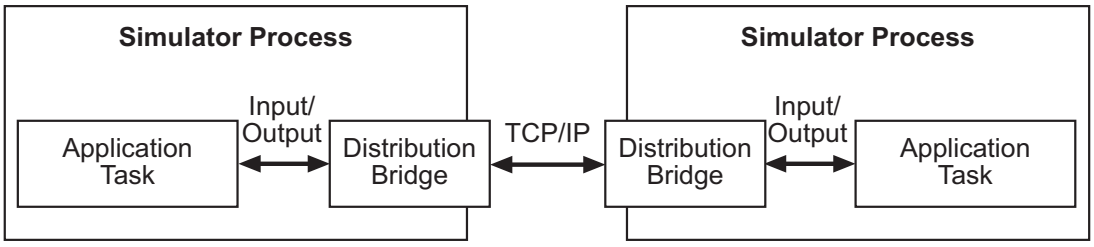

Figure 18. *Distribution of an application to two separate tasks.*

bridge also keeps the simulation processes synchronized and, hence, eliminates any chance of a causality problem.

If the partitioning to similar size tasks have been done successfully, the simulation runs roughly at the same speed on all the simulators. Otherwise, in a less than optimal partitioning the faster tasks will have to wait for the slower ones.

The bridge is implemented partly in VHDL and party in C. The communication between the VHDL code and the C code uses ModelSim's Foreign Language Interface (FLI). The VHDL code provides the input and output ports of the bridge so the application (also implemented in VHDL) can be connected with the bridge. The C code transforms these signals to ASCII format are then transmits over TCP/IP, exactly like the TCP/IP version of Distributed DTNS.

6 Case Studies

To determine the performance of the distributed DTNS a test case was set up. The scalability of the simulator was evaluated by running multiple simulations with two varying parameters. Firstly, the workload of simulated agents was modified, and secondly, the number of computers the simulation used for distribution was altered. The measured quantity for each simulation was the amount of wall clock time required for the simulation to complete.

There was also interest in the speed difference between the two distribution implementations, CORBA and TCP/IP. For that reason, all simulations were run with both versions so that they could be compared.

The distribution method used in DTNS was also tried with VHDL simulation and the test case was simulating a TUTWLAN network of eight terminals. The simulation times where measured using eight single processor Linux workstations and also using a single multiprocessor UNIX workstation.

6.1 DTNS Test Case

The test application was a traffic simulation on the Heterogeneous IP Block Interconnection version 1 (HIBI) [31] bus of a H.263 video encoder. No video encoding was done in the simulation but the communication profile was arranged to roughly match

an actual encoder in macro block encoding loop. The HIBI wrapper, which connects an IP block to the interconnection, was modeled accurately. An IP block and a HIBI wrapper together form an agent for the simulator. The structure of the simulated system is shown in Figure 19.

The HIBI bus in this case consists of a total of 70 std_logic signal values: 32 signals for data, 32 signals for address, three signals for command, and clock, reset and lock signals. This means a payload of 280 bytes for CORBA implementation or 70 bytes for TCP/IP implementation without protocol overhead. This payload needs to be transferred across the network between computers in both directions on every simulation cycle.

The actual agents that generate the bus traffic are so simple that their execution time can be neglected. A more complex system would require some processing at least with low-level modeling. For evaluation reasons, an artificial adjustable processing time was added to the agents. The processing time per simulation cycle was varied from zero to 5 millisecond with one millisecond intervals. Five agents were used to match the number of agents in the working H.263 implementation.

Simulation length was set to 20.000 simulated clock cycles. This means 40.000 cycles for the simulator as DTNS advances in half a clock cycle steps.

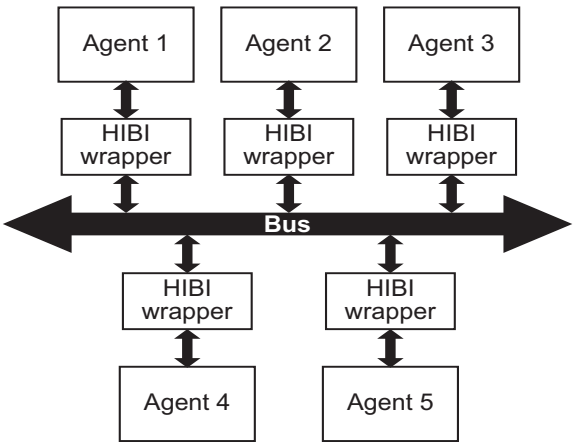

Figure 19. *Test case system structure.*

6.2 DTNS Test Environment

The test environment consisted of three ordinary Linux workstations. The configuration of computers is shown in Table 3. The workstations were connected to a 100 Mbps Ethernet with switches. Although the third computer is faster than the other two the artificial delay was still the same amount of milliseconds on all. This imitates a real situation when multiple computers are used in distributed simulation. Other processing was of course somewhat faster.

The simulations were executed with using one, two and three computers. The simulation kernel was always running on computer #1. The five agents where distributed as equally as possible among the used computers in each case. Load balancing in this case was rather easy because all the agents were equal with only parameters changed.

Table 3. *Computers used to run the test cases.*

Computer#	Processor	Memory	Operation System
1	Intel Pentium II 400 MHz	128 MiB	Red Hat Linux 7.1
2	Intel Pentium II 400 MHz	128 MiB	Red Hat Linux 6.2
3	AMD Athlon 1.0 GHz	256 MiB	SuSE Linux 7.1

6.3 DTNS Results

The speed-up gained from distributing the simulation to several computers was measured by timing the simulation runs described above. The results for CORBA and TCP/IP implementations are shown in Table 5 and Table 4 respectively and graphically in Figure 20 and Figure 21.

Table 5. *Results with CORBA (in seconds).*

	Processing time (ms)					
	0	***1***	***2***	***3***	***4***	***5***
one computer	22	151	277	413	536	665
two computers	86	148	230	282	352	425
three computers	135	163	204	260	310	333

Table 4. *Results with TCP/IP (in seconds).*

	Processing time (ms)					
	0	***1***	***2***	***3***	***4***	***5***
one computer	23	179	332	508	643	771
two computers	25	120	188	262	345	441
three computers	27	78	131	204	254	293

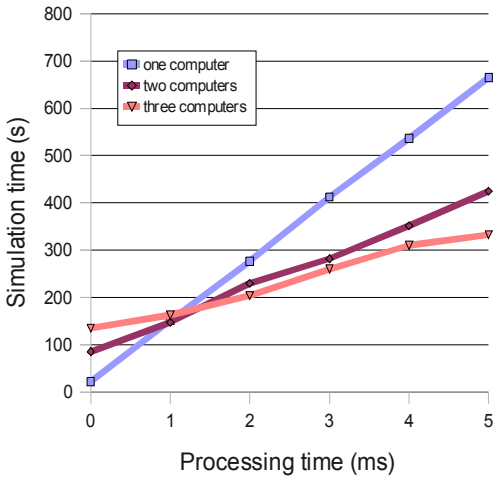

Figure 20. *Simulation results with CORBA.*

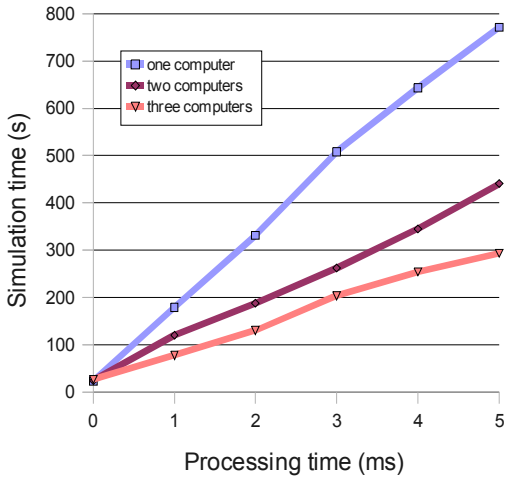

Figure 21. *Simulation results with TCP/IP.*

With very simple and fast-executing agents the distribution overhead causes the simulation to become slower than without distribution. TCP/IP implementation handles this case better because it has less overhead but the slowing down is noticeable.

When the agents are complex and need more processing time, distribution begins to make sense. With CORBA the turning point in this test case is at 2 ms delay. TCP/IP is a little better and it is faster that non-distributed already with just 1 ms delay. As the processing time increases, the difference between CORBA and TCP/IP implementations vanish.

The number of computers has a decreasing effect on the total simulation time. The simulations with 0 to 2 ms processing times would probably not benefit from extra computers. With 5 ms processing time maybe two extra computers could be used. The maximum amount with this test case is six, one for the simulator kernel and one for each five agents.

The TCP/IP implementation does not use the compressing algorithm presented in Section 5.3. Some simulations where run also with compression but the result where very close to the ones without compression. Therefore, it is determined that with such a low number of signal values as used here, there is no benefit from the compression. The explanation is that the minimum payload size of the Ethernet frame is so large (46 bytes) that the data (71 bytes uncompressed) is not significantly larger even without compression. For this reason, the actual data transfer over the physical network takes nearly the same amount of time.

6.4 Test Cases for Distributed VHDL Simulation

Because the real world test with DTNS do not show significant improvement in simulation time with distribution and the more theoretical test cases show some potential, some test cases where build where the distribution method from DTNS was borrowed and used with Mentor Graphics ModelSim. The core DTNS code handles the distribution and the agents are separate instances of ModelSim simulating hardware based on VHDL description.

The test cases were a simulation of a network of two to eight TUTWLAN [32] terminals, a total of seven different networks. A TUTWLAN terminal includes two processors, external memory, radio interface, and the blocks are connected by HIBI interconnection blocks, as shown in Figure 22. The networks were partitioned along the radio interface to form equal size tasks. The tasks were executed in their own simulator process. Each task also needs two ARM ISS processes to run the TUTMAC protocol and application software. The length of all simulations were 125,000 clock cycles.

The test cases were executed using one to eight ordinary single-processor Linux workstations. They all had a 2 GHz processor, 512 MiB of memory, and where connected to a 100 Mbps switched Ethernet network.

The same networks where also simulated using a multiprocessor UNIX workstation for comparison. The workstation has eight 1050 MHz processors and 64 GiB of memory. In addition to the distributed test cases, a simulation without distribution bridges was run with this workstation.

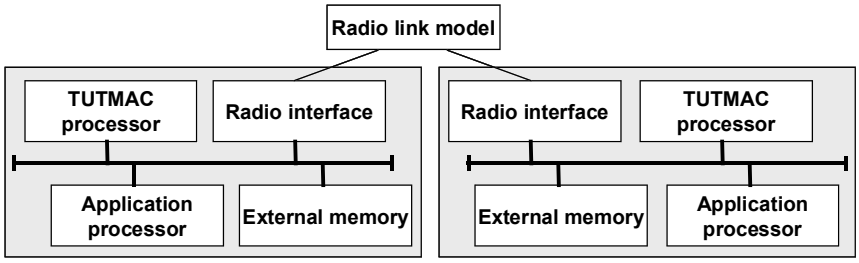

Figure 22. *TUTWLAN network with two terminals.*

6.5 Results of Distributed VHDL Simulation

The results from simulations using Linux PCs are shown in Figure 23. The maximum speed-up with eight computers is 7.3 times compared to non-distributed simulation. The simulation time stays nearly constant on simulations where the number of terminals equals the number of computers used. With one computer the simulation time increases almost linearly with the number of terminals. In odd cases the simulation time increases when an extra computer is added. This is because the partitioning to computers cannot be done equally and the additional computers add some overhead.

In Figure 24 the the results from the simulation run with the multiprocessor workstation are shown. With distribution to all eight processors a maximum speed-up of 5.5 is gained. This smaller speed-up is caused by the slight parallelization of even the non-distributed simulation, because every task had two extra processes running the ARM ISS. Again, the non-distributed simulation time increases almost linearly as the number of terminals increases. In the distributed case the simulation time also slightly increases as the number terminals (and processors) is increased.

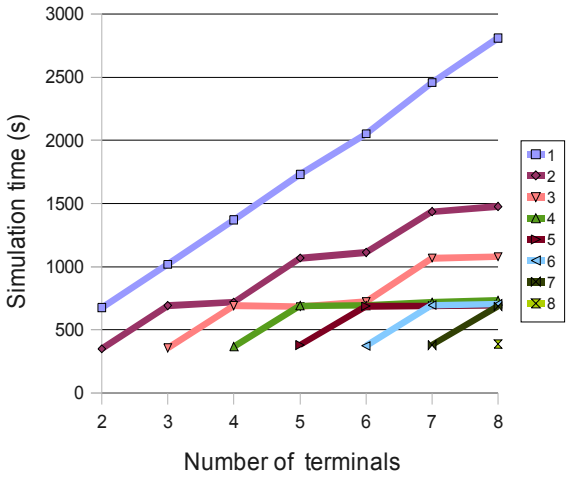

Figure 23. *Simulation times with different size TUTWLAN networks performed with Linux workstations.*

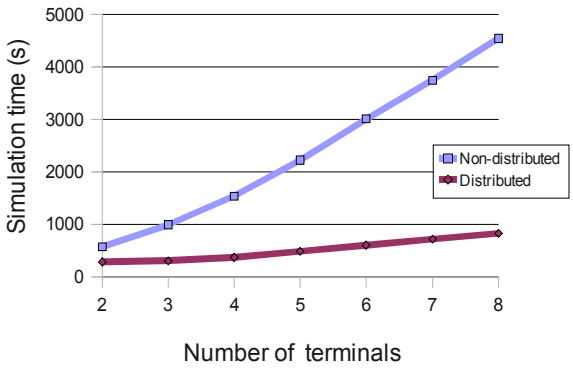

Figure 24. *Distributed and non-distributed simulation times with multiprocessor workstation.*

7 Conclusion

The distribution of DTNS was found partially successful. Simulation is sped up almost linearly to the available computers with relatively complex models. However, such complex models are not, at least currently, modeled with C or C++ language. Computers have also gained more speed, and will continue to do so. Therefore, the models would need to be increasingly complex for Distributed DTNS to stay useful.

The algorithm level simulation can still be done very well using DTNS with just a single workstation, and the the new version developed in this work is capable of parallel execution with more than one execution thread. With multi-core processors becoming common that ought to be more important and useful aspect.

The distribution of VHDL simulator, which used the same method that was developed for DTNS, however, has already proved to be successful and cut down simulation times significantly. This is mainly because the simulation of VHDL models is a lot slower than higher-level C models.

References

[1] Kuusilinna K., Riihimäki J., Hämäläinen, T., Saarinen J., "*DTNS: a Discrete Time Network Simulator for C/C++ Language Based Digital Hardware Simulations,"* 4th World Multiconference on Circuits, Systems, Communications and Computers, CSCC2000, Athens, Greece, 2000, pp. 3311-3316.

[2] X. Meng, "Distributed Simulation in a Loosely Coupled Environment Using the TCP/IP Protocol," *Proc. IEEE 14th Annual International Phoenix Conference on Computers and Communications*, 1995, pp. 122-127.

[3] TOP500 Supercomputer sites, http://top500.org/

[4] E. Naroska, "Parallel VHDL Simulation," *Proc. Design Automation and Test in Europe*, 1998, pp. 159-163.

[5] SETI@home Home Page, http://setiathome.ssl.berkeley.edu/

[6] George Coulouris, Jean Dollimore and Tim Kindberg, *"Distributed Systems: Concepts and Design, 3rd ed.,"* Addison-Wesley, 2001

[7] Nelson Minar, et al. , *"Peer-to-Peer: Harnessing the Power of Disruptive Technologies,"* O'Reilly, March 2001

[8] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys,* Vol. 18, No. 1, Mar. 1986, pp. 39-65.

[9] Y. M. Teo, et al., "Conservative Simulation Using Distributed-Shared Memory," *Proc. 16th Workshop on Parallel and Distributed Simulation,* 2002, pp. 3-10.

[10] D. Lungeanu and C. J. R. Shi, "Parallel and Distributed VHDL Simulation," *Proceedings of Design, Automation and Test in Europe,* 2000, vol. 2, pp. 658-662.

[11] Postel, J., Ed., "*Internet Protocol*", RFC 760, January 1980

[12] W.R. Stevens et al., *"UNIX Network Programming, Vol. 1: The Sockets Networking API, 3rd ed.,"* Addison-Wesley, 2003

[13] IANA Home Page, http://www.iana.org/

[14] Postel, J., Ed., "*Transmission Control Protocol*", RFC 761, January 1980

[15] Postel, J., *"User Datagram Protocol"*, RFC 768, August 1980

[16] Sun Microsystems, Inc., *"Remote Procedure Call,"* RFC 1050, April 1988

[17] Samba Home Page, http://www.samba.org/

[18] Homepage of CORBA, Object Management Group, Inc., http://www.corba.org/

[19] Java Home Page, http://java.sun.com/

[20] Support Readiness Document, Java 2 Standard Edition 1.3, Remote Method Invocation, Sun Microsystems Inc., 2000.

[21] Rose, M., "*The Blocks Extensible Exchange Protocol Core,*" RFC 3080, March 2001

[22] IEEE 1076 VHDL Language Reference Manual

[23] IEEE 1364-1995 Verilog Language Reference Manual

[24] Mentor Graphics Home Page, http://model.com/

[25] Synopsys Home Page, http://www.synopsys.com/

[26] Ptolemy II Home Page, http://ptolemy.eecs.berkeley.edu/ptolemyII/

[27] Open SystemC Initiative, http://www.systemc.org/

[28] Chang, H., et al, *"Surviving the SOC Revolution: A Guide to Platform-Based Design"*, Kluwer Academic Publishers, 1999

[29] Xerox Inter-Language Unification Home Page, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

[30] MICO Home Page, http://www.mico.org/

[31] V. Lahtinen et al., "Interconnection Scheme for Continuous-Media Systems-on-a-Chip," Microprocessors and Microsystems, vol. 26, no. 3, Apr. 2002, pp. 123-138.

[32] M. Hännikäinen *et al*., "TUTWLAN – QoS Supporting Wireless Network," *Telecommunication Systems - Modelling, Analysis, Design and Management*, vol. 23, no 3/4, 2003, pp. 297-333.