

WINDOWS NT 4.0 EMBEDDED TCP/IP NETWORK APPLIANCE:  
A PROPOSED DESIGN TO INCREASE NETWORK PERFORMANCE

A Thesis  
Presented to  
the Faculty of the  
California Polytechnic State University  
at San Luis Obispo

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science in Electrical Engineering

by  
Angel Hoi-Yee Yu,  
December 1999

**AUTHORIZATION FOR REPRODUCTION  
OF MASTER'S THESIS**

I grant permission for the reproduction of this thesis in its entirety or any of its parts, without further authorization from me.

---

Signature

---

Date

**APPROVAL PAGE**

TITLE: Windows NT 4.0 Embedded TCP/IP Network Appliance: A Proposed Design  
to Increase Network Performance

AUTHOR: Angel Hoi-Yee Yu

DATE SUBMITTED:

Dr. James Harris  
Advisor

\_\_\_\_\_  
Signature

Dr. Joe Grimes  
Committee Member

\_\_\_\_\_  
Signature

Dr. Chris Scheiman  
Committee Member

\_\_\_\_\_  
Signature

## **ABSTRACT**

Windows NT 4.0 Embedded TCP/IP Network Appliance:

A Proposed Design To Increase Network Performance

by:

Angel Hoi-Yee Yu

This thesis documents the design of a network architecture to maximize network performance. From previous studies on network performance, it was shown that the Windows NT operating system is the bottleneck during data transfers. The impetus for the proposed design is to minimize the amount of time spent in the kernel when transferring data.

The initial design required the use of a 3Com research and development network interface card. While investigating this design, it became apparent that it was not a viable solution for two reasons: the memory mapping utility, MapMem did not function properly in the Anaconda environment and the XINU operating system could not be ported within a reasonable amount of time.

As a result of the findings from the initial design, another design is proposed. This proposed design is a simplified version of the original plan. It involves the use of MapMem that will map memory from Intel's EBSA-285 board and porting the Linux operating system onto the board.

## ACKNOWLEDGEMENTS

First of all, I would like to thank 3Com Corp. for providing the resources used to conduct the research and for their advice on numerous issues.

I would also like to thank Dr. Jim Harris for his continued support and encouragement regarding my education in many ways throughout my years here at Cal Poly. His guidance has been invaluable. I thank Dr. Chris Scheiman for helping me with technical aspects of the research and for being a part of my committee. I also thank Dr. Joe Grimes for being a committee member. In addition, I thank Dr. Elmo Keller for providing a variety of Linux resources.

I would also like to acknowledge various contributions made by my peers on the project, especially Jim Fischer and Mauricio Sanchez.

Finally, I would like to thank my family for their love and believing in me. I did it!☺

# TABLE OF CONTENTS

|  | Page |
|--|------|
| List of Tables .....                           | ix   |
| List of Figures .....                          | x    |
| 1 Introduction.....                            | 1    |
| 2 Project Overview.....                        | 4    |
| Background.....                                | 4    |
| Network Architecture .....                     | 7    |
| TCP/IP Protocol Suite.....                     | 7    |
| Traditional Windows NT Model .....             | 8    |
| Design Plan .....                              | 11   |
| Initial Design Plan .....                      | 11   |
| Proposed Design.....                           | 16   |
| 3 Development Environment.....                 | 18   |
| Components .....                               | 18   |
| Setup.....                                     | 18   |
| Development Tools .....                        | 19   |
| 4 Analysis of Existing Anaconda Microcode..... | 21   |
| Transmit path.....                             | 21   |
| TCP Segmentation Path.....                     | 24   |

|   |  |    |
|---|--|----|
|   | Testing Transmit Path (TCP Segmentation Path)..... | 29 |
|   | Receive path.....                                  | 35 |
|   | Testing Receive Path.....                          | 38 |
| 5 | Windows NT MapMem Device Driver .....              | 41 |
|   | What is it? .....                                  | 41 |
|   | Verification .....                                 | 42 |
|   | Testing MapMem’s Read Capability .....             | 43 |
|   | Testing MapMem’s Write Capability .....            | 43 |
| 6 | Initial Design.....                                | 49 |
|   | Modifications to the Windows NT Application.....   | 49 |
|   | Incorporating MapMem.....                          | 49 |
|   | Custom Sockets.....                                | 49 |
|   | Socket Basics.....                                 | 50 |
|   | Socket Control Path .....                          | 52 |
|   | Incorporating Transmit Descriptor Mechanism .....  | 55 |
|   | Porting TCP/IP Protocol Stack .....                | 58 |
|   | Implementation Plan .....                          | 58 |
|   | Interface Between TCP/IP and Socket Layer .....    | 59 |
|   | Interface Between TCP/IP and NIC.....              | 60 |
|   | Results.....                                       | 61 |
| 7 | Proposed Design.....                               | 63 |

|   |     |
|---|-----|
| 8 Conclusion and Summary.....                               | 66  |
| Bibliography .....  | 67  |
| Appendix A: Execution Procedure for Anaconda Microcode..... | 69  |
| Appendix B: Packet Send from Anaconda Perspective .....     | 71  |
| Appendix C: Pseudocode for TCPSegSend().....                | 75  |
| Appendix D: Packet Receive from Anaconda Perspective .....  | 76  |
| Appendix E: Pseudocode for Receive.....                     | 79  |
| Appendix F: Ping Pong Application Program.....              | 80  |
| Appendix G: Send/Server Application Program .....           | 95  |
| Appendix H: Pseudocode for socket() .....                   | 107 |

## List of Tables

|   |    |
|---|----|
| Table 1. Initial Performance Measurement Results [6]..... | 6  |
| Table 2. Socket Function Calls .....                      | 52 |

## List of Figures

|  |    |
|--|----|
| Figure 1. TCP/IP Protocol Suite .....                            | 8  |
| Figure 2. High Level Networking Architecture in Windows NT ..... | 9  |
| Figure 3. Windows NT Networking Protocol .....                   | 10 |
| Figure 4. Initial Network Architecture.....                      | 12 |
| Figure 5. Control/Data Path in Initial Network Architecture..... | 14 |
| Figure 6. Four Control and Data Paths for Transmit .....         | 15 |
| Figure 7. Proposed Network Architecture.....                     | 16 |
| Figure 8. Isolated Development Network .....                     | 19 |
| Figure 9. Flow Chart of Transmit Path .....                      | 23 |
| Figure 10. Two-Workstation Testbed for Transmit.....             | 30 |
| Figure 11. Three-Workstation Testbed for Transmit.....           | 34 |
| Figure 12. Receive Path .....                                    | 37 |
| Figure 13. Testbed for Maptest2 Development .....                | 42 |
| Figure 14. Closed Loop Model Test Case.....                      | 47 |
| Figure 15. Initial Network Architecture Model.....               | 50 |
| Figure 16. Establishing a Socket Connection [8] .....            | 51 |
| Figure 17. Socket Control Path.....                              | 53 |
| Figure 18. Flow Control for Initial Design.....                  | 57 |
| Figure 19. TCP/IP / Socket Interface .....                       | 60 |
| Figure 20. Proposed Network Architecture.....                    | 64 |

## Typographic Conventions

To distinguish between various software objects, the following typefaces are used.

**FunctionName**

Names in bold font describe function names.

*VariableName*

Names in italic font describe variable names.



## CHAPTER 1

### Introduction

With the advances in technology, information has become easily accessible to people electronically. Instead of looking up textbooks and periodicals at a library to locate the desired topic, people can search the Internet on their personal computers. Information can travel from one side of the world to another within a given amount of time. Sometimes the amount of time spent waiting for the requested information is significant and very noticeable to the user. Since the distance between the two points is fixed, the time for the information to physically travel between the two points is also fixed. However, the time needed to process the information transmission and reception within the personal computer depends on its network architecture.

In order to maximize network performance, the amount of time spent processing data transmission and reception must be minimized. Network performance can be measured in terms of latency, throughput and central processing unit (CPU) utilization. Latency describes the amount of time for a given amount of data to be transferred. Throughput describes the transfer rate for a given amount of data. CPU utilization is the amount of time allocated to the CPU for processing in terms of percentage. To maximize network performance, latency must be minimized, throughput must be maximized, and CPU utilization minimized.

This thesis will discuss a proposed network architecture that will try to increase network performance by porting the Transmission Control Protocol (TCP)/ Internet Protocol (IP) protocol suite onto an external board. 3Com's Anaconda network interface card (a

development board) was chosen as the external board since it already contains some TCP/IP intelligence. The initial design of the architecture is divided into four areas: analysis of the Anaconda network interface card (NIC) microcode, application of the Windows NT MapMem device driver, modifications to Windows NT, and porting of the TCP/IP stack. The final proposed design is essentially a scaled-down version of the initial design.

To understand how the existing Anaconda microcode works, a detailed analysis was performed on the data transmission and reception paths. The analysis consisted of reading the microcode specifications, the microcode itself, and finally testing the transmit and receive paths. This provided insight on how to implement a new data path between the application data buffer and the microcode. The results of the analysis are described in Chapter 4.

The implementation of the MapMem utility is divided into three parts: reading from Anaconda memory, writing to Anaconda memory, and then applying this functionality inside the application. Chapter 5 describes the process in detail.

The initial network architecture design requires some modifications to the Windows NT environment. Chapter 6 describes a custom application that is needed to implement the MapMem utility and to create a socket interface to the ported TCP/IP stack (part of the XINU operating system [21]). By creating the custom application with these features, the Windows NT kernel and Network Driver Interface Specification (NDIS) layer can be bypassed. This chapter also discusses a plan to port the TCP/IP stack onto the NIC and defines the socket-to-TCP/IP stack and TCP/IP stack-to-NIC interfaces required for data transmission.

Chapter 7 describes the final, proposed design. In the process of designing the initial architecture, complications regarding MapMem and the TCP/IP stack arose. For these

reasons, the initial design was abandoned. The proposed design is similar to the initial design except for (1) Intel's EBSA-285 board will be used instead of the Anaconda board and (2) the whole Linux operating system will be ported to the EBSA-285.

Chapter 8 summarizes the work accomplished in this thesis.

## CHAPTER 2

### Project Overview

#### Background

The foundation for this thesis is based upon previous results obtained from network performance research done at Cal Poly. 3Com provided an isolated Ethernet local area network (LAN) to carry out the research. The network consists of five personal computers (one server and four clients) connected via 100BaseT Fast Ethernet. The four clients run Windows NT 4.0 operating system and each have a 3COM 3C905 Ehterlink-II NIC. Each client also has a Pentium Pro CPU. This isolated LAN will be referred to as the testbed.

The purpose of the research was to analyze network performance by measuring latency, throughput, and CPU utilization. The first step was to decide how to obtain these measurements. Two routes were taken. Yu Guang Liang proved that network performance measurements could be collected through hardware instrumentation with a logic analyzer [9]. Alternatively, Siu Ming Lo used software instrumentation at the application layer to acquire latency, throughput, and CPU utilization measurements [10].

Peter Xie also implemented software instrumentation to collect measurements [25]. However, Xie collected time stamps in the TCP/IP and User Datagram Protocol (UDP)/ IP stacks. Microsoft Research (MSR) IP version 6 (IPv6) was used.

With the proof that hardware instrumentation can be used to obtain network performance measurements, the pursuit of a network performance measurement tool began. Angel Yu began the development of a custom Windows NT parallel port driver that would be

used to send a signal level change to the parallel port [26]. A logic analyzer connected to the parallel port can be configured to trigger on the signal level change. Jim Fischer continued the development of the driver and also completed the measurement tool [6]. Fischer's application required a logic analyzer to be connected to the Peripheral Component Interconnect (PCI) bus and to the parallel port of the machine sending data. The application measured the time elapsed for the first byte of payload data to reach the PCI bus (start latency) and the time elapsed for the last byte of payload data to go across the PCI bus (stop latency). The rate at which the NIC receives data from the host (transfer rate) was indirectly measured as a result of the stop latency measurements.

The next step was to conduct tests on different platforms and compare the performance measurements. The comparisons were made between Windows NT and Linux operating systems running various combinations of TCP and UDP with IP version 4 (IPv4) and IPv6. As shown in Table 1, UDP gives a higher transfer rate than TCP. Also from these results, Linux has a larger transfer rate than Windows NT with UDP and IPv4.

Another conclusion can also be drawn from these transfer rate measurements. Theoretically, the PCI bus has a maximum transfer rate of 132 MB/sec and Ethernet transfers data at 12.5 MB/sec. Since the PCI bus transfers data at a faster rate than the Ethernet, clearly the bottleneck is not the PCI bus. Any transfer rate listed in Table 1 that is less than 12.5 MB/sec is therefore the bottleneck. The lowest transfer rates are found with Windows NT and TCP (8.5 MB/sec and 5.0 MB/sec).

Another platform was also used to conduct network performance measurements. At that time, a new technology was emerging, Intelligent Input/Output (I<sub>2</sub>O). Mauricio Sanchez

successfully ported the 3COM NIC driver for I<sub>2</sub>O [20] so that network performance measurements could also be obtained for this configuration.

**Table 1. Initial Performance Measurement Results [6]**

| <b>Platform Configuration</b>           | <b>Transfer Rate (MB/sec)</b> |
|---|-------------------------------|
| Windows NT 4.0 with TCP/IPv4            | 8.5                           |
| Windows NT 4.0 with UDP/IPv4            | 10.9                          |
| Windows NT 4.0 with TCP/IPv6            | 5.0                           |
| Windows NT 4.0 with UDP/IPv6            | 14.9                          |
| Redhat 6.0 Linux 2.2.5-15 with TCP/IPv4 | Not Available                 |
| Redhat 6.0 Linux 2.2.5-15 with UDP/IPv4 | 12.3                          |

Sanchez measured throughput and CPU utilization. The results showed that throughput was decreased with an increase in CPU utilization for the I<sub>2</sub>O configuration. As an emerging technology, I<sub>2</sub>O suggested that adding a coprocessor would offload the host processor. However, based upon the results of Sanchez's work, I<sub>2</sub>O actually added more work to the host [20].

The conclusions drawn from the research suggests that the Windows NT operating system is the limiting factor (bottleneck). A possible solution to this problem is to bypass the TCP/IP protocol stack and the NDIS NIC driver in NT. Off-loading the TCP/IP stack and the

NDIS NIC driver would reduce the number of transactions in the NT kernel, avoiding the bottleneck created by the operating system. The purpose of this thesis is to design a network architecture that off-loads the TCP/IP stack onto an intelligent network card that can be attached to the PCI bus.

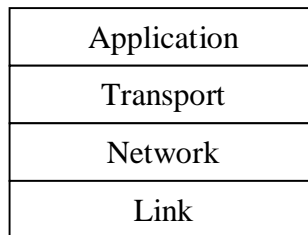
### Network Architecture

This section provides an overview of the network architecture fundamentals. In addition, it describes the typical network architecture for Windows NT. Finally, a description of the architecture with the TCP/IP stack and NDIS NIC driver removed from the NT operating system is provided.

### TCP/IP Protocol Suite

The TCP/IP Protocol Suite is composed of four layers as described in Figure 1. Each layer has a protocol that is in charge of a particular part in the process of communications. The highest layer is the application layer. Its task is to handle a specific application's details. Telnet and File Transfer Protocol (FTP) are examples of TCP/IP applications.

The layer immediately below the application is the transport layer. The main goal is to provide a flow of data between two hosts. Through this layer, the application can receive and transmit data. There are two protocols that are prevalent in this layer: TCP and UDP. TCP provides reliable transmission of data because of its concern with data arriving out of order, data loss and duplicity. UDP is simpler since it only sends out data via datagrams and there is no guarantee that the data will arrive at the destination.



**Figure 1. TCP/IP Protocol Suite**

The third layer is the network layer. This layer is concerned with routing. The protocol basically specifies how to assign addresses and how to route packets from one network to another. There are three network layer protocols in the TCP/IP suite: IP, Internet Control Message Protocol (ICMP), and Internet Group Management Protocol (IGMP).

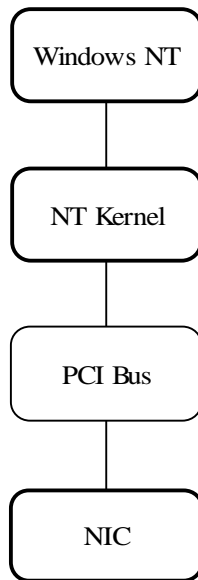
Finally, at the lowest layer is the link layer. The device driver in the operating system and NIC are part of this layer. Together, they manage the hardware details. This layer also decides how to handle data in terms of frames.

#### Traditional Windows NT Model

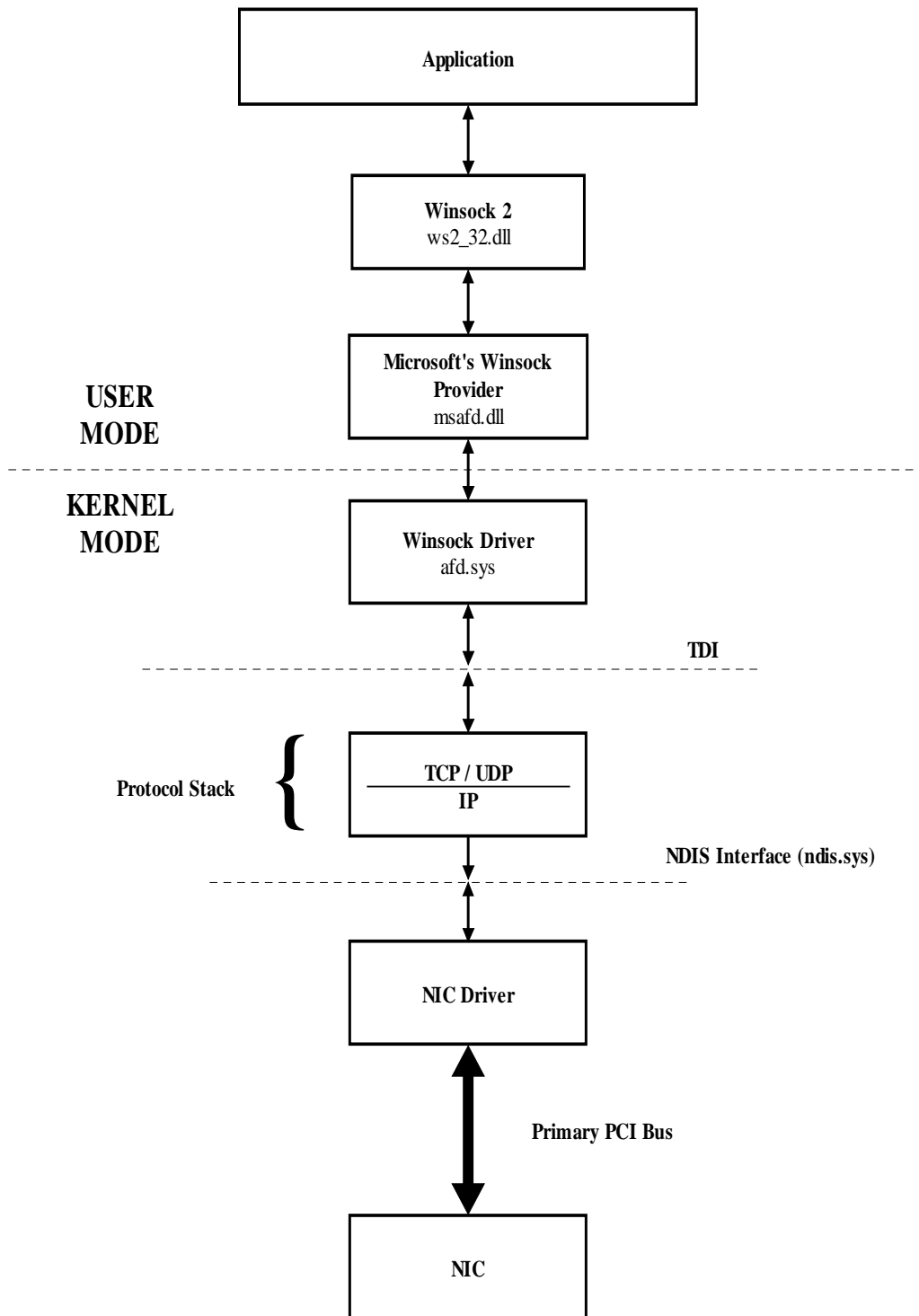
Figure 2 shows an overview of the traditional network architecture found with Windows NT. At the highest layer is the Windows NT operating system. The next layer contains the NT kernel that communicates with the NIC via PCI bus. With this structure, the TCP/IP stack is nestled in the NT kernel.

The networking protocol is displayed in Figure 3. The application makes a call to Winsock to create a socket for communication. The Transport-Device Independent (TDI) is the protocol interface at the kernel-level. The protocol stack, composed of TCP / UDP and IP

protocols, uses NDIS as the means to communicate with a NIC driver. The NIC driver transmits data to and from the NIC hardware via PCI Bus.



**Figure 2. High Level Networking Architecture in Windows NT**



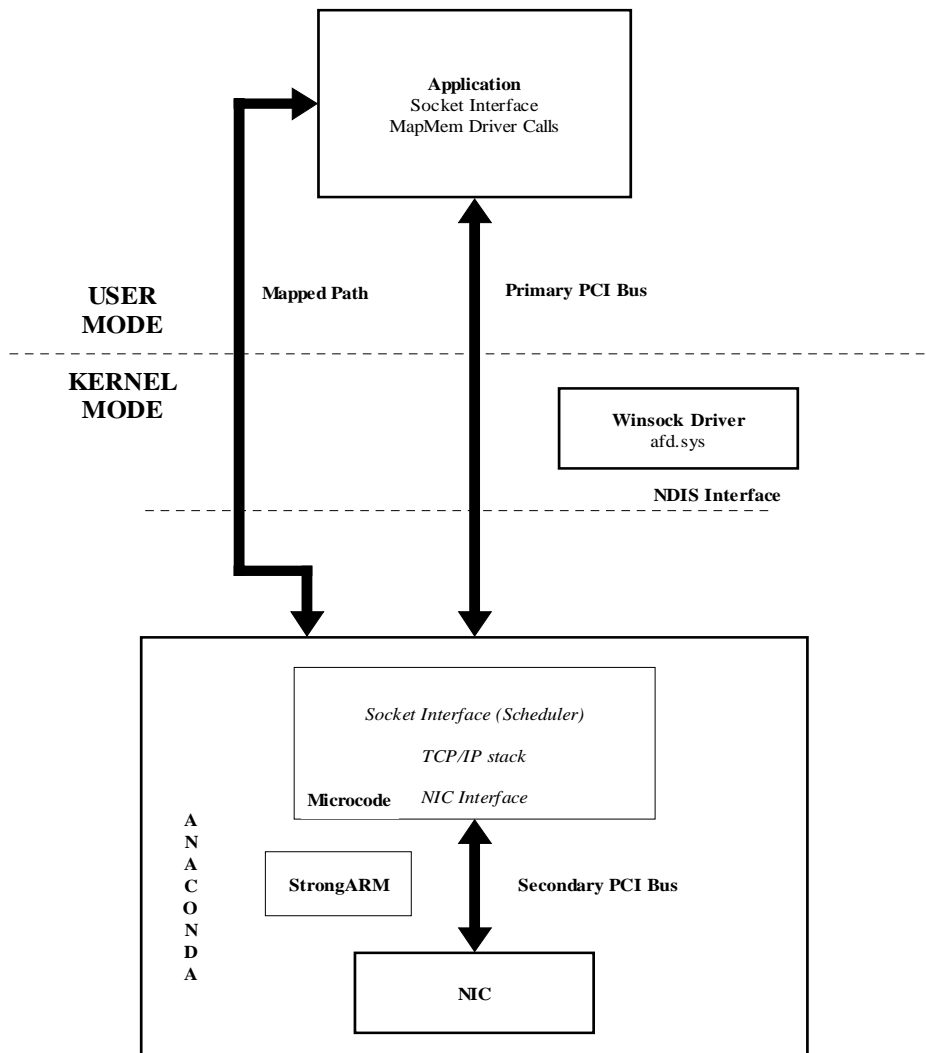
**Figure 3. Windows NT Networking Protocol**

### Design Plan

Originally, the network architecture model was to port the XINU [21] operating system onto the intelligent external board (3Com's Anaconda NIC). This initial design consisted of first understanding the Anaconda microcode and a memory mapping device driver, MapMem, provided by Microsoft Device Driver Kit (DDK) [11]. As the design progressed, it was found that porting XINU would require a tremendous effort. In addition, there were some complications with MapMem that has a key role in the process. As a result, a final prototype design is proposed: The entire Linux operating system will be ported onto the EBSA-285 board that contains a StrongARM ARM processor and 21285 controller chip (contains DMA capability).

### Initial Design Plan

In this architecture model, shown in Figure 4, the TCP/IP protocol is ported from the NT kernel to the Anaconda board. The memory mapping device driver, MapMem, is used to bypass the socket and NDIS layers in the traditional NT network architecture. Using this driver, a user application can access a memory region from a device on the PCI bus, i.e. the external board. Direct access (read/write) to an area in memory on the external board can be achieved in this fashion. A custom application with its own socket interface is required to obtain a connection with the ported TCP/IP stack. With this application, a socket can be created without the use of the Winsock components that reside in the kernel. The NDIS layer is bypassed through the use of the MapMem driver and the addition of a (new) mapped path. The end result is an intelligent network interface card.



**Figure 4. Initial Network Architecture**

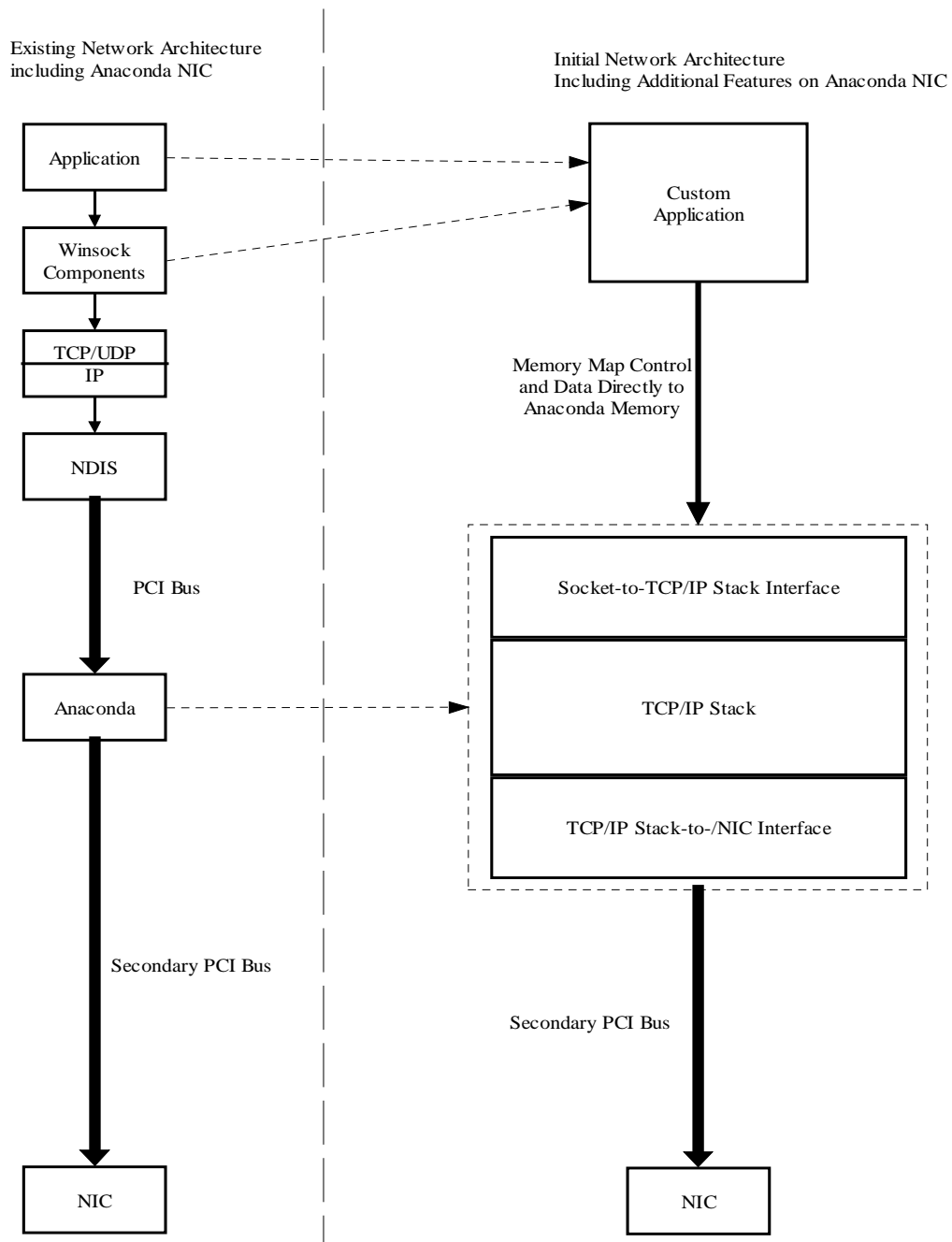
The main objective of the initial network architecture model is to bypass the Windows NT kernel. It achieves this by dividing the work into two parts: a memory mapping technique and porting the TCP/IP protocol stack to the NIC. As shown in Figure 5, the memory

mapping driver, MapMem is used in a custom application that possesses the same functionality as the application and Winsock components in the existing architecture.

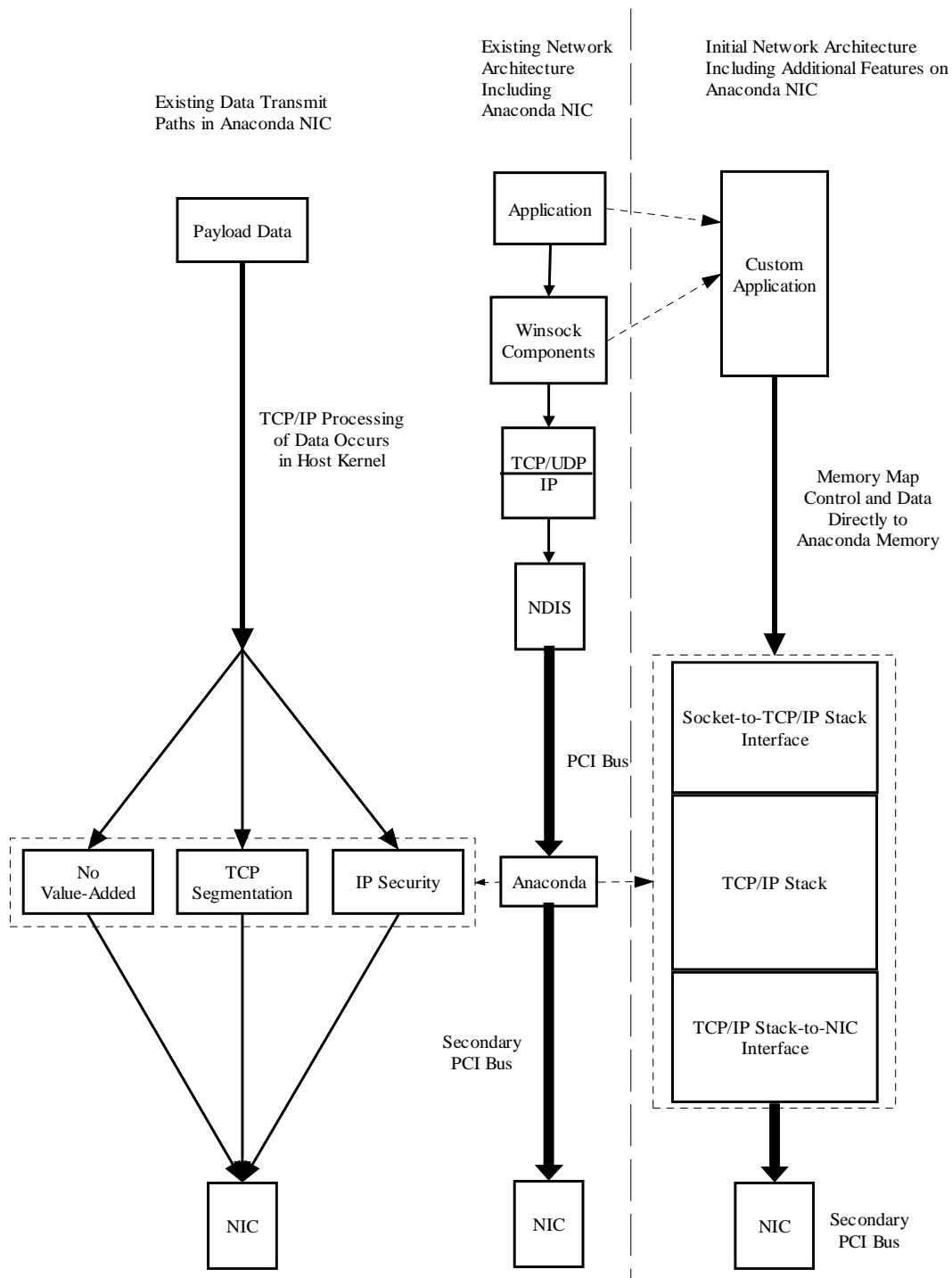
MapMem is used to map the control and data to Anaconda local memory, bypassing the NT kernel.

In order to port the TCP/IP stack onto the Anaconda NIC, a few new interfaces must be defined as well as modifying existing ones. Since a communication channel exists through a socket, an interface between the socket and TCP/IP stack must exist. The top-end of the stack needs to relay information to and from the socket. Similarly, the stack must also communicate with the Anaconda microcode since it will use one of the existing paths as a model to build the data packets for transmission. Finally, an interface must exist between the TCP/IP stack and NIC so that data can be passed to hardware. This interface should be similar to the existing Anaconda microcode/NIC interface. The interfaces are shown in Figure 5.

The mapped path described in the initial network architecture model does not replace any of the existing transmit paths as shown in Figure 6. Instead, the mapped path is an addition; data can follow any of the four paths. The memory mapping technique allows the NT kernel to be bypassed completely and allow the Anaconda board to process the data for transmission. In the existing data paths, the data is processed by the TCP/IP stack in the kernel. Then it is passed down to the Anaconda NIC for further processing. Details about each of the existing paths are described in Chapter 4.



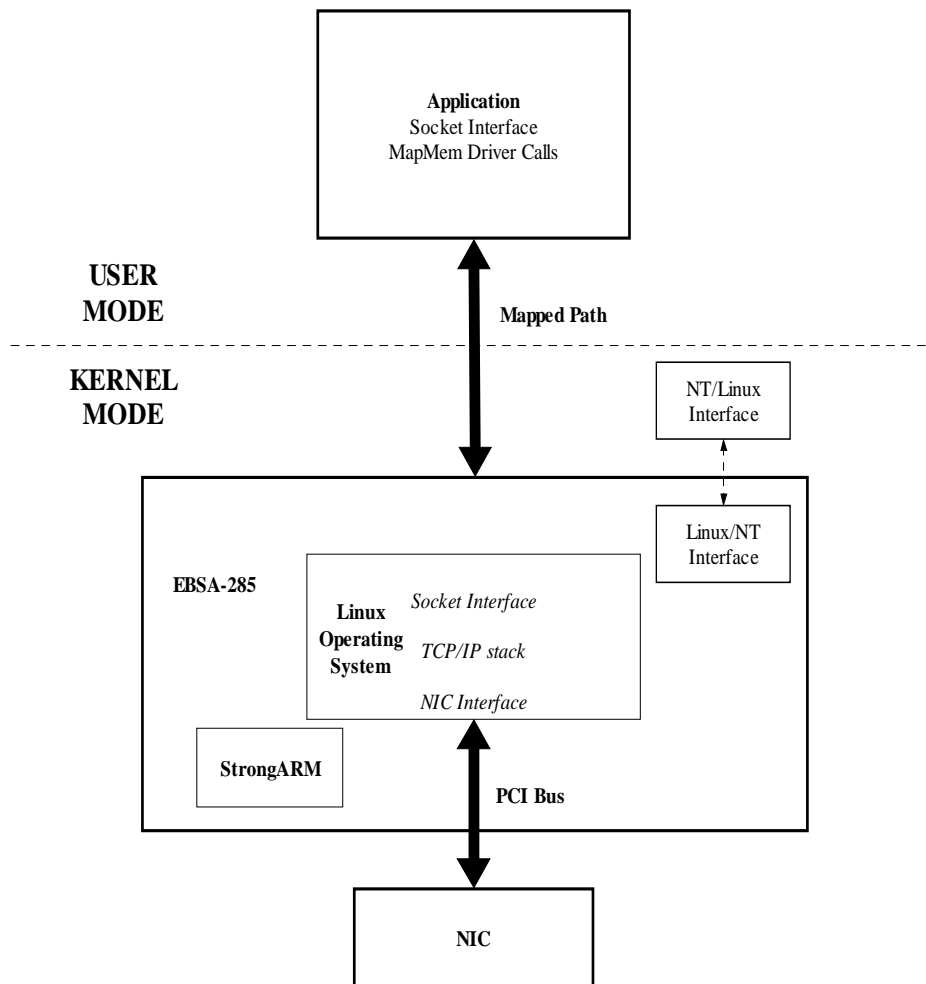
**Figure 5. Control/Data Path in Initial Network Architecture**



**Figure 6. Four Control and Data Paths for Transmit**

### Proposed Design

After discovering the difficulties with the XINU operating system and MapMem, the original design plan that consisted of porting XINU onto the Anaconda board was abandoned. Instead the network architecture as shown in Figure 7 is proposed.



**Figure 7. Proposed Network Architecture**

The design is very similar to the initial network architecture design, only simplified. A custom application is still required and will still have to include a socket interface and MapMem driver calls. The Linux operating system, which includes a socket interface, TCP/IP stack, and a NIC driver, is ported onto the EBSA-285 board.

## CHAPTER 3

### Development Environment

#### Components

The development environment consists of a target machine, Xena, and a host machine, Hercules. The target machine is a Windows NT 4.0 workstation with a Pentium Pro processor. It uses a 3Com development board, the Anaconda, as its NIC. The Anaconda houses a StrongARM 110 (SA-110) processor, two Hurricane ASICs, and a Footbridge DMA engine [23]. The Hurricane ASIC is 3Com's PCI 10/100 Mbps Ethernet chip. On the Anaconda, each Hurricane chip is controlled by the SA-110. There are 2MB of onboard SDRAM.

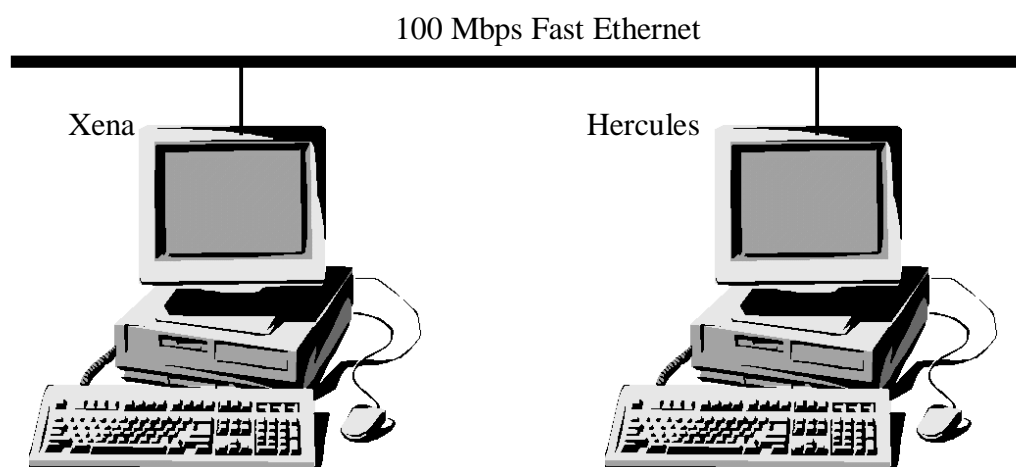
The host machine is a DELL server with dual 400 MHz Pentium II processors that is performing as a Windows NT 4.0 workstation. The ARM Software Development Toolkit v2.50 [1] is installed on the host for Anaconda microcode modifications and development.

#### Setup

As shown in Figure 8, Xena and Hercules are connected via 3Com's OfficeConnect Dual Speed Hub 8 onto an isolated 100 Mbps Fast Ethernet network. The two workstations are configured into an isolated network so that the traffic on the Ethernet can be controlled for testing purposes.

Before booting Xena, the Anaconda microcode must be up and running. Otherwise, when NT tries to initialize the NDIS Anaconda driver, the machine will blue screen with a page fault in non-pageable area. Appendix A: Execution Procedure for Anaconda Microcode

describes how to build and download the microcode. If Xena does not crash by the time the login dialog window appears, then the Anaconda microcode and NDIS driver have successfully started.



**Figure 8. Isolated Development Network**

### Development Tools

Throughout the design of the network architecture, a variety of development tools were utilized. The tools and brief descriptions of each are listed below. In addition, the tools are described in detail in the following chapters in the context of which they were used.

- ARM Software Development Toolkit v2.50 is used to analyze and modify the Anaconda microcode. The project manager is used to compile and build the microcode whenever changes were made. The debugger is used to set breakpoints in the microcode for tracing the different execution paths.
- NuMega SoftICE 3.0 is used to halt the NT operating system while debugging the Anaconda microcode. This tool was used to control NT in order to investigate the microcode.
- Send/Server Application program is used to send data payload. Data was transmitted when analyzing the microcode and during the verification of the MapMem utility.

- Ping/Pong Application program is used to send data payload. Data was transmitted when analyzing the microcode and during the verification of the MapMem utility.
- NetXRay 3.0.3 is used to capture the traffic on the wire. This tool is used to verify that a particular transmit path in the Anaconda microcode was indeed transmitting data.
- Microsoft Visual C++ 5.0 is used to modify the MapMem application, Maptest2. This tool is used when verifying the read and write capabilities memory mapping utility.

## CHAPTER 4

### Analysis of Existing Anaconda Microcode

Since the TCP/IP protocol stack is relocated from the Windows NT operating system to the Anaconda, packet processing will have to be handled directly on the Anaconda. This suggests that the data to be transferred must be accessed by Anaconda. One of the functions of the Anaconda microcode is to manage the data transmission. In order to modify the microcode to handle all packet processing, an understanding of the transmit and receive paths in the existing microcode must be obtained. The following sections describe each path in detail.

#### Transmit path

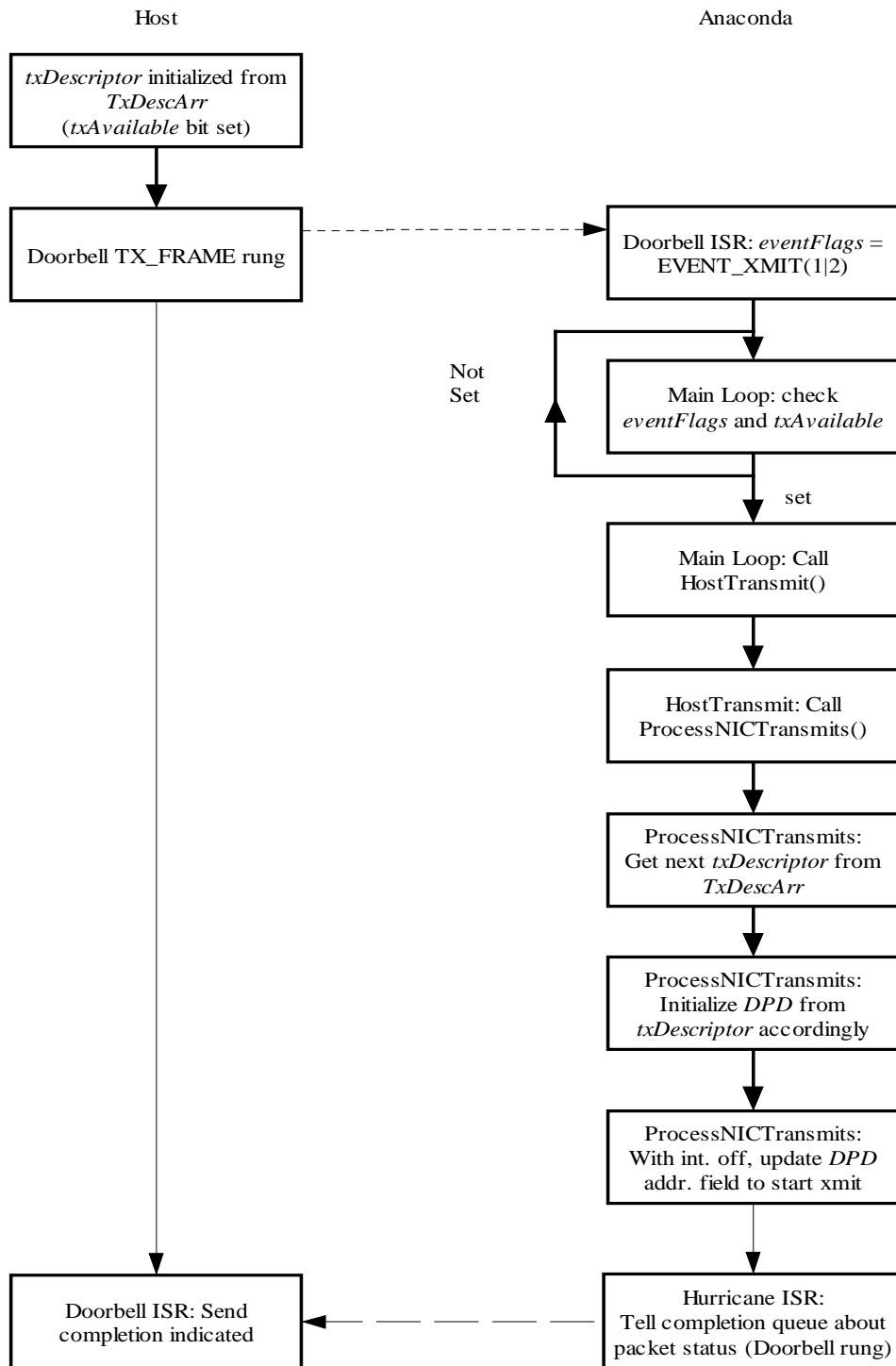
As shown in Figure 9, the general transmit path is initiated by a doorbell-interrupt mechanism or by the Anaconda microcode polling a bit (*txAvailable*). When the host (machine with the Anaconda board) has data to send, it sets up a transmit (TX) descriptor and writes to a register, indicating to Anaconda that there is data to be sent. The Anaconda doorbell interrupt service routine (ISR) sets a flag that will start execution of the transmit path. Once the data is processed and shipped to the NIC via the two functions **HostTransmit()** and **ProcessNICTransmits()**, another doorbell interrupt is used to inform the host that the data transmission is complete.

Although the doorbell-interrupt mechanism exists in the source code it has been commented out. The polling mechanism is used instead. When the host is ready to transmit

data it eventually sets *txAvailable*. The microcode polls this bit by checking it in the main loop. When it is set the general transmit path is executed.

The general transmit path in the Anaconda microcode is divided further into three individual transmit paths: no-value-added, TCP segmentation (TCP Seg), and IP security. To execute the no-value-added path, the data is sent directly to the NIC from host memory. The IP security consists of adding encryption to the data before transmission. This means that the whole data buffer (header information and payload data) must be copied from host memory into Anaconda local memory. The TCP segmentation path combines the strategies of the no-value-added and IP security paths. Only the header information is copied into the local memory on Anaconda. After processing the header, the payload data is sent to the NIC from host memory. Appendix B: Packet Send from Anaconda Perspective traces through the transmit algorithm in detail.

After analyzing the three different types of transmit paths available in the Anaconda microcode, the most logical path to implement in the porting of the TCP/IP stack is the TCP Segmentation path. This method gives the microcode access to the data payload without the cost of more local memory for buffering the data. This path also saves time since the microcode does not have to perform a copy operation for the potentially large data payload.



**Figure 9. Flow Chart of Transmit Path**

## TCP Segmentation Path

This section describes how a TCP segment packet is processed.<sup>1</sup>

TCP Segment packets (i.e., packets requesting TCP segmentation) are identified by the control field in the transmit (TX) descriptor (Anaconda Host Interface Specification v2.1, p.9). The transmission of TCP Segment packets is the same as normal packets (packets that do not require TCP segmentation or IP security) up to the point that the download packet descriptors (DPDs) for the packets are about to be sent out the NIC. For normal packets, the DPDs can be sent directly down to the NIC for transmission. For TCP Segment packets this is not possible since it is up to the microcode (ucode) to divide the packet into the appropriate number of TCP segments. When the DPDs for TCP Segment packets have been enqueued to the TCP Segment list, the function called to process them is **TCPSegSend()** (from *Tcpseg.c*).

The purpose of the **TCPSegSend()** function is to break up the packet supplied by the host device driver into segments specified by the maximum segment size (MSS). The segments are then passed down to Hurricane, for transmission.

When the **TCPSegSend()** function is invoked, the first action taken is to dequeue all DPDs that had been previously placed on the *TCPSegDpd* list. The dequeue process is accomplished through a call to **DPDListDequeue()** that returns the addresses to the head (*tcpHead*) and tail (*tcpTail*) DPDs, as well as the number of DPDs dequeued. With the DPDs dequeued, the next step is to individually process each DPD by traversing the DPD list. Each DPD is supposed to correspond to a complete packet that needs TCP segmentation. A packet

---

<sup>1</sup> This detailed analysis was obtained with the help of Mauricio Sanchez.

described by a DPD may contain up to sixteen data (DMA) fragments.<sup>2</sup> The fragmentation of a packet leads to some more work on part of the ucode, but usually is offset by the gains made in host memory allocation efficiency. See Anaconda Microcode Design Specification v1.5, p.43 for the details on how a DPD describes a packet.

After the DPDs are dequeued, each DPD must be processed via a primary for loop. The first step is to update the *FrameLength* field of the DPD by masking out the upper 16 bits of the 32-bit field. The upper 16 bits (starting at most-significant-bit) are control flags whereas the lower 16 bits contain the actual length. The modified *FrameLength* field is then stored in the next available entry in the *InLenArr*. The *InLenArr* is a global array consisting of 512 integers.

The first significant step in processing the current DPD is to copy the TCP/IP header from host to Anaconda memory. It is assumed that the first 128 byte described by the fragment array (each entry in *fragArr* contains the starting address and length for a particular fragment) for that particular DPD will contain the header. Whether part of a single fragment or multiple fragments, the first 128 bytes are copied from host memory (using the **MemCopyB()** function) to a raw data buffer (*tcpHdrTemplate*) in Anaconda memory. The raw data buffer is defined as an unsigned character array, so there is no easy way of manipulating individual header fields without using pointer arithmetic. To simplify the method, the beginning addresses and lengths of the TCP and IP headers are calculated for the

---

<sup>2</sup> "Fragment" is not used to describe data units resulting from IP fragmentation. The data payload is DMA'd and so each "fragment" is a DMA fragment. Each TX descriptor, and therefore DPD, may contain up to 16 fragments.

raw data buffer. Besides just knowing the starting address of the headers, the appropriate section of the raw data buffer is type cast into two data structures, the TCP and IP header structures. By type casting the data buffer into the TCP/IP combination, manipulating the individual fields is much easier.

At this point, certain fields in the TCP header are manipulated as follows:

1. A copy of the sequence number is swapped and stored into its own variable, *sequenceNumber*. The swapping turns little endian to big endian.
2. A copy of the code field is stored into its own variable, *tcpControlBits*. Control bits 2 and 1 get set if already set.

Modifications are made to the TCP header as follows:

1. The PUSH and FIN bits are turned off in code field in the actual header (not the copy of the code field)

After modifying the TCP header fields, the next step is to determine the beginning address and length of the TCP data payload. The total length header is calculated as the sum of the MAC header (always 14) + IP header length + TCP header length. The size of the TCP payload section is calculated as the difference between the *FrameLength* field (from the DPD) and the total header length. Calculating the starting address of the TCP payload requires traversing the fragment list of the DPD.

Knowing the start and length of the TCP payload, the next step is to perform the segmentation. Segmentation entails dicing the TCP payload into the necessary number of segments. For each segment an individual DPD and TCP/IP header must be created. Before beginning to setup DPDs it is necessary to know how much data each segment can (or will be

able to) handle. The limit is known as the Maximum Segment Size (MSS) and unless it is zero in the original (parent) DPD, the MSS is calculated as the parent's MSS + total header length. If the MSS in the parent was zero, then MSS is calculated as 256 + total header length. Note that the MSS applies only for regulating the limit of the TCP data payload section in determining how many segments (DPDs) are actually necessary. With knowledge of how much of the TCP data payload a segment can carry, the next step is to start allocation and stringing together the corresponding DPDs.

Individual DPDs are obtained from the free DPD pool via a call to **DpdAllocate()**. After initializing the control fields in the DPD to a default state, the next step is to setup the data fragment entries in the DPD to point to the local TCP/IP header and to the host TCP payload. Each segment (child DPD) requires its own TCP/IP headers with the only variation between headers being the length field in the IP header and the sequence number in the TCP header.

Unlike the TX descriptor and DPD rings that are pre-allocated data structures, TCP/IP headers are allocated from a block memory pool. Management of the memory pool is done by the microcode through the various routines from mempool.c. Why a memory pool is used rather than a static number of TCP/IP headers (in a ring configuration) is an implementation issue. Nevertheless, one call to **BlockPoolAllocate()** is made to allocate a chunk of memory for one child TCP/IP header. The previously modified TCP/IP header that is stored in the raw data buffer is then copied into the memory block. Much like the raw data buffer, the memory block has no organization. Therefore, the block is sectioned off into two appropriate chunks, which are then type cast into a TCP and an IP header.

The first data fragment entry in the DPD corresponds to the memory block containing the copied TCP/IP header. The subsequent data fragments will correspond to TCP payload data. While setting up the fragment entries with the TCP payload data information, it could occur that the MSS is reached and not all the payload data has been accounted for. This means that the TCP packet will traverse multiple segments (DPDs). If this case happens (which is very likely when doing bulk data transfers) then rather than immediately allocating another DPD, the processing of the current DPD is completed. This last phase of processing consists of modifying the IP length field, the TCP sequence number field, and setting up the length field in the DPD.

If the current DPD is not the last necessary DPD because the TCP payload was too large, then another DPD must be allocated, processed, and placed onto the linked-list of DPD. However, DPD allocation will not continue unabated, since after ten DPDs, the microcode will send out the thus far processed DPDs (ten in total) to not starve the NIC from working. Up until the threshold is reached, the DPDs will just collect on the linked-list.

When the final DPD is being processed, there is some extra setup necessary for it. The last DPD will have the PUSH and FIN bits set in the TCP control field, if the original parent packet had them set. Furthermore, the last child segment (DPD) will have its packet pointer set to that of the parent.

Up until now the DPDs were being strung together as they were allocated and configured. To send them out to the NIC these DPDs are queued onto the DPD queue via **DPDListEnqueue()** and then a call to **NicSend()** is made. See Anaconda Microcode Design

Specification v1.5, pp.34-35 for a visual description of the process. Appendix C:

Pseudocode for TCPSegSend() lists the pseudocode for the function **TCPSegSend()**.

### Testing Transmit Path (TCP Segmentation Path)

#### *Test Setup*

To test the TCP segmentation path, a buffer containing only '\*' of different sizes is sent from the socket layer. The microcode is modified so that it can detect the payload that contains an arbitrary number of '\*'s. When this payload is detected the TCP segmentation flag is set for that particular packet. With this flag set, the packet will be processed via the TCP segmentation path.

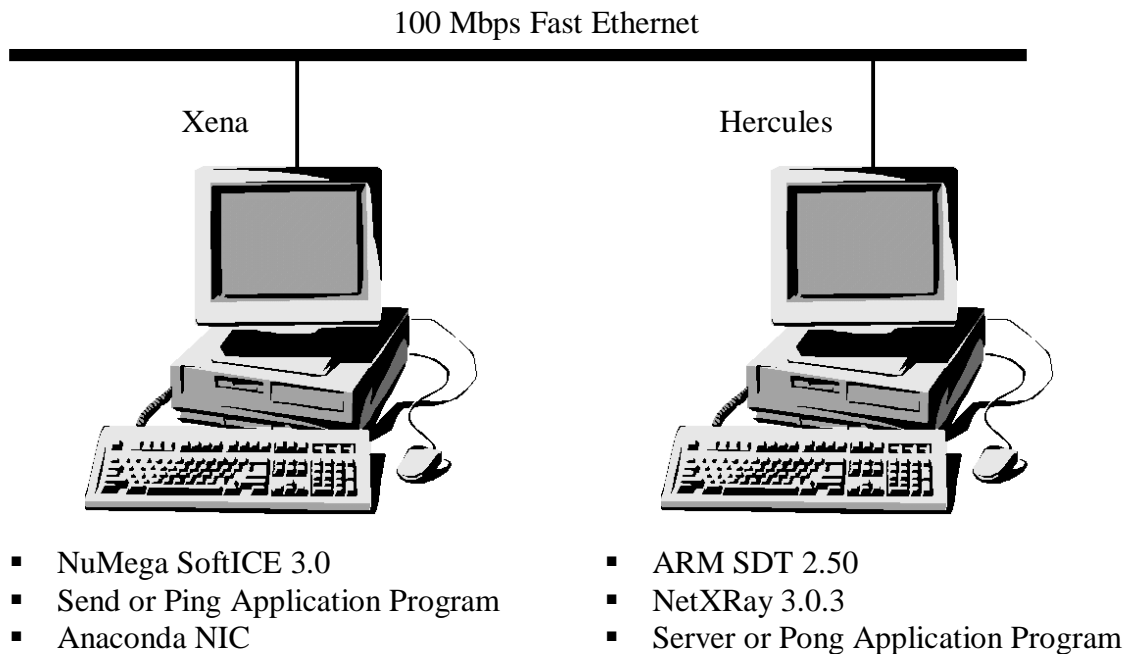
Two application layer programs were used as test drivers to send the buffer of '\*'s. Dr. Chris Scheiman's Ping Pong program (Appendix F: Ping Pong Application Program) sent 10 packets of 29 bytes in one run. Jim Fischer wrote a Send/Server program (Appendix G: Send/Server Application Program) where the size of the buffer could be specified. An option is provided so that the same size packet could be sent continuously.

The testbed consists of two NT workstations, Xena and Hercules, as shown in Figure 10. The two machines make up a 100 Mbps Ethernet network via 3Com's OfficeConnect Dual Speed Hub 8. Xena contains the Anaconda board, and Hercules is a DELL server but is actually running as an NT workstation. To test the transmit path, Xena sends data via either the Send or Ping application program to Hercules that is running the respective receiving end application; the Server or Pong application program. Hercules is also running NetXRay 3.0.3 to capture the traffic on the wire. Xena also has NuMega SoftICE installed so that the NT operating system can be controlled while debugging the Anaconda microcode. The ARM

Software Development Toolkit (SDT) is installed on Hercules for development and debugging via Angel debugger.

### *Setting Breakpoints in Microcode*

The Anaconda microcode must be running at all times in Xena or else the host machine crashes. In order to set a breakpoint in the microcode, the NT operating system in Xena must be halted first via SoftICE (<ctrl> d). Then a breakpoint may be inserted into the Anaconda microcode via Angel debugger.



**Figure 10. Two-Workstation Testbed for Transmit**

## **Test Cases**

### *Test Case 1:*

The first four bytes of the payload in host memory are copied down onto local memory. Assuming that the TCP/IP header is in the first 40 bytes associated with this DPD, the first byte of the payload follows immediately. To skip the TCP/IP header, the code used to copy the first 128 bytes into Anaconda local memory from host memory (**tcpSegSend()** in `TCPSeg.c`) was modified and used. (A call to copy host memory at this point will be a performance hit but the goal is to prove its functionality.)

Once the beginning of the payload is found (i.e. 40 bytes skipped), the first 4 bytes of the payload located in the current fragment was copied into a temporary buffer. The temporary buffer is compared to the string “\*\*\*\*”. If they match the TCP segmentation flag is set. This causes the DPD to be chained onto the TCP segmentation chain and **tcpSegSend()** is called.

When control is passed over to **tcpSegSend()**, the URG bit in the control field is set to mark that the packet had gone through the TCP segmentation path.

After running this test with different sizes of data, it was found that the TCP segmentation path was not being executed.

### *Test Case 2:*

Since an assumption was made about skipping 40 bytes in the buffer meant skipping the entire TCP/IP header, there was no verification that this was actually the case. Therefore the four bytes that were copied may not have been the first four bytes of the data payload. To guarantee the beginning of the payload, the first 128 bytes associated with the DPD is copied

into a buffer. Then the buffer is traversed to find the start of the payload. If the 128 bytes are not in the first fragment associated with the DPD, as many fragments as necessary are copied until all 128 bytes were in the temporary buffer. In order to index into the buffer, the TCP and IP header lengths and data payload start address and length are calculated. The code used to do all this is an exact replica of the code used in **tcpSegSend()** in TCPSeg.c. The first 4 bytes of the data payload is copied into a temporary buffer and compared to “\*\*\*\*\*”. If they match, the TCP segmentation flag is set. This causes the DPD to be chained onto the TCP segmentation chain and **tcpSegSend()** is called. A breakpoint is set at the call to **tcpSegSend()**.

Once the microcode stopped at the breakpoint, the breakpoint was removed. The debugger continued to execute the microcode until it reached an unknown breakpoint. After stepping through the microcode, the unknown breakpoint seemed to be caused by a read to memory. The microcode was trying to read the sequence number field in the TCP header. The read was not successful because the sequence and acknowledgement numbers are of type unsigned long and both straddle the 32-bit alignment boundary in memory. Since the alignment is off, a read to the sequence number causes an error.

To remedy the alignment problem, the sequence number is read byte by byte. The four bytes are stored into a temporary variable little endian style. Then the macro SWAPDWORD alters the sequence number to big endian.

With this last attempt, the payload data is transferred. The Anaconda microcode divides the payload so that every Ethernet frame contains no more than 256 data bytes. However, there was not an acknowledgement packet for each Ethernet frame that was sent for large

data transfers. Most of the time one acknowledgement came back for a group of frames. Also, some tests showed that a group of frames were sent and immediately following the frames was the respective number of acknowledgements. But the acknowledgements all contained the same sequence, acknowledgement, and window size numbers. We would expect the acknowledgement number to change according to how many bytes were received up until that point. Some possible explanations:

1. *The Anaconda microcode may perform in a non-standard fashion.*

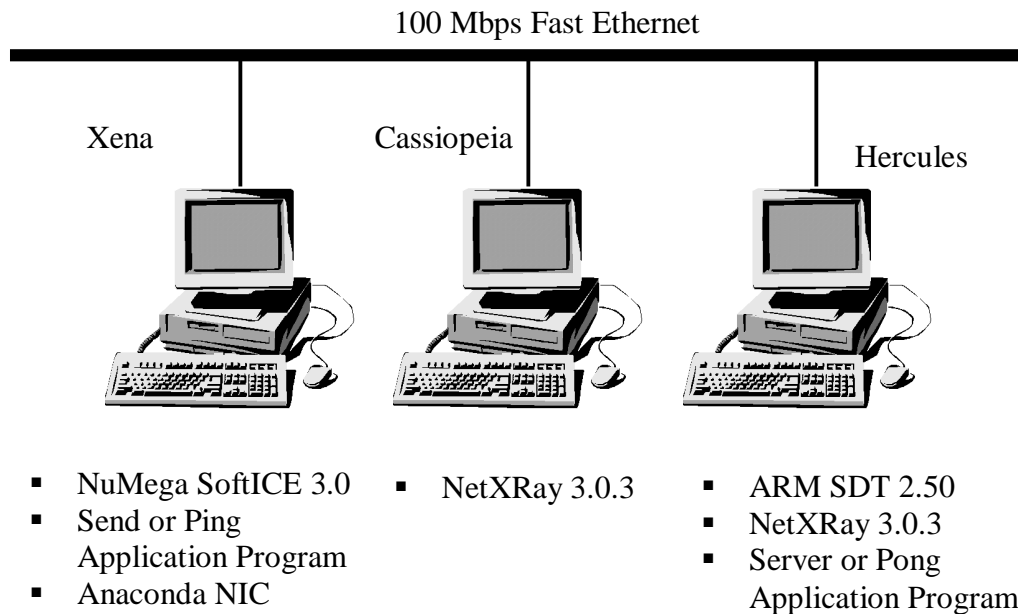
Results from tests that were run on machines with standard NICs showed that this was not the case. With the standard NICs, there was not an acknowledgement for every frame.

2. *NetXRay may be dropping packets.*

To test this explanation a third machine, Cassiopeia, was added to the test setup (Figure 11). The third machine is used to run NetXRay and since it is not running anything else the traffic between Xena and Hercules should be captured without any problems. Using a third machine that is in “promiscuous” mode is a better approach. Xena sent different data sizes (512, 900, 1000, 1460, 1500, 9000 bytes) to Hercules. Again, for each data size, not every frame received an acknowledgment. This leads to the conclusion that NetXRay is not dropping acknowledgement packets.

However, there were strange results when a data size of 9000 bytes was sent from Xena to Hercules. As seen in previous results, every frame did not receive an acknowledgement. Likewise, the acknowledgements that were received are cumulative as seen before. The NetXRay trace looks “normal” up until frames 49-61

when Hercules keeps sending acknowledgements with acknowledgement number 44572. This means that Hercules has received everything up to 44572. The data frames sent up to frame 49 add up to the 9000 bytes that were originally sent.



**Figure 11. Three-Workstation Testbed for Transmit**

Starting with frame 62, Xena finally retransmits the frame with sequence number 44572. We would expect Xena to retransmit 3852 bytes starting with sequence number 44572 so that the 9000 bytes would be received by Hercules. However, only 1460 bytes are retransmitted. Hercules sends out acknowledgements with acknowledgement number 48424. This would be the next sequence number if Xena had retransmitted the missing 2392 bytes. The next thing that happens is that the connection is terminated. It may have been disconnected prematurely. However,

the purpose of this test case was to verify that every frame does not receive an acknowledgement, and again this is what occurs.

3. *Protocol uses delayed acknowledgments where the acknowledgment that is sent is a cumulative acknowledgement (i.e., updates acknowledgement number according to how many bytes it has received up to that point). Acknowledgements may be time driven (acknowledgment sent before timer runs out or receives frame with PUSH bit set), not driven on how many bytes were received.*

Craig Zacker addresses the concept of delayed acknowledgements used by TCP in TCP/IP Administration [27]. Instead of acknowledging every segment that is received, acknowledgements are sent when the timer cycles.

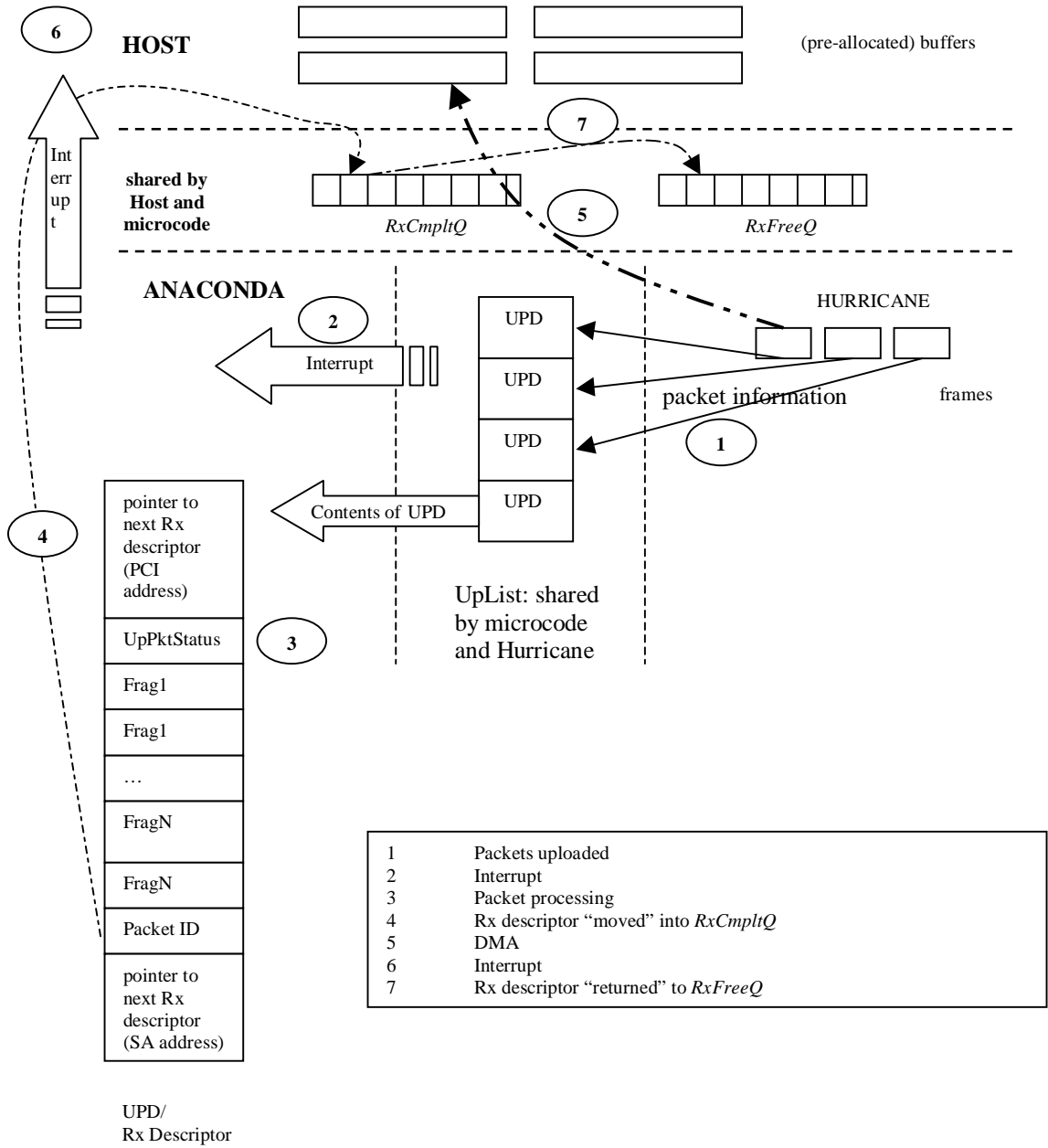
### *Summary*

The TCP segmentation path was verified through analysis and testing. During this process, the concept of delayed acknowledgements utilized by TCP was encountered.

### Receive path

The receive path in the Anaconda microcode is similar to the transmit path except that the data payload flows in the opposite direction. Figure 12 displays the receive path. First, Hurricane uploads the received packets to its local memory. It fills up the free receive (RX) descriptors in the *UpList* with packet information. Then an interrupt is raised to the microcode. Upon receiving the interrupt, the Anaconda microcode processes the packet according to the *UpPktStatus* field in the RX descriptor. Then the microcode places the packet identification (ID) number (field found in RX descriptor) into the *RxCmpltQ* to

indicate to the host that there is a packet for it. The actual data payload is DMA'd from Hurricane to host memory. When this process is complete, the Anaconda microcode raises an interrupt (RX\_COMPLETE) to the host. Finally the host passes the packet ID into the *RxCmpltQ* to the upper layers. The RX descriptor is returned to the *RxFreeQ* when all processing is complete. The two queues, *RxFreeQ* and *RxCmpltQ* are shared between the host and the Anaconda microcode. The *UpList* is shared between the Anaconda microcode and Hurricane. Appendix D: Packet Receive from Anaconda Perspective describes the receive path in detail.



**Figure 12. Receive Path**

## Testing Receive Path

### *Test Setup*

The testbed is exactly the same as the one for testing the TCP segmentation path where Xena and Hercules are two NT workstations and Xena has the Anaconda board. To verify the receive path, the Angel debugger was used to set breakpoints in specified locations in the microcode.

### *Test Case 1*

Xena sends 1200 bytes to Hercules. Xena is running the altered version of the TCP segmentation path and Hercules is running NetXRay. Breakpoints are set in the microcode in the receive path at the beginning of **NicUpComplete()** and at the call to **NicSetUpRxUpd()**.

On NetXRay, the connection and one acknowledgement were captured. The 1200 bytes were sent and received by Hercules. After stopping the capture (connection still exists), the receive path was stepped through without any problems via Angel debugger. It follows the route as outlined in Appendix D: Packet Receive from Anaconda Perspective.

### *Test Case 2*

Immediately after running Test Case 1, Xena sends 1400 bytes to Hercules. Xena is running the altered version of the TCP segmentation path and Hercules is running NetXRay. A breakpoint is set in the microcode in the receive path at the beginning of **NicUpComplete()**. A connection was not reestablished when running both Jim Fischer's send/server program and Dr. Chris Scheiman's ping-pong program.

Similar cases as described in Test Case 2 were tested, but these tests were run on just one original send/server connection. It seems that a connection between Hercules and Xena

cannot be reestablished once a breakpoint is set in the receive path. The Angel debugger also seems to crash at that point.

### *Another Approach*

Based upon the results, the testbed was reconfigured so that Hercules sends packets to Xena. With this method, the receive path in the Anaconda microcode would be traced with actual data payload packets instead of the acknowledgment packets.

### *Test Case 3*

A connection is established between Hercules and Xena. Xena is running the altered version of the TCP segmentation path. Hercules is running NetXRay. Hercules sends 1200 bytes to Xena. A breakpoint is set in the microcode in the receive path at the beginning of **NicUpComplete()**.

The breakpoint was reached. NetXRay captured packets sent by Hercules. Three Ethernet frames were sent adding up to 1200 bytes of data. The server program on Xena did not indicate that any bytes were received. Stepping through the microcode showed that the route as outlined in Appendix D: Packet Receive from Anaconda Perspective was executed. After reaching the end of the receive path, the server program on Xena indicated that it had received 1200 bytes.

### *Summary*

It seems that tracing the receive path with Xena sending data to Hercules was not a good idea. The source of the problems could have been because Xena had to request a connection from Hercules. When Hercules responded with a SYN (synchronization packet) it was not processed correctly since breakpoints were set. However, the key was to trace the receive

path for data packets. Having Hercules send data to Xena provided the ability to trace the path. With this approach, a connection was established first, before setting any breakpoints. This way, the handshaking was not stalled due to the breakpoints. This seems to be the cause of the problem instead.

## CHAPTER 5

### Windows NT MapMem Device Driver

#### What is it?

MapMem is a memory mapping device driver provided by Windows NT DDK. It allows a user mode application to read and write to physical memory. The DDK also provides Maptest, an application that uses the device driver to read from a specified physical address. An improved version of Maptest, Maptest2 was obtained from Carl Reinwald [17]. In this version, more bus interface types, including PCI, were added to the list of existing ones.

#### *PCI Bus Configuration Background*

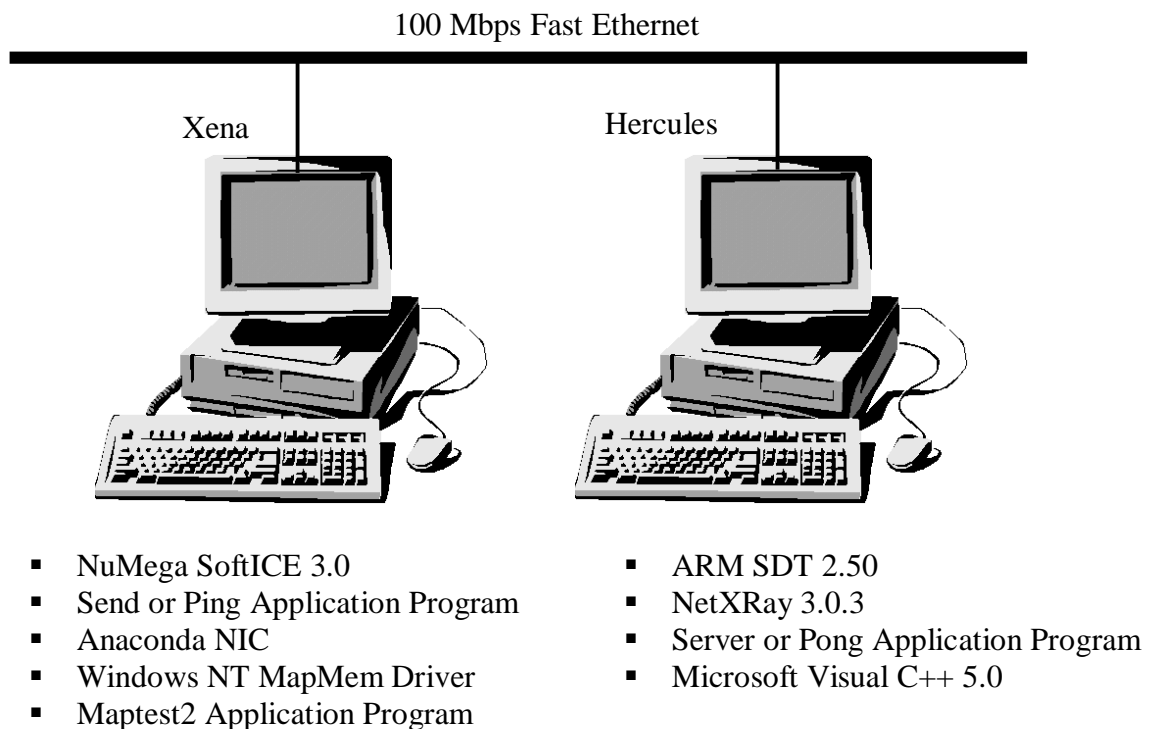
Each PCI device is assigned a PCI address space by the PCI BIOS during the boot sequence. Since each device can have at most six memory regions, there are base address registers numbered 0 through 5 that define the size and type of memory space for each corresponding region. After PCI BIOS initialization, the base address registers contain starting PCI address for its respective memory region.

Once the PCI base address is known, an application can map to that memory region by reading the corresponding base address register. An application, AMCCDIAG that was obtained from Carl Reinwald [17] has the ability to read the PCI base address registers for any PCI device. By running this application, the starting PCI address for the Anaconda card was found. However, the Anaconda card has five PCI devices on itself [23]. The application only saw Anaconda as a single load. The memory space on Anaconda is accessed via Footbridge and the starting address that was reported was actually for the Footbridge. The base addresses for the Anaconda memory space is located in the PCI configuration space of the Footbridge. The Angel debugger was used to trace through the Footbridge's PCI

configuration space. It turns out that the base address 2 register contains the starting address of the Anaconda memory space. From this information, it was also discovered that there are 64 Mbytes in the memory address space.

### Verification

The MapMem driver capabilities were verified before modifying it to meet the requirements for the initial network architecture. The testbed was updated, as shown in Figure 13, for the verification and development processes. MapMem driver<sup>3</sup> and Maptest2 application<sup>3</sup> was installed onto Xena. Microsoft Visual C++ 5.0 was also installed on Hercules for Maptest2 development.



**Figure 13. Testbed for Maptest2 Development**

---

<sup>3</sup> See Microsoft DDK for build and run instructions

### Testing MapMem's Read Capability

Initially, the MapMem driver was tested for the ability to only read Anaconda's memory space. With the Maptest2 application, many arbitrary memory locations were verified, including the address where the "Anaconda" signature resides. It was found that the Angel debugger reports an offset from the starting address of the memory address space instead of the actual physical address. When given a physical address, Maptest2 reads the data at the specified address. Thus, using the Angel debugger, the data reported by Maptest2 can be verified. The debugger has a feature that allows viewing of certain memory ranges. The physical address used by Maptest2 and the corresponding offset from the starting address of the Anaconda memory address space contained the same data for all memory locations tested.

### Testing MapMem's Write Capability

Testing the MapMem driver for its ability to write to memory involved some coding. Since the Maptest2 application was coded only to read memory (the writing capabilities were commented out), slight modifications were made. The comments only suggested how to write either one byte or four bytes. The code provided was augmented to allow for writing multiples of 4 bytes.

The Anaconda microcode also needed some modifications. A static buffer was created during the microcode's initialization in the **init()** function. This function allocates memory to the various data structures used in the microcode. The creation of the buffer takes place after each microcode data structure is allocated its required amount of memory. By allocating memory (1024 bytes) for the static buffer in this function, it will be global but always have the same address.

### *Test Case 1: Write to global buffer via Maptest2*

After changing the microcode to create a static buffer, *MapMemBuff*, the microcode was downloaded to the Anaconda board. Using the Angel debugger, the address (offset from the starting address of Anaconda memory space) was obtained. The Maptest2 application used this address (offset + starting address) as the physical address to write data to. After writing data to the buffer via Maptest2, Windows NT was halted and then so was the microcode. According to the Angel debugger, the buffer was not written too. It remained unchanged. However, when Maptest2 read the buffer (address specified), the data in the buffer matched the data that was to be written. Thus, according to Maptest2 the buffer was written to. For all the other data values that were tested the results were the same: Maptest2 (via read command) verifies the buffer write but the Angel debugger does not see the change.

### *Test Case 2: Cache problem?*

This outcome suggests that the Angel debugger's cache may not be flushing. If this is the case, then the buffer is actually being written to but the debugger is just not updating the change. An attempt was made to remedy the problem by adding another static buffer to the microcode. The original buffer, *MapMemBuff*, can then be copied into the second buffer (64 bytes), *MapMemBuffRead*. The copy takes place in the endless loop in **main** (), causing *MapMemBuff* to be written to *MapMemBuffRead* multiple times regardless of what Maptest2 does. For example, Maptest2 may only write data once to *MapMemBuff* but the buffer is copied repeatedly into *MapMemBuffRead* because of the loop in **main** (). In addition, data sizes of 0x80, 0x40, and 0x20, were written to the buffer. The goal was to use these two conditions to force the cache to flush.

With the two buffers, the following procedure was exercised to verify that the MapMem driver is able to write to Anaconda's memory space. The Maptest2 application was used to write data to a specified memory location. Windows NT was halted and then so was the microcode. Using the debugger, the contents of the buffers were viewed. Then the microcode was restarted followed by starting NT. Maptest2 was used again to read the contents of the second buffer, *MapMemBuffRead*.

Once again the results indicated that the write to the buffer via Maptest2 was unsuccessful. Maptest2 reported that the contents of *MapMemBuffRead* did not contain the contents that were supposedly written to *MapMemBuff*. The debugger also did not reflect any changes.

#### *Test Case 3: Microcode copy problem?*

The results from the first two tests suggest that the microcode may not be copying *MapMemBuff* into *MapMemBuffRead*. Another case was developed to test if the microcode was able to perform a copy in this part of its memory.

The setup is based upon Test Case 2 with some modifications. Once the buffers are created in *init()*, *MapMemBuff* is initialized with 8 bytes: 0xBABABABA ABABABAB. Immediately after the initialization, *MapMemBuff* is copied into *MapMemBuffRead*. Now the buffer is copied only once. With this test case, the Maptest2 application was used only to read the buffers since the copy taking place in the microcode must be verified.

The results showed inconsistencies. When the debugger is used to step through the microcode, Maptest2 and the debugger show different values in *MapMemBuffRead*. According to Maptest2, the copy did not take place, whereas the debugger shows that the values in *MapMemBuffRead* are the same as the values in *MapMemBuff*. However, when the

microcode is not stepped through via debugger, Maptest2 and the debugger both report that the copy has taken place.

Another inconsistent result is that in either case, stepping or not stepping through the microcode, Maptest2 and the debugger show different values in *MapMemBuff*. The Angel debugger reports that the buffer contains its initial values but Maptest2 sees 0's. Since the only values written to *MapMemBuff* are its initialization values, the buffer should still contain those values only. Maptest2 does not reflect this.

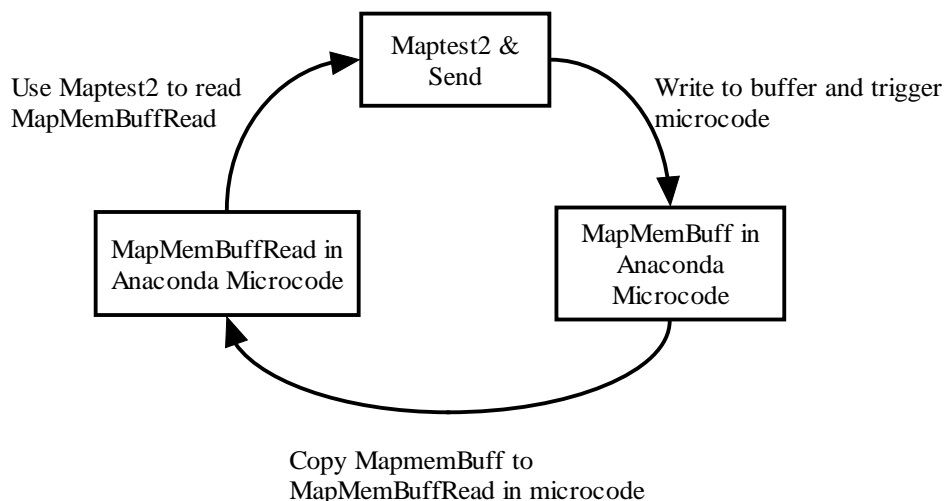
#### *Test Case 4: Microcode memory restricted?*

Based on the previous results, it seems that the Anaconda memory space may have certain characteristics in specific areas. For example, some areas of memory may be written to but not read. This could explain the outcomes of the earlier test cases.

A different approach was taken to test the write ability of the MapMem driver. This test case is based upon a "closed loop" model as shown in Figure 14. The first step is to use Maptest2 to write to *MapMemBuff*. Immediately after writing to the buffer, the Send application program is used to trigger the microcode to perform the copy from *MapMemBuff* to *MapMemBuffRead*. The code used to copy one buffer to the other was moved to the TCP segmentation path, so that the copy occurs only when the special payload ("\*\*\*\*\*") is recognized (see section TCP Segmentation Path). Assuming the time it takes the copy in microcode to finish is far less than the time it takes for a user to run Maptest2 again, Maptest2 is used to read *MapMemBuffRead*. If the data in *MapMemBuffRead* is the same as the data in *MapMemBuff*, then writing via MapMem driver is successful.

The results of this test were similar to the other test cases. Using Maptest2 to write 16 bytes as one test case and 32 bytes as another, Maptest2 displayed that in both cases the

values were written to MapMemBuff. However, when a read was performed on MapMemBuffRead via Maptest2, the data in that buffer was not the same as that in MapMemBuff. This test verifies that either writing to MapMemBuff via Maptest2 was unsuccessful and/or the copy in microcode was unsuccessful.



**Figure 14. Closed Loop Model Test Case**

#### *Test Case 5: Angel debugger problem?*

After reviewing the results from the previous test cases, a possible cause of the problem is the Angel debugger. Additional situations were tested accordingly. The closed loop model was tested again without viewing the Anaconda memory address space via debugger. The debugger was only used to retrieve the addresses of the two buffers and then it was used to download the microcode. After downloading the microcode, the debugger was left running but it was never used to view any memory locations. The results of writing 16, 32, and 64 bytes to *MapMemBuff* via Maptest2 were the same as in Test Case 4: *MapMemBuffRead* did not contain the same data as *MapMemBuff*.

A slight modification was made to this test case where after downloading the microcode, the debugger application was closed. Again, the results of writing 16, 32, and 64 bytes to *MapMemBuff* via *Maptest2* did not have favorable results. The two buffers still did not contain the same data at the end of the “closed loop” model procedure.

#### *Test Case 6: Variations on “closed loop” model*

More variations were made to the “closed loop” model procedure. Each buffer size was changed to 128 bytes. In addition, *Maptest2* wrote 16, 32, and 64 bytes of data to *MapMemBuffRead* instead of *MapMemBuff*. The microcode was then changed to copy *MapMemBuffRead* into *MapMemBuff*. However, the results were still unaltered. The buffers did not contain the same information after running the “closed loop” model test.

#### *Summary*

Based upon the results from the test cases, the read and write capabilities were not verified successfully. Another possible source of the problem may lie with the debugger. Some debuggers add an extra layer of indirection in order to access values of variables. For example if a char variable was declared, the debugger may access the variable through a pointer. Using a *MapMem* call to write to the char variable may write to the debugger pointer and the variable itself. To test this situation, a release version of the microcode should be built.

The undesirable outcomes from the *MapMem* verification tests may also be a result of the Anaconda board itself. The *MapMem* utility is functional as shown in Carl Reinwald’s thesis [17]. However, when it is used to map the Anaconda memory it does not seem to be operational. For this reason, the *MapMem* capabilities should be investigated on another external board, i.e. EBSA-285.

## CHAPTER 6

### Initial Design<sup>4</sup>

#### Modifications to the Windows NT Application

As shown in Figure 15, the application is modified so that it basically becomes an enhanced version of Maptest2. By combining the Send/Server and the Maptest2 applications, a new application, MapToAnaconda that is capable of minimizing its interaction with the NT kernel is developed. As a result, the program is not a typical Windows NT program at the application layer, i.e., Telnet. It can only send a stream of '\*'s, however the user can specify the size of the payload. The use of '\*'s will allow the Anaconda microcode to detect that this payload needs to execute the mapped path. The custom application uses the MapMem driver to bypass the NT kernel and a new socket interface to avoid the kernel Winsock components.

#### Incorporating MapMem

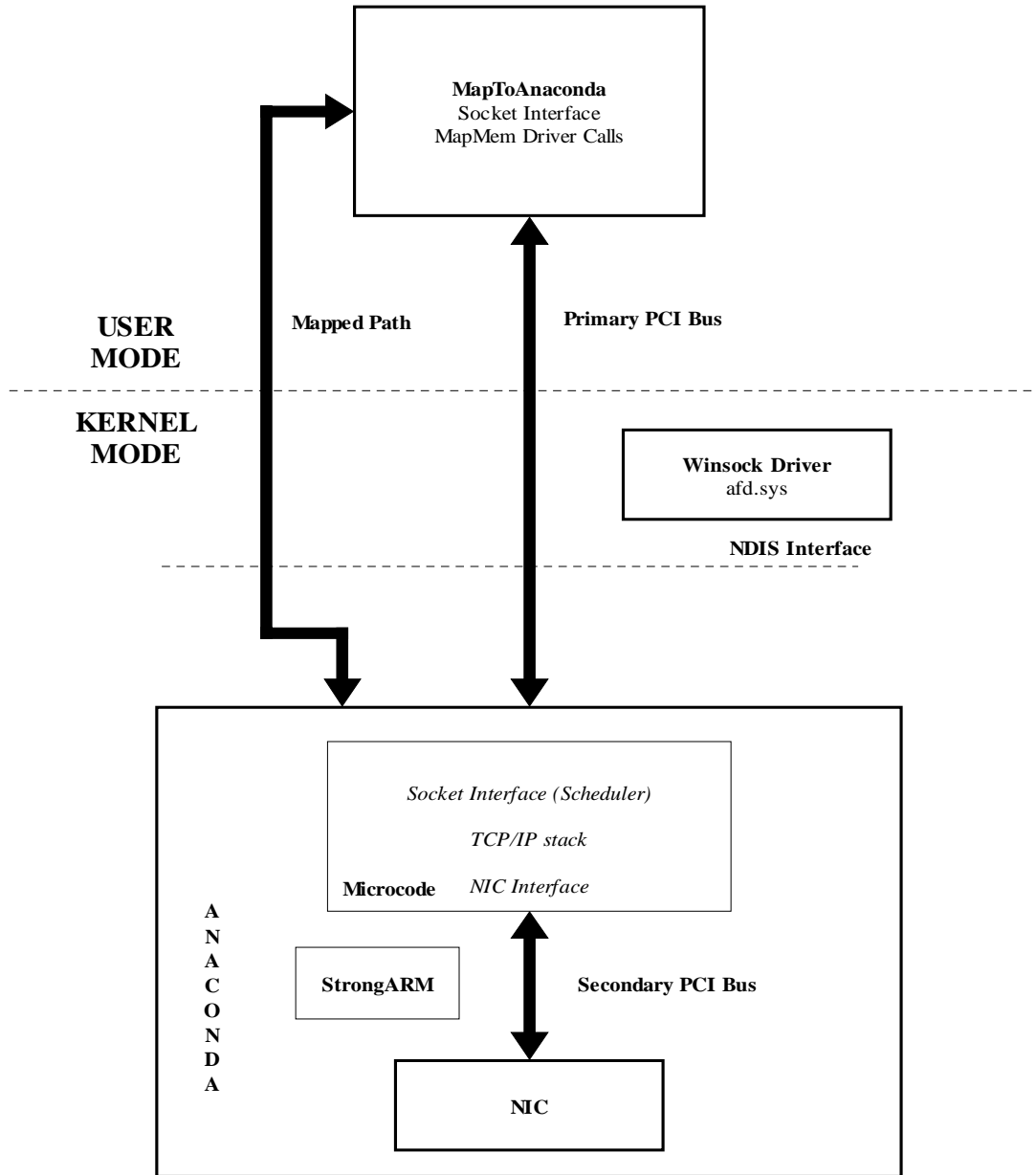
The MapMem driver calls will be used to establish shared memory between the host machine and the microcode residing on the Anaconda board without having to go through the NT kernel (see Custom Sockets). This memory mapping technique will also be used for the data control path (see Incorporating Transmit Descriptor Mechanism).

#### Custom Sockets

Instead of the application interfacing with Winsock to create a connection with the destination, it interfaces to Linux sockets. By using Linux sockets, it is easier to interface it with the ported TCP/IP stack since Linux is open source.

---

<sup>4</sup> Jim Fischer provided much insight to the initial design.

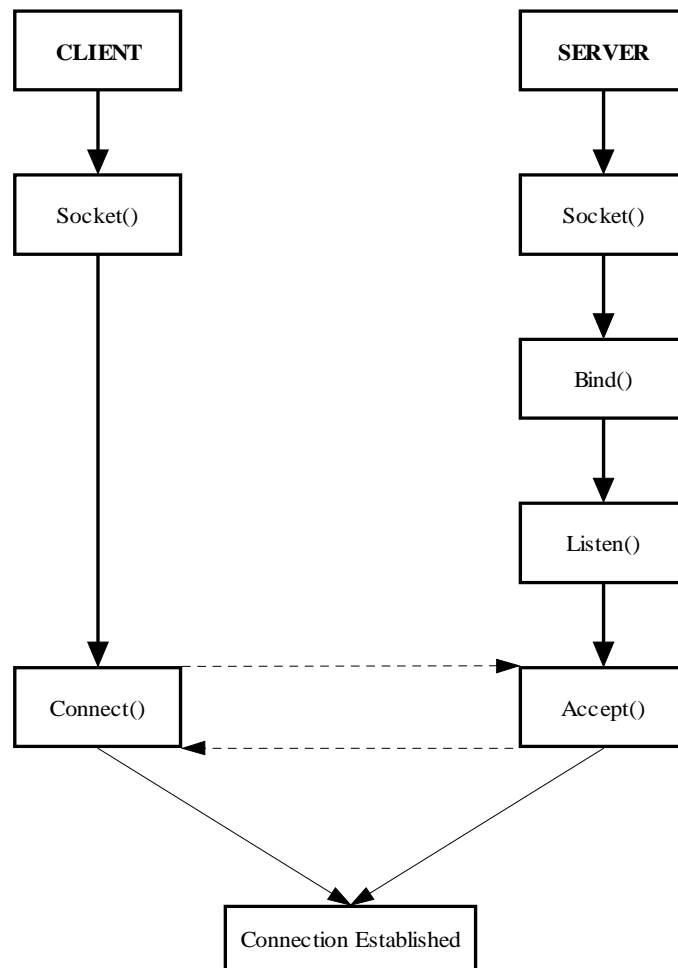


**Figure 15. Initial Network Architecture Model**

### Socket Basics

Establishing a connection with Linux sockets consists of five stages shared between a client and server [8]. Figure 16 shows a minimal, basic socket established between the client and server through the use of the typical socket function calls. Both the client and server

create sockets with a call to **socket()**. However, neither one of the sockets are connected just yet. Through the use of **bind()**, the server can attach a local address to the socket. Then the server calls **listen()** since it is now ready for clients to establish connections to the socket. When a client does try to establish a connection with the server by calling **connect()**, **accept()** on the server side establishes the connection. Once a connection is established, the socket is used for data transmission through **send()/write()** and **receive()/read()**. Table 2 lists the socket function calls commonly used.



**Figure 16. Establishing a Socket Connection [8]**

**Table 2. Socket Function Calls**

| <b>Socket Function Call</b> | <b>Description</b>   |
|-----------------------------|--|
| socket()                    | Returns a file descriptor for an uninitialized socket. The socket is used for the protocol specified in the argument list. |
| bind()                      | Binds local address to the socket  |
| connect()                   | Connect to address specified in the argument list. Client specific.  |
| listen()                    | Wait for another to establish connection. Server specific.   |
| accept()                    | Accept connection. Server specific.  |
| write()                     | Generic write using file descriptors.  |
| read()                      | Generic read using file descriptors.   |
| send()                      | Write to socket  |
| receive()                   | Read from socket   |
| close()                     | Close socket connection  |

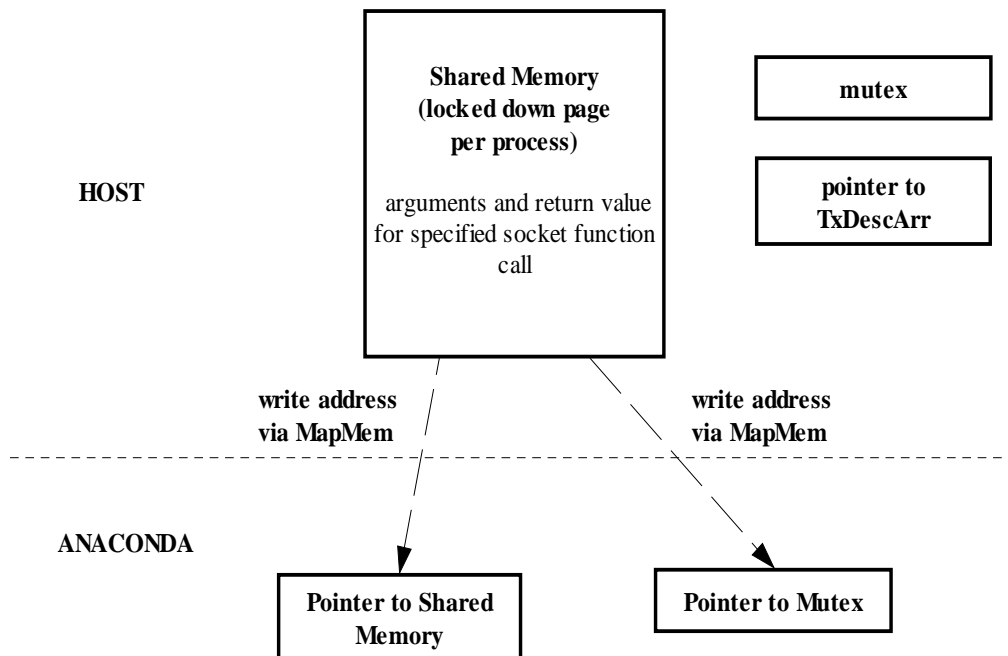
Only one system call handles all of the socket function calls [19]. A call to any one of the socket function calls at the application layer essentially calls the **sys\_socketcall()**. A argument in **sys\_socketcall()** is an integer that corresponds to each socket operation. The integer is used in a case statement that would eventually invoke the corresponding kernel socket function . For example, **socket()** calls **sys\_socketcall()** which in turn would call **sys\_socket()**.

#### Socket Control Path

The purpose of the socket control path is to establish (1) shared memory between the host and Anaconda microcode and (2) a socket connection as shown in Figure 17. Shared memory containing arguments and return value for a socket function call will reside on the host since the microcode should not be process dependent. The application will need to allocate a page of memory for this shared memory and it will have to make sure that this page is not swapped out, i.e., locked down while Anaconda has control.

The application must allocate memory for a pointer to the *TxDescArr* (see Chapter 4) that the microcode uses to process TX descriptors. Additional memory will also have to be

allocated for a mutex<sup>5</sup> that is used for transferring control between the host and the microcode. With this design, there is one mutex for each application that is initiated. For this reason, this design is for the test case when only one application is started. Multiple applications may run when a dynamic link library (dll) is created to take on the MapMem capability and allocating memory for mutex.



**Figure 17. Socket Control Path**

Before the application can use MapMem to write the shared memory address and mutex address to Anaconda, it must know where in Anaconda to write it to. For this reason, a kernel module needs to be developed so that it can access the Anaconda memory and pass the

<sup>5</sup> Similar to a binary semaphore that can have either a value of 0 or 1 only [22].

appropriate addresses to the application. Once, the application has the respective addresses, it can use MapMem to write the shared memory and mutex addresses to Anaconda. This kernel module is only used once to set up these address registers. After that, any exchanges between the host and Anaconda should be done through MapMem in order to avoid the NT kernel. Since *TxDescArr* is an existing, shared data structure between host and Anaconda, the method used to establish a shared area of memory already exists. This kernel module should follow this same method as described in the NDIS driver (see NDIS driver source code).

With the address registers on the host and Anaconda setup, the application is ready to create a socket and send data. There are basically two function calls that need to be called for a client to create a socket: **socket()** and **connect()**. Since Winsock will not be used, these functions along with the functions for the server must be modified. Because, Linux is open source, Linux/BSD sockets will be used.

Each socket function call at the application must first lock down the shared memory page. Then, it can prepare the arguments and return value required by that specific function call in the shared memory page. For safety purposes, the function must wait for Anaconda to finish any current processes, i.e., it must make sure the mutex is available. When the Anaconda releases the mutex, MapMem should be called to map the shared memory page to Anaconda. When the function sets the mutex, it implicitly invokes the function **sys\_socket()** in the Linux kernel on the Anaconda. **sys\_socket()** is the entry point into the kernel. When Anaconda is done executing, it releases the semaphore. The socket function must then retrieve the return value from the shared memory to analyze it. Then the page can be unlocked so that NT can swap it out if necessary. Finally, the return value should be returned

to the application itself. Appendix H: Pseudocode for `socket()` contains the pseudocode for the `socket` function call `socket()`.

### Incorporating Transmit Descriptor Mechanism

Once a connection is established, and `MapToAnaconda` is ready to send data, it must write (set) the bit `txAvailable` in the `ControlFlags` field of the TX descriptor. The Anaconda microcode polls this bit and to determine if it needs to transmit data. Setting `txAvailable` initiates the microcode to fetch and investigate the pending TX descriptor. After reading the control field of the TX descriptor, the existing Anaconda microcode decides which path it should execute. This will be referred to as the decision point. With the addition of the memory mapping capability, a fourth path is created in parallel with the original three. This path will be called the mapped path.

In order for the mapped path to process the payload, the data must be described by TX descriptors. For this purpose, the application will contain intelligence so that it can manage a TX descriptor pool. This pool is used to obtain TX descriptors for the payload and is independent so that it will not interfere with the pool of the original three transmit paths.

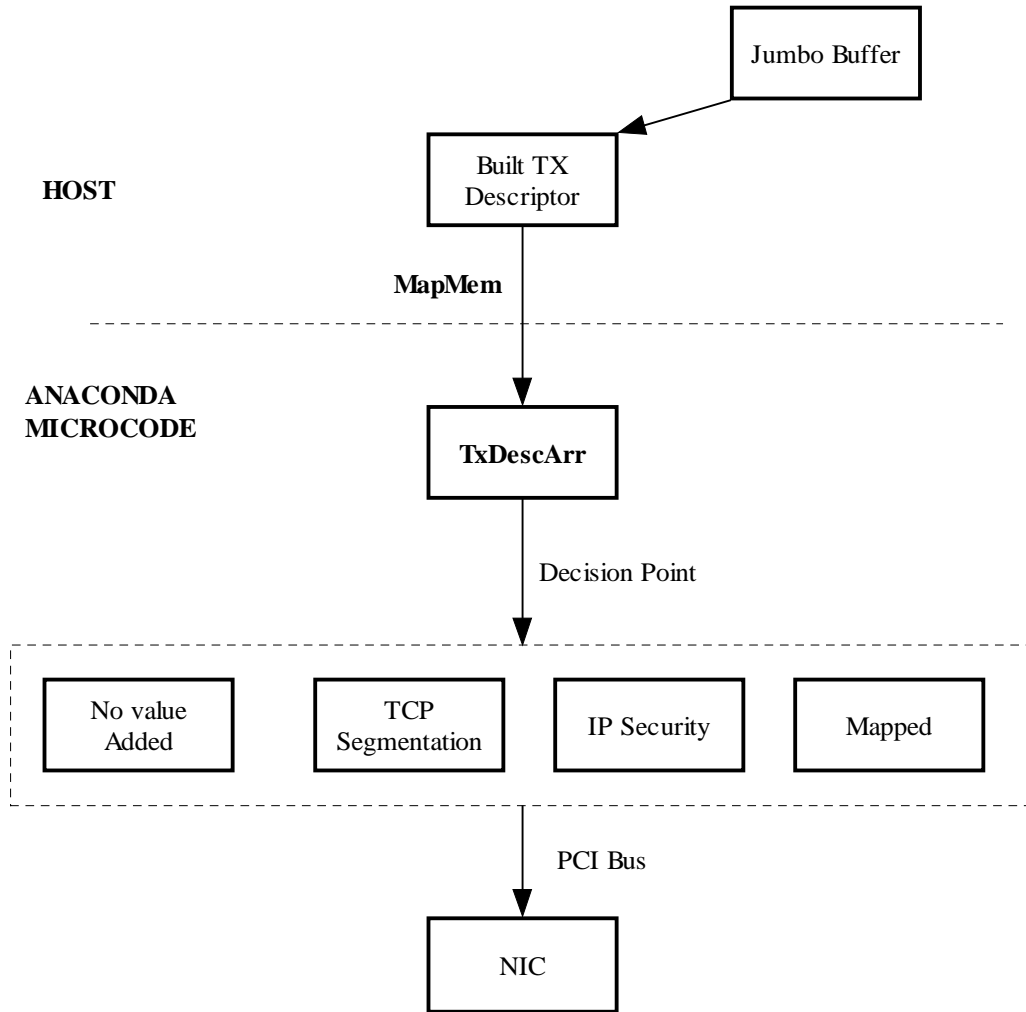
A separate TX descriptor pool can be created for the application's use in the Anaconda microcode. This allocation of memory can be done in `init()` by calling `BlockPoolCreate()` and `DpdPoolInit()`. Once the pool of descriptors is created, using the `MapMem` utility a TX descriptor can be retrieved and filled for the data waiting to be transmitted. The structure of the implementation is basically the same as the implementation in the microcode except for (hopefully) a few minor changes.

Building the TX descriptor was originally a task for the NDIS driver. However, since the initial network architecture adds a path to bypass the NT kernel, the TX descriptor will not be

built in the NDIS driver. Instead, the application will be responsible for filling out the various fields of the TX descriptor as it relates to the data to be sent. The eight fields required to be filled are *ScheduleTime*, *FrameLength*, *Fragments[]*, *MaxSegmentSize*, *SecAssociation*, *ControlFlags*, *PacketPointer*, and *VlanTag*. The most important fields for the mapped path are *FrameLength*, *Fragments[]*, *MaxSegmentSize*, and *ControlFlags*. *FrameLength* specifies the total data size to be sent. *Fragments[]* is an array that describes the data in terms of DMA fragments. The maximum segmentation size for TCP segmentation is specified in *MaxSegmentSize*. The execution of one of the paths depends on which bits are set in *ControlFlags*. For further details on the TX descriptor fields, see Anaconda Host Interface Specification and Cyclone ERS Specification.

In MapToAnaconda, it is critical to set a bit in *ControlFlags* to indicate that the mapped path should be taken. Since only bits 0-4 are defined in the specification, bit 5 will represent the mapped path. Setting bit 5 indicates that the frame to be sent is a “mapped” frame and should execute the mapped path. Otherwise, the frame is not a mapped frame.

The last bit to set in the TX descriptor is *txAvailable*. Setting this bit will eventually cause the microcode to fetch the descriptor from the *TxDescArr*. Once MapToAnaconda is finished filling out the TX descriptor, it should queue the descriptor in *TxDescArr* via MapMem. Seeing that *txAvailable* is set, the Anaconda microcode will then take the descriptor that has been queued and process it. At the decision point, if bit 5 in the *ControlFlags* field is set, the mapped path will be executed. This flow of control is shown in Figure 18.



**Figure 18. Flow Control for Initial Design**

## Porting TCP/IP Protocol Stack

### Implementation Plan

Because the Anaconda board has a limited amount of memory, a full-blown operating system with the TCP/IP protocol stack could not be ported onto it. However, XINU [21], is an operating system that focuses on the TCP/IP stack and has minimal system functions that are unrelated. The initial plan was to port this stack onto the Anaconda board and integrate it with the packet header-building concept of the TCP segmentation path. It was later found to be a more tedious task than intended because approximately thirty files were coded in Intel assembly language. To make XINU portable to the Anaconda board, those Intel assembly language files would have to be translated to StrongARM ARM assembly files.

With the TCP segmentation path, the first 128 bytes of the data is copied onto the Anaconda board from host memory. It is assumed by the microcode that the TCP/IP header is contained within this data. To segment the payload, the pointers to the payload's DMA fragments are modified so that each DMA fragment is a segment. The header is replicated as many times as needed so that each segment has a header associated with it. In this path, the whole data payload is not copied onto the Anaconda board, only the header information is initially copied.

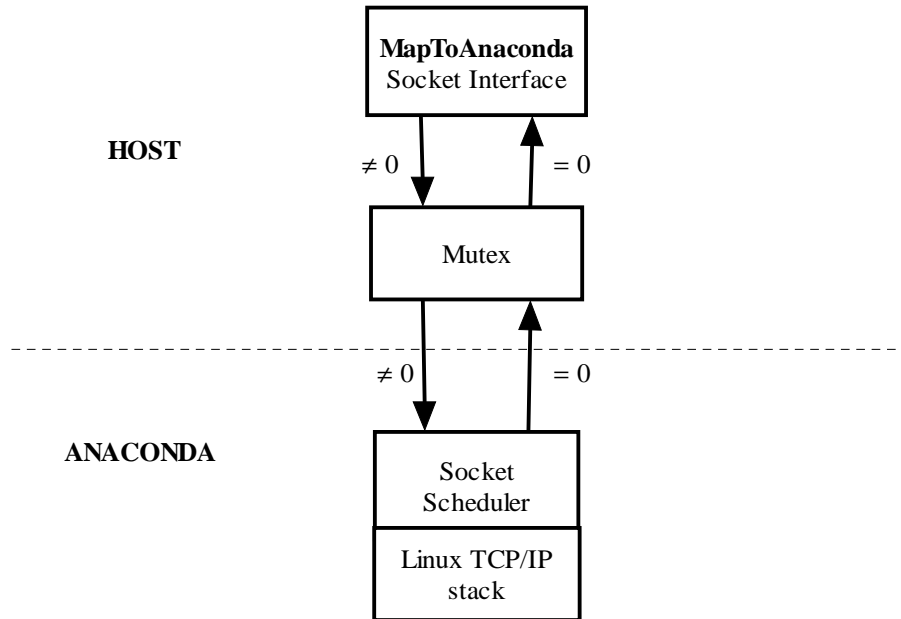
Using this method as a model, the ported TCP/IP stack will be able to process the data on the Anaconda board. In the TCP segmentation model, payload packets are decomposed into DMA fragments. Then these fragments are segmented according to the MSS so that the resulting DMA fragments are now segments. Similarly, for the mapped path, a jumbo buffer is fragmented. By using pointers to these fragments the TCP/IP stack can create packets. A new TX descriptor will be built for each packet. The new descriptor will copy all the

information as the original descriptor except for *Fragments[]*. This entry will contain the newly calculated pointers to DMA fragments. The ported stack will also create headers in a similar fashion as in the TCP segmentation path so that each packet will have its own header.

With the TCP/IP stack ported onto the Anaconda board, there are some interfaces that should be defined. There are two interfaces to the ported TCP/IP stack: top-end and bottom-end. The TCP/IP top-end interface is between the TCP/IP stack and the Linux socket. The bottom-end interface allows for the TCP/IP stack and the NIC to communicate. These two interfaces are discussed in detail in the following sections.

#### Interface Between TCP/IP and Socket Layer

The interface between the TCP/IP stack and the back-end of the socket layer in the application is basically a scheduler as shown in Figure 19. When a socket connection has not been established, the application, MapToAnaconda makes a series of socket function calls. However, before initiating a call, the arguments for that call must be made available to Anaconda. In order to allow only one process to be running at a time, MapToAnaconda must release the mutex (set it to a non-zero value) to the Anaconda board . With the mutex set, the scheduler can then call the respective TCP/IP kernel function call by invoking `sys_socketcall()` with the socket function name as a parameter. When the function has completed, the scheduler should clear the mutex so that control can be passed back to the application.



**Figure 19. TCP/IP / Socket Interface**

### Interface Between TCP/IP and NIC

In the original microcode, the communication object between the TCP segmentation processing and the NIC is the DPD (see Chapter 4). This mechanism is paralleled in the interface between the bottom-end of the TCP/IP stack and the NIC in the mapped path. After the data is processed by the TCP/IP stack, the interface creates DPDs for that payload so that the NIC can process it.

Creating the DPDs is fairly straightforward since DPDs basically contain all the fields in TX descriptors and a few others. With the descriptors filled out from the TCP/IP stack, each TX descriptor field is copied to its respective DPD field. An example of how to do this is located in **ProcessNiCTransmits()**. The DPDs are usually chained then sent down to the NIC for transmission. There are two fields, *NextDesAddr* and *SANextDesAddr* that are

responsible for linking the DPDs together. *NextDesAddr* is the PCI relative address whereas *SANextDesAddr* is the StrongARM ARM relative address. The code located at the end of **TCPSegSend()** demonstrates how to chain the DPDs together.

After the DPDs are chained together and are ready for transmission they should be queued for the NIC by calling **DPDListEnqueue()**. Then the transmission can begin by calling **NicSend()**.

### Results

Currently, the initial network architecture model is designed for transmitting data only. By concentrating on only transmission, the researcher has full control of what to send and when to send it. It is assumed that the acknowledgements will be received by executing the existing receive path as outlined in Chapter 4.

Since the verification of the Windows NT MapMem driver was unsuccessful, further investigation should be conducted. Questions regarding the Anaconda memory traits (read only, writeable but not readable, etc.) should be directed to the engineers who worked on the Anaconda board at 3Com. Although MapMem was unsuccessful in the Anaconda environment, Carl Reinwald has proved that it is capable of mapping memory [17]. Because of this, MapMem should be investigated with the EBSA-285 board.

For the initial network architecture, the MapMem driver was a service provided to the application. Once the architecture is implemented, the MapMem utility can be implemented in a custom kernel module.

It was found that the porting the XINU operating system would take a tremendous amount of effort. Another option is to use the Linux operating system. However, since the Anaconda board has a limited amount of memory, the whole Linux operating system is

unlikely to fit. With Intel's EBSA-285 board containing 16Mbytes of memory, the full-blown Linux operating system can be ported onto the board. With this method, the operating system will contain many other system services that do not pertain to the TCP/IP protocol. However, the goal is achieved: porting the TCP/IP stack.

Due to the findings from the tests on MapMem and the amount of effort required to port the XINU operating system, it is concluded that the initial design is not viable. This leads to a final proposed plan as described in Chapter 7.

## CHAPTER 7

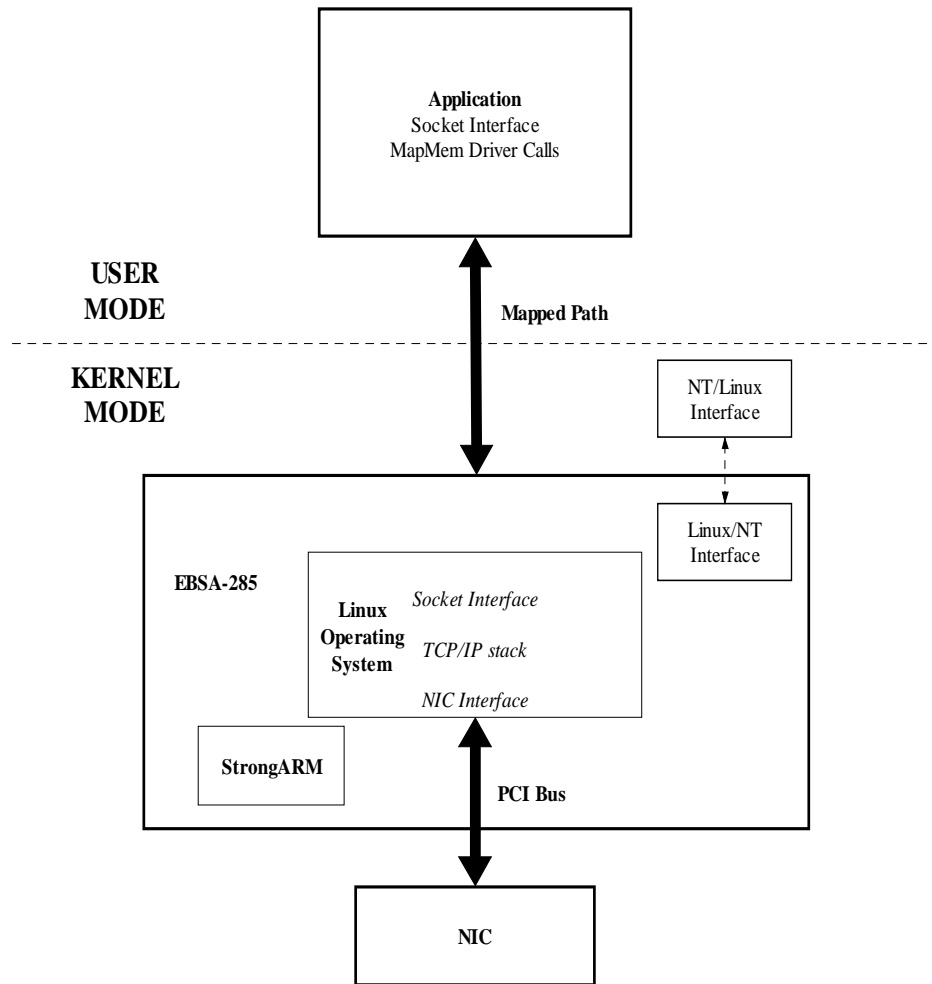
### Proposed Design

The proposed design is very similar to the initial network architecture design, only simplified as shown in

Figure 20. A custom application is still required and will still have to include a socket interface and MapMem driver calls. The Linux operating system, which includes a socket interface, TCP/IP stack, and a NIC driver is ported onto the EBSA-285 board. However, because an NDIS driver does not exist for the EBSA-285, some kernel process must perform initializations before the custom application can even run. For example, some kind of interface must be developed so that the NT host can find the EBSA-285 on the PCI Bus.

In Tim Pepper's senior project [18], he identified specific tasks that a Linux driver must perform for the Anaconda board. Although, the device driver is for a Linux host, the same issues must also be addressed in Windows NT. Such tasks should include PCI probing to locate the different hardware on the board itself, device initialization, and device cleanup/termination.

The application is basically the same as that described in the initial design plan (see Modifications to the Windows NT Application in the previous chapter). However, in this proposed design, the EBSA-285 board is used so the TX descriptor mechanism does not exist. Therefore, in the application, instead of using TX descriptors to describe each of the packets of payload, the whole jumbo buffer can be passed down to local memory on the EBSA-285 board through the 21285 controller chip that has DMA capabilities.



**Figure 20. Proposed Network Architecture**

With the proposed design, there are two data paths. One is for command and status that handles initialization, termination, and error conditions. The other data path handles the network data transactions. The control for the initialization, termination, and error conditions resides in the NT and Linux kernels (as a part of the NT/Linux interface). The application layer can then handle the data path for the network data.

Another difference between the proposed design and the initial design is that the former depends on having the Linux operating system ported onto the external board. The socket interface that Linux provides will also communicate with the socket interface at the application is similar to the socket control path described for the initial design. The socket control path is essentially the mapped path as shown in Figure 17. Creating a kernel module that will retrieve necessary addresses from the EBSA-285 for use with MapMem is also required.

Since the Linux operating system also includes NIC drivers, the TCP/IP stack-to-NIC interface is already developed. Hopefully, modifications, if any will be minimal. This reduces the amount of work on the external board side. Note that the EBSA-285 does not include a NIC on board like the Anaconda. The NIC is an actual device that is also attached to the PCI bus.

At this point it is not clear what the details of the PCI interface with the Linux operating system on StrongARM microprocessor are. For the proposed design, the NT host is the initiator while StrongARM with Linux is the target. A Linux/PCI microcode driver may be a solution to this issue, if indeed it is an issue.

## CHAPTER 8

### Conclusion and Summary

Due to the findings from previous research in network performance, it was concluded that the Windows NT operating system is the bottleneck when it comes to transmitting data from one computer to another. In order to test that conclusion, the purpose of this thesis was to document a design using the Windows NT architecture as a basis with some modifications to it.

The initial design consisted of porting the XINU operating system including the TCP/IP stack onto the Anaconda board. In order to achieve this, two technologies were specifically studied: MapMem and the Anaconda microcode. This design was abandoned for two reasons: (1) MapMem did not function in the Anaconda environment and (2) porting XINU would require much effort.

As a result, another network architecture was designed. The concept of this proposed design is similar to that of the initial design. However, the key differences are (1) the EBSA-285 board is used as the external board instead of the Anaconda board and (2) the whole Linux operating system is ported onto the EBSA-285. For this model, the functions of the interface between the NT host and Linux co-processor were identified. In addition, the proposed design is simpler than the initial design since the former does not require maintaining transmit descriptors.


## Bibliography

- [1] ARM Software Development Toolkit Version 2.50. ARM Ltd., 1998.
- [2] Beck, M., et al. *Linux Kernel Internals*. Harlow England: Addison Wesley Longman, 1998.
- [3] Chen, Charles. "Anaconda Host Interface Specification Version 2.1." 3Com Corp., August 18, 1998.
- [4] Comer, Douglas E., and David L. Stevens. Internetworking with TCP/IP Volume II: Design, Implementation, and Internals. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 1999.
- [5] Comer, Douglas E. Computer Networks and Internets. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 1997.
- [6] Fischer, Jim. "Measuring TCP/IP Performance in the PC Environment: A Logic Analysis/System Approach." Senior Project. California Polytechnic State University, San Luis Obispo, December 1999.
- [7] Hu, Tom, Ashraf Kaiser, and Anand Rajagopalan. "Anaconda Microcode Specification Version 1.5." 3Com Corp., October 21, 1998.
- [8] Johnson, Michael K., and Eric W. Troan. Linux Application Development. Reading, Massachusetts: Addison Wesley Longman, Inc., 1998.
- [9] Liang, Yu Guang. "Network Latency Measurement for I2O Architecture by Logic Analyzer Instrumentation Technique." Senior Project. California Polytechnic State University, San Luis Obispo, June 1998.
- [10] Lo, Siu Ming "Performance Measurement of Network Application." Master's Thesis. California Polytechnic State University, San Luis Obispo, June, 1999.
- [11] Microsoft Corporation. Microsoft Developer Network: DDK Platform. Redmond Washington, January 1996.
- [12] Microsoft Corporation. Microsoft Developer Network Library. Redmond Washington, July 1999.
- [13] Microsoft Corporation. Microsoft Developer Network: Win32 SDK. Redmond Washington, January 1996.
- [14] Patterson, David A., and John L. Hennessy. Computer Organization and Design: TheHardware/Software Interface. San Francisco, CA: Morgan Kaufman Publishers, Inc., 1998.



- [15] Pepper, Timothy Charles. "Linux Network Device Drivers for PCI Network Interface Cards using Embedded EBSA-285 StrongARM Processor Systems." Senior Project. California Polytechnic State University, San Luis Obispo, June 1999.
- [16] Quinn, Bob, and Dave Shute. Windows Sockets Network Programming. Reading, Massachusetts: Addison-Wesley Publishing Co., 1996.
- [17] Reinwald, Carl Joseph. "The Design and Implementation of the Digital Experimenter's Kit." Master's Thesis. California Polytechnic State University, San Luis Obispo, June 1999.
- [18] Rusling, David. "The Linux Kernel."  
<http://www.lug.calpoly.edu/Reference/tlk/net/net.htm> (November 29, 1999).
- [19] Sanchez, Mauricio. "Exection Analysis of Network Transactions in a Linux Client/Server Atmosphere using BSD Sockets over an Ethernet Network." June 1999.
- [20] Sanchez, Mauricio. "Iterations in TCP/IP Network Optimization." Master's Thesis. California Polytechnic State University, San Luis Obispo, June 1999.
- [21] Stevens, W. Richard. TCP/IP Illustrated, Volume 1: The Protocols. Reading, Massachusetts: Addison-Wesley Publishing Co., 1994.
- [22] Tanenbaum, Andrew S. Modern Operating Systems. Upper Saddle River, New Jersey: Prentice Hall, Inc., 1992.
- [23] Traber, Richard. "Anaconda Hardware Design Reference Revision 0.3." 3Com Corp., 1998.
- [24] Xie, Peter. "Network Protocol Performance Evaluation of Ipv6 for Windows NT." Master's Thesis. California Polytechnic State University, San Luis Obispo, June 1999.
- [25] Xie, Peter, Mei-Ling Liu, Jim Harris, and Chris Scheiman. "Profiling the Performance of TCP/IP on Windows NT." California Polytechnic State University, San Luis Obispo, August 26, 1999.
- [26] Yu, Angel "Custom Windows NT 4.0 Parallel Port Device Driver: A Component of a Network Performance Tool." Senior Project. California Polytechnic State University, San Luis Obispo, June 1998.
- [27] Zacker, Craig. TCP/IP Administration. Foster City, CA: IDG Books Worldwide, Inc., 1998.

## Appendix A: Execution Procedure for Anaconda Microcode

### Building the Anaconda Project

Start the ARM Project Manager from the Start Menu. Open the Anaconda project by selecting File → Open or click on the  icon located on the menu bar. This brings up a file browser window where the project file name needs to be specified. Change the 'Look\_in' field to C:\Anaconda\Ucode. Double click on Anaconda\_Ucode. A new window appears that contains the Anaconda project.

Double click on 'ARM Executable Image' or single click on the '+' sign at the left margin. The tree should be expanded to include 'DebugRel, Debug, Release' under 'ARM Executable Image.' Development requires using a debug version of the Anaconda microcode (ucode) so a debug version must be built. Click on 'Debug' to highlight it.

There are two types of builds: regular and force. The regular build only recompiles the files that have been updated. To perform a regular build select Project → Build Anaconda\_Ucode.apj 'Debug' or click on the  icon located on the menu bar. The force build recompiles all the files no matter what. To perform a force build select Project → Force Build Anaconda\_Ucode.apj 'Debug' or click on the  icon located on the menu bar.

### Executing the ARM Image


The result of compiling the Anaconda source code base is the anaconda.axf image file. Depending on whether a release or debug version was built, the file can be found in either the Release\ or Debug\ directory of the Anaconda project. Since all development efforts must be done utilizing the debug version, the building of the release version will probably never come into play.

With the anaconda.axf file available, there are three routes to getting the image down to the board and to start execution. Two of them are available via icon actions from the ARM Project manager and the third is the manual loading of the code from the ARM debugger. All three routes end up at the same point with the ARM debugger controlling execution of the Anaconda ucode.


### Three ways of downloading image file to Anaconda board

Assumption: For methods 1 and 2 ARM Project Manager is currently active with the debug version of the anaconda project.

#### *Method 1: Automatic debugger start, no execution*

From ARM Project Manager click on  the icon from menu Bar to start ARM debugger and download image file to board. After the download sequence is complete, the debugger brings up the source file where the very first instruction to be executed resides. The line of this first instruction is highlighted green since this is the next instruction that the processor will execute. At this point, the image is not running. To execute, select Execute → Go. Another instruction is highlighted with green and red since the debugger has reached a breakpoint. Select Execute → Go again. Now the image is executing.

#### *Method 2: Automatic debugger and execution start*

From the ARM Project Manager click  from the menu bar to start the ARM debugger and download the image file to the board. This method differs from method one only in that image execution starts automatically after downloading is finished. There are no extra steps necessary as in method 1 to get to continuous execution.

#### *Method 3: Manual debugger and execution start*

If the ARM Project Manager isn't currently active, it is still possible to download and execute the image file directly through the debugger. Going through the usual start menu and ARM program folder, the ARM

debugger is started and begins by asking whether to start a remote debugging session (assuming a previous remote debugging session had been conducted, if not see section *Initial Set-up for Remote Debugging Session*). Clicking on 'yes' in the dialog window starts the download of the Angel debugging monitor down to the board. After the code has downloaded, the debugger shows three windows, the execution window showing the assembly dump of the debugger software, the command window where debugger commands are entered, and the console window showing any debugging output generated by the board. At this point, the binary image file of the anaconda ucode can be downloaded.

From the 'File' menu the 'Load Image' entry is selected and this brings up the file browser window for .axf file selection. From here the anaconda.axf is selected in whatever directory the file resides and after clicking on 'Open' the downloading procedure commences. If the anaconda image file had been previously opened and downloaded, an entry of recently opened files should exist as a selection under the 'File' menu. This is an alternate way of selecting the file, rather than going through the file browser window.

After downloading the anaconda.axf file, the executing window changes to the init.s assembly file. Execution doesn't start automatically, so the 'Go' entry under the 'Execute' is used to start execution.

### **Initial Set-Up for Remote Debugging Session**

Bring up the ARM Debugger by selecting Start menu → ARM SDT v2.50 → ARM Debugger for Windows. A dialog box appears and asks to start in remote debugging. Click on 'No'. To configure the debugger to download the microcode, go to 'Options → Configure Debugger.' The debugger configuration window appears. Select 'remote\_a' for the 'Target Environment.' This configures the debugger to be in remote ARM mode (vs. Armulate). Click on 'Configure' to get the Angel Remote Configuration window. Under 'Remote Connection' make sure that 'serial' is selected. Also be sure to disable the 'heartbeat', since it sometimes causes the debugging session to crash. Furthermore, the baud rate is 38400 as this is fast enough to allow easy movement through the running code without causing spurious debugging session crashes. If the system becomes unresponsive, one of the fixes might be to lower the baud rate. At 9600 the speed is slow enough to rule out the baud rate as a possible source of trouble. See section *Downloading the Image* to load the anaconda microcode.

### **Development Caveats with the Anaconda board**

#### *On-board Memory*

Although the Anaconda board uses Flash memory to store the driver image while it is on, it does not conserve memory contents after power is turned off or after a reboot cycle. After each power, reboot, or crash recovery cycle, it will be necessary to download the image file back onto the board. This means that the Flash memory can be considered to act like normal DRAM memory.

3Com does have a utility to burn an image onto the board, however, during the development phase there is really no reason to do this. Only after a stable release version has been developed, does it make sense to go the permanent route.

The boards do contain some permanent initialization code for creating the initial connection between the board and the ARM debugger.

#### *NDIS Driver*

There is a fluke on some hardware systems that causes the computer to *sometimes* blue screen when doing a shutdown. It doesn't happen on every system, nor does it occur everytime on a system that has previously shown the problem.

The problem seems to be hardware dependent and is related to an unhandled fault that occurs when demapping the driver from PCI interrupt subsystem. 3Com knows about the problem and admitted that the driver needs further work in this area, but basic functionality is unaffected so they left it as is.

## Appendix B: Packet Send from Anaconda Perspective

### ***Three types of transmit***

1. TCPSEG (TCP segmentation)
  2. IPSEC (IP security)
  3. Hurricane
- each has its own input list

### ***Packet Send Algorithm (Overview)***

(For more details, see Anaconda Microcode Design Specification, pp. 30-31)

- 1) interrupt from host
- 2) copy transmit descriptor (filled out by host) into DPD (download packet descriptor) located in Anaconda private space
- 3) release transmit descriptor
- 4) call Host Transmit function
  - a) look at control field
    - is it TCPSEG?
      1. download header to SDRAM (local memory)
      2. download other data directly to FIFO
    - is it IPSEC?
      1. download all data into SDRAM
      2. download to FIFO after encryption
    - is it neither? i.e. Hurricane
      1. download data into FIFO

### ***TCPSEG function (Overview)***

(For more details, see Anaconda Microcode Design Specification, pp. 33-35)

input: transmit descriptor, possibly chained

output: list of DPDs that are passed to Hurricane for transmission

function: break up packet supplied by driver to maximum segment size (MSS) so that it can be downloaded to Hurricane for putting out on wire (transmission)

details:

- 1) get first descriptor from input list
- 2) download header into SDRAM via footbridge DMA
  - a) first 128 bytes are read in from the host buffer into the header template buffer
  - b) first 128 bytes should contain both IP and TCP headers completely
- 3) segment TCP data payload based upon MSS
- 4) create new headers, one for each segment → chain of DPDs
- 5) place DPDs on input list of Hurricane send

### ***IPSEC function (Overview)***

input: transmit descriptor

output: encrypted data payload

function: encryption/decryption of data

details:

1. copy transmit descriptor into DPD from local DPD pool
2. build DMA descriptor for transferring data from PCI memory to SDRAM.
3. get security association (SA) information and generate AH and/or ESP header based upon SA
4. give Sidewinder chip SA and payload for encryption. Encrypted data is read back to SDRAM by Sidewinder interrupt
5. DMA another packet, if any. repeat 4 until no more data.
6. place DPD on input list of Hurricane send

Hurricane – Cyclone-based Ethernet ASIC, 2 on Anaconda board

Cyclone – original Ethernet ASIC

**Detailed description of how to transmit from host to the wire**

\*Note:

- file names are underlined
- functions/macros are in **bold**
- actual code in *italics*

isr.s: DOORBELL INT

- 1) detects transmit interrupt from NIC 1 or 2
- 2) sets appropriate flag

anaconda.c: **main()**

- 1) calls function **FLUSH\_PCI()**
- 2) looks for transmit flag from NIC 1 or 2
- 3) if flag is set for either one call function **HostTransmit()**

hosttx.c: **HostTransmit()**

Input Parameters: None

Output Parameters: None

Description:

- 1) calls macro **TRANSMIT\_AVAILABLE** defined in hosttx.h to ensure that there is a transmit request from at least one of the NICS
- 2) calls macro **NIC1\_XMIT** defined in hosttx.h to see if host requested transmit on NIC 1
  - a) if so clear the flag from EventFlags and call function **ProcessNicTransmits** for NIC 1
- 3) calls macro **NIC2\_XMIT** defined in hosttx.h to see if host requested transmit on NIC2
  - a) if so clear the flag from EventFlags and call function **ProcessNicTransmits** for NIC 2

```
#define NIC1_XMIT(f)          (f & EVENT_XMIT1)
#define NIC2_XMIT(f)          (f & EVENT_XMIT2)
#define TRANSMIT_AVAILABLE(f) (NIC1_XMIT(f) || NIC2_XMIT(f))
```

hosttx.c: **ProcessNicTransmits()**

Input Parameters: nicID - The Nic on which the transmit is being requested  
 values: 0 → NIC 1  
 1 → NIC 2

Output Parameters: None

Description:

- 1) is NIC open
  - a) yes, go back to **main()**, i.e. no transmission
- 2) Check if the Threshold has been exceeded
  - a) if true, get out and come back at a later time to process the remaining. Threshold used – process 35 packets for one NIC so not to starve packets for other NIC
    - i) process *commands* from NIC 1 or 2
    - ii) event raised for other NIC so set flag for appropriate NIC
 

```
if (nicId == 0) {
                if (EventFlags & EVENT_XMIT2) {
                    // Frames are waiting for Nic 2 "the other nic"
                    EventFlags |= EVENT_XMIT1;
                    SwitchToNic2 ++;
                    break;
                }
            }
```
- 3) call **DPDListEnqueue** since direct transmit frames (a.k.a. no value add = no TCP segmentation, no IP security) are available
- 4) call **NicSend** to queue packets for transmission
- 5) allocate local descriptor from local DPD pool

- a) call **DpdAllocate**
  - i) if there are none get out and come back later
- b) copy tx descriptor into local memory via DPD
  - i) schedule time
  - ii) frame length
  - iii) max segment size
  - iv) security association
  - v) control flags
  - vi) packet pointers
  - vii) vlan tag
  - viii) fragment addresses and lengths
- c) free tx descriptor so that driver can reuse it, increment index to tx descriptor array
- d) by looking at control flags stored in DPD
  - i) if IPSEC is requested, create chain of frames for IPSEC, curIpssecChain
  - ii) if TCPSEG is requested, create chain of frames for TCPSEG, curTcpsegChain
  - iii) otherwise send frames directly to transmit task, curTxChain
- e) Now Q all frames recovered to specific tasks for further processing. Each chain that was created in the above loop is queued into the linked list specific to each task.
  - i) if direct transmit frames available (no value add)
    - (1) queue up packets by calling **DPDListEnqueue**
    - (2) call NIC transmit function **NicSend** to queue packets for transmission
  - ii) if IPSEC frames available
    - (1) queue up packets by calling **DPDListEnqueue**
  - iii) TCPSEG frames available
    - (1) queue up packets by calling **DPDListEnqueue**
    - (2) call **TCPSEgSend** to create TCP segments

#### DPDLinkedList.c: DPDListEnqueue

Input Parameters: l – DPD linked list to add chain of descriptors to  
 choose DPD linked list to be either NicDpd (direct transmit), IpSecDpd, or  
 TcpSegDpd  
 head – head of the descriptor chain  
 tail – tail of the descriptor chain  
 DPDCount – number of descriptors in chain

Output Parameters: l – DPD linked list

Description: Enqueues chain of DPDCount descriptors to the end of the linked list, l

#### nicsend.c: NicSend

Input Parameters: pNic – structure containing information about a NIC head as defined in  
nichead.h

Output Parameters: unsigned int (UINT) – reports the status of processing

Description:

- 1) if NIC is resetting do not append to DPD linked list and get out
- 2) call **DPDListDequeue** to dequeue DPDs
- 3) For NIC 1 or 2
  - a) copy some stuff over to another data structure OSD1 as defined in nichead.h
  - b) call **RingBufAdd**
- 4) calculate bytes in DPD linked list
- 5) if there's stuff to send (dpdcount != 0)
- 6) set some variables
- 7) call **IRQ\_DEVICE\_CLEAR\_SAVE**
- 8) set some more variables
- 9) call **IRQ\_DEVICE\_RESTORE**
- 10) if still stuff in queue reset timer by calling **NicSetCompleteTimer2**
- 11) return status: command success

Tcpseg.c: TCPSegSend

Input Parameters: pNic – NIC that requested TCP segmentation  
 Output Parameters: None  
 Description: See “Trace Analysis for TCP segment packets” by Angel/Mauricio

DpdLinkedList.c: DPDListDequeue

Input Parameters: l – DPD linked list to remove chain of descriptors from; choose DPD linked list to be either NicDpd (direct transmit), IpSecDpd, or TcpSegDpd  
 DPDCount – number of descriptors to remove  
 Output Parameters: head – head of the descriptor chain  
 tail – tail of the descriptor chain  
 returns the actual number of descriptors removed  
 Description: returns all of DPDs in list if DPDCount == 0. otherwise returns part of the list (a number of descriptors as specified by DPDCount).

Ringbuf.c: RingBufAdd

Input Parameters: obj – an object to be added to the ring buffer  
 ring – previously created ring buffer where the obj will be added  
 id – caller defined value that gets stored in the interanl address field of the obj  
 Output Parameters: None  
 Description: Adds an object pointed to by “\*obj”, to a previously created ring buffer “\*ring”.

globals.h: IRQ\_DEVICE\_CLEAR\_SAVE (INT\_DEVICE, save\_val)

Description: Save the IRQ enable bit of device “INT\_DEVICE” into the variable “save\_val”.  
 Then clear the same bit to disable IRQs from that device

globals.h: IRQ\_DEVICE\_RESTORE (save\_val)

Description: Restore the previously saved device IRQ enable bit in the variable “save\_val” Thus re-enabling IRQs from that device if that bit is set

## Variables

EventFlags unsigned int used to keep track of events that occurred

## Appendix C: Pseudocode for TCPSegSend()

Dequeue all enqueued DPDs from tcpseg packet linked list

For next DPD from dequeued DPDs

*Copy TCP/IP header from host memory to temp buffer in Anaconda*

- Clean out upper 16 bits of *FrameLength* field
  - Save *FrameLength* in next *InLenArray* entry
- Get first fragment address from DPD
- For first 128 bytes of packet (header length)
  - Extract fragment length from current fragment
  - Calculate whether current fragment has entire header or just part (bytes to transfer calculation)
  - Grab calculated amount of header from host memory and save in local buffer

*Make copies of headers and modify TCP/IP fields*

- Calculate IP header length and its address in local buffer
- Calculate TCP header length and its address in local buffer
- Swap TCP header sequence number from little endian to big endian and save in its own variable
- Ensure that copy of TCP control field bits [2:1] are set
- Turn off PUSH and FIN bits in TCP control field of header
- Calculate length of TCP payload
- Calculate starting address of TCP payload

*Segment the TCP data payload*

Allocate a DPD (child)

While there is still TCP payload

Copy TCP/IP header into first fragment of child DPD

Fill rest of fragments (size and address, not actual data) with chunks of TCP payload in MSS sizes

Modify IP length field

Modify TCP sequence number

If TCP payload was too large and did not fit into 1 DPD

Allocate a DPD (another child)

If 10 DPDs are in the linked list

send DPDs to NIC

start new linked list

else

continue with existing linked list

Set PUSH and FIN bits in last segment if set in original header

Send DPDs on linked list to NIC

Process next dequeued DPD (from TCPSEG linked list)

Free DPD that was just used (dequeued DPD from TCPSEG linked list)

## Appendix D: Packet Receive from Anaconda Perspective

Note:

- file names are underlined
- functions/macros are in **bold**
- variable names in *italics*
- actual code in Courier font

### ***Packet Receive Algorithm (Overview)***

(For more details, see Anaconda Microcode Design Specification Version 1.5, pp.17-18)

- 1) Interrupt from Hurricane
- 2) Locate Up Packet Descriptor (UPD) in UpList
  - a) Do any processing necessary
- 3) Place packet ID in “Receive Complete” queue (*RxCmpltQ*)
- 4) Raise interrupt notifying host of packet receive completion
- 5) Grab free Rx descriptors from “Receive Free” queue (*RxFreeQ*) and add to UpList for Hurricane to use
- 6) Host driver clears the corresponding entry in *RxCmpltQ*
- 7) Host driver resets contents of RX descriptor and places it in *RxFreeQ*

### ***Detailed description of how to receive packets from the NIC (Hurricane)***

isr.s: check4moreIRQ

- detect Hurricane interrupt from Hurricane 1 or 2
- call function **NicIsr()**

nicisr.c: **NicIsr()**

Input parameters:        *nicID* – The NIC on which the receive is from

Output parameters:      none

Description:

This function handles the interrupt causes. It should examine hardware status to determine the reason(s) for the interrupt and invoke the appropriate handlers. It is expected to loop handling interrupt causes until there are no remaining. At that point it should re-enable interrupts from the adapter and return. [as stated in the code]

- 1) Check interrupt status bits
  - a) Interrupt associated with receive: NIC\_INT\_UP\_COMPLETE
- 2) Dispatch the appropriate handler
  - a) Handler associated with receive: **NicUpComplete()**

nicrecv.c: **NicUpComplete()**

Input parameters:        *pNic* – The data structure containing all information about the NIC on which the receive is from

Output parameters:      none

Description:

This function processes UPDs from Hurricane’s UpList. It takes the UPD from the UpList and enters its packet ID into the *RxCmpltQ* for the host driver. Packet data is DMA’d in this function from Hurricane to host memory. The host is notified of the receive packet completion.

- 1) Get UPD

While still UPDs to be processed (not free UPDs) in UpList

- 2) Check for runt frame (frame with length less than the minimum Ethernet frame size)
- 3) If runt frame or any error with frame
  - a) Call **NicRxReject()**
  - b) Chain up current UPD with list of error UPDs (to be reused later)
- 4) If no errors, update receive packet counter and receive byte counter
- 5) Check if *RxCmpltQ* is full, go to 14
- 6) If *RxCmpltQ* is empty (call to **RxQEmpty()**) and if gathering receives (*rx\_event*) before issuing DRBL\_RX\_COMPLETE interrupt to the host, start timer

- 7) Copy UPD status and packet ID over to the specified *RxCmpltQ* entry
- 8) If gathering many receives before issuing an interrupt (DRBL\_RX\_COMPLETE) to the host
  - a) Increment number of receives
  - b) Issue DRBL\_RX\_COMPLETE interrupt to host and stop timer when number of receives exceeds the maximum defined
- 9) If not gathering receives before issuing DRBL\_RX\_COMPLETE interrupt to the host, issue now
- 10) Flush Footbridge Outbound write FIFO; aka DMA packet data to host
- 11) Increment index to *RxCmpltQ*
- 12) Get next UPD and release current one. Go back to 2 if more UPDs.
- 13) If a chain of error UPDs exist end that chain.
- 14) Restore UPD to be processed again
- 15) Call **NicSetupRxUpd()**
- 16) Clear interrupt status bit (NIC\_INT\_UP\_COMPLETE)

Note: If a ringbuffer (RINGBUF) is defined in preprocessor, then there is some extra code that is executed in *NicUpComplete()*. Ring buffer is used for logging information/objects for the debugger.

#### nicrecv.c: NicRxReject()

Input parameters: *pNic* – The data structure containing all information about the NIC on which the receive is from  
*upPktStatus* – status of packet received

Output parameters: none

Description:

Counts errors associated with the current receive packet. Specific errors are *StatRxAlignment*, *StatRxBadCRC*, *StatRxOversize*.

#### nicrecv.c: RxQEmpty()

Input parameters: *head* – The NIC on which the receive is from

Output parameters: BOOLEAN

Description:

Returns true if the *RxCmpltQ* is empty. Otherwise returns false.

#### nicutil.c: NicSetupRxUpd()

Input parameters: *pNic* – The data structure containing all information about the NIC on which the receive is from

Output parameters: none

Description:

This function gets free Rx descriptors for Hurricane's UpList. It grabs all free Rx descriptors from *RxFreeQ* and also any error UPDs waiting to be reused. These are all added to the UpList so that Hurricane has UPDs to fill when more packets are received.

- 1) Get next free Rx descriptor address from the *RxFreeQ*
- 2) If Rx descriptor is already used go to 7
- 3) Clear the entry from the *RxFreeQ*
- 4) Increment index to *RxFreeQ*
- 5) Add all other free Rx descriptors from *RxFreeQ* to the current Rx descriptor by chaining each one to the end of the list
- 6) If a previous list of UPDs exist
  - a) Chain the list of error UPDs to the end of the current UPD list
  - b) Add current UPD list to previous UPD list
- 7) If there is no UpListPtr and came from 2, issue (DRBL\_NEED\_RX\_DES) interrupt to NIC (specified by a component of *pNic*) driver
- 8) If there is no UpListPtr and still UPDs, traverse list to find Rx descriptor that has not completed its receive. Write the address of this Rx descriptor into the UpListPointer.

The Uplist and UpListPtr are critical sections of code. The Uplist is accessed by the Hurricane driver and the microcode. The UpListPtr is accessed by the microcode. Because of this, the microcode must "stall" the receive

engine on the adapter before making any modifications to the UpList or the UpListPtr. After the modifications the microcode must “uninstall” the receive engine. So for 6b and 8, calls to stall and un stall are made before and after modifying the critical sections, respectively. [Cyclone Adapters ERS version 1.0 p.31, p.41]

### Terminology

| <i>Term</i>                 | <i>Description</i>   |
|-----------------------------|--|
| Up Packet Descriptor (UPD)* | Packet descriptors that contain pointers to fragment buffers for receive data. Similar to Rx descriptor but is used as a communication tool between Hurricane and microcode.   |
| UpList*                     | Linked list of UPDs from Hurricane   |
| UpListPtr*                  | Points to the head (current upload packet) of the UpList   |
| Receive (Rx) descriptor**   | Contains fragment information for pre-allocated receive buffers in host memory. Chained to form Hurricane’s UpList. Similar to UPD but is used as a communication tool between microcode and host.                           |
| RxFreeQ**                   | Stores addresses of free Rx descriptors that can be queued into Hurricane’s UpList. Host driver writes available Rx descriptor addresses. Microcode retrieves addresses when it needs Rx descriptors and zeroes the entries. |
| RxFreeQCurrIndex            | Microcode’s current index into the array.  |
| RxCmplQ**                   | Stores the packet ID when the packet has been received. Microcode writes the packet ID when the packet is ready for the host driver. The entry is zeroed by the host driver.   |
| RxCmplQCurrIndex            | Microcode’s current index into the array.  |
| CurrentUpd                  | current Rx descriptor for processing (in UpList?)  |
| NextUsedUpd                 | next Rx descriptor waiting for completion (in UpList?)   |

\*Cyclone Adapters ERS Version 1.0

\*\*Anaconda Microcode Specifications Version 1.5

### Associated functions

#### nicisr.c: NicFillIntDispatch()

Input parameters: *pNic* – The data structure containing all information about the NIC on which the receive is from

Output parameters: none

Description:

This function fills in interrupt dispatch table, *IntDispatch*, which is used by the interrupt handler, **nicisr()**, to dispatch the handler for the highest priority interrupt reason in *IntStatus*. [as stated in code] This function is only called by **NicInit()** in nicinit.c to setup the dispatch table during initialization.

#### nicrecv.c: ReceiveStall()

Input parameters: none

Output parameters: none

Description:

This function starts the receive action again because the UpListPtr is Null. This function is called by **main()** in Anaconda.c.

For each NIC:

- 1) check IPG tuning (can’t find what IPG means)
- 2) check link speed and duplex state via *oldTimer3Tick*
- 3) If UpListPtr is null, call **NicUpComplete()**

## Appendix E: Pseudocode for Receive

While still UPDs to be processed in Uplist  
    check for errors in frame  
    if error, attach UPD to error UPD list (for reuse)  
    Copy UPD status and packet ID to the specified RxCmpltQ entry  
    DMA packet data to host  
    Get next UPD and release current one

Get next free Rx descriptor from RxFreeQ  
Chain all other free Rx descriptors from RxFreeQ to current Rx descriptor  
Chain error UPDs to current UPD list  
Chain current UPD list to previous UPD list  
Ensure that an UpListPtr exists by either raising interrupt to NIC or look for Rx descriptor that has not completed its receive

## Appendix F: Ping Pong Application Program

```

file: common.h
// -----
// IPv6 is now defined (or not) in sources
// #define IPv6

// #define SENDSTRING "Small test message[number x]"

#if !defined(WIN32_LEAN_AND_MEAN)
#define WIN32_LEAN_AND_MEAN
#endif

#include <winsock2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#if defined(IPv6)
#include <ws2ip6.h>
int parse_addr(char *s, struct sockaddr_in6 *sin6);
#endif

// instrumentation/sending stuff
#define DEFAULT_BUFLEN 29
#define ITERATIONS 10
#define MAXITERATIONS 10000

// NTtimer stuff. originally us/millisecc timer from ttcp code.
void prepNTQueryTimer(void);
void readNTQueryTimer(int i, int getTimes);
void printNTQueryTimes(int iterations);
void printNTQueryHelp();
extern __int64 NTQtime0; // used to hold last value read
extern __int64 NTQtime1; // used to hold current value read
extern __int64 NTQfreq;
extern DWORD NTQtimes[MAXITERATIONS]; // times in microseconds between read_timer calls

extern short int numberOfReadsPerIteration[MAXITERATIONS];

// network connectivity stuff
#define DEFAULT_PORT 5001
#define DEFAULT_PROTO SOCK_STREAM // TCP

#define DEFAULT_IP_SUBDOMAIN "129.65.26."

//-----other stuff

int sendBuffer(SOCKET msgsock, char *Buffer, int BufLen, int iter, int thisisPingNotPong, int
getTimes);
int receiveBuffer(SOCKET msgsock, char *Buffer, int BufLen, int iter, int thisisPingNotPong,
int getTimes);
void dumpTimesAndExit(int exitCode, int iterations, int getTimes);

void extraSetup(char *params, int *ExtraFlag);
void allocateBuffers(char **BufferIn, char **BufferOut, int size);
void doExtraAfterReceive(int iteration, char *Buffer, int BufLen);
void doExtraBeforeSend(int iteration, char *BufferOut, int BufLen);
void startupMessage(int getTimes);

#include "penty.h"

```

```

file: common.c
/* this is included into ping.c/ping.c instead of as separate file
   because of (1) #define chris... at top of those files.
   (2) SOCKET define. MIGHT be hard to diff between ip4/ip6 sockets here
   need to define those in makefile or something

this file has:
    send/receive routines used by ping and pong.
    these routines are instrumented for timing measurements
    NTtimers: routines using QueryPerformanceCounter for measurements
*/

#include "common.h"

/* global variables ----- globals */
// NTtimer stuff. originally us/millisecond timer from ttcp code.

__int64 NTQtime0; // used to hold last value read
__int64 NTQtime1; // used to hold current value read
__int64 NTQfreq;
DWORD NTQtimes[MAXITERATIONS]; // times in microseconds between read_timer calls

short int numberOfReadsPerIteration[MAXITERATIONS];

/*-----end globals */

#include "version.h"
void startupMessage(int getTimes)
{
    printf("version: %s ",PPVERSION);
    #if defined(IPv6)
        printf("IPv6 ");
    #else
        printf("IP4 ");
    #endif

    if (getTimes==0)
        printf("getTimes? NONE. no NTQuery or PentCounters");
    else if (getTimes==1)
        printf("getTimes? NTQuery only. no pent ctrs");
    else
        printf ("getTimes? NTQuery AND pentCounters");
    printf("\n");
}

#define ALIGN_SIZE (4*1024)
char *Array1,*Array2;
void allocateBuffers(char **BufferIn,char **BufferOut,int size)
{
    Array1 = malloc(size+ALIGN_SIZE);
    Array2 = malloc(size+ALIGN_SIZE);
    if (Array1 == NULL || Array2 == NULL) {
        printf("malloc failed to allocate space\n");
        exit(1);
    }
    *BufferIn = (char *) (((unsigned int) Array1+ALIGN_SIZE-1) & ~(ALIGN_SIZE-1)) ;
    *BufferOut= (char *) (((unsigned int) Array2+ALIGN_SIZE-1) & ~(ALIGN_SIZE-1)) ;
}
void unAllocateBuffers()
{
    free(Array1);
    free(Array2);
}

void extraSetup(char *params, int *ExtraFlag)
{
    if (params[1] == 'p')
        *ExtraFlag=1;
    else {
        printf("unknown Extra flag %s\n",params);
    }
}

```

```

}
void doExtraAfterReceive(int iteration, char *Buffer, int BufLen)
{
    char string[80];

    strcpy(string, "\treceive ");
    itoa(iteration, string, 16);
    strcat(string, ", buffer is ");
    if (BufLen < 20 )
        strncat(string, Buffer, BufLen);
    else {
        strncat(string, Buffer, 10);
        strcat(string, " ... ");
        strncat(string, Buffer+BufLen-5, 5);
    }
    printf(string);
}
void doExtraBeforeSend(int iteration, char *BufferOut, int BufLen)
{
    int i;
    for (i=0; i<BufLen; i++)
        BufferOut[i] = 'A' + iteration ;
}

/* ----- send/rcv routines */

int sendBuffer(SOCKET msgsock, char *Buffer, int BufLen,
               int iter, int thisisPingNotPong, int getTimes)
{
    int retval;
    if (getTimes>1) pentyMARK(USER+2, ATstart);
    retval = send(msgsock, Buffer, BufLen, 0);
    if (getTimes>1) pentyMARK(USER+2, ATend);

    if (retval == SOCKET_ERROR) {
        fprintf(stderr, "send() failed at iteration %d : error
%d\n", iter, WSAGetLastError());
        if (thisisPingNotPong) {
            dumpTimesAndExit(-1, iter, getTimes);
        }
    }
    return retval;
}

int processReceiveError(int retval, SOCKET msgsock,
                        int iter, int thisisPingNotPong, int getTimes);
int receiveMoreBuffer(SOCKET msgsock, char *buf, int BufLenLeft,
                      int iter, int thisisPingNotPong, int getTimes);

int receiveBuffer(SOCKET msgsock, char *Buffer, int BufLen,
                  int iter, int thisisPingNotPong, int getTimes)
{
    int retval;

    if (getTimes>1) pentyMARK(USER+3, ATiteration|iter);
    retval = recv(msgsock, Buffer, BufLen, 0 );
    if (getTimes>1) pentyMARK(USER+3, ATend);

    if (retval == SOCKET_ERROR || retval == 0)
        processReceiveError(retval, msgsock, iter, thisisPingNotPong, getTimes);

    if (retval < BufLen ) {
        retval = receiveMoreBuffer(msgsock, Buffer+retval, BufLen-
retval, iter, thisisPingNotPong, getTimes);
        if (retval>1) retval = BufLen;
    }

    // printf("Received %d bytes, data [%s] from client\n", retval, Buffer);
    return retval;
}

```

```

int receiveMoreBuffer(SOCKET msgsock, char *buf, int BufLenLeft,
                    int iter, int thisisPingNotPong, int getTimes)
{
    int retval;
    int TotalToRead = BufLenLeft;
    int iter2=1; // already did one
    while (BufLenLeft>0) {
        iter2++;
        if (getTimes>1) pentyMARK(USER+5,ATiteration2|iter2);
        retval = recv(msgsock,buf,BufLenLeft,0 );
        if (getTimes>1) pentyMARK(USER+5,ATend);

        numberOfReadsPerIteration[iter]++;

        if (retval == SOCKET_ERROR || retval == 0)
            return processReceiveError(retval,msgsock,iter, thisisPingNotPong, getTimes);
        if (retval>BufLenLeft)
            printf("read %d bytes more than expected at iteration %d\n",
                retval-BufLenLeft,iter);

        BufLenLeft -= retval;
        buf += retval;
    }
    // stupid: wasnt returning anything. added this, but need to check
    // to fix this up.
    return TotalToRead;
}

int processReceiveError(int retval, SOCKET msgsock,
                    int iter, int thisisPingNotPong, int getTimes)
{
    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"recv() failed at iteration %d: error
%d\n",iter,WSAGetLastError());
        closesocket(msgsock);
        if (thisisPingNotPong) {
            dumpTimesAndExit(-1,iter,getTimes);
        }
        else
            return 0;
    }
    if (retval == 0) {
        if (thisisPingNotPong) {
            printf("Read back nothing at iteration %d. Will quit.\n",iter);
            closesocket(msgsock);
            dumpTimesAndExit(-1,iter,getTimes);
        }
        else {
            printf("Client closed connection at iteration %d\n",iter);
            closesocket(msgsock);
            return 0;
        }
    }
    return 0;
}

// only exits if exitcode is non-zero
void dumpTimesAndExit(int exitCode, int iterations, int getTimes)
{
    WSACleanup();
    if (getTimes>0) printNTQueryTimes(iterations);
    if (getTimes>1) pentyStop();

    unAllocateBuffers();
    if (exitCode != 0)
        exit(exitCode);
}

/* ----- timing routines */

```

```

void
prepNTQueryTimer()
{
    (void) QueryPerformanceFrequency((LARGE_INTEGER *)&NTQfreq);
    (void) QueryPerformanceCounter((LARGE_INTEGER *)&NTQtime0);

    { int i;
      for (i=0; i< MAXITERATIONS; i++)
          numberOfReadsPerIteration[i] = 0;
    }
}

void
readNTQueryTimer(int i, int getTimes)
{
    // DWORD hold;

    if (getTimes>1) pentyMARK(USER+4,ATstart);
    (void) QueryPerformanceCounter((LARGE_INTEGER *)&NTQtime1);
    if (getTimes>1) pentyMARK(USER+4,ATend);

    NTQtimes[i] = (DWORD) (NTQtime1-NTQtime0);
    NTQtime0 = NTQtime1;
    return;
}

void
printNTQueryTimes(int iterations)
{
    int i;
    printf ("NTQueryTimes in usec. NT PerfCounter has frequency of %.2f MHz (%.2f ns)\n",
            NTQfreq/1e6, (double) 1e9/NTQfreq);
    if (iterations<20)
        for (i=0; i<iterations; i++) {
            printf("from start of recv %i to next recv: %ld (%d reads)%s\n",
                    i,
                    (DWORD) ( (double) NTQtimes[i] * (1e6/ (double) NTQfreq) ), /*
convert to usec */
                    (i==0)? 0: numberOfReadsPerIteration[i]+1,
                    (i==0)? "no receives; just startup": "");
        }
    else {
        printf("times in uSec from start of 1 recv to start of next\n %4d ",0);
        for (i=0; i<iterations; i++) {
            printf(" %5ld",
                    (DWORD) ( (double) NTQtimes[i] * (1e6/ (double) NTQfreq) ) );
            if (i>0 && numberOfReadsPerIteration[i]>0)
                printf("(%d)",numberOfReadsPerIteration[i]+1);
            if (i%10 == 9) printf("\n %4d ",i+1);
        }
    }

    // print average
    { int startat=2;
      DWORD sum;
      for (i=startat,sum=0; i<iterations; sum+= NTQtimes[i++])
          ;
      printf("average time from start of iteration %d to last (%d iters) is %.2f\n",
            startat,iterations-startat,
            (double) ( sum* (1e6/(double)NTQfreq) / (double)(iterations-startat)));
    }
    return;
}

void
printNTQueryHelp()
{
    printf("----- info on NTQuery counters\n");
    printf("This uses NT function QueryPerformanceCounter\n\
it ticks every N microseconds. N determined by QueryPerformanceFrequency\n\
value of N is printed when run this program\n\

```

```
    on a 200Mhz machine, it takes just over 10us to call QueryPerformanceCounter\n");  
    printf("----- end info on NTQuery counters\n");  
}
```

```
/* ----- end of timing routines */
```

```

file: ping.c
/*****\
* ping.c
* Originally from simplec.c - Simple TCP/UDP client using Winsock 1.1
* This is a part of the Microsoft Source Code Samples.
* Copyright (C) 1996 Microsoft Corporation.
*****/

#include "common.h"

void Usage(char *progname) {
    startupMessage(2);
    fprintf(stderr,"Original format (as simplec) [-l option gone]:\n\
\t%s -p [protocol] -n [server] -e [endpoint]\n\
\t -p protocol:\t TCP or UDP [default: TCP]\n\
\t -n server\t server name or IP address [default: localhost]\n\
\t -e endpoint\t port to listen on [default: 5001]\n\n",progname);

    fprintf(stderr,"NEW params (above still work. and note, usually need -n):\n\
\t -i iterations -S sizeOfBuffer (NO max, default: %d)\n",
        DEFAULT_BUFLEN);
    fprintf(stderr,"\t -x (no perf measurements) -t (no pentium meas.)\n\
\t -K [!n!k] connect to kernel (!n means: but not to NDIS)\n\
\t -v verbose\n");

    fprintf(stderr,"You can truncate server number (e.g., -n 5 gives %s5)\n",
        DEFAULT_IP_SUBDOMAIN);
    fprintf(stderr,"Examples:\n\
\t -n 75 -i 100 -S 1024\n\
\t -n 129.65.29.75 -i 1000 -S 100k -t (can use k in size)\n\
\t -n ::129.65.29.75 (: : if using ip6)\n");

    WSACleanup();
    exit(1);
}

#ifdef IPv6
int
parse_addr(char *s, struct sockaddr_in6 *sin6)
{
    struct hostent *h;

    sin6->sin6_family = AF_INET6;

    if (!inet6_addr(s, &sin6->sin6_addr)) {
        // Presumably the user gave us a DNS name.

        h = getnodebyname(s, AF_INET6, AI_DEFAULT);
        if (h == NULL)
            return FALSE;

        memcpy(&sin6->sin6_addr, h->h_addr, sizeof(struct in6_addr));
        freehostent(h);
    }

    return TRUE;
}
#endif

int __cdecl main(int argc, char **argv) {
    int retval;

    // network connectivity stuff
    char *server_name= "localhost";
    unsigned short port = DEFAULT_PORT;
    int socket_type = DEFAULT_PROTO;
#ifdef IPv6
    struct sockaddr_in6 server;
#else
    struct sockaddr_in server;
    struct hostent *hp;
#endif

```

```

        unsigned int addr;
#endif
    WSADATA wsaData;
    SOCKET conn_socket;

    char fixedIPAddress[80]; // for turning "5" into "129.65.29.5"

    // vars for sending stuff
    int i, iterations = ITERATIONS;
    int getTimes=2; // do performance measurements
    int BufLen=DEFAULT_BUFLLEN;
    char *BufferIn;
    char *BufferOut;
    int ExtraFlags=0;

/* ----- parse arguments-----*/
    if (argc >1) {
        for(i=1;i <argc;i++) {
            if ( (argv[i][0] == '-') || (argv[i][0] == '/') ) {
                switch(argv[i][1]) {

                    case 'p': // from old prog: TCP or UDP
                        if (!strcmp(argv[i+1], "TCP") )
                            socket_type = SOCK_STREAM;
                        else if (!strcmp(argv[i+1], "UDP") )
                            socket_type = SOCK_DGRAM;
                        else
                            Usage(argv[0]);
                        i++;
                        break;

                    case 'n': // from old prog: get server address
                        server_name = argv[++i];
                        break;
                    case 'e': // from old prog: port number
                        port = atoi(argv[++i]);
                        break;

                    //----- new parameters
                    case 'i':
                        iterations = atoi(argv[++i]);
                        break;
                    case 's':
                        BufLen = atoi(argv[++i]);
                        if (tolower(argv[i][strlen(argv[i])-1]) == 'k') BufLen *=1024;
                        break;
                    case 't':
                        getTimes=1; // don't call pentyMark
                        break;
                        case 'x':
                            getTimes=0; // don't do any performance measurements
                            break;
                    case 'K':
                        pentySetFlags(argv[i]);
                        break;
                        case 'v':
                            pentyHelp();
                            printNTQueryHelp();
                            Usage(argv[0]); //exits
                            break;
                    case 'E': // -E[p] extra stuff, such as print buffer.
                        extraSetup(argv[i], &ExtraFlags);
                        default:
                            Usage(argv[0]);
                            break;
                }
            }
            else
                Usage(argv[0]);
        }
    }

```

```

    }
}

startupMessage(getTimes);

/* ----- get address -----*/

if (WSAStartup(0x202,&wsaData) == SOCKET_ERROR) {
    fprintf(stderr,"WSAStartup failed with error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}

if (port == 0){
    Usage(argv[0]);
}

// new: added this to turn number into full ip address.
// if looks like a number, and no periods, tack on default subdomain
if (isdigit(server_name[0]) && !strchr(server_name, '.')) {
#   if defined(IPv6)
#       strcpy(fixedIPAddress, "::");
#   else
#       strcpy(fixedIPAddress, "");
#   endif
    strncat(fixedIPAddress,DEFAULT_IP_SUBDOMAIN,60);
    strncat(fixedIPAddress,server_name,5);
    //sprintf(fixedIPAddress,"%s%s",DEFAULT_IP_SUBDOMAIN,server_name);
    server_name = fixedIPAddress;
}
//
// Attempt to detect if we should call gethostbyname() or
// gethostbyaddr()
#if defined(IPv6)
    memset(&server,0,sizeof(server));
    if (!parse_addr(server_name,&server)) {
        fprintf(stderr,"Client: Cannot resolve address [%s]: Error %d\n",
            server_name,WSAGetLastError());
        WSACleanup();
        exit(1);
    }
    server.sin6_port = htons(port);
#else
    if (isalpha(server_name[0])) { /* server address is a name */
        hp = gethostbyname(server_name);
    }
    else { /* Convert nnn.nnn address to a usable one */
        addr = inet_addr(server_name);
        hp = gethostbyaddr((char *)&addr,4,AF_INET);
    }
    if (hp == NULL) {
        fprintf(stderr,"Client: Cannot resolve address [%s]: Error %d\n",
            server_name,WSAGetLastError());
        WSACleanup();
        exit(1);
    }
    //
    // Copy the resolved information into the sockaddr_in structure
    //
    memset(&server,0,sizeof(server));
    memcpy(&(server.sin_addr),hp->h_addr,hp->h_length);
    server.sin_family = hp->h_addrtype;
    server.sin_port = htons(port);
#endif

/* ----- connect ---*/

#if defined(IPv6)
    conn_socket = socket(AF_INET6,socket_type,0); /* Open a socket */
#else

```

```

conn_socket = socket(AF_INET,socket_type,0); /* Open a socket */
#endif
if (conn_socket <0 ) {
    fprintf(stderr,"Client: Error Opening socket: Error %d\n",
            WSAGetLastError());
    WSACleanup();
    return -1;
}

//
// Notice that nothing in this code is specific to whether we
// are using UDP or TCP.
// We achieve this by using a simple trick.
// When connect() is called on a datagram socket, it does not
// actually establish the connection as a stream (TCP) socket
// would. Instead, TCP/IP establishes the remote half of the
// ( LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
// This enables us to use send() and recv() on datagram sockets,
// instead of recvfrom() and sendto()

#if defined(IPv6)
    printf("Client connecting to somewhere\n");
#else
    printf("Client connecting to: %s\n",hp->h_name);
#endif
if (connect(conn_socket, (PSOCKADDR)&server,sizeof(server))
    == SOCKET_ERROR) {
    fprintf(stderr,"connect() failed: %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}

/* -----start of test and loop-----*/
/* -----*/
{
    int bufError=0;

    // init buffer
    // BufLen default is 29, set by -S option
    allocateBuffers(&BufferIn,&BufferOut,BufLen);
    for (i=0; i<BufLen; i++)
        BufferOut[i] = '*' ; // '0' + (i%10);

    if (getTimes>0) prepNTQueryTimer();
    if (getTimes>1) pentyStart(1); // prep and start pentium timers

    for (i=0; i<iterations; i++) {
        BufferOut[BufLen-1] = '0' + (i%10);
        if (ExtraFlags) doExtraBeforeSend(i,BufferOut,BufLen);

        if (getTimes>0) readNTQueryTimer(i,getTimes);

        retval = sendBuffer(conn_socket,BufferOut,BufLen,i+1,1,getTimes);
        if (retval == SOCKET_ERROR) return -1;
        // printf("Sent Data [%s]\n",Buffer);

        retval = receiveBuffer(conn_socket,BufferIn,BufLen,i+1,1,getTimes);
        if (retval == SOCKET_ERROR || retval == 0) return -1;
        if (retval != BufLen) {
            printf("Error: Received %d bytes of %d from server (iteration
%d)\n",retval,BufLen,i);
            bufError = i+1;
        }
    }
    printf("Out of loop. Terminating connection\n");
    closesocket(conn_socket);
    dumpTimesAndExit(0,iterations,getTimes);
    if (bufError) printf("not all data received starting at iteration %d\n",bufError);
}

```

```
    }  
    return 0;  
}
```



```

        case 'I':
            iterations = atoi(argv[++i]);
            break;
        case 'i':
            iterations = atoi(argv[++i]);
            break;
        case 's':
            BufLen = atoi(argv[++i]);
            if (tolower(argv[i][strlen(argv[i])-1]) == 'k') BufLen *=1024;
            break;
        case 'e':
            port = atoi(argv[++i]);
            break;
    case 't':
        getTimes=1; // don't call pentyMark
        break;
    case 'x':
        getTimes=0; // don't do any performance measurements
        break;
    case 'K':
        pentySetFlags(argv[i]);
        break;
    case 'Z':
        testPentyAndExit();
        break;
    case 'E': // -E[p] extra stuff, such as print buffer.
        extraSetup(argv[i], &ExtraFlags);
        break;
        case 'v':
            pentyHelp();

    printNTQueryHelp();
    Usage(argv[0]);

        break;
    default:
        Usage(argv[0]);
        break;
    }
    }
    else
        Usage(argv[0]);
}
}
startupMessage(getTimes);

/* ----- get address -----*/

    if (WSAStartup(0x202, &wsaData) == SOCKET_ERROR) {
        fprintf(stderr, "WSAStartup failed with error %d\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }

    if (port == 0) {
        Usage(argv[0]);
    }

#ifdef IPv6
    local.sin6_family = AF_INET6;
    //local.sin6_addr.s_addr =
    // (!interface)?INADDR_ANY:inet6_addr(interface, &local.sin6_addr);
    /* Port MUST be in Network Byte Order */
    local.sin6_port = htons(port);
    listen_socket = socket(AF_INET6, socket_type, 0); // TCP socket
#else
    local.sin_family = AF_INET;
    local.sin_addr.s_addr = (!interface)?INADDR_ANY:inet_addr(interface);
    /* Port MUST be in Network Byte Order */
    local.sin_port = htons(port);
    listen_socket = socket(AF_INET, socket_type, 0); // TCP socket
#endif

```

```

if (listen_socket == INVALID_SOCKET) {
    fprintf(stderr,"socket() failed with error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}
//
// bind() associates a local address and port combination with the
// socket just created. This is most useful when the application is a
// server that has a well-known port that clients know about in advance.
//
if (bind(listen_socket, (PSOCKADDR)&local, sizeof(local) )
    == SOCKET_ERROR) {
    fprintf(stderr,"bind() failed with error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}

//
// So far, everything we did was applicable to TCP as well as UDP.
// However, there are certain steps that do not work when the server is
// using UDP.
//

// We cannot listen() on a UDP socket.

if (socket_type != SOCK_DGRAM) {
    if (listen(listen_socket,5) == SOCKET_ERROR) {
        fprintf(stderr,"listen() failed with error %d\n",WSAGetLastError());
        WSACleanup();
        return -1;
    }
}
printf("%s: 'Listening' on port %d, protocol %s\n",argv[0],port,
       (socket_type == SOCK_STREAM)?"TCP":"UDP");

fromlen =sizeof(from);
msgsock = accept(listen_socket, (PSOCKADDR)&from, &fromlen);

if (msgsock == INVALID_SOCKET) {
    fprintf(stderr,"accept() error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}
printf("accepted connection from %s, port %d\n",
#if defined(IPv6)
        inet6_ntoa(&from.sin6_addr),
        htons(from.sin6_port)) ;
#else
        inet_ntoa(from.sin_addr),
        htons(from.sin_port)) ;
#endif

/* -----start of test and loop-----*/
/* -----*/
{
    int bufError=0;

    // init buffer
    // BufLen default is 29, set by -S option
    allocateBuffers(&BufferIn,&BufferOut,BufLen);
    for (i=0; i<BufLen; i++)
        BufferOut[i] = '*' ; // 'a';

    if (getTimes>0) prepNTQueryTimer();
    if (getTimes>1) pentyStart(0);

    for (i=0; i<iterations; i++) {

```

```

    if (getTimes>0) readNTQueryTimer(i,getTimes);

    retval = receiveBuffer(msgsock,BufferIn,BufLen,i+1,0,getTimes);
    if (!retval) continue; // error
        if (retval != BufLen) {
            printf("Error: Received %d bytes of %d from client (iteration
%d)\n",retval,BufLen,i);
            bufError = i+1;
        }
    if (ExtraFlags) doExtraAfterReceive(i,BufferIn,BufLen);

    // printf("Echoing data back to client\n");
    retval = sendBuffer(msgsock,BufferOut,BufLen,i+1,0,getTimes);
}
printf("Out of loop. Terminating connection\n");
closesocket(msgsock);
WSACleanup();
if (getTimes>0) printNTQueryTimes(iterations);
if (getTimes>1) pentyStop();
if (bufError) printf("not all data received starting at iteration %d\n",bufError);
}
return 0;
}

// called when do -Z option. just do a few timers and that is it.
void testPentyAndExit()
{
    int i;
    prepNTQueryTimer();
    pentyStart(1);
    for (i=0; i<1; i++) {
        pentyMARK(1,ATstart);
        readNTQueryTimer(i,1);
        pentyMARK(1,ATend);
    }
    pentyStop();
    exit(0);
}

```

## Appendix G: Send/Server Application Program

### Send source files

file: buffer.hpp

```
//
// buffer.hpp
//

#ifndef BUFFER_HPP
#define BUFFER_HPP

class Buffer {
private:
    // prevent copy operations
    Buffer( const Buffer& );
    Buffer& operator = ( const Buffer& );
protected:
    char* m_pBuffer;           // pointer to the actual buffer
    const unsigned m_xuSize;   // size of buffer in bytes
public:
    // ctor & dtor
    Buffer( unsigned uSize = 256, const char xcInit = 0x00 );
    ~Buffer();
    // typecast operator
    operator char*();
    // accessor(s)
    unsigned Size();
};

/////////////////////////////////////////////////////////////////
// ctor
inline Buffer::Buffer( unsigned uSize, const char xcInit )
    : m_pBuffer( new char[ uSize ] ),
      m_xuSize( uSize )
{
    memset( m_pBuffer, xcInit, uSize );
}

/////////////////////////////////////////////////////////////////
// dtor
inline Buffer::~~Buffer()
{
    delete[] m_pBuffer ;
}

/////////////////////////////////////////////////////////////////
// typecast operator
inline Buffer::operator char*()
{
    return m_pBuffer ;
}

/////////////////////////////////////////////////////////////////
// accessor
inline unsigned Buffer::Size()
{
    return m_xuSize ;
}

#endif // BUFFER_HPP
```

```

file: main.cpp
//-----
#include <vcl.h>
#include <Registry.hpp>
#pragma hdrstop

#include "main.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
const AnsiString g_sRegistryKeyName = "\\Software\\CalPoly3Com\\Anaconda\\Client" ;
const AnsiString g_sServerIpAddrValueName = "ServerIpAddr" ;
const AnsiString g_dwScreenPosition = "ScreenPos" ;
//-----

__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner),
  buffer( BUFFERSIZE, '*' ),
  bSizeChanged( true ),
  nConnectAbortCount( 0 )
{
}
//-----

void __fastcall TForm1::ConnectSpeedButtonClick(TObject *Sender)
{
  if ( ConnectSpeedButton->Down )
  {
    DoConnect();
  }
  else
  {
    if ( ClientSocket1->Active )
    {
      ClientSocket1->Close();
      OnDisconnectControlConfig();
    }
    else
    {
      // The system was in the process of trying to establish a
      // connection with the server when the user unpressed the
      // connect button (i.e., the user aborted the connect attempt).
      // A socket has been created, but it has not been connected yet.
      // So close (deallocate) the socket, and keep track of the number
      // of aborted connection attempts. This information is needed
      // later, when we eventually get a connection failed message
      // from the Winsock DLL (see: ClientSocket1Error()).

      closesocket( (SOCKET) ClientSocket1->Socket->Handle );
      ++ nConnectAbortCount ;
      OnDisconnectControlConfig();
    }
  }
}
//-----

void __fastcall TForm1::ContinuousCheckBoxClick(TObject *Sender)
{
  SendSpeedButton->GroupIndex = ( ContinuousCheckBox->Checked ? 1 : 0 );
}
//-----

void __fastcall TForm1::DoConnect()
{
  if ( ServerIpAddrEdit->Text.IsEmpty() )
  {
    PleaseSpecifyIpAddrMsg();
    return ;
  }
}

```

```

// Disable the IP address edit field
ServerIpAddrEdit->Enabled = false ;

// Provide visual feedback that we're trying to connect
ConnectSpeedButton->Down = true ;
ConnectSpeedButton->Caption = "Connecting..." ;
ConnectSpeedButton->Repaint();
Screen->Cursor = crHourGlass ;

// try to connect with the server app at the new IP address
if ( ClientSocket1->Active )
    ClientSocket1->Close();
ClientSocket1->Address = ServerIpAddrEdit->Text ;

try { ClientSocket1->Open(); }
catch(...)
{
    ReportConnectFailure();
    return ;
}

if ( ClientSocket1->Active )
    EnableSendControls();
}
//-----

void __fastcall TForm1::EnableSendControls( bool Value )
{
    Screen->Cursor = crDefault ;
    Timer1->Enabled = false ;
    SizeMaskEdit->Enabled = Value ;
    ContinuousCheckBox->Enabled = Value ;
    SendSpeedButton->Enabled = Value ;
}
//-----

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    TRegistry* Registry = new TRegistry ;
    try
    {
        if ( Registry->OpenKey( g_sRegistryKeyName, true ) )
        {
            // Save the server's IP address
            Registry->WriteString( g_sServerIpAddrValueName, ServerIpAddrEdit->Text );

            // Save the screen position
            screenPosition.nLeft = Left ;
            screenPosition.nTop = Top ;
            Registry->WriteBinaryData( g_dwScreenPosition, &screenPosition,
                sizeof(screenPosition) );
        }
    }
    catch ( ... ) { }

    Registry->CloseKey();
    delete Registry ;
}
//-----

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    // Load the last used IP address (if any) from the registry
    TRegistry* Registry = new TRegistry ;
    try
    {
        if ( Registry->OpenKey( g_sRegistryKeyName, false ) )
        {
            // Load in the previously specified server IP address
            ServerIpAddrEdit->Text = Registry->ReadString( g_sServerIpAddrValueName );
        }
    }
}

```

```

        try
        {
            // Load in the previous screen position
            Registry->ReadBinaryData( g_dwScreenPosition, &screenPosition,
                sizeof(screenPosition) );
            Left = screenPosition.nLeft ;
            Top = screenPosition.nTop ;
        }
        catch ( ... ) { }
    }
}
catch ( ... ) { }

Registry->CloseKey();
delete Registry ;

Show();
Update();
}
//-----

void __fastcall TForm1::OnDisconnectControlConfig()
{
    Screen->Cursor = crDefault ;
    ConnectSpeedButton->Caption = "Connect" ;
    ConnectSpeedButton->Down = false ;
    ConnectSpeedButton->Enabled = true ;
    ServerIpAddrEdit->Enabled = true ;
    SendSpeedButton->Down = false ;
    DisableSendControls();
}
//-----

void __fastcall TForm1::PleaseSpecifyIpAddrMsg()
{
    OnDisconnectControlConfig();
    ServerIpAddrEdit->Text = "0.0.0.0" ;
    Application->MessageBox(
        "Please type in the IP address of the machine\t\n"
        "that's running the \"server\" application, and\t\n"
        "then press the \"Connect\" button.\t\n",
        "ERROR",
        MB_ICONINFORMATION | MB_OK );
    ServerIpAddrEdit->SetFocus();
    ServerIpAddrEdit->SelStart = 0 ;
    ServerIpAddrEdit->SelLength = -1 ;
}
//-----

void __fastcall TForm1::ReportConnectFailure()
{
    OnDisconnectControlConfig();
    Application->MessageBox(
        "Unable to locate the server application at the\t\n"
        "specified IP address.\n\n"
        "Make sure you have specified the IP address of the\t\n"
        "machine that is running the server.exe application,\t\n"
        "and be sure the server.exe application is running\t\n"
        "before you try to establish a connection.\t\n",
        "Connect Error",
        MB_ICONERROR | MB_OK );
    ServerIpAddrEdit->SetFocus();
}
//-----

void __fastcall TForm1::Sending( bool Value )
{
    ConnectSpeedButton->Enabled = !Value ;
    SizeMaskEdit->Enabled = !Value ;
    ContinuousCheckBox->Enabled = !Value ;
    Timer1->Enabled = Value ;
}

```

```

}
//-----
void __fastcall TForm1::SendSpeedButtonClick(TObject *Sender)
{
    // did the user specify a new data size ?
    if ( bSizeChanged )
    {
        nSendSize = SizeMaskEdit->Text.Trim().ToIntDef(4);
        bSizeChanged = false ;
    }

    if ( 0 == SendSpeedButton->GroupIndex )
    {
        // Simulate a timer timeout by manually calling Timer1Timer().
        // This function will then send the specified amount of data
        // to the server via the socket.
        //
        Timer1Timer(Sender);
    }
    else
    {
        Sending( SendSpeedButton->Down );
    }
}
//-----

void __fastcall TForm1::ServerIpAddrEditKeyPress(TObject *Sender,
char &Key)
{
    if ( VK_RETURN == Key )
    {
        ConnectSpeedButton->Down = true ;
        ConnectSpeedButtonClick(Sender);
        Key = 0 ;
    }
}
//-----

void __fastcall TForm1::SizeMaskEditChange(TObject *Sender)
{
    bSizeChanged = true ;
}
//-----
#pragma warn -aus

void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    char* p = buffer ;
    int nResult = ClientSocket1->Socket->SendBuf( p, nSendSize );
}
#pragma warn .aus
//-----

void __fastcall TForm1::ClientSocket1Connect(TObject *Sender,
TCustomWinSocket *Socket)
{
    ConnectSpeedButton->Caption = "Connected" ;
    EnableSendControls();
}
//-----

void __fastcall TForm1::ClientSocket1Disconnect(TObject *Sender,
TCustomWinSocket *Socket)
{
    OnDisconnectControlConfig();
}
//-----

void __fastcall TForm1::ClientSocket1Error(TObject *Sender,
TCustomWinSocket *Socket, TErrorEvent ErrorEvent, int &ErrorCode)

```

```
{
  switch ( ErrorEvent )
  {
    case eeConnect :
      {
        switch ( ErrorCode )
        {
          case WSAECONNREFUSED :
          case WSAETIMEDOUT :
            if ( Socket->Handle != HWND(INVALID_SOCKET) )
              Socket->Close();
            else
              OnDisconnectControlConfig();
            ErrorCode = 0 ;
            break ;
          }
        }
      }
    break ;
  }
}
//-----
```

```

file: main.h
//-----
#ifndef mainH
#define mainH
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Buttons.hpp>
#include <Mask.hpp>
#include "buffer.hpp"
#include <ExtCtrls.hpp>
#include <ScktComp.hpp>
//-----
class TForm1 : public TForm
{
__published: // IDE-managed Components
    TMaskEdit *SizeMaskEdit;
    TLabel *Label1;
    TSpeedButton *SendSpeedButton;
    TCheckBox *ContinuousCheckBox;
    TTimer *Timer1;
    TGroupBox *ServerInfoGroupBox;
    TLabel *Label2;
    TEdit *ServerIpAddrEdit;
    TSpeedButton *ConnectSpeedButton;
    TClientSocket *ClientSocket1;
    void __fastcall SendSpeedButtonClick(TObject *Sender);
    void __fastcall ContinuousCheckBoxClick(TObject *Sender);
    void __fastcall SizeMaskEditChange(TObject *Sender);
    void __fastcall Timer1Timer(TObject *Sender);
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
    void __fastcall ConnectSpeedButtonClick(TObject *Sender);
    void __fastcall ServerIpAddrEditKeyPress(TObject *Sender, char &Key);
    void __fastcall ClientSocket1Connect(TObject *Sender,
        TCustomWinSocket *Socket);
    void __fastcall ClientSocket1Disconnect(TObject *Sender,
        TCustomWinSocket *Socket);
    void __fastcall ClientSocket1Error(TObject *Sender,
        TCustomWinSocket *Socket, TErrorEvent ErrorEvent,
        int &ErrorCode);

private: // User declarations

    enum { BUFFERSIZE = 100000 };
    Buffer buffer ;
    bool bSizeChanged ;
    int nSendSize ;
    int nConnectAbortCount ;
    struct tagScreenPosition {
        int nLeft, nTop ;
    } screenPosition ;

    void __fastcall DisableSendControls() { EnableSendControls(false); }
    void __fastcall DoConnect();
    void __fastcall EnableSendControls( bool Value = true );
    void __fastcall OnDisconnectControlConfig();
    void __fastcall PleaseSpecifyIpAddrMsg();
    void __fastcall ReportConnectFailure();
    void __fastcall Sending( bool Value );

public: // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//-----
extern PACKAGE TForm1 *Form1;
//-----
#endif

```

```
file: send.cpp
//-----
#include <vcl.h>
#pragma hdrstop
USERES("send.res");
USEFORM("main.cpp", Form1);
//-----
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//-----
```

**Server (receive) source files**

```

file: main.cpp
//-----
#include <vcl.h>
#include <Registry.hpp>
#pragma hdrstop

#include "main.h"

//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
const AnsiString g_sRegistryKeyName = "\\Software\\CalPoly3Com\\Anaconda\\Server" ;
const AnsiString g_dwScreenPosition = "ScreenPos" ;
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner),
  uTotalBytesReceived( 0 ),
  buffer( 256 * 1024 )
{
}
//-----

void __fastcall TForm1::ServerSocket1ClientConnect(TObject *Sender,
  TCustomWinSocket *Socket)
{
  ConnectedCheckBox->Checked = true ;
}
//-----

void __fastcall TForm1::ServerSocket1ClientDisconnect(TObject *Sender,
  TCustomWinSocket *Socket)
{
  ConnectedCheckBox->Checked = false ;
}
//-----

void __fastcall TForm1::ServerSocket1ClientRead(TObject *Sender,
  TCustomWinSocket *Socket)
{
  int nBytesRead = Socket->ReceiveBuf( buffer, buffer.Size() );
  BytesReceivedLabel->Caption = nBytesRead ;
  uTotalBytesReceived += nBytesRead ;
  TotalBytesReceivedLabel->Caption = uTotalBytesReceived ;
}
//-----

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
  ServerSocket1->Close();

  TRegistry* Registry = new TRegistry ;
  try
  {
    {
      if ( Registry->OpenKey( g_sRegistryKeyName, true ) )
      {
        screenPosition.nLeft = Left ;
        screenPosition.nTop = Top ;
        Registry->WriteBinaryData( g_dwScreenPosition, &screenPosition,
          sizeof(screenPosition) );
      }
    }
  }
  catch ( ... ) { }

  Registry->CloseKey();
  delete Registry ;
}
//-----

```

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    TRegistry* Registry = new TRegistry ;
    try
    {
        if ( Registry->OpenKey( g_sRegistryKeyName, false ) )
        {
            Registry->ReadBinaryData( g_dwScreenPosition, &screenPosition,
                sizeof(screenPosition) );
            Left = screenPosition.nLeft ;
            Top = screenPosition.nTop ;
        }
    }
    catch ( ... ) { }

    Registry->CloseKey();
    delete Registry ;
}
//-----

void __fastcall TForm1::ResetButtonClick(TObject *Sender)
{
    uTotalBytesReceived = 0 ;
    BytesReceivedLabel->Caption= "0" ;
    TotalBytesReceivedLabel->Caption = "0" ;
}
//-----
```

```

file: main.h
//-----
#ifndef mainH
#define mainH
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ScktComp.hpp>
#include "e:/jfisher/common/include/buffer.hpp"
//-----
class TForm1 : public TForm
{
__published: // IDE-managed Components
    TServerSocket *ServerSocket1;
    TCheckBox *ConnectedCheckBox;
    TLabel *BytesReceivedLabel;
    TLabel *Label1;
    TLabel *Label2;
    TLabel *TotalBytesReceivedLabel;
    TButton *ResetButton;
    void __fastcall ServerSocket1ClientConnect(TObject *Sender,
        TCustomWinSocket *Socket);
    void __fastcall ServerSocket1ClientDisconnect(TObject *Sender,
        TCustomWinSocket *Socket);
    void __fastcall ServerSocket1ClientRead(TObject *Sender,
        TCustomWinSocket *Socket);
    void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall ResetButtonClick(TObject *Sender);
private: // User declarations
    struct tagScreenPosition {
        int nLeft, nTop ;
    } screenPosition ;
    unsigned uTotalBytesReceived ;
    Buffer buffer ;
public: // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//-----
extern PACKAGE TForm1 *Form1;
//-----
#endif

```

```
file: server.cpp
//-----
#include <vcl.h>
#pragma hdrstop
USERES("server.res");
USEFORM("main.cpp", Form1);
//-----
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//-----
```

## Appendix H: Pseudocode for socket()

```

/* someheader.h */

typedef struct tagX {
    int domain ;
    int type ;
    int protocol ;
    int result ;
} X ;

/* anaconda */
int _socket( )
{
    struct X x_anaconda ;
    ....
}

int g_a_socket_mutex = 0 ;

/* host socket API */

int socket ( )
{
    int result = -1 ;

    // allocate a block of memory that's large enough to hold
    // a type X object, and lock that memory down so that it
    // cannot be paged out...
    struct X* px = locked_page ;

    //
    // code that initializes 'x' here...
    //

    // wait for anaconda to complete the previous socket call
    // [add: timeout code; code that reduces CPU utilization]
    while ( g_a_socket_mutex );

    // pass anaconda the address of the locked down memory block
    mapmem( px --> px_anaconda );

    // set the semaphore
    g_a_socket_mutex = SYS_SOCKET ;

    // wait for anaconda to complete the socket call
    // [add: timeout code; code that reduces CPU utilization]
    while ( g_a_socket_mutex );

    // code that analyzes the results here...
    if ( px->result )
    {
    }
    else
    {
    }

    return result ;
}

```