TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

MIDDLEWARE SOLUTIONS FOR NETWORK INTERFACE CARDS IN BARE MACHINE COMPUTING

by

Faris Al Mansour

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree

Doctor of Science

Department of Computer & Information Sciences

Towson University

Towson, Maryland 21252

May, 2018

TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by Faris Almansour, entitled "MIDDLEWARE SOLUTIONS FOR NETWORK INTERFACE CARDS IN BARE MACHINE COMPUTING" has been approved by the dissertation committee as satisfactorily completing the dissertation requirements for the degree Doctor of Science in Information Technology.

Ramesh K. Karne — 4-27-2018
Chair, Dissertation Committee, Dr. Ramesh K. Karne — Date

4/27/18
Committee member, Dr. Alexander L. Wijesinha — Date

4/27/2018
Committee member, Dr. Chao Lu — Date

Subrata Acharya — 04/27/18
Committee member, Dr. Subrata Acharya — Date

5-9-18
Dean of Graduate Studies — Date

Acknowledgements

I would like to express my appreciation to all those who have supported my efforts to complete this dissertation. I am greatly appreciative of my research committee: Dr. Ramesh K. Karne (chair), Dr. Alexander Wijesinha, Dr. Chao Lu and Dr. Subrata Acharya for supporting this research. I am especially thankful to Dr. Karne and Dr. Wijesinha for all the long hours in the lab especially on many weekends, and for their support and valuable advice.

To my beloved wife Wafa and my five children; Abdullah, Mohammed, Lamar, Salman, and Mansour, you've been there for me all the time-without you all I wouldn't achieve my dream. I would also like to thank my Father in law Mohammed A Zammal for his counsel, sympathetic ear and continuous support. He was always there for me. I offer my heartfelt thanks to my beloved parents Wefqah and Abdullah, my brothers, Mohammed, Mana, Husain, Ali, Zamil, and all my sisters, whose words have encouraged me to complete my doctoral study.

To my colleagues in Towson University ;Majed Aljazaeri, Mohammed Alyami, Hamdan Alabsi and Rasha Almajed I say thank you for all the help and intimacy we shared during the years of doctoral research. I also would like to thank Dr. Sidd Kaza, Chair of the Department of Computer and Information Sciences, and Dr. Janet V. Delany Dean of Graduate Studies at Towson University, for facilitating this work. I am also grateful to the late Frank Anger (National Science Foundation) for his support of the Application Oriented Object Architecture (AOA), which evolved into Bare Machine Computing research and consequently made this dissertation possible.

ABSTRACT

MIDDLEWARE SOLUTIONS FOR NETWORK INTERFACE CARDS IN BARE MACHINE COMPUTING

Faris Abdullah Al Mansour

Bare machine computing (BMC) applications, which run without the support of an operating system (OS) or kernel, are based on the BMC paradigm and programming methodology. The necessary hardware interfaces and network interface card drivers are integrated with the BMC application. This dissertation deals with the design of novel OS-independent device drivers for Ethernet network interface controllers/cards (NICs) used with BMC applications.

We first develop OS-independent middleware that allows different Ethernet NIC drivers to be used with BMC applications. Although network interface cards evolve over time with new models and enhanced functionality, it is observed that one could homogenize the design of network interface cards and develop a generic architecture for NICs. We then implement Ethernet bonding on a BMC Web server using device drivers for dual NICs, where both NICs can send packets but only one NIC can receive them. We finally implement Ethernet device drivers for a BMC Web server that can migrate with minimal changes for evolving IBM compatible PCs with Intel NICs. The BMC NIC driver is compared with OS-based drivers to identify design differences and tradeoffs in NIC driver complexity versus their functionality.

Currently, device drivers vary depending on platform, vendor and CPU architecture. Our work shows how to implement Ethernet device drivers that are

independent of any platform, and a simple API for applications, where applications directly communicate to its underlying hardware. It provides an in-depth understanding of BMC Ethernet device drivers and serves as a foundation to construct BMC and OS-based device drivers that can be made upward compatible with minimal changes.

Table of Contents

List of tables ix
List of figures x
1. INTRODUCTION 1
2. Related Work 5
2.1 Bare Machine Computing Background 5
2.2 BMC Applications 6
2.3 OS-based Drivers 7
2.4 Ethernet Bonding 8
2.5 User Space Drivers and Networking 8
3. Middleware for Network Interface Cards (NICs) in Bare PC Applications 10
3.1 NIC Heterogeneity Problem 10
3.2 System Architecture 11
3.3 MiddleWare Design 12
3.3.1 NIC Internals 12
3.3.2 Differences in the NIC Interfaces 13
3.3.3 Middleware Taxonomy 17
3.3.4 Middleware Class 18
3.4 Implementation and Testing 20
3.5 Generic NIC Architecture 22
4. Ethernet Bonding on A Bare PC Web Server with Dual NICs 24
4.1 Dual NIC Bare PC Web Server 24
4.1.1 Architecture 24
4.1.2 Design 26
4.1.3 Implementation 28
4.2 Performance Measurements 30
4.2.1 Load Balancing Strategy 31
4.2.2 Two Clients 32
4.2.3 Three Clients 33
5. Upward Compatible Ethernet Device Driver for Bare PC Applications 35
5.1 NIC Driver 35
5.1.1 Architecture 35
5.1.2 Interfaces 36
5.1.3 Design 39
5.1.4 Implementation 41
5.2 Upward Cmpatible Ethernet Device Driver 42
5.2.1 Dell Optiplex 960 44
5.2.2 Dell Optiplex 9010 and HP Elite Book 8460P 46
5.3 OS Based NIC Drivers 47
5.4 Observation and Findings 48
6. Significant Contributions 50
7. Conclusion 51

8. Appendix 53
8.1 Middleware Implementation 53
8.2 Ethernet Bonding Implementation 63
8.3 Upward Compatibility of NIC Driver Implementation 67
8.3.1 Obtaining device address for NIC 67
8.3.2 Original functions in Optiplex 260 (Intel 82540EM) 69
8.3.3 Upgrade to Optiplex 960 72
8.3.4 Upgrade to Optiplex 9010 74
8.3.5 Porting to HP Elite Book 75
9. References 76
10. Curriculum Vitae 80

LIST OF TABLES

Table 1. Differences in Transmit Descriptor 16

Table 2.Differences in Receive Descriptor 17

Table 3. Differences in NIC API 21

Table 4. Generic NIC Architecture 23

Table 5. Interfaces 37

Table 6. Transmit Control Register 40

Table 7. Receive Control Register 40

Table 8 .Simillar Functions in Both NICs 54

Table 9. Vendor and Device IDs 67

LIST OF FIGURES

Figure 1. TCP Level Code Calls 14

Figure 2. System Architecture..... 15

Figure 3. Circular List Data Structures 18

Figure 4. Descriptor Details 19

Figure 5. Middleware Taxonomy..... 20

Figure 6. Integrated HTTP/TCP Protocol 25

Figure 7. Single NIC Bare Web Server Flow 27

Figure 8. Dual NIC Web Server Architecture..... 29

Figure 9. Task State Transition Diagram 30

Figure 10. Each Client has a Dedicated Send NIC 31

Figure 11. Connection Time (Two Clients) 32

Figure 12. Response Time (Two Clients) 33

Figure 13. Connection Time (Three Clients) 34

Figure 14. Response Time (Three Clients) 34

Figure 15. BMC based Architecture 36

Figure 16. OS based Architecture 36

Figure 17. Sequence of Operations 38

Figure 18. SetTDTail 39

Figure 19. IPInsertPacket 42

Figure 20. Descriptor Ring Data Structure 43

Figure 21. Transfer Descriptor (TDL) 45

Figure 22. Receive Descriptor (RDL) 45

Figure 23. Classification of Interfaces 48

1. INTRODUCTION

Bare PC applications are written as a single monolithic executable and do not depend on any conventional operating system (OS) or lean kernel. Such applications eliminate OS overhead and OS-related security vulnerabilities. They currently run on any Intel Architecture 32-bit (IA-32) compatible PC and do not use a hard disk. Bare PC applications are not the same as applications running on an embedded system or in a virtual machine. Many bare PC applications [1, 2, 3, 4, 5] have been developed earlier. Bare PC applications are based on the bare machine computing (BMC) paradigm, which has some features of the minimal systems discussed in [6]. The methodology for designing and implementing BMC applications is described in [7]. There are many approaches that reduce OS overhead including Exokernel [8], Tiny-OS [9], Palacio and Kitten [10], IOLite [11], and OS-Kit [12]. These approaches retain some form of an OS or minimal kernel.

In a bare PC application, an OS-independent network interface controller/card (NIC) device driver is integrated with the application code. It is thus useful to develop OS-independent middleware that will allow an existing bare PC application integrated with a bare PC driver for a given NIC to work with a different NIC and its bare PC driver. The problem is challenging because the bare PC driver for the second NIC is itself integrated with a different bare PC application. The first part of this dissertation describes the design and implementation of middleware that enables a bare PC Web server with an Intel NIC driver and a bare PC Webmail server with a 3COM NIC driver to use both cards with their respective drivers. The middleware design also identifies common features of NICs that will be useful for designing an OS-independent generic NIC architecture for bare and non-bare applications in the future.

Ethernet NIC bonding [15] combines network interfaces on a device. It is widely used in practice, for a variety of reasons including load balancing and reliability. Ethernet bonding on Web servers typically requires some form of OS or kernel support. The second part of this dissertation deals with the implementation of a novel form of Ethernet bonding using dual NICs with a bare PC Web server application. In the bare PC Ethernet bonding implementation, both NICs can send but only one NIC can receive. The approach can be extended to work with more than two NICs and a variety of send/receive NIC configurations. In future, NIC bonding can be used with bare PC Web servers running on multi-core architectures and with other high performance or high security BMC applications.

To further illustrate the dual NIC split send-receive approach in a BMC application, consider the intertwined HTTP/TCP protocols as implemented in a bare PC Web server with a single NIC [16]. Protocol intertwining is inherent in BMC applications due to their underlying task design discussed later. While a bare Web server can use TLS for security, in this dissertation (for reasons of simplicity), we assume that HTTP connections are to port 80. Fig. 6 shows the intertwined protocol messages exchanged by client and server due to a single request from a client, which can be any conventional OS-based Web browser. A bare PC NIC driver consists of send and receive data structures as shown in Fig. 3. In the driver, the send path using the transcript descriptor list (TDL) and the receive path using the receive descriptor list (RDL) are separate and parallel paths in the hardware. Send and receive controls such as enable and disable, and associated configuration parameters are also different in the driver. In essence, in the NIC and also in the driver, send and receive paths can be treated as two separate entities.

In the bare Web server single NIC implementation shown in Fig. 7, send and receive paths are also different. When a packet is received in RDL, its Ethernet header is removed and IP processing is done on the packet as usual. After the IP header is removed, TCP and HTTP processing are done in an intertwined manner on the packet. For a given client's IP address and port, a unique entry for the request is created in the TCP table. This unique entry is kept in the table until the completion of the client's request.

A client HTTP Get or Post requires a server response, which consists of the response data prefixed with an HTTP header. The response is inserted into one or more packets with TCP, IP and Ethernet headers. These packets are inserted into the TDL. The Ethernet NIC sends packets from the TDL and receives packets from the RDL. All of these steps are part of the Web server application since there is no OS or kernel in a BMC system.

The monolithic executable, unique tasking (discussed later), and integration of all protocol code with the application facilitates the separation of send and receive paths in the design of the bare PC Web server. In fact, send and receive paths are naturally disjoint in a bare Web server at the task, NIC, driver, protocol, and application levels. This enables Ethernet bonding to be implemented without any OS or kernel support.

Writing a device driver poses daunting challenges as it requires multi-disciplinary skills including an understanding of the operating system (OS), underlying CPU architecture, hardware interfaces and the software specification for a driver. As the platform, hardware, software and technology change rapidly, no consideration has been given to preserve upward compatibility for device drivers. In particular, Ethernet drivers evolved over the years in a heterogeneous manner without a design focus on upward compatibility. Some efforts in developing platform independent driver interfaces such as the network driver

interface specification (NDIS) [36] have proved to be successful in making driver applications independent of their platforms. However, NDIS hides the heterogeneity by creating middleware between the actual driver and a driver application. It does not eliminate vendor specific drivers and does not address the issues of simplicity and upward compatibility for a given driver. The problem is pushed back into the underlying OS, which handles the middleware and the device driver. Is it possible to eliminate proliferation of device drivers and reduce their complexity? Ethernet driver functionality is universal in nature and its architecture can be unified and standardized to cope with this heterogeneity and obsolescence. However, no such architecture exists at present. The last part of the dissertation will give some insight into designing unified upwardly compatible or migratable Ethernet device drivers with less complexity.

2. RELATED WORK

In this chapter, we discuss research related to our work presented in this dissertation. In addition to previous work on bare machine computing, we consider OS-based drivers, Ethernet bonding, and user space drivers and networking.

2.1 Bare Machine Computing Background

Since the beginning of the computers over 50 years ago, software complexity has been growing rapidly. The massive growth in computing hardware and software has created unmanageable electronic waste [17]. This is partly because new software cannot work with legacy hardware. Software applications, operating systems, tools and gadgets become quickly obsolete in years-sometimes in months. The operating system size also increases dramatically. For example, for Microsoft window XP professional version with SP3, the size of OS is over one GB.

The Bare Machine Computing (BMC) paradigm was invented by Dr. Ramesh Karne at Towson University, which was also referred to earlier as a dispersed operating system computing (DOSC) system. The key concepts of the BMC paradigm are as follows:

1) Computing applications run in a bare machine without Operating System, centralized kernel, or any system software pre-loaded into the machine.
2) The applications can be loaded and carried in a portable device such as flash memory, and run in a bare machine anywhere.

The BMC approach is a novel approach for developing computing applications. Unlike traditional approaches, it is application-centric, and completely differs from conventional computing approaches that are environment and platform-centric. There is little need to upgrade or patch the computing environment often; the focus is on the applications

themselves. Once a computing box is made bare, the expense to protect it will be minimized since it only has memory, CPU, a basic user interface (input/output), and a network interface. All persistent data is either stored in a removable device such as USB flash memory or on the network.

In the BMC paradigm, an Application Object (AO) is a self-contained, self-controllable and self-executable unit [18] so that when an AO is developed, it can be run in any bare hardware such as desktop, laptop, and hand-held, or other electronic device.

The BMC approach makes computing application simpler and more secure. An AO is developed in a single programming language and run a bare machine, so the AO developer only needs to know one computer language and AO domain knowledge. The AO has no particular ownership and is self-contained and self-executed so that it can run in any bare machine. As the AO controls both application and execution aspects, and also avoids all the system and kernel related vulnerabilities by making the device bare, it will be more secure. The BMC paradigm changes the way applications are developed today.

2.2 BMC Applications

Several complex bare applications have been developed in the bare machine computing laboratory at Towson University-most were the outcomes of doctoral research work. They clearly show the feasibility of the BMC paradigm. He [16] developed the first bare PC Web server and demonstrated the feasibility of building complex software that runs on a bare PC with thousands of threads and outperforms other compatible commercial Web servers. Khaksari [19] developed the first VoIP soft-phone that runs on a bare PC and provides secure communication on an end-to-end basis. Alexander [20] built a SIP server and a bare SIP user agent to demonstrate the feasibility of running high performance SIP servers with

secure communication using the SRTP protocol. George H. Ford built the first Email server that runs on a bare PC and provides compatible performance to related commercial email servers [3, 21]. Emdadi [22] implemented the complex TLS protocol for a bare Web server. Yasinovsky [23] implemented the IPv6 protocol for a bare PC VoIP softphone client. Rawal [4] developed a unique split protocol concept and applied it to Web servers that run on a bare PC. He also developed mini-cluster configurations for Web servers based on the split protocol concept that offer high performance [24] and run on a bare PC. Appiah-Kubi developed a secure Webmail server using TLS [25]. These previous doctoral research projects made possible the discovery of many novel characteristics that are unique to BMC and that would be applicable to future computing applications that run on bare devices.

2.3 OS-based Drivers

OS-based device drivers have been the focus of many studies. In [26], static analysis tools are used to analyze the code of Linux device drivers. Furthermore, the interaction between drivers, the OS, and the hardware is examined. In [27], hardware abstraction and APIs for hardware and software interfaces are used as a basis for developing device drivers. A device object model is also presented that is used to separate OS-dependent and device-dependent driver components. In [28], a technique for reverse engineering device driver binaries is given, and Windows drivers are reverse engineered and ported to other OSs. In [29], an approach for driver reuse is proposed, where a virtual machine is used to run a driver with its original OS. As these examples show, earlier work on device drivers has been done using OS-based drivers.

2.4 Ethernet Bonding

The growing popularity of virtualized OSs and containers have resulted in the recent introduction of several so-called minimalist OSs [30], including the Docker platform, Ubuntu Core, Core OS, and Atomic Host. Minimalist OSs may focus on protecting against failures and attacks, target embedded systems, IoT and the cloud, or be designed for speed. There are also many lightweight Linux distributions that run on older x86 hardware. One of the earliest attempts to provide a small kernel with minimal functionality for applications is Exokernel [8]. Subsequently, lightweight OSs supporting high-performance applications were introduced as in [10]. In contrast, BMC systems are non-virtual and non-embedded. They enable applications to run without any OS or kernel support by providing direct interfaces to the hardware [31].

Ethernet bonding on Linux systems [15] has numerous driver options and allows bonding configurations to be based on requirements of high availability or maximum throughput for single and multiple switches. The performance of Ethernet bonding is studied in [32], and it is found that active backup mode has low switch-over time in case of failure, while round robin mode with dual NICs almost doubles the throughput achieved without bonding. In Oracle VM, network bonding is designed primarily for redundancy although increased throughput requirements are also supported [33].

2.5 User Space Drivers and Networking

User space drivers and user space networking in conventional systems move packet processing out of the kernel. In [37], upstream and downstream interfaces exported by the Linux kernel for user space drivers are discussed. In [38], performance aspects of user space IO device drivers in real time systems are the focus. In [39], performance, security

and architecture related issues due to user space networking are highlighted. In [40], user space networking is viewed as a means to improve NFV performance. In [41], some disadvantages of user space networking are given. In [42], the design and implementation of a system for running untrusted Linux device drivers in user space are described. Although there are some similarities, a significant difference is that user space approaches still require kernel support unlike bare PC drivers and applications that communicate directly to the hardware.

3. MIDDLEWARE FOR NETWORK INTERFACE CARDS (NICS) IN BARE PC APPLICATIONS

In this chapter, we discuss the middleware solution for NICs and their drivers in bare machine computing. We first consider NIC heterogeneity.

3.1 NIC Heterogeneity Problem

In general, heterogeneity [13] exists in hardware and software systems and applications, as well as in NICs and their drivers. There are approaches to homogenize software applications such as [14], but they are based on a conventional OS environment and are thus not suited for bare PC applications. NIC heterogeneity comes in many forms. Some older external NICs have local buffers that need to be addressed by the device driver as first-in-first-out buffers. Newer NICs support higher data rates and larger buffers, and are integrated with the motherboard. Different Ethernet NICs, even those built by the same vendor, often have differences that make their drivers incompatible. Ideally, bare PC applications should use an OS-independent device driver integrated with the application for any NIC installed in the bare machine. Such a driver is difficult to develop since there are a variety of NICs for different network technologies.

Instead, we build middleware to address the driver problem for the case of two specific Ethernet NICs and their bare PC device drivers. The first is the bare PC device driver for the external 3COM 905CX NIC [12]. It is used with some bare PC applications e.g., [1]. This NIC is designed to use buffers in main memory instead of internal buffers, and the driver works with circular buffer data structures. The second is the bare PC device driver for the gigabit Ethernet Intel 82540EM NIC [34]. It is used with some other bare PC

applications e.g., [4]. These two drivers are not compatible, and the Intel driver is not totally compatible with drivers for other gigabit Ethernet Intel NICs.

To understand the middleware architecture and design, it is necessary to note that bare PC applications include the necessary network protocols and directly use the Ethernet interface without going through intermediate layers. This optimization avoids the need for buffer copying and also reduces procedure call and other overhead. An API is provided for bare PC applications to access the Ethernet interface as there in no centralized OS or kernel in the system.

For example, the code segment in Fig. 1 illustrates how the TCP level code in a bare PC Web server application using an Intel gigabit NIC interacts with the Ethernet interface. Here, calls are made by the TCP code in the application to methods that format the TCP, IP, and Ethernet headers before sending the packet. These calls are made within a single thread of execution avoiding the need for switching between layers. The Ethernet buffers are accessed directly and the send buffer address is formed in step 1. Formatting headers and aligning the data is done in steps 2 through 7. These seven steps enable the bare PC Web server application to send a packet through the Ethernet interface via the NIC driver without the overhead of invoking an OS or kernel. The OS-independent middleware is designed to allow the application to use the desired Ethernet interface when sending packets.

3.2 System Architecture

The system architecture shown in Fig. 2 enables the 3COM and Intel Web servers to access either the 3COM or the Intel NIC drivers using the middleware. There are four possible configurations shown as paths through the middleware, and labeled 0-3. These

configurations/paths correspond respectively to the 3COM Web server accessing the 3COM driver; the Intel Web server accessing the Intel driver; the 3COM Web server accessing the Intel driver; and the Intel Web server accessing the 3COM driver. The “0” and “1” paths are the same as each server accessing its original driver except that access is now through the middleware. The “2” and “3” paths require middleware functions that allow the respective server to access the second driver.

Ideally, the middleware should be designed so that the existing driver and server code remain unchanged. However, it was necessary to make a few changes to the existing server code as explained in the next section. To choose the driver and NIC to be used by a server (i.e., to select one of the four paths), a variable called “NIC_FLAG” taking values 0-3 is used. This flag can be set at any time, which enables a NIC to be chosen dynamically.

3.3 MiddleWare Design

The bare PC middleware design depends on NIC internals and NIC interfaces. In this section, we provide details of NIC internals, describe the differences in the NIC interfaces, give a taxonomy for NIC middleware, and discuss the middleware class.

3.3.1 NIC Internals

The basic architectural element used for communication between a NIC and its driver is a circular list data structure. There are two circular list data structures, one for sending packets and one for receiving packets via the NIC. Fig. 3 shows these data structures. Each list consists of **“descriptor”** elements that store information needed to communicate with the NIC. A bare PC application can directly access these circular lists. Each list has pointers to insert and remove items: SIN and SOUT for the send list, and RIN and ROUT for the

receive list. There are also two head pointers for these lists: TPTR for the send list, and RPTR for the receive list.

These abbreviated names are used in this dissertation for convenience; the respective 3COM and Intel specifications [12, 34] use different names. The size of a circular list depends on the queue size required for sending and receiving packets. The sizes used for the 3COM and Intel NICs were 4096 and 20,000 respectively. These sizes are sufficient to buffer packets based on the data rates for the NICs (100 Mbps for 3COM and 1 Gbps for Intel). The descriptors for each NIC differ in size and format. The sizes are 32 bytes for 3COM and 16 bytes for Intel. Fig. 4 shows the names and sizes of descriptor fields. As the middleware is working at the driver interface level, it only needs details of some fields e.g., memory address, and previous and next link pointers. The differences between 3COM and Intel NICs for the transmit (send) and receive descriptors are shown in Table 1 and Table 2 respectively. These differences are relevant to the generic NIC architecture discussed later.

3.3.2 Differences in the NIC Interfaces

Two different bare PC server applications used different NICs in this study. A Web mail server is used with the 3COM NIC and a Web server is used with the Intel NIC. As already discussed and illustrated using the code segment in Fig. 1, the NIC interfaces are part of, and tightly integrated with, the respective bare PC application

```
//1.GET Transmit Buffer Pointer
x = EO.getTDLPointer() + EO.getSendInPtr()*16;
p1= (long*)x;
send_buffer = (char*)*p1;
//2.Add Ethernet and IP Header Lengths
send_buffer = send _buffer+14+20;
//3.Format TCP Packet
TCPPack_size = FormatTCPPacket(send_buffer,
tcp->IP,tcb->PORT,tcb->tempflags,tcb->RCVWND,
tcb->tempSeqNum,tcb->RCVNXT,sendbuffer,
TCPSegSize,tcb->tempIndex,currenttask);
//4.Adjust for IP Header
send_buffer=send_buffer-20;
//5.Format IP Packet
retcode=ip.FormatIPPacket(send_buffer,TCPPack_size,tcb
->IP,tcb_>destmac,TCP_Protocol,currenttask);
//6.Adjust for Ethernet Header
send_buffer=send_buffer-14;
//7.Format Ethernet Header and Send Data
retcode = EO.FormatEthPacketN(send_buffer,
TCPPack_size+20,IP_TYPE,tcb->destmac,Inptr,
tcb->count1,tcb->sendtype,currenttask);
```

Figure 1. TCP Level Code Calls

The integration of NIC interfaces in the above applications with respect to code is different. For example, the Init() functions in the two applications are specific to each application and perform different tasks. It is necessary to analyze the application code to identify differences in the NIC interfaces when designing a common interface in the middleware. Table 3 summarizes the NIC differences.

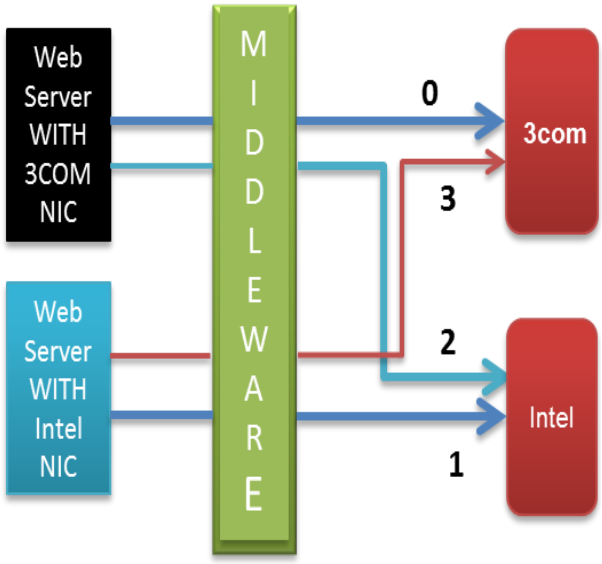

Figure 2. System Architecture.

In some cases, these differences result in a few changes to the application itself in order to work with the middleware. This is because the applications may use not only NIC interfaces (functions), but also global variables (static in a class) to compute the values of other variables or data. For example, an application uses SIN and SOUT pointers to compute the descriptor address to access the data structure and modify its controls. When middleware is used, these pointers are not directly addressable by the application as the application uses the middleware while the actual NIC driver is only accessible to the middleware (Fig. 2). To handle this issue, getSIN(), setSIN(), getSOUT(), and setOUT() functions was written to replace the SIN and SOUT variables. This means in general that all NIC variable instances must be replaced with equivalent functions to access the variables. As a result, the application code has to be modified.

In another case, global variables are used in an expression to derive some data. For example, TDLPointer may occur in some expression E that is used to compute a value X. Then a function getTDLPointer() is defined in the middleware that returns the value of

TDLPointer, and this function is used instead of TDLPointer in the expression E to form a new expression newE. Also, a new function getXNew() is defined in the middleware that computes the value X using the expression newE instead of E.

Fields	3COM	INTEL
NXT DPD PTR	4 Bytes (Address)	N/A
FRAME START	4 Byte	N/A
DATA BUFF ADDR	4 Bytes 32 bit address	Buffer Address 8 Bytes (64 bit Address)
DATA BUFF LENGTH	4 bytes	Length 2 byte
PACKET ID	4 bytes	N/A
PREV DPD PTR	4 Bytes	N/A
TX STATUS(*)	4 Byte TX Status has 1 Byte includes, TX Under run, max Collision, Tx complete, interrupt requested, TxStatusOverflow TxReclaimError TxJabber.	4 bits includes, Transmit Under run, Late Collision, Excess Collisions and Descriptor Done
SEND FLAG	4 Bytes	N/A
CSO	N/A	1Byte
CSS	N/A	1Byte
CMD	N/A	1Byte has the main 3 bits; End of Packet Insert Frame Check Sequence Report Status

Table 1. Differences in Transmit Descriptor

Again, this requires changes to the applications. However, all changes made to the applications were small and limited to only a few files.

Fields	3COM	Intel
Next UPD Ptr	4 Byte	N/A
FRAME START HDR	4 Bytes	N/A
DATA BUFF ADDRESS	4 Bytes 32 bit address	Buffer Address 8 Bytes (64 bit Address)
DATA BUFF LENGTH	4 Byte Bit 31 = 1 (last frame))	2 Byte length
PACKET ID	4 bytes	N/A
NOT USED	4 Byte	N/A
NOT USED	4 Byte	N/A
NOT USED	4Byte	N/A
Status	N/A	1 byte -Passed in-exact filter,- IP Checksum Calculated on Packet,- TCP Checksum Calculated on Packet,- Reserved,- Packet is 802.1Q (matched VET),- Ignore Checksum Indication, - End of Packet,- Descriptor Done
Errors	N/A	1Byte includes,-RX Data Error,- IP Checksum Error,TCP/UDP Checksum Error,- Carrier Extension Error,- Reserved,-Sequence Error ,Symbol Error -CRC Error or Alignment Error

Table 2.Differences in Receive Descriptor

3.3.3 Middleware Taxonomy

Fig. 5 shows a middleware taxonomy for the two NICs chosen in this dissertation. This taxonomy is very similar to other middleware solutions that homogenize applications. It includes functions, variables and constants, which may have conflicts. Conflicts can be further classified into three types of attributes: same, different and not applicable. Unlike in OS-based middleware approaches, these conflicts must be resolved with an OS-independent middleware design that works with bare PC applications.

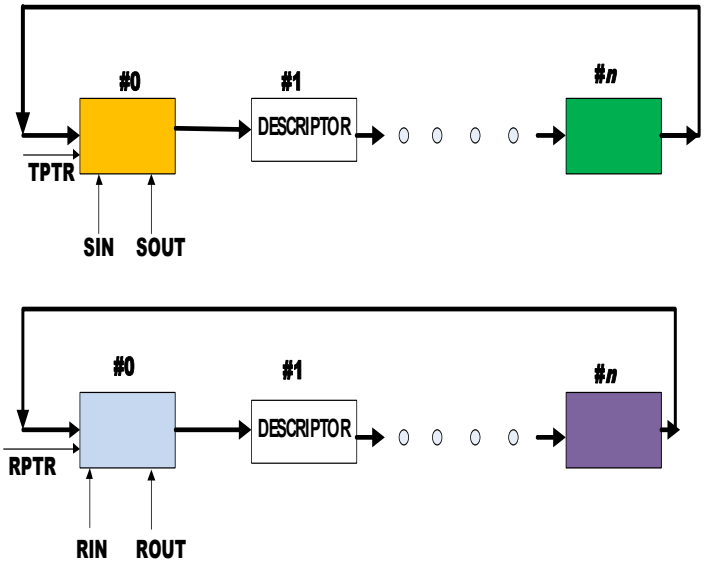

Figure 3. Circular List Data Structures

3.3.4 Middleware Class

The applications instantiate the NIC interfaces using the middleware object, which is called EtherObj (EO). The EO object in turn uses the 3COM or Intel driver objects as shown in Fig. 2. Since the middleware object uses the same name EtherObj as a NIC object for its applications, the existing Ethernet objects were renamed as EthernetObjIntel and EtherObj3Com to distinguish the drivers. The new middleware object EO is included in the system as part of the application. This requires some changes to the bare PC compile and link commands. For the two NICs and two server applications considered here, there are four new system configurations as shown in Fig. 2. The design can be extended in the future enabling one application to work with both NICs at the same time.

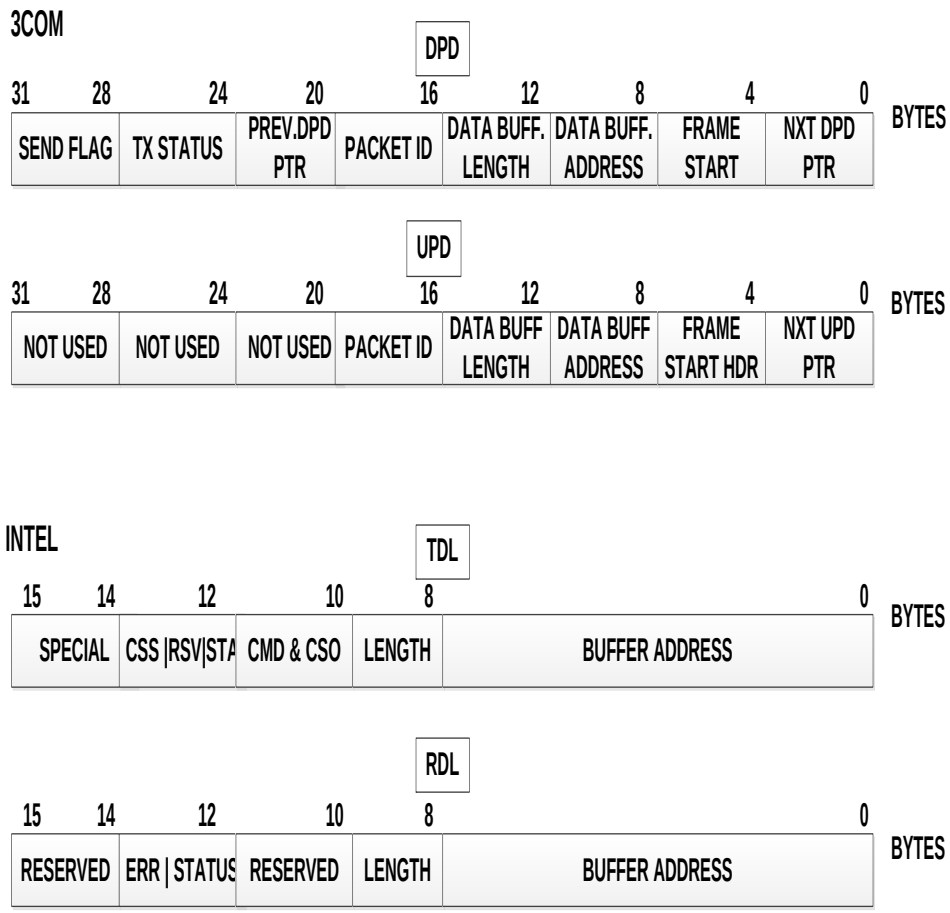

Figure 4. Descriptor Details

The middleware class consists of a header and a class file. The header file consists of all function prototypes needed for the middleware, and some constants needed for initializing data structures. There was a separate memory map for NICs in the existing server applications. In the middleware design, a new memory area was created for data structures. This allows an application to use the Intel memory map for the data structures when it goes to the Intel NIC instead of the 3COM NIC, and vice versa. The memory map defines the DPD, UPD, TDL, and RDL addresses, and their sizes.

These entities, which were shown in Fig. 4, are different for the 3COM and Intel server applications and needed to be redefined. The implementation file for this class consists of

functions that call the 3COM and Intel driver methods and adheres to the middleware taxonomy in Fig. 5.

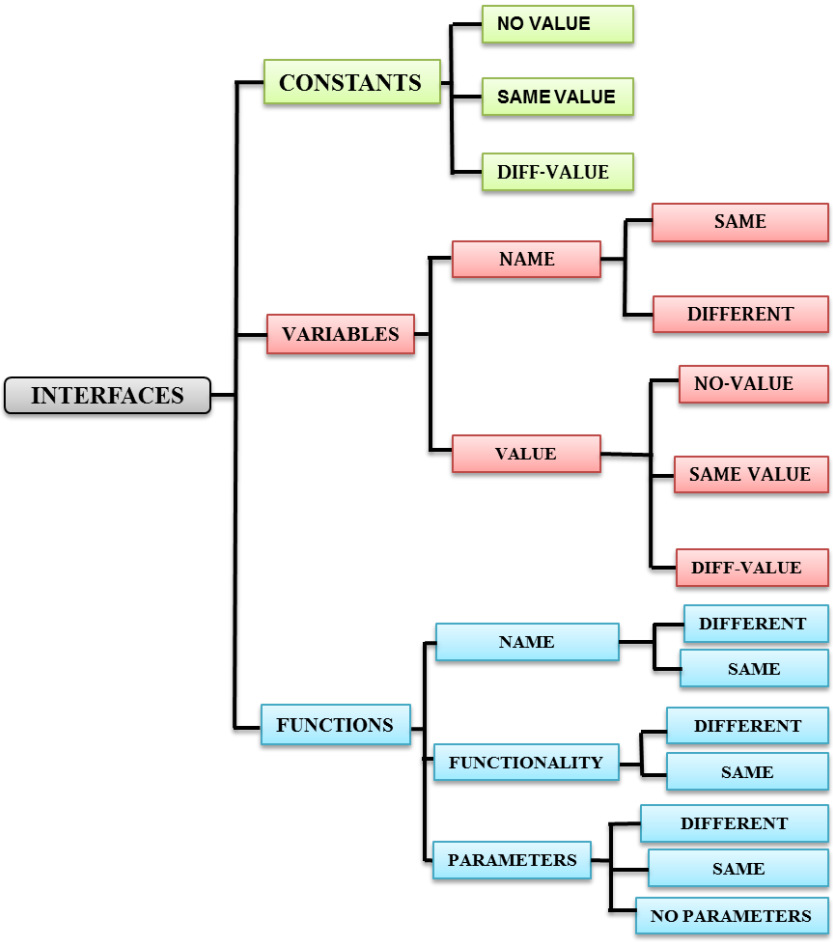

Figure 5. Middleware Taxonomy

3.4 Implementation and Testing

The middleware code is written in C++, while the existing applications were written in C/C++. The middleware was tested on a Dell OptiPlex 960 desktop with 3COM905X and Intel 82540EM NICs. There is no hard disk in the system, and the application (including boot and load) were resident on a USB flash drive. The bare PC applications used were a Webmail server integrated with a 3COM NIC driver, and a Web server integrated with an Intel NIC driver. The NIC middleware was tested by accessing an HTML file in the Web

browser. The middleware was also tested with a file transfer application that downloads files to the Web server or the Webmail server from a Windows client. All four configurations shown in Fig. 2 were tested for functionality and validated for correct operation. The tests demonstrate the feasibility of bare PC NIC middleware and its potential for use with other bare PC applications.

Attributes	3COM	INTEL
Initialization	Init()	Inittest()
Create DS	Createdatastrcucture()	N/A
Transmit	enableTransmit()	TEnable()
Receive	enableReceive()	REnable()
MAC Address	getMAC()	GemtMACEPROM()
Full	TDLFull()	DPDFull()
Increment Send-OutPtr	IncSendOutPtr()	IncSendPtr()
Transmit Ptr	DownListPointer()	TDLPointer()
Receive Ptr	Uplistpointer()	RDLPointer()
Cold Reset	N/A	Coldreset()
Hot Reset	N/A	Hot reset
Close	Close()	N/A
Transmit Data Ptr	DownListDataPointer()	TDLDataPointer()
Receive Data Ptr	N/A	RDLDataPointer()
Get Status Received Packet	readPacket()	N/A

Table 3. Differences in NIC API

3.5 Generic NIC Architecture

Due to different vendors and different NIC architectures, there are many differences in NIC interfaces. It is possible in principle to design a common NIC API for initialization, transmit and receive controls, reset, managing circular list pointers, setting descriptor values, checking circular list limits, and obtaining the MAC address.

The key elements used to communicate with a NIC are its data structure and controls. Tables 1 and 2 give details of the descriptor fields and their contents for transmit and receive operations. The device driver for a NIC manages these descriptors and ensures that they are used correctly. The data buffer and length fields in the transmit descriptor (Table 1) are required in both NICs. The transmit status field occupies 4 bytes in 3COM and 4 bits in Intel. Frame start, send flag, and previous pointers are not available in Intel; and the fields CSO, CSS and CMD are not applicable to 3COM. Similarly, in the receive descriptor (Table 2), many fields are not used or reserved. Also, the status and error fields are not applicable to 3COM. In effect, each NIC vendor designs the descriptor fields in a different way resulting in heterogeneity among NIC drivers. The differences between the NIC APIs are shown in Table 3. However, we found that the overall functionality of these NICs is the same at the application level. This functionality is split among the architecture, design and implementation in various ways.

It may be possible to unify descriptors and develop a generic architecture for NICs making them compatible at the data structure and descriptor level. A generic NIC architecture for transmit and receive descriptors is shown in Table 4. In this architecture, a uniform 32-byte descriptor is used for transmit and receive, and includes common functionality such as descriptor command, status, data pointer, and data length. A generic

NIC architecture can be used as a first step towards defining a common interface for NICs, which will help to reduce the differences between them.

Fields	Transmit	Receive
Descriptor Command	4 Byte	4 Byte
Status	4 Byte	4 Byte
Data Buffer Pointer	4 Byte	4 Byte
Data Buffer Length	4 Byte	4 Byte
Enable / Disable	4 Byte	4 Byte
Packet ID	4 Byte	4 Byte
Next DPD Ptr	4 Bytes	4 Bytes
Next UPD Ptr	4 Byte	4 Byte

Table 4. Generic NIC Architecture

4. ETHERNET BONDING ON A BARE PC WEB SERVER WITH DUAL NICS

In this chapter, we consider Ethernet bonding for NICs in bare machine computing applications. Our implementation uses dual NICs in the bare PC Web server.

4.1 Dual NIC Bare PC Web Server

This section describes the architecture, design and implementation of the dual NIC bare Web server. We also provide details of the tasks used by the Web server.

4.1.1 Architecture

As noted earlier, due to the integration of the TCP and HTTP protocols in the BMC Web server (Fig. 6), it is natural to split send and receive logic in a BMC application from the NIC hardware level to the application level. So we modified the single NIC bare PC Web server design shown in Fig. 7 so it can use dual NIC Ethernet bonding by splitting the send and receive paths. Fig. 8 shows the bare PC Web server architecture for Ethernet bonding with dual NICs. Many clients can connect to the Web server where each client request is identified as usual by the unique IP address and port number combination. The two NICs are associated with respective IP addresses IP1 and IP2, and MAC addresses MAC1 and MAC2. The server uses only one MAC address to receive packets, which is the server's MAC (MAC1) for the receive NIC (R-NIC). All packets from the clients are thus received at the server on MAC1. ARP broadcasts are used to ensure that the default gateway or any local clients only send packets to the server using MAC1. No switch configuration is needed. The server can use either NIC for sending data based on load balancing requirements.

When a BMC system is booted, it goes to the MAIN task, which runs continually whenever no other task is running. This is different from the way an OS/kernel-based

system works: in a bare PC, tasks are created and used as needed by a given BMC application suite. When a packet arrives, the RCV task is run to read a packet from the RDL and it continues running while Ethernet, IP and TCP processing is done on the packet until the state is updated in the TCP table. When an HTTP Get or Post command is received, a unique HTTP task is created to process the client request. A connection may be alive for many GET or POST requests. However, a unique HTTP task handles a given request.

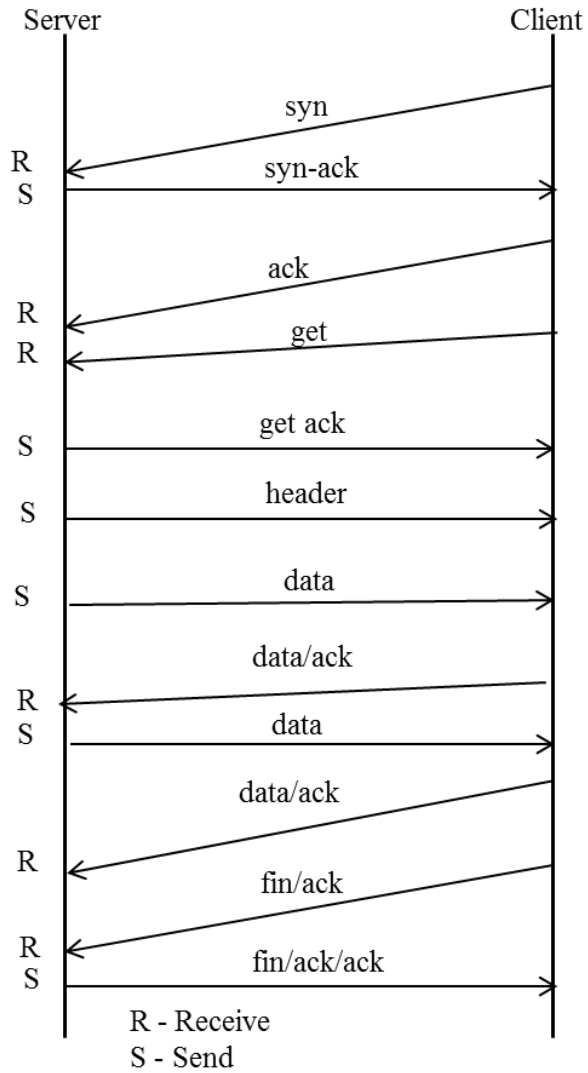

Figure 6. Integrated HTTP/TCP Protocol

The relation between the MAIN, RCV and HTTP tasks used in the bare Web server is shown in Fig. 9. The RCV and HTTP tasks process client packets that are received or that need to be sent. In an OS-less BMC system, tasks run to completion and only suspend themselves when waiting for an event. Suspended tasks are resumed when they are ready to run. Notice that the bare server has an inherently parallel design with respect to client requests.

4.1.2 Design

In a dual NIC system, there are many ways to send and receive packets using the NICs. Referring again to Fig. 6, it is seen that the server receives SYN, SYN-ACK-ACK, GET, DATA-ACK and FIN-ACK from the client; and sends SYN-ACK, GET-ACK, Data and FIN-ACK-ACK to the client. The HTTP and TCP protocols can be split based on send and receive interfaces. Also, for HTTP requests, packets received by the server except for GET are small (about 60 bytes), while data packets sent to the client are large. Thus, when dual NICs are used for send and receive separately, the load is not balanced with respect to the two cards.

One approach with dual NICs is to dedicate one NIC for sending and one NIC for receiving. Sent packets use IP2 and MAC2 but the send NIC does not receive any data from a client. Similarly, a receive NIC do not send any data. This simplex connection of dual NICs may have potential security benefits due to isolation of send and receive paths. Such benefits were not investigated in this dissertation.

Another alternative is to receive on one card and send on two cards to load balance the data. When two cards are used for sending, small packets can be sent on one card and large

data packets on the other. The decision of which card to use for sending a given packet can be made based on load balancing requirements.

This dissertation investigated a variety of strategies for load balancing NICs based on the novel bare PC Web server design. As shown in previous studies using OS-based systems, simple load balancing based on the data alone is not sufficient to achieve optimal performance in a dual NIC architecture. It is also known that using two different cards for the same request does not improve performance, as the client has to handle two NICs simultaneously (data level parallelism).

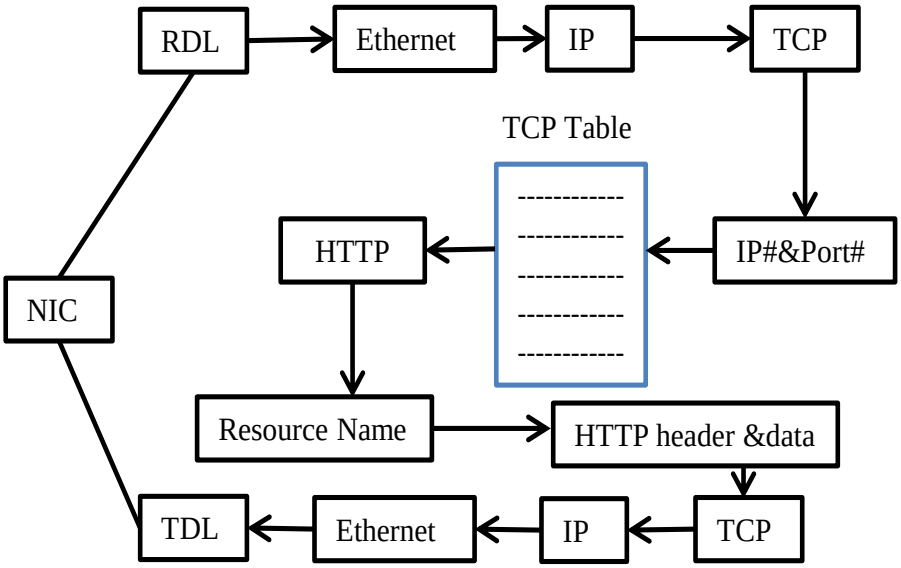

Figure 7. Single NIC Bare Web Server Flow

In order to achieve optimal dual NIC performance using the bare PC Web server design, the parallelism in the client requests was exploited, where each request can be treated as being independent of the others (request level parallelism). Although not investigated in this dissertation, the single core dual NIC design can be extended to multi-core level parallelism using essentially the same approach.

4.1.3 Implementation

Implementation of Ethernet bonding using dual NICs in a bare PC Web server for optimal performance requires that the NIC for sending data to clients be dynamically adapted based on the load. To simplify the design, two identical Ethernet NICs were used. One NIC is enabled for receive and send, and the other NIC is only enabled for sending data. This strategy is simple to implement in the bare PC Web server as the NIC driver code is part of an application program.

In the bare PC Web server design, two instances of the Ethernet object are created to interface with the two NIC drivers. When a SYN packet arrives from a client, the load balancing strategy is implemented based on request level parallelism. In addition, for a given client, a dedicated NIC will send the data and all other packets on the connection. For example, if there are two clients sending requests, client 1 will get data from NIC 1 and client 2 from NIC 2. If a third client comes at the same time, half of its responses are sent using NIC 1 and the other half using NIC 2. If there are n simultaneous requests then n/2 use NIC 1 and remainder use NIC 2. This strategy is easily implemented in the bare PC application itself (without any OS or kernel involvement).

When a TCP SYN packet arrives, a send-id (1 or 2) is set in the TCP table to handle sending data using NIC 1 or NIC 2 respectively. A new function is written to send data based on the send-id. Minor modifications in the code enabled us to extend the existing single NIC Web server to a dual NIC architecture with load balancing implemented in the application itself. As the client requests are parallel, the two NICs and the two HTTP tasks run in parallel to provide a parallel path from the NIC hardware to the application. With a single core, this optimizes the use of dual NICs.

The Web server and the NIC driver are implemented using C/C++. Intel gigabit NICs are used to implement the Web server system. About 20 lines of assembly code are used in the NIC driver, which required two functions to read and write to the host controller configuration registers.

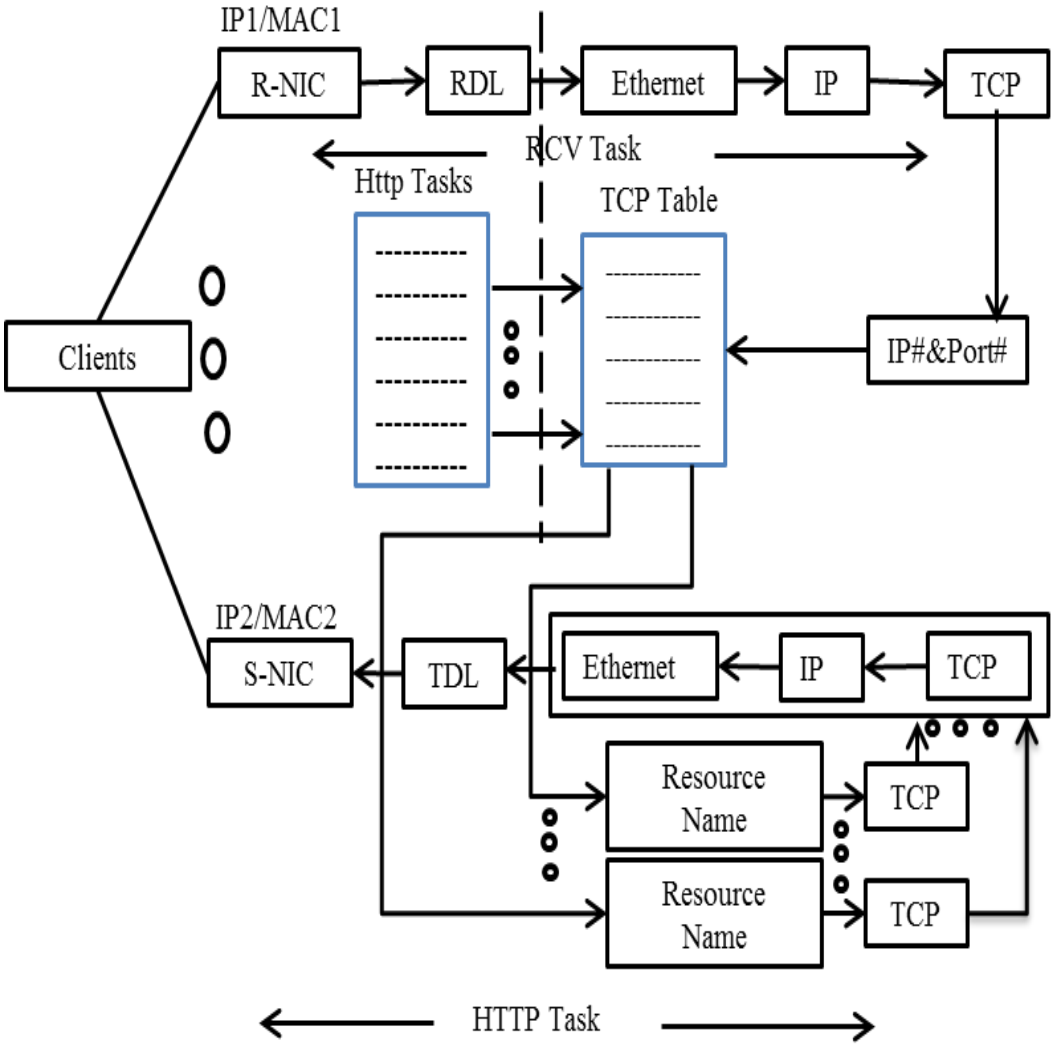

Figure 8. Dual NIC Web Server Architecture

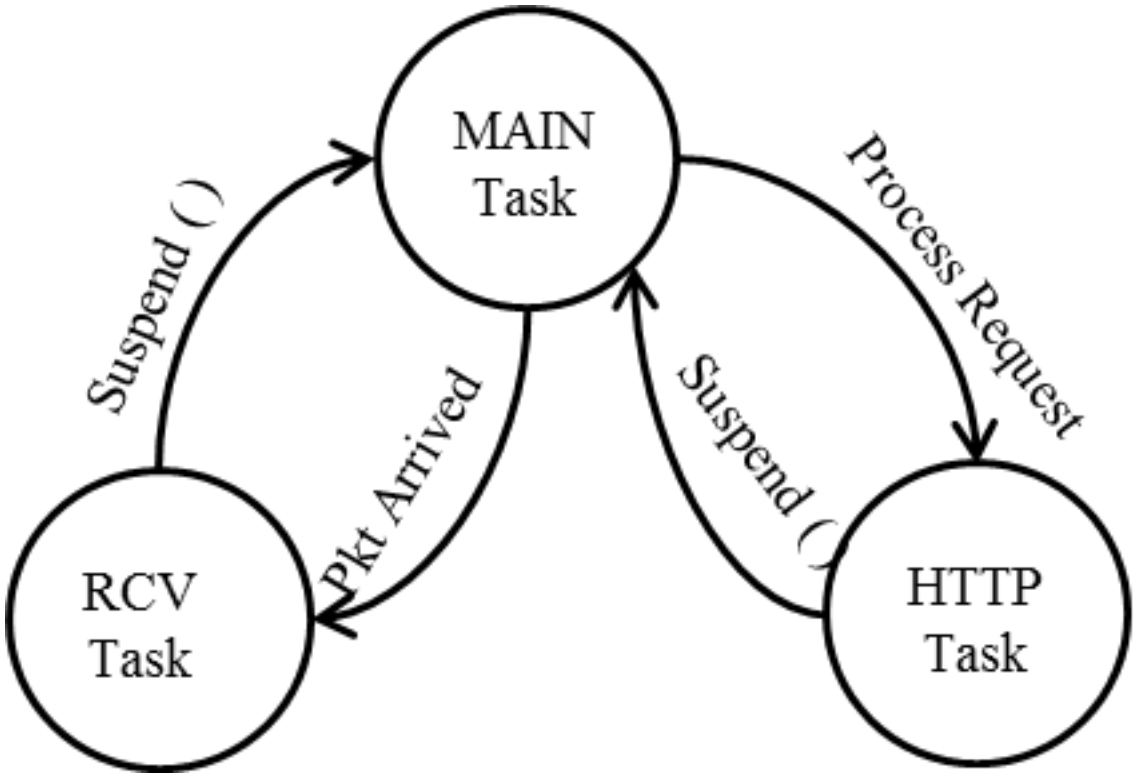

Figure 9. Task State Transition Diagram

4.2 Performance Measurements

This section gives performance measurements for single and dual Intel external gigabit NICs [34] installed on a bare PC Web server running on an Optiplex 960 desktop machine. The HTTP stress tool http_load [35] is used to send requests and collect measurement data.

The experiments are used to study bare PC Web server performance differences between single and dual NICs for different values of the load parameters. The requested file sizes vary from 4K to 1MB. Connection time and initial response times are used as performance metrics.

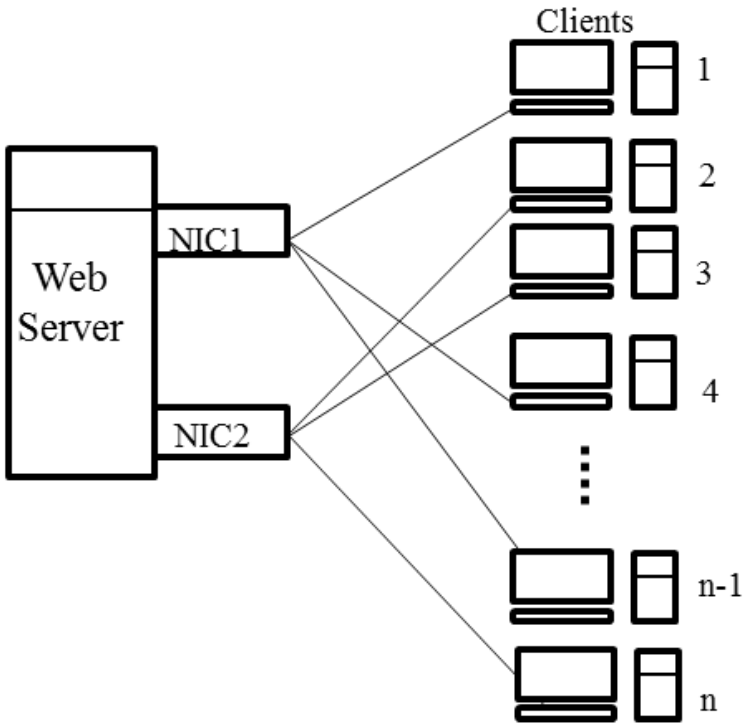

Figure 10. Each Client has a Dedicated Send NIC

4.2.1 Load Balancing Strategy

Two NICs can be used in a variety of ways to send and receive packets. Load balancing for dual NICs with split send and receive paths were considered, where one NIC is dedicated for receiving packets and either NIC is able to send packets to a client.

Several strategies can be easily implemented to choose the NIC for sending packets. For example, packets can be sent by randomly picking the NIC to use for a given client, or alternatively, the NIC to send can be chosen based on throughput, packet size, connection time and/or response time. Considering a simple strategy, where a given NIC is dedicated for sending data to a given client as shown in Fig. 10. For an even number of clients, both NICs send data by sharing the number of requests equally. For an odd number of clients, the server uses NIC 1 for half of the responses to send data and NIC 2 for the other half.

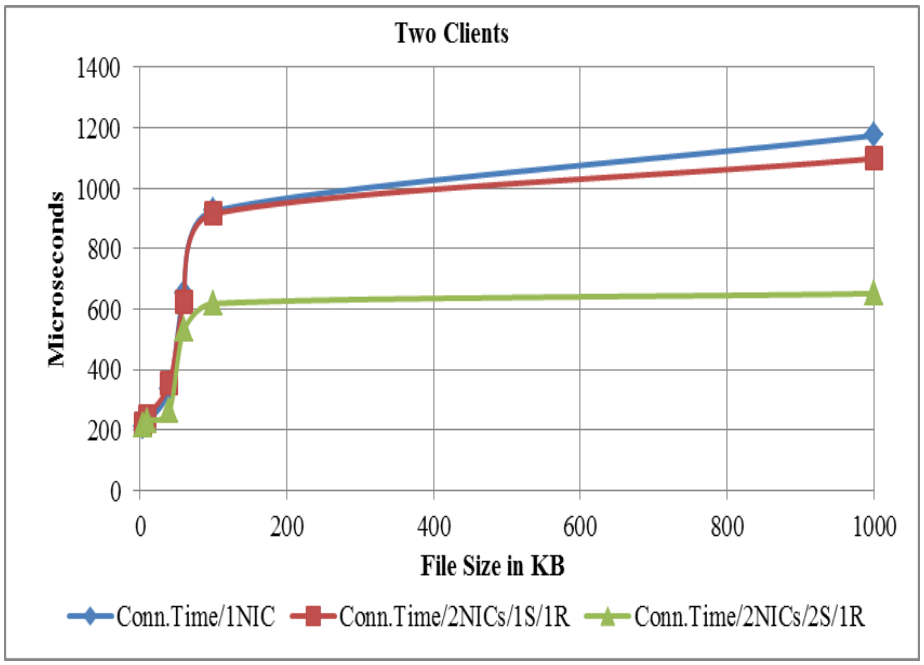

Figure 11. Connection Time (Two Clients)

4.2.2 Two Clients

Fig. 11 shows connection times for a single NIC (one NIC) and dual NICs: (one send, two receive) or (two send, one receive). Connection time with one NIC increases as the file size increases, which is expected as it takes more time to send larger files. When dual NICs are used, one for sending only and one for receiving only, the performance is very close to one NIC. This is because the receive card is not as heavily loaded as the send card. Also, received packets associated with GET requests are small (about 60 bytes each except for the GET request itself). Since the two NICs are not sharing the load equally, there is no significant performance improvement.

When two NICs are used, where both can send data, the lowest connection time is seen because the NICs are now parallelized with respect to clients (each NIC serves one individual client). For 4K and 1 MB file sizes, with a respective rates of 1000 and 30

requests per second for each client. For large files, the request rate is limited by buffers allocated at the server.

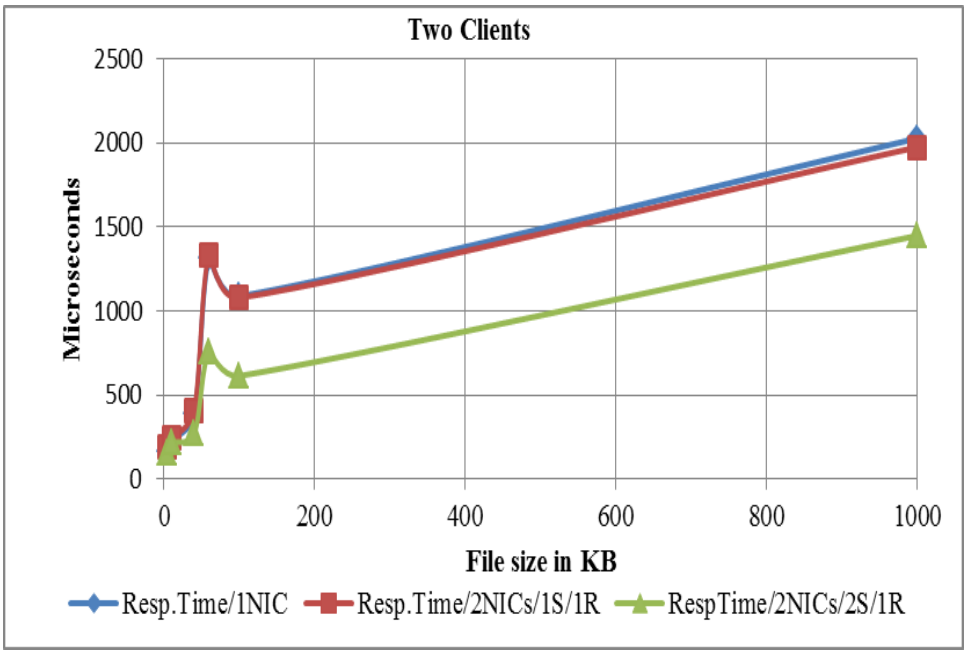

Figure 12. Response Time (Two Clients)

The maximum improvement in connection time is about 44% for a 1 MB file. At this point there was no any connection time improvement for small file sizes as the NICs are not load balanced. As shown in Fig. 12, the improvement in initial response time for a 1 MB file is 29%.

4.2.3 Three Clients

Fig. 13 shows connection times for three clients. Client 1 is served by NIC 1 and Client 2 is served by NIC 2. Half of Client 3's requests are served by NIC 1 and the other half are served by NIC 2. For a 1 MB file, the connection time improvement is 42%, and the initial response time improvement (shown in Fig. 14) is 21%.

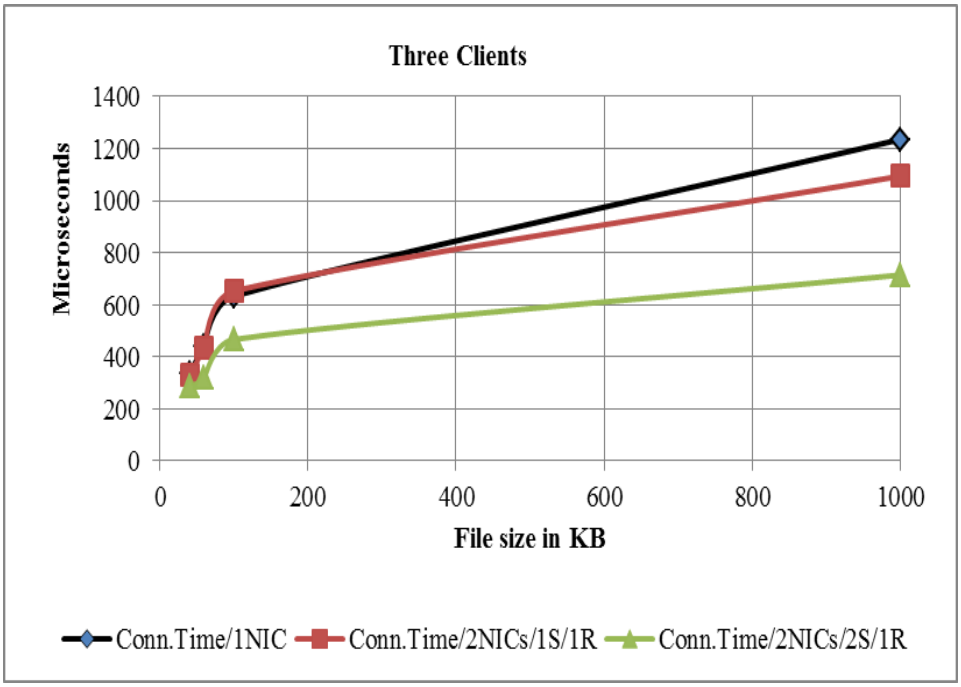

Figure 13. Connection Time (Three Clients)

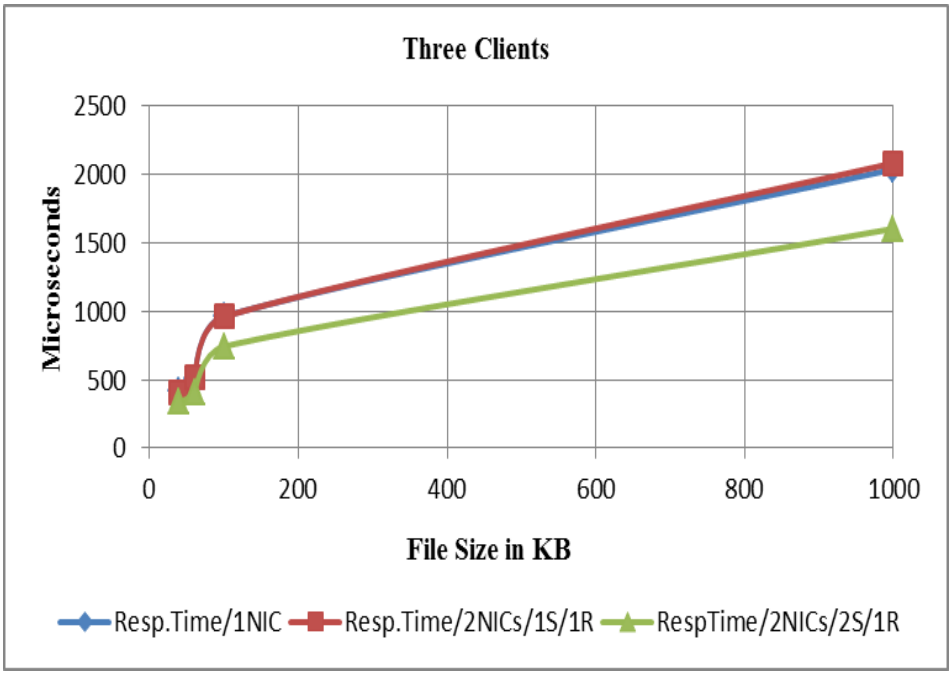

Figure 14. Response Time (Three Clients)

5. UPWARD COMPATIBLE ETHERNET DEVICE DRIVER FOR BARE PC APPLICATIONS

This chapter describes the architecture, design and implementation of an upward compatible or migratable Ethernet device driver (MEDD) that runs on a bare PC or a laptop. Conventional device drivers operate with an underlying OS or kernel in a general purpose computing environment. The MEDD device driver presented here directly communicates with a given application program and it is a part of an application.

5.1 NIC Driver

The device driver interfaces are used by an application programmer to create a given application. There is no kernel or OS running in the bare PC. The driver is designed for an underlying CPU instruction set architecture (ISA) rather than for a given OS or platform. The following subsections describe more details of MEDD that illustrates the BMC approach.

5.1.1 Architecture

The MEDD architecture is illustrated in Fig. 15. A software application can use the NIC driver API that directly communicates with the host controller (HC) in a BMC architecture. The HC in turn controls the NIC hardware. PC BIOS interrupts are used to obtain a device address for the NIC. The software application and the NIC hardware communicate through user memory to exchange data, commands and status information. The data structures required for communicating to NIC hardware reside in shared memory. The HC acts as middleware to provide this communication to the software application and hardware.

In an OS based environment in contrast, the user has no direct communication with the device driver. As shown in Fig. 16, the user application uses system calls for I/O which are

provided by the underlying OS or kernel. The device and the device driver, which use shared memory to communicate, are under the control of an OS and hence not visible to application programmers. BIOS interrupts are used to obtain the NIC device address. However, whereas applications use the BMC API to communicate to BIOS in Fig. 15, they use OS system calls in Fig. 16. The design and development of a MEDD requires a thorough understanding of the "Software Development Manual" for a given NIC [16].

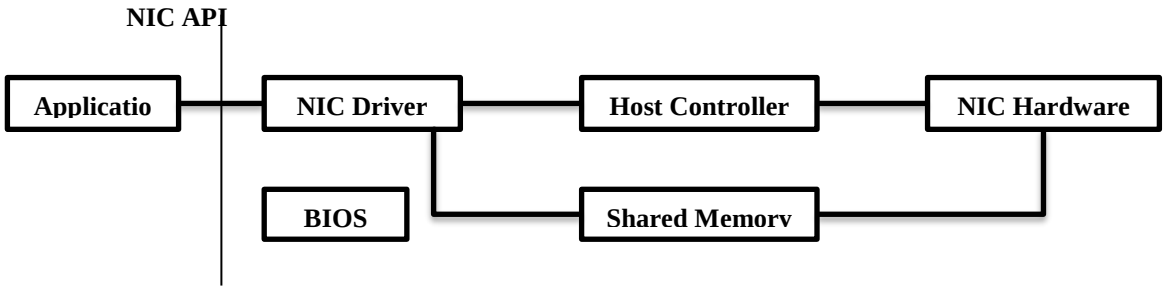

Figure 15. BMC based Architecture

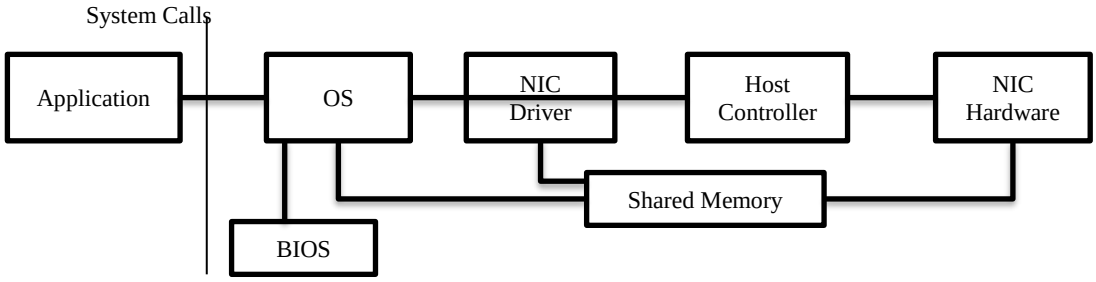

Figure 16. OS based Architecture

5.1.2 Interfaces

The MEDD design provides about 150 interfaces for the API. About half of these interfaces are used to debug the driver and some are internal function calls. They include some C++ interfaces used by the BMC application programmer and two assembly calls as shown in Table 5. A total of 26 interfaces (API) are sufficient to build BMC applications.

Two assembly call interfaces are used to access the control registers [16] of the NIC. Fig. 17 shows a sequence of operations that are needed before the NIC is ready to receive and send data. An example use of an assembly interface is shown in Fig. 18. Two assembly calls setRegister() and getRegister() are sufficient to write this device driver in C++. This example illustrates writing to a transmit descriptor tail register, which has a control register address of 0x3818 [16].

AOAgetShared Mem()	setBaseAddress()
getMACEPROM(SrcMAC)	ColdeReset()
Inittest(Data Structure Parameters)	TEnable()
REnable()	ReadData()
getMAC()	IPInsertPkt()
ARPInsertPkt()	sndCall()
getTDTail()	getRDTail()
setTDTail()	setRDTail()
IncSendInPtr()	initRDL()
incSendOutPtr()	initTDL()
TDLfull()	RDLfull
setRegister()	getRegister()
IsRdescDone()	isTdescDone()

Table 5. Interfaces

Writing values to a register requires two steps. In the first step, the selected register uses IOADDR and in the second step, writing to the register uses IODATA. Reading from a register is also similar, except that setRegister() is used in the first step and readRegister() is used in the second step.

Fig. 19 illustrates the IPinsertPkt() interface as used in a BMC application. In order to insert a packet, the application gets the transmit ring slot to be inserted, forms the packet, inserts the packet in the descriptor, and calls sndCall(). The sndCall() function will read the tail register, increment it and set up the pointer to be transmitted. Notice that in BMC, the entire process of sending an IP packet is controlled by an application programmer. The

details of driver operation is hidden in the IPinsertPkt() function. Similarly, all other interfaces can be controlled by an application programmer and the details are encapsulated in the interface object.

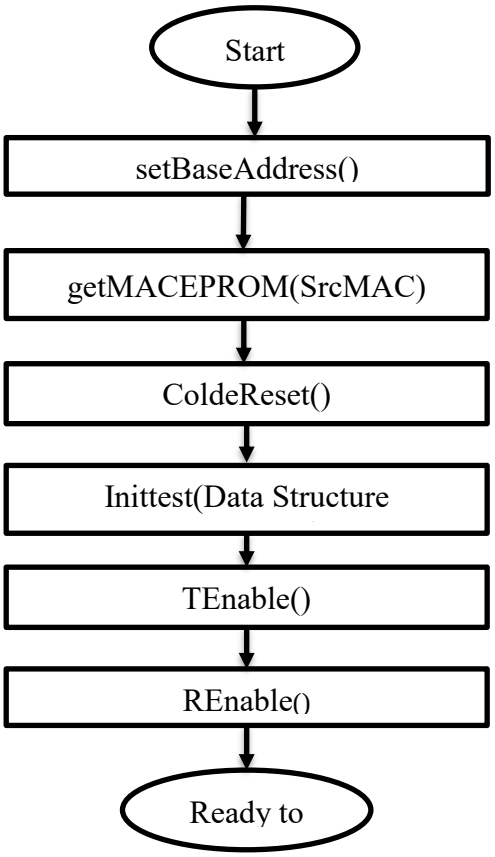

Figure 17. Sequence of Operations

The device driver designed and implemented as above is for an Optiplex 260. It was then migrated to other Dell PCs and also to a laptop. During this process, two control registers had some impact and needed to be modified. The TEnable and REnable functions set control registers. Tables 6 and 7 show transmit and receive control register fields that will be discussed in the next section.

5.1.3 Design

Software design of an Ethernet driver depends upon the internal data structures of the Ethernet hardware [16] and the type of NIC. The Intel NIC hardware defines transmitter and receiver descriptor rings. A descriptor ring as shown in Fig. 20 consists of a set of descriptors organized as a circular list. It has a head and tail to indicate the beginning and end of a ring. It also has IN and OUT pointers for each ring.

```
int EtherObj::setTDTail(int  tailindex)
{
    int retcode=0;
    int temp=0;
    // TDT  0x3818
   setRegister(0x3818, IOADDR);
   //mask higher 16 bits
   temp = tailindex;
   temp = temp & 0x0000ffff;
   setRegister(temp, IODATA);
   return retcode;
}
```

Figure 18. SetTDTail

Each descriptor in this design consists of a 16 byte data structure. There can be up to 4096 descriptors and a minimum of 8 is required for this architecture.

A transmit descriptor (16 bytes) is shown in Fig. 21 [16]. It has 8 bytes of packet address (64 bit address) and 8 bytes of control. Control fields include: Length (2 bytes of packet length), CSO (1 byte), CMD byte, STA (status 4 bits TU, LC, EC, DD) [16], RSV (4 bits reserved), CSS (1 byte) and Special (2 bytes). Similarly, a receive descriptor (16 bytes) is shown in Fig. 22 [16]. It consists of 8 bytes of packet address and 8 bytes of control information. The status field consists of 8 bits (PIF, IPCS, TCPCS, RSV, VP, IXSM, EOP, DD) and Error byte (RXE, IDE, TCPE, CXE, RSV, SEQ, SE, CE)). Some of the fields shown here are reserved and not used in a given driver.

Bit No.	Field
0	Reserved
1	Transmit Enable
2	Reserved
3	Pad Short Packets
4 : 11	Collision Threshold
12 : 21	Collision Distance
22	Software XOFF Transmission
23	Reserved
24	Re-transmit on Late Collision
25	No Re-transmit on underrun
26 : 31	Reserved

Table 6. Transmit Control Register

Bit No.	Field	Bit No.	Field
0	Reserved	16 : 17	Receive Buffer Size
1	Receiver Enable	18	VLAN Filter Enable
2	Store Bad Packets	19	Canonical Form Indicator Enable
3	Unicast Promiscuous Enabled	20	Canonical Form Indicator bit value
4	Multicast Promiscuous Enabled	21	Reserved
5	Long Packet Reception Enable	22	Discard Pause Frames
6 : 7	Loopback mode.	23	Pass MAC Control Frames
8 : 9	Receive Descriptor Minimum Threshold Size	24	Reserved
10	Reserved	25	Buffer Size Extension
12 : 13	Multicast Offset	26	Strip Ethernet CRC from incoming packet
14	Reserved	27 : 31	Reserved
15	Broadcast Accept Mode.		

Table 7. Receive Control Register

The NIC driver designed here uses polling instead of interrupts. Polling is convenient in BMC applications as control flow returns to the main task, which is idle most of the time. When a packet arrives from the network, NIC hardware places it in the receive descriptor and sets the status bit DD in the descriptor. During the polling process, a packet is received by an application by checking the DD bit in the receive descriptor status field. The “isDescDone()” function will perform this polling for receiving packets. Similarly, the “incSendOutPtr()” function will perform polling of sent packets to acknowledge the transmitted packet.

Once the initialization process as shown in Fig.17 is complete, an application will receive packets based on polling. When a packet is ready to be sent, it is simply placed in the transmit ring and control returns to its caller. If the transmit or receive buffer is full, an error occurs and the machine stops. Many other error conditions are also detected and resolved as needed.

5.1.4 Implementation

The Ethernet device driver is implemented as a single class with EtherObj.cpp and EtherObj.h files. In addition, it has two assembly functions consisting of 26 lines of code. The C++ code in this driver is about 3600 lines including comments and its object file size is 37KB. The driver is implemented in C, C++ with a small amount of assembly code and compiled using the Microsoft Visual Studio 12.0 C++ compiler. When this compiler is used, no system libraries or include files are used from this OS based compiler (/NODEFAULTLIB option). More implementation details of BMC applications are given in [15].

5.2 Upward Cmpatible Ethernet Device Driver

Initially, the Ethernet device driver was designed for bare PC applications running on a Dell Optiplex 260 with an onboard Intel chip 82540EM (Gigabit). This driver was used in a BMC Web server application [3] and many other applications as well. The Dell Optiplex 260 model eventually became obsolete and new models were introduced. As a result, the onboard Intel NIC was replaced with an equivalent external NIC. In BMC research, evolution of NIC models have caused road blocks due to heterogeneity and obsolescence in hardware.

```
int EtherObj::IPInsertPkt(char* PACK, int PACK_Size, int
PROTOCOL, char* Target_HAdd, int CurrentTask)
{
  // variables not included …..
x = TDLPointer + SendInPtr * 16 - ADDR_OFFSET;
      //TDL address is absolute
//……
PACK = PACK - 14;
    p1 = (long*)x; // p1 is address of tdlPointer
//….error code not included
SendInPtr++;
if (SendInPtr == NO_OF_TDL)
     {
      SendInPtr = 0; //circular list
     }
    p1++;
    p1++;
     sizeOfPacket = PACK_Size + 14;
   for(i=0; i< 6; i++)
    {
       PACK[i] = Target_HAdd[i];
      PACK[i+6] =  mac[i];
   }
       PACK[12] = ((PROTOCOL>>8) & 0x00FF);
       PACK[13] = ((PROTOCOL) & 0x00FF);
    temp = *p1;
    temp = temp & 0xffff0000;
    temp = temp + sizeOfPacket;
   *p1 = temp;  //TDL entry for length
   retcode = sndCall();
   return 0;
```

Figure 19. IPInsertPacket

In OS based systems, vendors write device drivers as the platforms change and this enables them to keep up with the evolution. Our experience in writing NIC drivers and other drivers [18] led us to adopting an upward compatibility path for BMC applications. Due to the evolution of Ethernet NIC architectures, many vendors ultimately moved from external NICs to internal onboard chips.

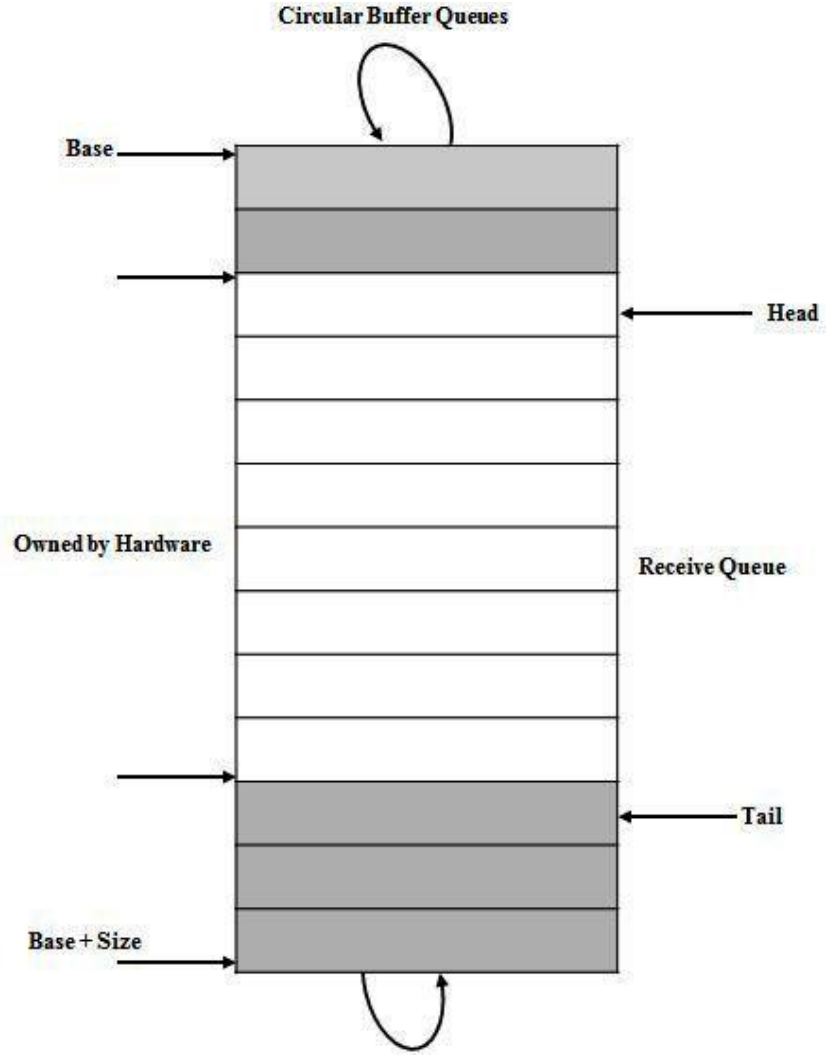

Figure 20. Descriptor Ring Data Structure

However, each vendor chip is different in its hardware and software specifications causing heterogeneity. One of our findings is that even though vendors and hardware differ, the basic internal structure and organization of NICs are similar. However, newer models

added new facilities and features abandoning upward compatibility. This resulted in different device drivers for different NICs. This problem is addressed somewhat by NDIS, but its focus is different as discussed before.

The following sections describe the upward compatible device driver designed for an onboard NIC in a Dell Optiplex 260. This driver design is designated as D1.

5.2.1 Dell Optiplex 960

The Dell Optiplex 960 model has an internal NIC Intel 82567 (Gigabit chip). The D1 driver was upgraded to work with this model. The upgraded driver is referred to as D2. The following changes to D1 were made.

1. In order to obtain the device address, device id (0x10de) was changed for BIOS interrupt call.
2. The TEnable function that uses transmit control register (0x400) [19] was changed as shown below (Refer to Fig. 21 for details of the register fields).

In D1, the transmit control register was initialized to 0x002000f2. The reserved bit (28) is 0, collision distance bits (21-12) are 0x200, and bits 29, 28 are 0s. In D2, the reserved bit 28 must be 1 [19], collision distance bits (21-12) are 0x3f, and read request threshold bits (30, 29) are 01. Also, in D2 the reserved bit setting is strictly enforced by the device hardware and it is 1 instead of 0. The read request threshold does not have much impact on the driver; however, more threshold was kept. Since the collision distance in D1 is not used, it is 0. The collision distance in D2 is very sensitive as the processor is faster and may have been the reason for more collisions.

3. The getMACEPROM() function in D2 does not work. The reasons for this are:

(1) Optiplex 260 uses ICH4 and the EERD [19] register is used to read MAC from the EPROM; (2) Optiplex 960 uses ICH9 and the EERD register is not available as per our knowledge.

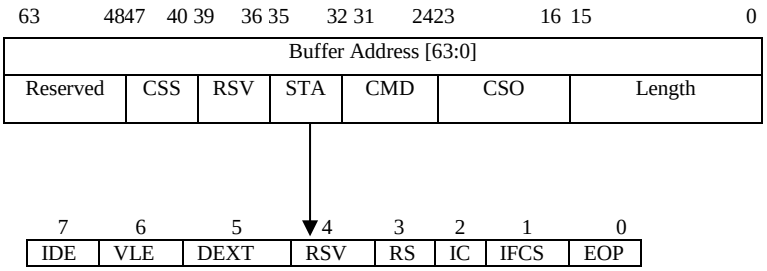

Figure 21. Transfer Descriptor (TDL)

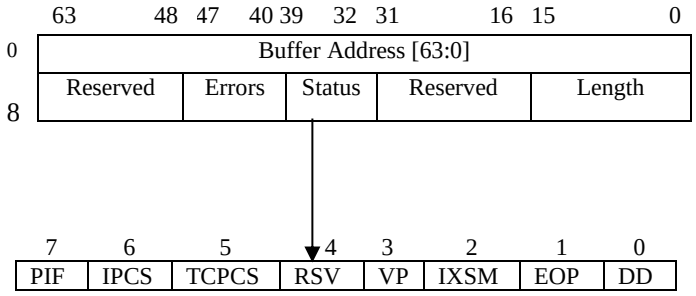

Figure 22. Receive Descriptor (RDL)

This problem was addressed by reading the MAC address using the resource address obtained from a Windows machine for this model. For the machine under test, the resource

address was 0xfe6e0000. This memory mapped address can be used to read the MAC address from memory. Another finding is that each machine may have a different resource address. With the above few changes, the D2 driver runs successfully on an Optiplex 960 with the on board NIC.

5.2.2 Dell Optiplex 9010 and HP Elite Book 8460P

The Dell Optiplex 9010 models have an Intel 82579LM (Gigabit chip) onboard. The D1 driver was upgraded to work with this model. This driver is referred to as D3. The following changes to D1 were made.

1. In order to obtain the device address, the device id (0x1502) was changed for the BIOS interrupt call.

2. TEnable function that uses the transmit control register (0x400) [20] was changed as shown below. In D1, transmit control register was initialized to 0x002000f2. The reserved bit (28) is 0, collision distance bits (21-12) are 0x200, and bits 29, 28 are 0s.

In D3, the reserved bit 28 must be 1 [20], collision distance bits (21-12) are 0x3f, and read request threshold bits (30, 29) are 01. This is same as D2. The REnable function that uses the receive control register (0x100) was changed. The code for D1 was 0x04408002. Bit 22 (discard pause frames) is 1. In Optiplex 9010, this bit is reserved and it must be 0. The new code is 0x04008002.

3. The getMACEPROM() function in D3 does not work. The reasons for this are: (1) Optiplex 260 uses ICH4 and the EERD register to read MAC from the EPROM; (2) Optiplex 9010 does not use an ICH controller. It uses the Intel 6 series chipset [20] for Optiplex 9010, which is different from the ICH architecture. Similarly, for the HP Laptop, a different chip Intel QM67 is used [20].

This problem was addressed by reading the MAC address using the resource address obtained from a Windows machine for this model. For the machine under test, the resource address was 0xf7do0000 for the Optiplex 9010 and 0xd480000 for the HP Laptop. This address can be used to read the MAC address from memory. With the above few changes, the D3 driver runs successfully on the Optiplex 9010 and the HP Laptop. We noted that the behavior of the 9010 and the HP Laptop drivers were the same except for their resource addresses.

5.3 OS Based NIC Drivers

This section describes some details of OS or kernel based Ethernet device drivers. The following three models were studied: Optiplex 260 (O1), Optiplex 960 (O2), and Optiplex 9010 (O3) (HP Laptop) with their related driver binary executables on Windows machines. In all these models, their drivers interface with the NDIS, Kernel and HAL (hardware abstraction layer) modules. None of these drivers operate in standalone mode. Table 4 summarizes some characteristics of these drivers obtained by using the OBJDUMP [21] tool and the BMC driver.

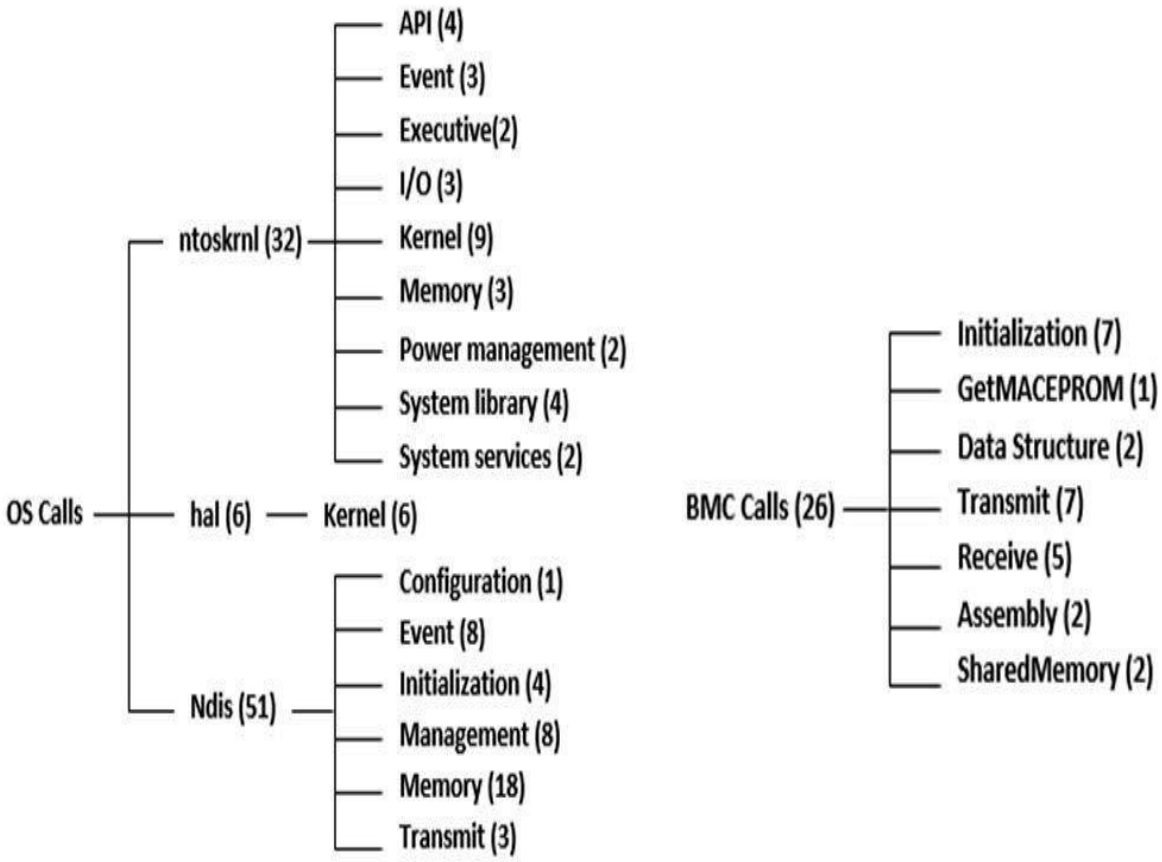

Figure 23. Classification of Interfaces

5.4 Observation and Findings

We now summarize our observations and findings from this study. Ethernet drivers have common architectural features such as transmit and receive data structures, initialization, transmit and receive control facilities, and status and command controls. These basic elements are same in all NIC designs. Within one vendor (Intel), there is no upward compatibility maintained in the NIC design with respect to hardware and software. Obviously, there is no compatibility across different vendors and different NIC designs.

The basic NIC architecture has not changed in these models, but their implementations and interfaces are different resulting in heterogeneity.

While the new designs may have some enhanced functionality, there is no need to abandon the original architecture and its design. It is fairly easy to enhance functionality by preserving upward compatibility. Using the reserved bits, this goal can be achieved. The NDIS approach does not address this issue as its focus is at the application level. It is also possible to make the driver code upward compatible as illustrated in the MEDD approach. When upward compatibility is the focus, it is possible to migrate the hardware and software to handle enhanced functions while eliminating waste and obsolescence. Our observations and findings are also applicable to other hardware and software designs.

6. SIGNIFICANT CONTRIBUTIONS

BMC computing applications have their own drivers for network interfaces cards (NIC) or chips. As the NICs are different for each PC, there is a need to write new drivers for each version. The NICs also vary depending on a vendor. We tried to solve the problem through middleware, which helps to communicate to 3COM or Intel NICs thus avoiding changes to the application (Webserver). We also provide a novel approach where send and receive packets can be handled by a different NIC. This technique also resulted in better performance compared to a single NIC and provided isolation in send and receive paths for the Webserver. Finally, the NIC driver written for Optiplex 260 and Intel 82540EM (motherboard chip) was made to work with Optiplex 960 (Intel 82567 LM), Optiplex 9010 (Intel 82579 LM), and HP Elite Book 8460P (Intel 82579LM). The changes made were limited to a few lines of code in the driver. This illustrates that it is possible to port NIC driver code with minimal effort in BMC computing applications.

7. CONCLUSION

We presented a novel approach for designing middleware that enables existing bare PC applications to use integrated NIC drivers. Two different server applications, a Web server with an Intel NIC and a Webmail server with a 3COM NIC, were used to test the middleware implementation. A taxonomy was given to highlight the issues in bare PC middleware design. A generic NIC architecture was proposed as a means to unify NICs at the data structure and descriptor levels. Further study is needed to extend the middleware so that it can be used with a variety of bare PC applications and NICs. In future, homogenized NIC interfaces based on the generic NIC architecture could be enhanced and moved into the hardware.

We also described the architecture, design and implementation of a dual NIC Web server with split send-receive paths that runs on a bare PC. Performance measurements were given to illustrate the improvement in connection and initial response times as the load is increased. Associating a given client's request with a given NIC for sending data was found in our studies to be optimal for load balancing with dual NICs in a bare PC Web server. Other strategies for load balancing multiple NICs can be investigated in the future.

The dual NIC approach was shown to exploit the natural partition on send and receive logic at the NIC level, Ethernet driver level and the Web server implementation level. Isolating send and receive paths provides simplicity and better performance. Although not investigated here, multiple NICs can be used to completely separate the send and receive channels for security purposes. This approach for Ethernet bonding on servers without any OS or kernel support extends to multicore systems. With a single NIC for communication (as in an ordinary desktop), the multi-core processors share memory and the NIC. When

dual or multiple NICs are used, they can be allocated to cores thus improving performance. It will also be possible to integrate NICs with multicores on the same chip.

Finally, we implemented a novel Ethernet device driver that is simple, small and upward compatible. Its design showed how device driver code can be easily made upward compatible by only making a few changes in the control registers. We also compared the design of our driver with the design of existing OS based device drivers by classifying driver interfaces. Our observations and findings can be used to design new device drivers with similar characteristics. In particular, maintaining upward compatibility in device driver design can reduce heterogeneity and lead to a standard architecture for drivers.

8. APPENDIX

This appendix gives details of the implementation of middleware, multiple NICs and upward compatible drivers in BMC applications.

The bare PC NIC driver has a small set of direct interfaces that can be invoked by a bare PC application program. The following calls illustrate those interfaces, which are implemented in C++. We have 3COM and Intel drivers in BMC; this middleware provides access to either NICs which is transparent to the application program. A total of 26 functions implemented to build the middleware.

8.1 Middleware Implementation

The above NIC interface calls from the bare PC application program use EO (ethernet object) and its related member functions. We have created a middleware class object (ethernet.cpp), where it can direct the calls to 3COM or Intel based on a NIC_FLAG as shown defined below and also shown in Fig. 2:

- NIC_FLAG 0 – 3COM code to 3COM NIC
- NIC_FLAG 1 - Intel code to Inel NIC
- NIC_FLAG 2 – 3COM code to Inel NIC
- NIC_FLAG 3 - Intel code to 3COM NIC

3COM and Intel NIC drivers used have some heterogeneity with respect to variables and functions. Some of the heterogeneity is shown in Table 8.

INTEL	3COM
TDLPointer = 0;	DownListPointer=0;
TDLDataPointer = 0;	DownListDataPointer = 0;
RDLPointer = 0;	UpListPointer = 0;
SendInPtr = 0;	SendInPtr = 1;
SendOutPtr = 0;	SendOutPtr = 1;
ReceiveInPtr = 0;	ReceiveInPtr = 0;
ReceiveOutPtr = 0;	ReceiveOutPtr = 0;
isRDescDone(	getCount()
IncSendOutPtr()	IncSendPtr(0);
TDLFull ()	DPDFull()

Table 8 .Simillar Functions in Both NICs

The actual implementation code for the above interfaces is shown here.

1. Set base address in the object IOBASE for 3COM and IOBASE1 for intel 0x9c and

```
void EtherObj::setBaseAddress()
  {
       int addr;
        if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
          {
           addr  = io.AOAgetSharedMem(0x9c);
          EO3C.setBaseAddress(addr);
          }
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       {           addr  = io.AOAgetSharedMem(0x58);
          EOI.setBaseAddress(addr);
       }
       else return;
  }
```

2. getMACEPROM() is in Intel Driver, but not in 3COM

```
void EtherObj::getMACEPROM(char *dMAC)
{
  this is an intel NIC function
  if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
        EOI.getMACEPROM (dMAC); //to intel
  else if  ((NIC_FLAG == 0) || (NIC_FLAG == 3))
        EO3C.getMAC(dMAC);
       else return;
}
```

3. Initialization function is different for both cards

```
 int EtherObj::inittest(long tbaseAddress, long tbufferPointer, long rbaseAddress,
 long rbufferPointer, char *srcmac)
  {
  int retcode=0;
  if  ((NIC_FLAG == 1) || (NIC_FLAG == 2))
  return EOI.inittest(tbaseAddress, tbufferPointer,rbaseAddress,
rbufferPointer,srcmac);
    else  if  ((NIC_FLAG == 0) || (NIC_FLAG == 3))
           {
               EO3C.Init();
              retcode = EO3C.createDataStructure(RcvLstSize,  0, UPD_ADDR_3COM,

PD_DATA_3COM,0x80000640,SndLstSize,0x1e008000,DPD_ADDR_3COM,DPD_DATA_3COM,
              0x80000640);
              return 0;
     }
        else return -1;
  }
```

4. Enable transmitter

```
int EtherObj::TEnable()
  {
   if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
        return EOI.TEnable();
        else if ((NIC_FLAG == 0) || (NIC_FLAG ==3))
        return  EO3C.enableTransmit();
       else return -1;
  }
```

5. Enable receiver

```
  int EtherObj::REnable()
  {
    if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
      return EOI.REnable();
       else if ((NIC_FLAG == 0) || (NIC_FLAG ==3))
       return   EO3C.enableReceive();
       else return -1;
  }
```

6. Insert ARP packet

```
int EtherObj::ARPInsertPkt(char* PACK,int PACK_Size, intPROTOCOL,
char*Sender_HAdd,    char* Target_HAdd, int sendtype, int CurrentTask)
{
    if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.ARPInsertPkt( PACK, PACK_Size, PROTOCOL, Sender_HAdd,
       Target_HAdd,sendtype,CurrentTask);
        else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
         return EOI.ARPInsertPkt(PACK,PACK_Size, PROTOCOL, Sender_HAdd,
              Target_HAdd,sendtype,CurrentTask);
              else return -1;
}
```

7. ReadData

```
int EtherObj::ReadData(char** Data, int* Type, char*ipaddr, char*macaddr)
{
   int retcode = 0;
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       {
          retcode = EO3C.ReadData(Data, Type, ipaddr, macaddr);
          return retcode;
       }
        else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
          return EOI.ReadData( Data, Type, ipaddr, macaddr);
         else return -1;
}
```

8. IPInsertPacket

```
int EtherObj::IPInsertPkt(char* PACK, int PACK_Size, int PROTOCOL, char*
Target_HAdd, int CurrentTask)
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       {
       return EO3C.IPInsertPkt(PACK, PACK_Size, PROTOCOL, Target_HAdd,
       CurrentTask);
       }
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.IPInsertPkt(PACK, PACK_Size, PROTOCOL, Target_HAdd,
       CurrentTask);
       else   return -1;

  }
```

9. Format TCP Packet

```
  int EtherObj::FormatEthPacket(char* PACK, int PACK_Size, int PROTOCOL, char*
  Target_HAdd, long InPtr, int sendtype, int CurrentTask)
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       {
        return
       EO3C.FormatEthPacket(PACK,PACK_Size,PROTOCOL,Target_HAdd,InPtr,sendtype,
        CurrentTask);
        }
         else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
          return
        EOI.FormatEthPacket(PACK,PACK_Size,PROTOCOL,Target_HAdd,InPtr,sendtype,
        CurrentTask);
           else return -1;
}
```

10. Format N ethernet packets

```
int EtherObj::FormatEthPacketN(char* PACK, int PACK_Size, int PROTOCOL, char*
Target_HAdd,  long InPtr, int count,int sendtype, int CurrentTask)
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       {
       return
       EO3C.FormatEthPacketN(PACK,PACK_Size,PROTOCOL,Target_HAdd,InPtr,count,
       sendtype,CurrentTask);
       }
        else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
        return EOI.FormatEthPacketN(PACK,PACK_Size,PROTOCOL,Target_HAdd,InPtr,
              count,sendtype,CurrentTask);
              else return -1;
}
```

11. Get ethernet buffer count

```
int EtherObj::getCount()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.getCount();
         else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
        return EOI.isRDescDone(EOI.ReceiveOutPtr);
              else return -1;
}
```

12. Increment the SendOutPtr

```
int EtherObj::IncSendPtr(int ctask )
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.IncSendPtr(ctask);
       if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.IncSendOutPtr();
        else return -1;
}
```

13. Get Downlist Data Pointer

```
int EtherObj::getDownListDataPointer()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
        return EO3C.DownListDataPointer;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.TDLDataPointer;
       else return -1;
}
```

14. Get DownList Pointer

```
int EtherObj::getDownListPointer()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.DownListPointer;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.TDLPointer;
       else return -1;
}
```

15. Set the downlist pointer

```
void EtherObj::setDownListPointer()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.DownListPointer = value;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.TDLPointer = value;
       else return -1;
}
```

16. Get SendIn Pointer

```
int EtherObj::getSendInPtr()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.SendInPtr;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.SendInPtr;
       else return -1;
}
```

17. Get SendOut Pointer

```
int EtherObj::getSendOutPtr()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.SendOutPtr;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.SendOutPtr;
       else return -1;
}
```

18. Get Receive In Pointer

```
 int EtherObj::getReceiveInPtr()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.ReceiveInPtr;
          else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
         return EOI.ReceiveInPtr;
       else return -1;
}
```

19. Set Receive Out Pointer

```
void EtherObj::setReceiveInPtr(int value)
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       EO3C.ReceiveInPtr = value;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       EOI.ReceiveInPtr = value;
}
```

20. Get Receive Out Pointer

```
int EtherObj::getReceiveOutPtr()
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.ReceiveOutPtr;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       return EOI.ReceiveOutPtr;
        else return -1;
}
```

21. Set Receive Out Pointer

```
void EtherObj::setReceiveOutPtr(int value)
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
        EO3C.ReceiveOutPtr= value;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       EOI.ReceiveOutPtr = value;
}
```

22. Set Send In pointer

```
void EtherObj::setSendInPtr(int value)
{
       if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       EO3C.SendInPtr = value;
       else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       EOI.SendInPtr = value;
}
```

23. Increase Send In pointer

```
void EtherObj::incSendInPtr()
{
   if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       EO3C.SendInPtr = EO3C.SendInPtr + 1;
   else if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
       EOI.SendInPtr = EOI.SendInPtr + 1;
}
```

24. Get TDL pointer

```
int EtherObj::getTDLDataPointer()
{
 if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
  return EOI.TDLDataPointer;
   else if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
       return EO3C.DownListDataPointer;
        else return -1;
}
```

25. Get TDL Data pointer

```
int EtherObj::getTDLDataPointer()
{
	if ((NIC_FLAG == 1) || (NIC_FLAG == 2))
	return EOI.TDLDataPointer;
	else if ((NIC_FLAG == 0) || (NIC_FLAG == 3))
	return EO3C.DownListDataPointer;
	else return -1;
}
```

26. Get Next x is to get the buffer address for both drivers x 3COM and Intel.

```
int EtherObj::getNextX()
{
   int retcode=0;
   long x = 0;
   long *p1;
   char *send_buffer;
   long InPtr=0;
   int value;

        if (NIC_FLAG == 0 || NIC_FLAG == 3)
           {
        value = getDownListPointer() + getSendInPtr() * 32 +8 - ADDR_OFFSET;

       //error checking
       if ((value-8+ADDR_OFFSET) > (getDownListPointer() + (SndLstSize-1) * 32))
            {
                return -112;
            }

       p1 = (long*)value;
        //now check if this data buffer is within the limits of DPD data buffers
        if ((*p1) >= (getDownListDataPointer() + SndLstSize * getPacketSize()))
          {
              return -113;
              }
              return value;
          }

     else if (NIC_FLAG == 1 || NIC_FLAG == 2)
       {
            value = getTDLPointer() + getSendInPtr() * 16 - ADDR_OFFSET;
           //check if the TDL pointer in the range
           if ((value+ADDR_OFFSET) > (getTDLDataPointer() + (NO_OF_TDL-1) * 16))
           {
              return -122;
           }

           p1 = (long*)value;
          //now check if this data buffer is within the limits of TDL data buffers
          if ((*p1) >= (getTDLDataPointer() + NO_OF_TDL * T_BUFFER_SIZE))
           {

              return -123;
           }
          return value;
       }
       else
       {
              io.AOAPrintText("Bad NIC_FLAG", Line20+80);
              io.Application_Status = 2;
              io.AOAExit();
       }

       return 0;
}
```

8.2 Ethernet Bonding Implementation

In this case, we used Optiplex 960 with two Intel 82540EM external cards. Send and receive paths use different NICs. Some sample code is shown here to illustrate this concept. EO1 is used to send data and EO2 is used to receive data.

```
       id = io.AOAgetSharedMem(0x9c); //get NIC 1 device address
       EO1.setBaseAddress(id);        //set the base address in the object
       id = 0;
       id = EO1.getBaseAddress();
       id = io.AOAgetSharedMem(0x58); //get NIC 2 device address
       EO2.setBaseAddress(id);        //set the base address in the object
       id = 0;


       EO1.getMACEPROM(RcvMAC);//get the MAC address NIC 1 and set as reciver MAC
address
       EO2.getMACEPROM(SrcMAC); //get the MAC address NIC 2 and set as source MAC
address


       retcode = EO1.ColdReset();
       retcode = EO2.ColdReset();

       retcode = EO1.inittest(DPD_ADDR, DPD_DATA, UPD_ADDR, UPD_DATA,RcvMAC);
       retcode = EO2.inittest(DPD_ADDR1, DPD_DATA1, UPD_ADDR1, UPD_DATA1,SrcMAC);

       retcode = EO1.TEnable();
       retcode = EO2.TEnable();

       retcode = EO1.REnable();
       retcode = EO2.REnable();

RXSize =   EO2.ReadData(&Data, &PacketType, ip.msourceIP, macaddr);
        TotalDataRcvd = TotalDataRcvd + RXSize; //accumulate the data packets
arrived



int IPObj::sendData(char* data, int len, char DestIP[4], char DestMAC[6], int
protocol, int currenttask)
         {
          retcode = EO1.IPInsertPkt(data,sizeOfPacket,IP_TYPE,DestMAC,
currenttask);
         }
             tcb->sendid = 1; //send card
```

The following function shows how we use two NICs in TCP code to send data. As our experiments allow to send or receive from either card, we need to modify the code to

accomplish this task. Notice that sendid is used to keep track of which card is sending at a given time. This function illustrates how a TCP send works in a BMC application. When a TCP packet is formed, it gets the ethernet buffer, adds TCP, IP and Ethernet headers and inserts the packet into the ethernet buffer. It is done in one call and there is no need to go through the layers. The actual buffer at the NIC is inserted at a TCP level.

```
int TCPObj::SendMisc2Cards (int tcbno, char *destIP, char * destPort, char Flags,
              char *TargetMAC, int sendtype, int currenttask)
{
       TCBRecord *tcb;
       char *send_buffer;
       long *p1;
       long c3;
       char c4;
       long x = 0;
       int i;
       int retcode = 0;
        char data[10];
       int TCPPack_size=0;
       long InPtr=0;
       int TCBRecordNum=0;
       int sendid=0;

       //int TCBRecordNum = SearchTCB(destIP, destPort);
       //TCBRecord *tcb;
        tcb=(TCBRecord*)(TCBBase + (tcbno * sizeof(TCBRecord) ));
       TCBRecordNum = tcbno;
       io.AOAPrintText("K2", Line2+18);
       sendid = tcb->sendid;
       p1 = &c3; //dummy address
       send_buffer = &c4;  //dummy address
       if (tcb->sendid == 2)
       {
       if (EO2.TDLFull() == 1)
          return -30;   //DPD is full
        }
       else if (tcb->sendid == 1)
       {
       if (EO1.TDLFull() == 1)
          return -300;   //DPD is full
         }
       else
       {
              io.AOAPrintText("Wrong sendid in SendMisc2Cards", Line23+20);
              io.Application_Status = 2;
              io.AOAExit();
       }
```

```
        if (tcb->sendid == 1)
        {
           x = EO1.TDLPointer + EO1.SendInPtr * 16 - ADDR_OFFSET;
          InPtr = EO1.SendInPtr;
        }
        else if (tcb->sendid == 2)
        {
           x = EO2.TDLPointer + EO2.SendInPtr * 16 - ADDR_OFFSET;
          InPtr = EO2.SendInPtr;
        }

        //check if the TDL pointer in the range
        if (tcb->sendid == 2)
        {
        if ((x+ADDR_OFFSET) > (EO2.TDLPointer + (NO_OF_TDL-1) * 16))
           return -221;
        }
        else if (tcb->sendid == 1)
        {
        if ((x+ADDR_OFFSET) > (EO1.TDLPointer + (NO_OF_TDL-1) * 16))
        {
               return -222;

        }
        }

        p1 = (long*)x;

        if (tcb->sendid == 2)
        {
        //now check if this data buffer is within the limits of TDL data buffers
        if ((*p1) >= (EO2.TDLDataPointer + NO_OF_TDL * T_BUFFER_SIZE))
               return -23;
        }
        else if (tcb->sendid == 1)
        {
        //now check if this data buffer is within the limits of TDL data buffers
        if ((*p1) >= (EO1.TDLDataPointer + NO_OF_TDL * T_BUFFER_SIZE))
               return -232;
        }

        send_buffer = (char*)*p1;
//address from DPD pointing to next available slot

        if (tcb->sendid == 2)
        {
          EO2.SendInPtr++;    //this is like an InPtr which inserts packets
          if (EO2.SendInPtr == NO_OF_TDL)
            EO2.SendInPtr = 0; //circular list
        }
        else if (tcb->sendid == 1)
        {
         EO1.SendInPtr++;    //this is like an InPtr which inserts packets
         if (EO1.SendInPtr == NO_OF_TDL)
            EO1.SendInPtr = 0; //circular list
        }
```

```
        //add TCP header in front of the packet
        send_buffer = send_buffer + 14 + 20 - ADDR_OFFSET;//add header before data
         if (TCBRecordNum ==-1)
             TCPPack_size = FormatTCPPacket(send_buffer, destIP, destPort, Flags,
                         0, 0, 0, data, 0, 0, currenttask);
         else

             {
              tcb = (TCBRecord*)(TCBBase + ( TCBRecordNum * sizeof(TCBRecord) ));
              TCPPack_size = FormatTCPPacket(send_buffer, tcb->IP, tcb->PORT,
               Flags, tcb->RCVWND,tcb->SNDNXT, tcb->RCVNXT, 0, 0, 0, currenttask);

             }

        send_buffer = send_buffer - 20; //20 byte IP header

         retcode = ip.FormatIPPacket(send_buffer, TCPPack_size, destIP, TargetMAC,
                     TCP_Protocol, currenttask);
        if(retcode != 0)
           return retcode;

        send_buffer = send_buffer - 14; //14  byte ethernet header

        SendCountTotal++;

        if (tcb->sendid == 2)
        {
        retcode = EO2.FormatEthPacket(send_buffer, TCPPack_size+20, IP_TYPE ,
TargetMAC,
                                InPtr, sendtype, currenttask);
        }
        else if (tcb->sendid == 1)
        {
        retcode = EO1.FormatEthPacket(send_buffer, TCPPack_size+20, IP_TYPE ,
TargetMAC,
                                InPtr, sendtype, currenttask);
        }
        if(retcode != 0)
          {
           return retcode;
          }

   return 0;
}
```

8.3 Upward Compatibility of NIC Driver Implementation

This section shows details of code changes that are needed to provide upward compatibility of a NIC driver from an Optiplex 260 to an HP Laptop.

8.3.1 Obtaining device address for NIC

Each NIC has a vendor id and device id. When a system boots, we obtain the NIC device address using the vendor id and device id and store it in shared memory in real memory area. BIOS interrupts are used to obtain the NIC address.

Table 9 shows vendor and device IDs for the PCs used in our work. These are used in the following assembly code sample to obtain the NIC device address.

Model	**NIC**	**Vendor id**	**Device id**
Optiplex 260	Intel 82540 EM	8086	100e
Optiplex 960	Intel 82567LM	8086	10de
Optiplex 9010	Inel 82579 LM	8086	0085
HP Elite Book 8460P	Intel 82579 LM	8086	1502

Table 9. Vendor and Device IDs

The following assembly code shows how we obtain the device address using a vendor id and device id and BIOS interrupts.

```
Start         PROC
              mov     ax,RDataSeg
              add     ax,RELOCATE1
              mov     ds,ax
              mov    al,'Y'
              mov    ah,14
              int    10h

            ;*******************************************************
            ; get the NIC device address and store it in shared memory
            ; this has to be done only in real mode
            ; Optiplex 9010
            ;*******************************************************
              mov ah, 0b1h  ; function  PCI BIOS p23
              mov al, 02h   ; find PCI device PCI BIOS p23
              ;3com
              ;mov cx, 9200h       ; device id p71
              ;mov dx, 10b7h   ; subsystem vendor id p71
              ; intel
              ;mov cx, 1502h
              mov cx, 0085h   ; Intel Optiplex 9010
              mov dx, 8086h
              mov si, 0
              int 1ah       ; calls PCI BIOS interrupt
              jc carry1       ; error
              xor cl, cl      ; good
              jmp next100
              carry1:
              mov cl, 1       ; cl is 1 or 0
              next100:
              mov eax,ebx;
              sal eax, 16     ; move to left
              mov bl, cl
              mov ax, bx
                mov esi, eax
              ; eax has the following data
              ;----------------------
              ; BH |  BL | AH | CL
              ; bus no | device no in 5 bits | return code | 0 - success 1 -fail
              ;----------------------
              sar eax, 16
              mov ebx, 0
              mov ebx, eax   ; bh, bl has bus no and device no

              mov eax, 0
              mov ah, 0b1h  ; function  PCI BIOS p23
              mov al, 0ah   ; read config dword p19 PCI
              ;3com
              ;mov di, 10h   ; offset 10h p/65
              ; intel
              mov di, 18h
              int 01ah      ; calls PCI BIOS interrupt
```

```
mov eax, ecx
and ecx, 0fffeh        ; make the last bit zero
mov eax, ecx           ; this is the device address

push ds
push eax
mov ax, 0
mov ds, ax

mov ebx, S_Base   ;00008600h
add ebx, S_IOBASE ;009Ch
pop eax
mov DWORD PTR ds:[ebx], eax  ; store it in shared memory

mov ebx, S_Base   ;00008600h
add ebx, 58h      ;offset
mov DWORD PTR ds:[ebx], esi

pop ds
```

8.3.2 Original functions in Optiplex 260 (Intel 82540EM)

The following three functions in Optiplex 260 get modified during the driver upgrade.

1. **<u>getMACEPROM():</u>** The following code shows the internals of reading MAC address for Optiplex 260.

```
void EtherObj::getMACEPROM(char *dMAC)
{
    int retcode=0;
    int i=0;
    int temp=0;

    // EERD (EEPROM Read Register)
    //31               16 15         8 7        5   4    3      1    0
    //-------------------------------------------------------------------------
    //|        Data       |   Address  | Reserved |DONE | Reserved | START |
    //-------------------------------------------------------------------------
    // Word 00h -> Ethernet Address Byte 1 and Ethernet Address Byte 0
    // Word 01h -> Ethernet Address Byte 3 and Ethernet Address Byte 2
    // Word 02h -> Ethernet Address Byte 5 and Ethernet Address Byte 4

    // EERD 0x14
    setRegister32(0x14, IOADDR);

    // Step 1
    // ------
    // Write 00h in address field, set START to 1 and wait until DONE is 1
    // value read is in Data which is ethernet address byte0 and byte1
    setRegister32(0x00000001, IODATA);
```

```
do
{
   retcode = getRegister32(IODATA);
}while((retcode & 0x00000010) == 0);

retcode = getRegister32(IODATA);

mac[0] = (retcode & 0x00ff0000) >> 16; // mask bit(16-23) and right shift by
16 to get byte0
mac[1] = (retcode & 0xff000000) >> 24; // mask bit(24-31) and right shift by
16 to get byte1

// Step 2
// ------
// Write 01h in address field, set START to 1 and wait until DONE is 1
// value read is in Data which is ethernet address byte2 and byte3
setRegister32(0x00000101, IODATA);

do
{
   retcode = getRegister32(IODATA);
}while((retcode & 0x00000010) == 0);

retcode = getRegister32(IODATA);

mac[2] = (retcode & 0x00ff0000) >> 16; // mask bit(16-23) and right shift by
16 to get byte2
mac[3] = (retcode & 0xff000000) >> 24; // mask bit(24-31) and right shift by
16 to get byte3

// Step 3
// ------
// Write 02h in address field, set START to 1 and wait until DONE is 1
// value read is in Data which is ethernet address byte4 and byte5
setRegister32(0x00000201, IODATA);

do
{
   retcode = getRegister32(IODATA);
}while((retcode & 0x00000010) == 0);

retcode = getRegister32(IODATA);

mac[4] = (retcode & 0x00ff0000) >> 16; // mask bit(16-23) and right shift by
16 to get byte4
mac[5] = (retcode & 0xff000000) >> 24; // mask bit(24-31) and right shift by
16 to get byte5
//copy from static variable mac to passed variable dMAC
for(i=0; i<6; i++)
    dMAC[i] = mac[i];
```

```
   temp = 0;
        // copy first 4 bytes into temp and print it
        for(i=0; i<4; i++)
        {
            temp = temp | (mac[i] & 0x000000ff);
            if(i < 3)
            {
               temp = temp << 8;
            }
        }
        io.AOAprintHex(temp, Line24+60);

        temp = 0;
        // copy next 2 bytes into temp
        for(i=0; i<2; i++)
        {
            temp = temp | (mac[i+4] & 0x000000ff);
            temp = temp << 8;
        }
        // fill the remaining 2 bytes with ff so that it will be easy to read mac
address
        for(i=0; i<2; i++)
        {
            temp = temp | 0x000000ff;
            if(i < 1)
            {
               temp = temp << 8;
            }
        }

        io.AOAprintHex(temp, Line24+80);
```

2. **Transmit Enable():** The following code shows TEnable function that will be modified later to upgrade to new NICs.

```
        int EtherObj::TEnable()
        {
         int retcode=0;
        // bit1 = Transmit enable
        // make bit1 = 1
        retcode = setTCtrl(0x002000f2);
        return retcode;
        }
```

3. **Receiver Enable():** The following code shows REnable function that will be modified later to upgrade to new NICs.

```
{
     int EtherObj::REnable()

      int retcode=0;

     // bit1 = Receiver enable
     // make bit1 = 1
     // bit3 = 1 (Unicast Promiscuous Enabled)
     // bit15 = 1 (Broadcast Accept Mode)
     // bit22 = 1 (Discard Pause Frames)
     // bit26 = 1 (Strip Ethernet CRC from incoming packet)

     //retcode = setRCtrl(0x0440800a);//bit3 = 1
     retcode = setRCtrl(0x04408002); //bit3 = 0

     return retcode;
}
```

8.3.3 Upgrade to Optiplex 960

The following changes in the code are made to port to Optiplex 960.

1. **getMACEPROM()**

This function is replaced with the following code. Optiplex 960 does not have the same mechanism to read MAC address from EPROM as it has a different chipset for the controller. In order to address this problem, we obtain NIC controller address (using a Windows machine) 0xfe6e0000 and its memory location at 64. This MAC address can be used to address the NIC.

```
void EtherObj::getMACEPROM(char *dMAC)
{
 int retcode=0;
 int i=0;
 int temp=0;
 char *mptr;
 mptr = (char*)(0xfe6e0000-ADDR_OFFSET);
 for (i=0;i<6;i++)
 {
      dMAC[i]= mptr[64+i];
 }
 return ;
```

2. **Transmit Enable()**

The transmit control register values are changed from 0x002000f2 to 0x3003f0f2. Bit 28 was reserved in Optiplex 260, but needs to be 1 in Optiplex 960. Bits 29-30 are threshold bits in 960 and set to '11'. Bit 12 – 21 represening the collision distance should be 0x03f in 960 model. All other bits are same in both models.

```
int EtherObj::TEnable()
{
 int retcode=0;
// bit1 = Transmit enable
// make bit1 = 1
//Optiplex 260 retcode = setTCtrl(0x002000f2);
  retcode = setTCtrl(0x3003f0f2);  //Optiplex 960
return retcode;
}
```

3. **Receiver Enable()**

The bit 22 is 960 model is reserved and it must be 0. This is the only change in this register value.

```
int EtherObj::REnable()

 int retcode=0;

// bit1 = Receiver enable
// make bit1 = 1
// bit3 = 1 (Unicast Promiscuous Enabled)
// bit15 = 1 (Broadcast Accept Mode)
// bit22 = 1 (Discard Pause Frames)
// bit26 = 1 (Strip Ethernet CRC from incoming packet)

//retcode = setRCtrl(0x04408002);//Optiplex 260
retcode   = setRCtrl(0x04008002); //Optiplex 960

return retcode;
}
```

8.3.4 Upgrade to Optiplex 9010

The following changes in the code are made to port to Optiplex 9010.

1. **getMACEPROM ()**

   This function is replaced with the following code. Optiplex 9010 does not have the same mechanism to read MAC address from EPROM as it has a different chipset for the controller. In order to address this problem, we obtain NIC controller address (using a Windows machine) 0xf7d00000 and its memory location at 64. This MAC address can be used to address the NIC.

```
void EtherObj::getMACEPROM(char *dMAC)
{
 int retcode=0;
 int i=0;
 int temp=0;
 char *mptr;



 mptr = (char*)(0xf7d00000-ADDR_OFFSET);
 for (i=0;i<6;i++)
 {
      dMAC[i]= mptr[64+i];
 }
 return ;
```

2. **Transmit Enable ()**

   This code is same as optiplex 960.

3. **Receiver Enable( )**

   This code is same as optiplex 960.

8.3.5 Porting to HP Elite Book

The following changes in the code are made to port to Optiplex HP Elite Book.

1. **getMACEPROM ()**

   This function is same as Optiplex 9010 except the controller address is 0xd4800000.

2. **Transmit Enable ()**

   ```
   This code is same as optiplex 960.
   ```

3. **Receiver Enable()**

   ```
   This code is same as optiplex 960.
   ```

9. REFERENCES

[1] L. He, R. K. Karne, A. L. Wijesinha, and A. Emdadi, A study of bare PC Web server performance for workloads with dynamic and static content, 11th IEEE International Conference on High Performance Computing and Communications (HPCC), 2009, pp. 494-499.

[2] P. Appiah-Kubi, R. K. Karne, and A. L. Wijesinha. The design and performance of a bare PC Webmail server, 12th IEEE International Conference on High Performance Computing and Communications, (HPCC), 2010, pp. 521-526.

[3] G. H. Ford, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, The design and implementation of a bare PC email server, 33rd IEEE International Computer Software and Applications Conference (COMPSAC), 2009, pp. 480-485.

[4] B. Rawal, R. Karne, and A. L. Wijesinha, Splitting HTTP requests on two servers, 3rd Conference on Communication Systems and Networks (COMSNETS), 2011.

[5] H. Chang, R. Karne, and A. Wijesinha, Migrating a Bare PC Web server to a multi-core architecture, IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), 2016, pp. 216-221.

[6] S. Soumya, R. Guerin and K. Hosanagar, Functionality-rich vs. minimalist platforms: A two-sided market analysis, ACM Computer Communication Review, vol. 41, no. 5, pp. 36-43, Sept. 2011.

[7] G. H. Khaksari, R. K. Karne and A. L. Wijesinha. A Bare Machine Application Development Methodology, International Journal of Computers and Their Applications (IJCA), Vol. 19, No.1, March 2012, pp. 10-25.

[8] D. R. Engler and M.F. Kaashoek, Exterminate all operating system abstractions, Fifth Workshop on Hot Topics in Operating Systems, USENIX, 1995, p. 78.

[9] TinyOS Community Forum||An open-source OS for the networked sensor regime, http://www.tinyos.net, [Accessed: 28-Apr-2016].

[10] J. Lange, et. al, Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing, 24th IEEE International Parallel and Distributed Processing Symposium, Apr. 2010.

[11] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system, ACM Transactions on Computer Systems, Vol.18 (1), Feb. 2000, pp. 37-66.

[12] 3COM; 3C90xC NICs Technical Reference [Pdf]. (1999).

[13] A. Shan, Heterogeneous processing: a strategy for augmenting Moore's Law. Linux Journal (142), February 2006.

[14] W. Kim, I. Choi, S. Gala and M. Scheevel, On resolving schematic heterogeneity in multidatabase systems, Modern Database Systems, ACM Press, pp. 521-550, 1995.

[15] Linux Ethernet bonding driver HOWTO, https://www.kernel.org/doc/Documentation/networking/bonding.txt, accessed: Sep 2017.

[16] L. He, R. K. Karne, and A. L. Wijesinha, "The design and performance of a bare PC Web server", International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[17] http://news.bbc.co.uk/2/hi/technology/5107642.stm. "30 million PCs being dumped each year in the US alone," BBC News: PC users

[18] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," 6th International Conference, HPC Asia 2002, Centre for Development of Advanced Computing, Bangalore, Karnataka, India, December 2002.

[19] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIPApplication," CCNC'07, Proceedings of the IEEE Consumer and Communications and networking conference, pp. 803-807, IEEE Press, Las Vegas, Nevada, January 2007

[20] A. Alexander, A. L. Wijesinha, and R. Karne. "A Study of Bare PC SIP Server Performance," The Fifth International Conference on Systems and Networks Communications. ICSNC 2010, August 22-27, Nice, France.

[21] G. Ford, R.K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. "The performance of a Bare Machine email server," SBAC-PAD"09, 21st International Symposium on Computing Architecture and High Performance Computing, pp. 143-150, IEEE, October 2009.

[22] A. Emdadi, R. K. Karne, and A. L. Wijesinha. "Implementing the TLS Protocol on a Bare PC," ICCRD2010, The 2nd International Conference on Computer Research and Development, Kaula Lumpur, Malaysia, May 2010.

[23] R. Yasinovskyy, A. L. Wijesinha, R. K Karne and G. Khaksari. "Comparison of VoIP Performance on IPv6 and IPv4 Networks", The 7th ACS/IEEE International Conference on Computer Systems and Applications I(AICCSA), 2009

[24] B. Rawal, R. K. Karne, and A. L. Wijesinha. "Mini Web server clusters for HTTP request splitting," IEEE Conference on High Performance, Computing and Communications (HPCC), 2011, pp. 94-100.

[25] P. Appiah-Kubi, R.K. Karne and A.L. Wijesinha, "A Bare PC TLS Webmail Server," Proc. Of IEEE Workshop on Computing, Networking and Communications, ICNC-CNC, January 2012, Maui, HI, pp. 156-160.

[26] A. Kadav and M. M. Swift, Understanding modern device drivers, 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[27] A. Amar, S. Joshi and D. Wallwork, Generic Driver Model, Available: http://www.design reuse.com/articles/a8584/generic-driver-model.html, [Accessed: 28-Apr-2016].

[28] V. Chipounov and G. Gandea, Reverse engineering of binary device drivers with RevNIC, 1st EuroSys Conference on Systems, 2006.

[29] J. LeVasseur, V. Uhlig, J. Stoess and S. Gotz, Unmodified device driver reuse and improved system dependability via virtual machines, 6th Symposium on Operating Systems Design and Implementation, 2004.

[30] P. Salvatore, "The new minimalist operating systems", https://blog.docker.com/2015/02/the-new-minimalist-operating-systems/, accessed: Sep 2017.

[31] R.K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ applications on a bare PC," 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD), 2005, pp. 50-55.

[32] H. Tran-Viet, et. al, "Experimental study on the performance of Linux Ethernet bonding", Inernational Conference on Testbeds and Resaerch Infrastructures: Development of Networks and Communities (TridentCom), 2014, pp. 307-317.

[33] Network Bonding, https://docs.oracle.com/cd/E27300_01/E27309/html/vmusg-network-bonding.html, accessed: Sep 2017.

[34] Intel; PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual.

[35] http_load-multiprocessing http test client, http://www.acme.com/software/http_load/, accessed: Sep 2017.

[36] Network Driver Interface Specification. Retrieved from https://technet.microsoft.com/en-us/library/cc958797.aspx#mainSection

[37] N. Brown, Linux drivers in user space — a survey, Oct 2016, https://lwn.net/Articles/703785/Online

[38] H. Koch, Userspace I/O drivers in a realtime context, 11th Real-Time Linux Workshop, OSADL, 2009.

[39] S. Hemminger, Networking in Userspace, 2013, https://www.slideshare.net/shemminger/uio-final, Online.

[40] User Space Networking Fuels NFV Performance, 2015, https://software.intel.com/en-us/blogs/2015/06/12/user-space-networking-fuels-nfv-performance, Online.

[41] M. Majkowski, Why we use the Linux Kernel's TCP Stack, 2016, https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/, Online.

[42] S. Boyd-Wickizer and N. Zeldovich, Tolerating Malicious Device Drivers in Linux, USENIX Annual Technical Conference (USENIX ATC'10), 2010.

10. CURRICULUM VITAE

Faris Abdullah Al-Mansour
[redacted]

OBJECTIVE

Seeking for an academic position where my teaching and research experience in addition to the intellectual potential can be utilized in a creative environment at an academic institution.

EDUCATION

- Doctoral of Science in Information Technology at
  Towson University Towson, MD, USA
  Anticipated graduation May 2018

- Master of Science Information Technology from
  NC A&T State University in Greensboro, NC, USA May 2013
- Bachelor of Science Information Technology& Computing
  Arab Open University, Riyadh, Saudi Arabia Sept 2008

Doctoral Dissertation

My dissertation focused on an in-depth study of Ethernet device drivers used in bare machine computing applications including their design, implementation and performance. As a result of my research, it becomes possible to integrate the Ethernet driver architecture with the bare PC Web server and Web client, as well as other applications. More generally, my research shows that the architecture of the bare PC allows the programmer to integrate many other components and improve the integrated components with ease. My work required a thorough understanding of the Bare Machine Computing(BMC) application development process. My research was based on previous work to integrate a network Ethernet card with BMC applications in order to communicate. I extended the earlier work, expanded it to include more than one NIC, and enabled device drivers to work with more than one application within the BMC paradigm.

Research Experience

Over a four year period, my research has provided me with a wealth of experience and knowledge. During this time, I have explored many areas in both research and academic fields where I learned how to logically identify and solve problems in the real world. My area of research enabled me to master various network protocols and concepts including HTTP, TCP/IP, and Ethernet. It also provided me with valuable experience working with switches and routers when building test networks to conduct experiments, measure performance, and address a variety of practical issues. Overall. I have enriched my knowledge theoretically and technically in the field of networking and operating systems. In addition, other skills that I have mastered include programming with the C and C++ language in a BMC environment, identifying research issues, solving technical problems, planning and executing research plans, writing peer reviewed papers, and making conference presentations.

Teaching Experience

- Instructor Spring 2016 – Present

  - Teaching COSC 111 (Information Technology for Business) This course is an introduction to the use of Information Technology in a business environment where students will be exposed to and learn the key elements of computer-based technology. Topics include computer technology used to retrieve, filter, classify, process, sort, and evaluate information. Problem-solving, creative thinking, effective communication, team building, and professional ethics within an information system environment are stressed.

  - Teaching COSC 109 (Computers and Creativity). The course gives student experience with creative activities involving symbolic manipulation and computer graphics, animation, dynamic storytelling, computer music, visual effects, Web publishing, artwork, and multimedia.

Research Publications

1. Faris Almansour, Ramesh K. Karne, Alexander L. Wijesinha, Bharat S. Rawal "Ethernet Bonding on a Bare PC Web Server with Dual NICs", The 33rd ACM Symposium On Applied Computing SAC 2018, April 2018, Pau, France.
2. Hamdan Z. Alabsi, W. V. Thompson, R. K. Karne, Alexander Wijesinha, Rasha. Almajed, F. Almansour, A Bare Machine RAID File System for USBs, SEDE 2017: 26th International Conference on Software Engineering and Data Engineering, pp 113-118.
3. Faris Almansour, Ramesh Karne, Alexander Wijesinha, Hamdan Alabsi, and Rasha Almajed "Middleware for NICs in Bare PC Applications", The 26th International

Conference on Computer Communications and Networks Jul 2017, Vancouver, Canada

4. Kateeb, I. A., & Burton, L., & El-Bathy, N., & Almansour, F. A. (2012, June), *Future Energy and Smart Grid* Paper presented at 2012 ASEE Annual Conference & Exposition, San Antonio.

Conferences/Workshop

- Career & Leadership Development Conference Greensboro, NC, USA April 2012
- Dominion Power Renewable Energy Protection Workshop July 2012
- Cisco Corporate Technology Day in Raleigh, NC, USA Nov 2012
- Conflict Management Training Institute in Greensboro, NC, USA Nov 2012
- Center for Energy Research Technology (CERT) Poster Presentation Nov 2012
- National Instruments Technical Symposium Energy and Power Workshop Dec 2012
- IEEE Early College at GTCC Renewable Energy Workshop, Dec 2012