INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeib Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0590
Implementing Multicast in a Software Emulation of the
Virtual Interface Architecture

by

Damian Dobric

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Master of Science

APPROVED, THESIS COMMITTEE:

John K. Bennett, Associate Professor, Chair
Electrical and Computer Engineering

Joseph R. Cavallaro, Associate Professor
Electrical and Computer Engineering

J. R. Jump, Professor
Electrical and Computer Engineering

Houston, Texas
May, 2000
ABSTRACT

Implementing Multicast in a Software Emulation of the
Virtual Interface Architecture

by

Damian Dobric

The Virtual Interface Architecture (VIA) is an emerging standard for low-latency, high-bandwidth, user-level communication designed to achieve high performance by minimizing data copying and kernel/user transitions. Currently very few network controllers provide VIA support, and the current specification for VIA does not include multicast, a useful mechanism for distributed applications.

This thesis tests two ideas by experiment. Whether a software implementation of VIA can provide useful performance enhancement, and whether multicast support can be incorporated into VIA with tangible benefit. I designed a Windows NT driver software implementation of VIA for Gigabit Ethernet that achieved an average of 57% lower latency than Ethernet (UDP) for messages of one to 64K bytes. These low-level benefits translated to a reduction in execution time of 10-14% over UDP for several distributed applications, and with multicast, an additional reduction of 1% to 15%. We conclude that multicast support would be a useful extension to the VIA specification that could be added without difficulty.
ACKNOWLEDGEMENTS

First, I would like to thank God for giving me the opportunity to come to Rice and have this experience. I am also very grateful to my parents for always encouraging me to continue learning.

I especially thank my advisor John Bennett for being very supportive of me in completing this thesis research. I would also like to thank my committee members, J. R. Jump and Joe Cavallaro for reviewing my thesis and providing useful comments.

I thank Hazim Shafi for the unbelievable amount of help he has given me, interesting discussions that we have had, and for being very supportive of me in finishing this research. I also thank Evan Speight for all the help he has also given me along the way. I thank my officemate Bill Balabanos for his company during the many late nights while I was completing this work.

I would like to thank Compaq Computer Corporation for all of their equipment donations, especially the video switches, without which I would have surely ended up in a mental ward.

Finally, I thank my friend Norman and his wife for their encouragement and support in the last couple months while I have been finishing my thesis, and for all the wake-up calls without which I would have surely slept for months. I also thank my friend Christa Clark for her encouragement and support as well as all my other friends and fellow graduate students here at Rice. Thanks go to Bahareh & Farbod for their being continually helpful.
# Table of Contents

1. **INTRODUCTION** ........................................................................................................ 1

2. **VIRTUAL INTERFACE ARCHITECTURE OVERVIEW** ............................................. 5
   
   2.1. **VI ARCHITECTURE COMPONENTS** .................................................................. 6
       2.1.1 Virtual Interfaces ......................................................................................... 6
       2.1.2 Completion Queues ..................................................................................... 7
       2.1.3 VI Provider .................................................................................................. 8
   
   2.2. **MEMORY REGISTRATION** ............................................................................ 9
   
   2.3. **CONNECTION PROCEDURES** ..................................................................... 10
   
   2.4. **DATA TRANSFER MODES** .......................................................................... 10
       2.4.1 Send / Receive ............................................................................................. 11
       2.4.2 Remote Direct Memory Access (RDMA) ....................................................... 11
   
   2.5. **RELIABILITY LEVELS** ................................................................................ 12
   
   2.6. **EMULATING VI-NIC OPERATION** ................................................................. 14

3. **VIRTUAL INTERFACE ARCHITECTURE PROVIDER LIBRARY (VIPL)** ............ 15

4. **WINDOWS NT ARCHITECTURE DRIVER-RELATED DETAILS** ......................... 20
   
   4.1. **WINDOWS NT I/O SUBSYSTEM** ................................................................... 20
       4.1.1 Describing I/O Requests .............................................................................. 23
           4.1.1.1 I/O Request Packets (IRPs) ................................................................. 23
       4.1.2 Describing Requestor Data Buffers .............................................................. 24
           4.1.3 I/O Function Codes & I/O Control Functions .......................................... 24
   
   4.2. **WINDOWS NT LAYERED DRIVER I/O MODEL** ........................................... 25
4.2.1 Driver Types ........................................................................................................... 26
  4.2.1.1 Kernel-Mode Drivers ......................................................................................... 26
  4.2.1.2 Intermediate Drivers ......................................................................................... 26
  4.2.1.3 NDIS Miniport Drivers ...................................................................................... 26
  4.2.2 Device Driver Stacking (Layering) ...................................................................... 27
  4.2.3 Inter-Driver Communication Alternatives .............................................................. 29
  4.2.4 Fast I/O .................................................................................................................. 29

5. VI IMPLEMENTATION .................................................................................................... 31
  5.1 DESIGN OVERVIEW .................................................................................................. 32
  5.2 CONNECTING TO DRIVER .......................................................................................... 33
  5.3 VI ENDPOINT CREATION/DESTRUCTION ................................................................. 34
  5.4 VI CONNECTION MANAGEMENT .............................................................................. 35
  5.5 MEMORY PROTECTION/REGISTRATION ................................................................. 37
  5.6 DATA TRANSFER & COMPLETION OPERATIONS .................................................... 38
    5.6.1 Sending Data ......................................................................................................... 38
    5.6.2 Receiving Data ..................................................................................................... 41
    5.6.3 Completion Mechanisms ..................................................................................... 44
  5.7 COMPLETION QUEUE MANAGEMENT ..................................................................... 48
  5.8 ERROR HANDLING ................................................................................................... 49
  5.9 MULTITHREAD SYNCHRONIZATION ...................................................................... 50
  5.10 HARDWARE SUPPORT FOR VI ON PEGNIC ......................................................... 51

6. MULTICAST PROTOCOL OVERVIEW ............................................................................ 54
  6.1 IP MULTICAST .......................................................................................................... 55
# Table of Contents

7. **Multicast VI Implementation** ................................................................. 58
   7.1. **Additional Support in VIPL** .............................................................. 59
   7.2. **Connecting to Driver** ................................................................. 60
   7.3. **VI Multicast Connection Management** ......................................... 62
       7.3.1 Setting Up a VI Multicast Connection ......................................... 62
       7.3.2 Tearing Down a VI Multicast Connection .................................... 64
       7.3.3 Multicast Address Receive Filtering ........................................... 64
   7.4. **Data Transfer** ................................................................................. 66
   7.5. **VI Hardware Support for Multicast** ............................................... 67
   7.6. **Extending Brazos to Use VI Multicast** .......................................... 68
       7.6.1 MC Address Distribution ................................................................. 69
       7.6.2 Allocating VI Buffers ................................................................. 70
       7.6.3 Setting Up Multicast Connections .............................................. 71

8. **Experimental Methodology** ................................................................. 76
   8.1. **Performance Benchmarks** .............................................................. 76
       8.1.1 Bandwidth and Latency ................................................................. 76
       8.1.2 Time to Service Multiple Requests .............................................. 77
   8.2. **Brazos DSM Application Benchmarks** ............................................. 77

9. **Experimental Results** ........................................................................... 81
   9.1. **Low-Level Benchmark Results** ...................................................... 81
       9.1.1 Observed Bandwidth and Latency ................................................. 81
   9.2. **VI Multicast** .................................................................................. 84
   9.3. **Brazos DSM Application Results** .................................................. 86
9.3.1 3D-FFT ................................................................................................. 87
9.3.2 Barnes Hut ............................................................................................ 91
9.3.3 LU ........................................................................................................ 92
9.3.4 Water .................................................................................................... 93
9.3.5 SOR ........................................................................................................ 94

9.4. SUMMARY OF RESULTS ......................................................................... 95

10. RELATED WORK .......................................................................................... 98

10.1. USER-LEVEL NETWORK ACCESS TECHNIQUES .................................. 98
  10.1.1 U-Net ............................................................................................... 98
  10.1.2 Virtual Memory-Mapped Communication (VMMC) ......................... 102
  10.1.3 Active Messages ............................................................................. 106
  10.1.4 OS Mechanisms ............................................................................. 107
  10.1.5 VI Implementations ...................................................................... 109
  10.1.5.1 VI Proof-of-Concept Tests .......................................................... 109
  10.1.5.2 BobNet ..................................................................................... 110
  10.1.5.3 Other VI Work ........................................................................ 111

10.2. MULTICAST ............................................................................................. 111
  10.2.1 IP Multicast in ATM Environment .................................................. 111
  10.2.2 Reliable Multicast Protocols ............................................................ 113
  10.2.3 Vulnerabilities of Reliable Multicast Protocols ............................... 114
  10.2.4 Secure Agreement Protocols ........................................................... 115

11. CONCLUSIONS AND FUTURE WORK .................................................. 117

REFERENCES ................................................................................................. 120
List of Figures

Figure 2.1 Block Diagram of the Virtual Interface Architecture.............................6
Figure 2.2 Virtual Interface..................................................................................7
Figure 2.3 Completion Queue..............................................................................8

Figure 4.1 I/O Subsystem and User Application Relationship...............................21

Figure 5.1 Layered Driver Design.........................................................................32
Figure 5.2 Inter-Driver Communication Via a Common Structure.........................33
Figure 5.3 Connection Setup Process....................................................................36
Figure 5.4 Measured Latency with Varying DMA Size...........................................40
Figure 5.5 Measured Latency with Varying Rx/Int................................................44
Figure 5.6 Example Synchronization between User and Kernel..............................47
Figure 5.7 VI States............................................................................................50
Figure 5.8 PEGNIC Block Diagram.......................................................................51
Figure 5.9 Modified Hamachi Core and Memory Interface.....................................52

Figure 6.1 Unicast...............................................................................................54
Figure 6.2 Multicast............................................................................................54
Figure 6.3 Multicast Address Format.....................................................................55
Figure 6.4 Multicast Distribution..........................................................................56

Figure 7.1 IP Multicast to Ethernet Address Mapping............................................62
Figure 7.2 CAM Multicast Address Filter Structure..............................................65
Figure 7.3 One-to-Many VI Communication.........................................................67
Figure 7.4 Procedure for setting up MC Connections............................................72
Figure 7.5 VI Multicast Connection Setup Procedure............................................75

Figure 9.1 Observed Throughput with Varying Message Size...............................83
Figure 9.2 Observed Latency vs. Ideal Latency (wire time)....................................84
Figure 9.3 Time to Service Multiple Requests......................................................85
Figure 9.4 3D-FFT Relative Execution Times.........................................................88
Figure 9.5 Barnes Hut Relative Execution Times.....................................................92
Figure 9.6 LU Relative Execution Times...............................................................93
Figure 9.7 Water Relative Execution Times...........................................................94
Figure 9.8 SOR Relative Execution Times.............................................................95

Figure 10.1 U-Net Interface Diagram.................................................................99
Figure 10.2 U-Net Emulated Endpoints...............................................................100
Figure 10.3 Automatic Update Mapping.............................................................103
Figure 10.4 Mapping Receive Buffer into Sender’s Destination Space.................104
List of Tables

Table 3.1  VI Provider Library Routines ................................................................. 16
Table 7.1  PEGNIC Driver Add/Remove Multicast Address Handlers .................. 65
Table 8.1  Shared Memory Application Data Set Sizes ........................................... 77
Table 9.1  Message Distributions for DSM Benchmarks ......................................... 87
Table 9.2  Reduction in Total Execution Times ....................................................... 89
Table 9.3  Reduction in Time Spent Waiting for Data ............................................ 90
Table 9.4  Reduction in Number of Sent Messages ............................................... 91
1. Introduction

Despite significant increases in network performance over the last two decades, message latencies seen by end-user applications have experienced little corresponding performance improvement. Network access remains expensive primarily because network resources have traditionally been managed exclusively by the kernel, with the host operating system multiplexing access to the network hardware across process-specific endpoints set up through a standard set of network APIs. In this type of network protocol, nearly every network access requires one or more traps or function calls into the kernel, significantly increasing the amount of overhead associated with network API calls. Copying data into and out of kernel data buffers, both within device drivers and network protocol stacks, remains another significant performance detriment to network high performance.

Academic research into user-level communication mechanisms including U-Net [10], VMMC [42], Active Messages [51], application device channels [7] and fbufs [21] prompted Compaq, Intel, and Microsoft to jointly develop the Virtual Interface (VI) Architecture Specification [4]. The VI Architecture Specification describes the mechanisms needed to implement a network that provides user-level data transfer intended to achieve low message latencies, even for small messages. Although support for the VI architecture is intended to be provided in NIC (Network InterConnect) hardware, very few commercial products currently support “VI-aware” network interfaces. The majority of VI implementations to date have been implemented in software, usually implemented as a device driver “Kernel Agent”, which supplies
software emulation of VI hardware. Software implementations of VI are currently available on ServerNet (Tandem), Myrinet (Myricom), and Ethernet (Intel).

Multicast is often a useful communication primitive in parallel programming and distributed computing systems. Multicast has been shown to be useful in implementing parallel languages. Languages such as Orca [3] use multicasting in their implementation for replicating global information. The Brazos DSM system [17] uses multicast to explicitly reduce the amount of coherence traffic on the network, as opposed to its use in Orca for data object replication.

In order to simplify implementation and deployment, the VI Architecture Specification does not support multicast. Implementing an efficient, reliable mechanism for multicast in software is difficult, and there are a number of problems involved, such as buffer management and flow control. These issues have been addressed in implementations such as Fast Messages on Myrinet [49], but the added reliability code has been shown to introduce significant overhead.

One of the goals of this research was to explore the viability of multicast support for the Virtual Interface Architecture. While the VI Architecture has been shown to reduce the latency of messages in small networks, many non-VI networking protocols include support for multicast, offering an apparent advantage over VI with their ability to send data to multiple destinations at once. We contend that the omission of multicast reduces the effectiveness of VIA when compared to other networking protocols that include multicast. If multicast can be implemented efficiently the associated potential performance benefits should make the VI Architecture an attractive solution to communication in clustered computing networks.
To test this idea I developed a prototype software emulation of the Virtual Interface Architecture for use with the Packet Engines GNIC II (PEGNIC) Gigabit Ethernet card, including the addition of software support for one-to-many communication. I have designed a Kernel Agent driver that augments the existing Packet Engines driver to allow support of the VI Programming Library (VIPL), while maintaining backwards-compatibility with the existing Winsock functionality provided by the PEGNIC driver.

This thesis is organized as follows. I first describe the VI Architecture and the VI Programming Library that provides the user with the network access API functions. In Section 4, I describe some key issues in the Windows NT I/O subsystem, as well as in the layered device driver model that Windows NT employs. My device driver is a tightly-coupled intermediate driver, interfacing with both the PEGNIC driver and the Network Driver Interface Specification (NDIS) library, and its design is described in detail in Section 5. In Sections 6 and 7, I describe the operation of IP multicast and how I incorporated this concept into the connection-oriented VI architecture. I also explore the design issues encountered when I extended my software emulation of VIA to include one-to-many communication. In addition, I describe the use of VI multicast connections in Brazos, a distributed shared memory system for clusters of PCs running Windows NT. Buffer management is also considered. In Sections 8 and 9, I describe low-level and distributed application experiments that demonstrate the performance potential of my VIA software emulation. I also measure the performance benefit obtained using multicast, and present an evaluation of the results of these experiments.
My VI implementation allows for both Winsock and VI to be used simultaneously, providing maximum flexibility in choosing network protocols. The driver performs zero-copy send operations and single-copy receive operations. The resulting performance of the layered driver is an average of 57% better in terms of latency than Ethernet (UDP) performance provided by WinSock for message sizes from 8 to 64 Kbytes. In addition, my software emulation of VIA achieves a 10-14% reduction in application execution time for Brazos DSM applications that have high communication rates. The multicast extensions to VIA developed as part of this work proved to be useful in applications that do not have pair-wise sharing and that share data with several other nodes. In addition, the multicast support does not introduce unnecessary overhead, and was relatively easy to incorporate with VIA. My results show that multicast is a useful extension to the VI Architecture that merits consideration for incorporation into the VIA protocol specification.
2. Virtual Interface Architecture Overview

The Virtual Interface (VI) Architecture is a network OS interface that provides the user with more direct access to the network than standard OS protocol stack interfaces, while maintaining the same level of inter-process protection. The VI Architecture uses hardware support to provide direct, virtualized access to the interface without kernel intervention. The design of the VI Architecture addresses three performance problems associated with network communication:

- Low bandwidth – Network-related software overhead limits the usable bandwidth of the network, resulting in utilization of only a fraction of the possible network bandwidth.
- High latency of synchronization messages – Because processes in a distributed system must synchronize using network messages, high latency for (typically) small synchronization messages can significantly reduce overall performance.
- Host processing load – The overhead of message processing can reduce processing time that the CPU could otherwise dedicate to application code.

VI Architecture implementations attempt to reduce these overheads by avoiding a kernel trap on every network call. Instead, the user is provided with a direct, protected network interface to the Network Interface Card (NIC) without having to perform a kernel operation to send and receive data. This direct interface is referred to as a virtual interface (VI) and is analogous to the socket endpoints in a typical TCP connection. Each VI is bi-directional and provides point-to-point transfer of data.
2.1. VI Architecture Components

The Virtual Interface Architecture is comprised of four basic components (Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers), as shown in Figure 2.1. The VI Provider consists of a software Kernel Agent and a VI network adapter. The Virtual Interfaces and Completion Queues are entirely placed on the VI network adapter in a normal hardware implementation of VI, as will be described further in Section 2.1.3. The VI Consumer is typically an application program in conjunction with an operating system communication facility such as sockets or MPI that uses the VI Provider's interface, although some applications that are VI-aware interface directly with the native VI API.

![Block Diagram of the Virtual Interface Architecture](image)

**Figure 2.1 Block Diagram of the Virtual Interface Architecture**

2.1.1 Virtual Interfaces

A VI is composed of two queues (Send and Receive) used for posting requests in the form of Descriptors for sending and receiving data. Descriptors contain pointers to data buffers as well as any other information required by the VI Provider to complete requests. VI Providers asynchronously process posted descriptors by sending the
information contained in a data buffer over the network or copying received information to a data buffer and setting a flag in the descriptors when these send or receive operations complete. The VI Consumers remove completed descriptors from the Send and Receive queues and re-post them to handle subsequent requests. Both the Send and Receive Queues have an associated “Doorbell” that is used to notify the network adapter that a new descriptor has been posted to either queue. In a hardware implementation of the VI Architecture, the NIC contains this doorbell and no kernel intervention is required.

![VI Consumer Diagram](image)

**Figure 2.2 Virtual Interface**

### 2.1.2 Completion Queues

A completion queue mechanism is provided that allows the VI Consumer to combine descriptor completion notifications for multiple VIs by checking a single queue without requiring a kernel call or interrupt. When the VI is created, it is associated with a
Completion Queue and from that point on all completion synchronization takes place on that Completion Queue. When a descriptor is completed on an associated VI's receive or send queue, an entry is also placed on the Completion Queue corresponding to the VI whose queue has a completion, as well as an indication as to whether or not it is a send or receive. Thus, a VI Consumer must poll or wait on a Completion Queue until it finds an entry and then retrieve the completed descriptor from the appropriate queue.

![Completion Queue Diagram]

**Figure 2.3 Completion Queue**

2.1.3 VI Provider

The VI Provider is a set of hardware and software components that provide all of the functionality of instantiating and managing Virtual Interfaces. The VI Provider generally consists of a VI-aware network interface controller (NIC) and a software Kernel Agent.

The VI-NIC contains memory structures that implement all of the send and receive queues for Virtual Interfaces and the Completion Queues as well as directly
handling send and receive operations. Although using a VI-NIC to handle all of these queues was the original intent of the VI design because it provides the highest performance for time-critical data operations, an emulation of this functionality can be designed using an alternative NIC and a special device driver. This is discussed in more detail in Section 2.6.

The Kernel Agent is a privileged part of the operating system that is generally a device driver supplied by a VI-NIC vendor and performs all of the operations necessary to setup and maintain VI connections between the VI consumer and the NIC. Standard operating system mechanisms are used by a VI Consumer to access the Kernel Agent in the VI Provider. The operations that require a trap into the kernel in order to access the Kernel Agent are: VI creation and destruction, VI connection setup and teardown, interrupt processing, registration of system memory used by the VI NIC, and error handling. Since the Kernel Agent is not required for network send and receive routines (when using a VI-NIC), the overhead of trapping to the kernel does not impede the performance during actual data transfer. This results in latencies significantly lower than standard network protocols such as TCP/IP.

2.2. Memory Registration

A large portion of the overhead in traditional network protocol stacks results from copying data between user and kernel buffers. The VI Architecture eliminates this problem by requiring the VI Consumer to register its send and receive buffers with the VI Provider. This registration process locks down pages in user memory, allowing DMA operations to move data without kernel intervention and without the possibility of page faulting. Once the pages are locked in memory, an opaque handle is provided to the VI
Adapter for each registered memory region. This handle is used on subsequent accesses to the memory region. Memory registration helps to improve the performance of sending and receiving data by allowing the VI Consumer to reuse registered memory buffers and avoid duplication of locking and translation procedures. In addition, memory registration takes page-locking overhead out of the performance-critical data transfer path. Memory registration is usually performed once for each memory region at the beginning of execution of an application.

2.3. Connection Procedures

Virtual Interfaces provide connections that are similar to those provided by standard connection-oriented networking protocols such as TCP. The programmer makes traditional operating system calls to the Kernel Agent in order to create a VI and connect it to a corresponding VI on a remote system. Once a connection is established between a local and a remote VI, send and receive requests are simply posted directly to the local VI. Setting up and tearing down VIs is a relatively expensive process and is typically only performed at the start and end of program execution.

2.4. Data Transfer Modes

The VI Architecture provides two different modes of data transfer: traditional send / receive, and remote memory reads and writes (RDMA). These are described in detail in this section.
2.4.1 Send / Receive

Sending and receiving data between VIs follows the common endpoint approach, except that all send and receive operations complete asynchronously. The VI Consumers on both the sending and receiving sides of the connection specify the data buffer locations. The VI Consumer on the sending side has to specify the location of the data buffer that contains the data to be sent to the remote VI. The process on the receiving side must specify the data buffer location to place any data that it receives. In addition, the VI Consumer at the receiving end must post a descriptor on its Receive Queue before the remote VI Consumer at the sending end can post a message to be sent. The send / receive mode requires that VI Consumers are notified when a descriptor completes (send or receive) in order to synchronize properly.

2.4.2 Remote Direct Memory Access (RDMA)

RDMA operations allow a process to read data from or write data to another node’s address space without the remote node taking any action other than the initial connection setup. The initiator of the data transfer specifies both the source buffer and the destination buffer for the data transfer.

For an RDMA Write operation, the VI Consumer specifies the local source data in a registered memory region, and the destination of the remote data that has been registered by the remote VI consumer. The local data to be sent can consist of a list of buffers, but the remote destination must be a single, virtually contiguous region. This is to make the operation simpler so the VI provider does not have to manage many virtual addresses and memory handles on the remote end. The RDMA Write operation implies that before the operation has been requested the remote VI Consumer has informed the
initiator of the virtual address of a registered memory region to be used for the operation, specified by a virtual address and memory handle, and the region is enabled for RDMA. Also note that no descriptor is used on the remote node’s receive queue by RDMA operations and the remote node is not informed about an RDMA operation completion, but a descriptor must be posted on the initiator node’s send queue to request the operation. If an RDMA Write request specifies Immediate Data, a descriptor will be used on the remote end’s receive queue when the data transfer completes to insure synchronization. Thus, the VI Consumer on the remote end must have already posted a receive descriptor before the sender initiates an RDMA Write. An error will be posted and the connection can be broken if there are no RDMA enabled descriptors posted.

An RDMA Read operation is performed in a very similar manner to an RDMA Write operation. The initiator again must already know the virtual address and memory handle for the registered remote region it wants to read from, and it must specify the local registered region to place the transferred data. The operation is started by placing a descriptor on the send queue and also a descriptor on the receive queue to receive the data from the operation. Unlike RDMA Write, immediate data is not allowed for an RDMA Read.

2.5. Reliability Levels

There are three levels of communication reliability supported by the VI Architecture at the NIC level: Unreliable Delivery, Reliable Delivery, and Reliable Reception. It is a requirement that a VI NIC at least support the Unreliable Delivery Level, but support for one or both of the higher levels is recommended because it allows
for the consumer software to be less complex. The reliability level is specified as an attribute for a VI, and only VIs with the same level can be connected.

An Unreliable Delivery VI guarantees that a Send or RDMA Write is delivered at most once (only one receive descriptor consumed) and corrupt transfers are detected on the receiving side. Sends and RDMA Writes may be lost and this condition is not detected nor are retransmissions implemented. Additionally, requests are not guaranteed to be delivered in order by the sender. It is noted that RDMA Writes should only be used with unreliable connections if the application can tolerate late arrival of data or loss of data, however RDMA Reads are not allowed on an Unreliable VI and will result in a descriptor format error. Finally, an unreliable VI will not break its connection with a remote VI if a request processing error is detected, the VI Consumer is simply informed and the VI does not transition to the Error state.

A Reliable Delivery VI guarantees that any data submitted for a transfer will arrive at the remote destination exactly once, intact, in the order submitted, and without errors. The VI provider will deliver an error to the Consumer if one of the above is not satisfied, but it is considered a catastrophic error if this happens since it should be extremely rare. As a result of such an error being detected, the VI transitions to the Error state, its connection is broken, and all posted descriptors are completed with an error status. In addition, any error that occurs on the initiating system will result in the associated descriptor being completed with an error status. Any error that occurs after a descriptor has been completed will be delivered to the VI Consumer asynchronously using a callback setup when the VI Consumer gets a handle to the NIC.
A Reliable Reception VI behaves just as a Reliable Delivery VI with the following exception: A descriptor will only be completed with successful status when the data has been copied to the target memory location, and if an error occurs it will be reported through the descriptor status. This guarantees that no subsequent descriptors will be processes after a descriptor that causes an error.

2.6. Emulating VI-NIC Operation

In order to emulate the functionality of a VI NIC, essentially all of the VI queues and Completion Queues in Figure 2.1 have to be stored in host memory and managed by the Kernel Agent, which should be a device driver. Depending on how this device driver is implemented, this could require a kernel trap on every operation for a VI, or the device driver can share some memory with the user in order to avoid this overhead. The basic overview of an emulated VI system consists of the VI Provider Library (VIPL) at the highest level linked with the user application, then the device driver that will field requests from the VIPL, and finally the NIC that will do the actual data sending and receiving in some type of format. Depending on the capability of the NIC and again on the device driver software, this can also introduce more copying of the data from registered memory locations to packets and vice versa. In the next section information on the VIPL will be provided and in Section 5 all of these issues and how they were solved will be discussed in more detail.
3. Virtual Interface Architecture Provider Library (VIPL)

At the highest level of any VI implementation is the Virtual Interface Architecture Provider Library (VIPL). This library can be either used directly by a user application or by another communication library that the application uses, but in any case it provides the user process with the means of communicating directly with the NIC. Table 3.1 contains a listing of all of the routines that the VIPL provides to the user, and a brief description of what functionality each procedure provides. Refer to the Virtual Interface Architecture Specification [5] for more detailed information on each call and its specific parameters and return values. Notice that for each routine there is a note as to whether or not the routine incurs one or more kernel traps. This is specific to my design, since I am replacing the VI-aware NIC with a device driver that implements all of the VI components. All of the routines that require kernel traps are those that either can’t avoid a kernel trap because they are creating a VI or a CQ, etc. and need to setup structures in the driver, or they are not time-critical operations because they are only done at initialization and at the end of a program and do not need to be extremely fast. The one exception to this is VipPostSend( ), for reasons which will be explained in Section 5. Specific details about the Windows specific mechanisms the VIPL uses to call the driver for all of the API routines that require a kernel trap will be described in Section 4 and the details as to how the driver handles each of these calls will be described in Section 5.

The categories that the VI API have been organized into are the following: hardware connection, VI instance creation and destruction, connection management, memory protection and registration, data transfer and completion operations, completion queue management, querying information, and error handling. The categories are self-
explanatory so I will leave it to the reader to refer to Table 3.1 to gain an understanding of the functionality that each API routine in these categories provides.

Table 3.1 VI Provider Library Routines

<table>
<thead>
<tr>
<th>Service Category</th>
<th>VIPL Routines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Connection</td>
<td><strong>VipOpenNic()</strong> <em>(kernel trap)</em></td>
<td>Associates a process with a VI NIC, and returns a NIC Handle. The Handle is used in subsequent routines to specify this particular NIC. In this implementation the handle is actually a handle to an instance of the Kernel Agent device driver</td>
</tr>
<tr>
<td></td>
<td><strong>VipCloseNic()</strong> <em>(kernel trap)</em></td>
<td>Removes the association between the calling process and the VI NIC that was setup previously in VipOpenNic().</td>
</tr>
<tr>
<td>VI Creation / Destruction</td>
<td><strong>VipCreateVi()</strong> <em>(kernel trap)</em></td>
<td>Creates an instance of a VI for the specified NIC, setting up its specified attributes and associating one or both Work Queues with a Completion Queue, and setting its state to Idle. A VI Handle is returned to the user.</td>
</tr>
<tr>
<td></td>
<td><strong>VipDestroyVi()</strong> <em>(kernel trap)</em></td>
<td>Tears down a VI, only if it is in the Idle state and all Descriptors have been de-queued. Use of the destroyed VI Handle on a subsequent operation will fail.</td>
</tr>
<tr>
<td>Connection Management</td>
<td><strong>VipConnectWait()</strong> <em>(kernel trap)</em></td>
<td>Used to look for incoming requests by specifying a local network address used to filter requests. Waits for a request until Timeout period has expired. If a request is found that matches the local address, the remote address of the requestor and a connection handle for subsequent calls to VipConnectAccept() or Reject() is returned.</td>
</tr>
<tr>
<td></td>
<td><strong>VipConnectAccept()</strong> <em>(kernel trap)</em></td>
<td>Used to accept a connection request and associate the connection with a local VI. If the attributes of the local VI do not match the attributes of the remote VI, the call fails and the remote VI is not notified.</td>
</tr>
<tr>
<td></td>
<td><strong>VipConnectReject()</strong> <em>(kernel trap)</em></td>
<td>Used to reject a connection request; notification is sent to the machine that made the request.</td>
</tr>
<tr>
<td></td>
<td><strong>VipConnectRequest()</strong> <em>(1 or 2 kernel traps)</em></td>
<td>Requests a connection between the local VI and a remote VI using a local and remote network address and waits until a Timeout period has expired. If the connection is established, the local address is bound to the local VI, and the attributes of the remote VI are returned. If the remote end rejected the connection a rejection error is returned.</td>
</tr>
<tr>
<td></td>
<td><strong>VipDisconnect()</strong> <em>(kernel trap)</em></td>
<td>Used to terminate a connection. When the local endpoint is disconnected, all posted Descriptors are marked complete, and the local VI transitions to the Idle state.</td>
</tr>
</tbody>
</table>
| Memory Protection / Registration | VipCreatePtag()  
*kernel trap | Creates a new protection tag for the calling process that can be used to protect access to VIs and memory regions. This protection tag must match for all memory references to associated memory regions or VIs. |
| --- | --- | --- |
|  | VipDestroyPtag()  
*kernel trap | Destroys a protection tag. If the protection tag is currently associated with a VI or registered memory region, the call fails. |
|  | VipRegisterMem()  
*kernel trap | Allows a process to register a region of user pre-allocated memory with a VI NIC. sets the memory attributes (ptag, RDMA) according to the user’s input, and returns a new memory handle. |
|  | VipDeregisterMem()  
*kernel trap | De-registers memory previously registered using VipRegisterMem( ) and unlocks the associated pages from memory. The contents and attributes of the region are not altered. |
| Data Transfer/Completion Operations | VipPostSend()  
*kernel trap | Adds a descriptor to the tail of the send queue of a VI, informs the NIC that a new descriptor is available, and returns immediately. |
|  | VipSendDone()  
*NO kernel traps | Checks the descriptor at the head of the send queue of a VI and if it is marked complete, removes the head and returns its address, otherwise it returns an error. |
|  | VipSendWait()  
*NO kernel traps | Checks the descriptor at the head of the send queue of a VI and if it is marked complete, removes the head and returns its address, otherwise it blocks on the queue until the head completes or a specified Timeout expires. Cannot be used with a send queue associated with a CQ. |
|  | VipPostRecv()  
*I kernel trap on first post to an empty queue or ALL DONE queue | Adds a descriptor to the tail of the receive queue of a VI, informs the NIC that a new descriptor is available, and returns immediately. |
|  | VipRecvDone()  
*NO kernel traps | Checks the descriptor at the head of the receive queue of a VI and if it is marked complete, removes the head and returns its address, otherwise it returns an error. |
|  | VipRecvWait()  
*NO kernel traps | Checks the descriptor at the head of the receive queue of a VI and if it is marked complete, removes the head and returns its address, otherwise it blocks on the queue until the head completes or a specified Timeout expires. Cannot be used with a send queue associated with a CQ. |
|  | VipCQDone()  
*NO kernel traps | Polls the Completion Queue for a completion entry and if one is found it returns the VI handle and a flag indicating whether the completed descriptor is on the send or receive queue, otherwise it returns an error. The calling process must de-queue the completed descriptor using VipSendDone or VipRecvDone, but it may not block using VipSendWait or VipRecvWait. |
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VipCQWait()</strong></td>
<td>Checks the Completion Queue for a completion entry and if one is found it returns the VI handle and a flag indicating whether the completed descriptor is on the send or receive queue, otherwise it blocks until an entry is generated or a Timeout expires. The calling process must de-queue the completed descriptor using <em>VipSendDone</em> or <em>VipRecvDone</em>, but it may not block using <em>VipSendWait</em> or <em>VipRecvWait</em>.</td>
</tr>
<tr>
<td><em>NO kernel traps</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipSendNotify()</strong></td>
<td>Checks if the descriptor on the head of the send queue is completed and if so it removes it and invokes a Handler, otherwise interrupts are enabled for the VI send queue. When a single descriptor is completed, the handler will be invoked but must be called multiple times for multiple callbacks. Destroying the VI cancels any pending callbacks. Cannot be used with a send queue associated with a CQ.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipRecvNotify()</strong></td>
<td>Checks if the descriptor on the head of the receive queue is completed and if so it removes it and invokes a Handler, otherwise interrupts are enabled for the VI receive queue. When a single descriptor is completed, the handler will be invoked but must be called multiple times for multiple callbacks. Destroying the VI cancels any pending callbacks. Cannot be used with a receive queue associated with a CQ.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipCQNotify()</strong></td>
<td>Checks for an entry on the head of the CQ and if one is found it removes it and invokes a Handler, otherwise interrupts are enabled for the CQ. When a single entry is generated, the handler will be invoked but must be called multiple times for multiple callbacks. Destroying the CQ cancels any pending callbacks.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipCreateCQ()</strong></td>
<td>Creates a new CQ with a specified number of entries and returns a handle to the CQ if successful.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipDestroyCQ()</strong></td>
<td>Destroys a CQ: if any VI work queues are associated with the CQ it is not destroyed and an error is returned.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipResizeCQ()</strong></td>
<td>Modifies the size of the CQ by specifying a new number of entries it should hold. Useful when the number of entries that could be placed on the queue changes dynamically.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipQueryNic()</strong></td>
<td>Used to query for information for a specific NIC instance. The information is returned in the NIC attributes data structure.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td><strong>VipSetViAttributes()</strong></td>
<td>Attempts to modify the attributes of a VI, but if the VI Provider does not support the specified attributes or the VI is in a state that does not allow attribute modification, it returns an error.</td>
</tr>
<tr>
<td><em>kernel trap</em></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>VipQueryVi()</code></td>
<td>Used to query for information for a specific VI. The VI attributes data structure and the state of the VI are returned.</td>
</tr>
<tr>
<td>*kernel trap</td>
<td></td>
</tr>
<tr>
<td><code>VipSetMemAttributes()</code></td>
<td>Modifies the attributes of a registered memory region, but if the VI Provider does not support the specified attributes it returns an error. The memory attributes should not be modified while a data transfer operation is in progress that refers to that region, because this can result in undefined behavior.</td>
</tr>
<tr>
<td>*kernel trap</td>
<td></td>
</tr>
<tr>
<td><code>VipQueryMem()</code></td>
<td>Returns the attributes of a registered memory region to the caller.</td>
</tr>
<tr>
<td>*kernel trap</td>
<td></td>
</tr>
<tr>
<td><code>VipQuerySystemManagementInfo()</code></td>
<td>Returns system management information for a specified NIC. The user can specify specific types of information to return and the information returned is VI Provider specific.</td>
</tr>
<tr>
<td>*kernel trap</td>
<td></td>
</tr>
<tr>
<td><strong>Error Handling</strong></td>
<td></td>
</tr>
<tr>
<td><code>VipErrorCallback()</code></td>
<td>Used to register an asynchronous error handling function with the VI Provider. If the user does not register an error handling function using this call, a default error handler will be called. Asynchronous errors are those that cannot be returned directly into a descriptor such as loss of connection or packet droppage due to an empty receive queue.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Windows NT Architecture Driver-Related Details

The Windows NT Architecture has a complex inter-driver and user communication process. Thus, driver development requires understanding of not only the I/O subsystem and processing of I/O requests, but many other NT architecture issues that impact the performance of device drivers. This section summarizes the manner in which drivers process I/O requests and how those requests are described, and presents the best performance solution to I/O processing as well as presenting several techniques for minimizing communication latency between drivers. These techniques are used in the Kernel Agent driver and the discussion in this section will provide a background for understanding the Kernel Agent driver implementation and why certain implementation choices were made.

4.1. Windows NT I/O Subsystem

The I/O Manager combined with all of the Kernel mode device drivers in a Windows NT system comprise the Windows NT I/O Subsystem. There are several aspects to the I/O Subsystem that are of particular interest to device driver writers.

The first important aspect is that the I/O Subsystem is highly structured and consistent. The I/O Manager provides the interface between device drivers and the rest of the OS. This interface is provided by I/O system services, which provides basic I/O support to the OS. The user application (or the VIPL in our case) calls functions NtCreateFile(), NtReadFile(), NtWriteFile, and NtDeviceIoControlFile() to get a handle to a device driver and to communicate with it. The I/O Manager also exhaustively defines the manner in which drivers interact, and defines a detailed and specific set of
interfaces to be implemented by device drivers. This can be seen in Figure 4.1 below, which is a depiction of the different layers and interaction involved between users and drivers in user and kernel space.

**Figure 4.1 I/O Subsystem and User Application Relationship**

A second important aspect of the I/O subsystem is that it is pre-emptible and interruptible. Low-level components such as the I/O subsystem strive to keep their IRQLs as low as possible as much of the time as possible. This allows more important and more time-critical operations to complete rapidly in the system. The I/O Manager attempts to keep drivers pre-emptible and interruptible as much as it can by calling driver entry points at the lowest possible IRQL, namely PASSIVE_LEVEL. This issue of keeping the IRQL low as much as possible is very important for device driver writers,
because not only can other important system events be delayed, but user-level applications that need to run may not get as much time to run. This issue will be discussed in Section 5 in relation to the interaction between the user and the Kernel Agent driver in receive processing.

A third important aspect is that the NT I/O Subsystem is multiprocessor safe. NT is designed to run on multiprocessor systems and this means that there must be appropriate guarding of shared data that can be accessed simultaneously on multiple CPUs. Data being accessed at PASSIVE_LEVEL presents a number of techniques for serializing access. Shared data that is accessed at DISPATCH_LEVEL or above requires the use of Spin Locks. A Spin Lock is a Windows NT data structure in non-pageable memory that associates an IRQL with a lock. It is basically impossible to provide a fully exclusive mechanism for accessing shared data between a thread running at DISPATCH_LEVEL and one running at PASSIVE_LEVEL. This is because the nature of NT is that IRQL DISPATCH_LEVEL is the level at which the Dispatcher itself runs, and thus if a process is running at this level it cannot be pre-empted and resultantly cannot spin or sleep in order to acquire a shared lock for such synchronization.

Another important aspect of the NT I/O Subsystem is that it is packet-driven. Thus, each I/O request that the system receives can be described by a single I/O request packet (IRP). An IRP is built by the I/O Manager in response to a request to an I/O System Service, and once it is initialized properly it is passed to the appropriate driver. This is described in more detail in the next section. One final important aspect of the NT I/O Subsystem is that it is layered. Thus, layers or stacks of drivers work together to
process I/O requests. This is described in more detail in Section 4.2, as it is a very essential part of the VI implementation that I have produced.

4.1.1 Describing I/O Requests

In this section I present a brief overview of all of the different parts involved in describing I/O requests in order to communicate with a driver.

4.1.1.1 I/O Request Packets (IRPs)

An IRP is a standard NT structure of type "IRP" that contains all of the information necessary to describe an I/O request. It essentially has two parts, a fixed part and an I/O Stack. The fixed part is information that might not change when the IRP is passed between drivers or does not need to be preserved between drivers, while the I/O Stack contains a set of locations that each correspond to a driver in a stack and hold information specific to the associated driver. Although the size of an IRP is fixed when the I/O Manager creates it, an IRP is not always the same size. The I/O Manager allocates as many I/O Stack locations as there are drivers in the stack processing this request, just in case the IRP needs to traverse all the way down the stack.

The fixed portion of the IRP contains some fields that are interesting in terms of the discussion in the following sections. The MDL Address field points to a Memory Descriptor List (MDL) that describes the requestor's buffer when a driver is using Direct I/O. The AssociatedIrп.SystemBuffer field points to an intermediate buffer in non-paged pool for the requestor's data, when the driver is using Buffered I/O. Both of these buffer representations for requestor's data will be discussed in the next section. The I/O Status field contains the ultimate completion status of the IRP.
Each I/O Stack Location in the IRP contains information for a specific driver corresponding to that stack position for the I/O request. The fields that are interesting for the discussion on my Kernel Agent device driver are the *MajorFunction* and *MinorFunction* fields. The *MajorFunction* field indicates the type of I/O operation to be performed, while the *MinorFunction* field indicates the minor I/O function code associated with the request, which modifies the major function code when used. The *MinorFunction* field is ignored by most device drivers.

### 4.1.1.2 Describing Requestor Data Buffers

There are three different options provided by Windows NT for describing the requestor’s data buffer associated with an I/O operation to a driver: Direct I/O, Buffered I/O, and Neither I/O. In Direct I/O, a memory descriptor list (MDL) structure is used to describe a locked-down buffer in its original location by physical addresses for each of the requestor’s user mode virtual addresses. In Buffered I/O, data from a user buffer is copied from user space to an intermediate location in non-paged system space before the driver gets the IRP, and the driver is provided with a pointer in the IRP with which to access the data. In Neither I/O, the driver is simply provided with the buffer’s user virtual address without locking down the pages. Neither I/O only works if the called routine is in the same context as the calling process.

### 4.1.1.3 I/O Function Codes & I/O Control Functions

I/O Function Codes indicate the type of operation to take place on a device driver file object, specified in the *MajorFunction* and *MinorFunction* fields in an IRP described
previously. *MinorFunction* codes are rarely used, so I will present the most common Major I/O function codes used.

IRP_MJ_CREATE creates a new file object by accessing an existing device or file or creating a new file. This is issued via the Win32 function *CreateFile()* and is used in obtaining a handle to an instance of a driver (such as for *VipOpenNic()*).

IRP_MJ_CLOSE closes a previously opened file object and is issued via the *CloseHandle()* Win32 function.

IRP_MJDEVICE_CONTROL performs a driver-defined function on an existing file object. This is issued using the Win32 function *DeviceIoControl()* and is used extensively in allowing custom driver functions to be requested by a user application. All of the VIPL kernel traps to the driver are in processing these I/O Control requests, because they allow for the execution of Custom I/O Control Functions to service the VIPL.

IRP_MJ внутренний_драйвер_CONTROL performs a driver-defined function on an existing file object. There are no user-level APIs, however, and this function is typically just used for inter-driver communication.

4.2. Windows NT Layered Driver I/O Model

As mentioned previously, an IRP is created as a result of a request to an I/O Systems Service. Since the Windows NT I/O subsystem is a *layered* driver model, a single IRP may pass through many driver layers (or stacks) working together to process the I/O request asynchronously. The remainder of this section briefly describes the different types of drivers found in Windows NT, the layering model, and alternate methods of driver communication.
4.2.1 Driver Types

4.2.1.1 Kernel-Mode Drivers

Kernel-mode drivers form part of the Windows NT Executive layer and run in Kernel mode only. The I/O Manager accesses these drivers as the result of the execution of an I/O system service. All standard NT Kernel mode drivers share a common basic structure, and contain specific entry points called by the I/O Manager to perform certain functions. When a driver is initially loaded, the DriverEntry entrypoint is called. This function is responsible for determining the driver’s configuration and performing any required driver- and device-initialization processing. A driver associates a Major I/O function code for each type of I/O operation it supports, which must have a corresponding Dispatch routine, and a Minor I/O function code for each operation of that type.

4.2.1.2 Intermediate Drivers

Intermediate drivers form the middle layer of the NT driver hierarchy, positioned below File systems drivers and above device drivers. They typically provide some added feature for a device and rely upon the device driver below them in the hierarchy for access to the device. The Kernel Agent I developed to support VI is an intermediate driver that supports added functionality for the underlying PEGNIC NDIS miniport driver.

4.2.1.3 NDIS Miniport Drivers

Miniport Drivers exist within a wrapper that typically restricts the driver’s interfaces to those provided by the wrapper. The PEGNIC miniport driver exists within
the NDIS (Network Driver Interface Specification) wrapper, which describes the interface by which NIC drivers communicate with underlying network interface cards, overlying protocol drivers, and the operating system. The NDIS library is meant to remove the burden of coding multiprocessor support and other more difficult operating system programming issues from the NIC driver developer, allowing developers to focus on writing only hardware-specific code. The interface mechanism provided by NDIS does not utilize IRPs, but rather is a call and return interface. This means that drivers call functions in the NDIS library, which may call functions in other drivers before returning.

A packet that is transmitted or received through the NDIS interface is described by an NDIS_PACKET, a structure that is opaque to NDIS drivers. The NDIS Library provides functions necessary to retrieve information about an NDIS_PACKET or manipulate the packet and message that it describes. The NDIS_PACKET comprises one or more NDIS_BUFFERS, which are also opaque handles. Each NDIS_BUFFER describes a virtually contiguous part of the message located in kernel virtual memory. NDIS_BUFFERS may be chained together to allow for messages of virtually non-contiguous buffers.

4.2.2 Device Driver Stacking (Layering)

Typically, kernel-mode drivers are grouped together in stacks that work together to completely process a request targeted at a particular Device Object (a structure that represents a physical or logical device). Device driver stacks are created when a system starts. These stacks are created by the device drivers themselves, and each driver in the stack depends on other drivers being started at the correct time. When a driver is loaded, it first creates its own device object and attaches it to the top of the stack. A device
object contains a StackSize field specifying how high on the device stack it resides, which corresponds to how high its driver is on the driver stack. Certain driver properties, such as the method of data buffering used, are generally maintained throughout all levels of the driver stack.

Upon receipt of an I/O system service request, the I/O Manager calls the driver at its Dispatch routine corresponding to the request's Major function code. A pointer to the IRP describing the request is passed along with a pointer to the device object to the driver's entry point corresponding to the function code. Upon receiving an IRP, the driver may satisfy the I/O request and complete the IRP with success or an error, queue the IRP to be processed and return STATUS_PENDING (indicating that it has retained ownership of the IRP), or pass the IRP to a lower layer driver on the stack. If the IRP is passed to a lower layer driver, it must traverse back up to the top-level driver's completion routine and return either error or other status upon completion. To pass an IRP to a lower layer, a driver obtains a pointer to the next I/O Stack location. This pointer is used to fill in the parameters of the request to be passed to the underlying driver, causing the I/O Manager to push the I/O Stack and causing the next location to become the current driver. The I/O Manager must next find the driver associated with the target device object, and call that driver's dispatch routine with the Major function code for the current I/O Stack location. In addition to passing IRPs through to lower layers, drivers may create additional IRPs to pass to lower-level drivers to satisfy certain requests.
4.2.3 Inter-Driver Communication Alternatives

There are several alternatives to the manner in which NT drivers layer themselves upon one another. It is possible for two drivers to have an inter-driver communication mechanism that does not conform to the standard I/O processing model described previously. The simplest method for two drivers to communicate is to simply call each other’s routines directly. This is possible because all drivers reside within the system process’ address space. This can be implemented using a DLL common to both drivers. This DLL contains code used to set up pointers to each of the driver’s entry points. When the drivers wish to communicate, they only have to call a set of functions in the DLL to call the appropriate function through the provided pointers. This is the method used by intermediate layer drivers to interface with NDIS drivers. The NDIS wrapper is the common library DLL used. Another communication mechanism is the creation of a common structure in the non-paged pool. These non-standard methods have the advantage that two drivers can transfer more information than is conveniently possible with an IRP structure. Also, if two drivers communicate very frequently, the significant overhead inherent in creating IRPs for every communication is avoided.

4.2.4 Fast I/O

Fast I/O is an optimized method that high-level drivers can use to deal with I/O requests. A driver that supports fast I/O creates a Fast I/O Dispatch Table. When the I/O Manager receives an I/O request, it checks to see if a driver supports Fast I/O for this request prior to building an IRP, and if it does it calls the driver directly at its Fast I/O entry point in the context of the requesting thread along with the parameters supplied with the request. If the driver is able to completely handle the request in its Fast I/O
entry point, it returns TRUE, otherwise it returns FALSE. If FALSE is returned, the I/O
Manager creates an IRP for the request in the normal manner and calls the driver with the
IRP. This Fast I/O method is used for all of the entry points for the VIPL calls in the
Kernel Agent driver, as it provides the fastest method of fielding a request.
5. VI Implementation

This section describes the low-level implementation details of the software emulation of VI that I have designed. The two main components involved in the VI implementation are the Kernel Agent and PEGNIC (Packet Engines GNIC) NT drivers. The VI Architecture Kernel Agent Driver is an intermediate NT network driver tightly coupled with the PEGNIC driver. The PEGNIC driver provides the interface to the GNIC II Gigabit Ethernet card and services all interrupts for sending and receiving data as well as reusing packets. I have modified the existing PEGNIC driver to support both WinSock and VI packets. When a packet is received, it is either passed to WinSock or to the VI Kernel Agent intermediate driver depending on a specifier in the header.

The VI Architecture Kernel Agent Driver was designed to handle the tasks that would normally be handled by VI hardware, while taking advantage of the features available to the Windows NT device driver developer. Some of the important tasks that must be maintained by the driver include memory registration, VI management, Completion Queue management, and send and receive completion notification.

The Kernel Agent driver was developed using the Microsoft Device Driver Kit (DDK 5.0 Beta) and the Microsoft C/C++ (Version 12.00.8168) compiler. To date, the VI driver has only been tested on x86 architectures.

In the remainder of this section, I describe the implementation details for all aspects of emulating VI across both the Kernel Agent and PEGNIC drivers, including implementation details on how each of the main groups of VIPL calls mentioned in Section 3 are handled.
5.1. Design Overview

Figure 5.1 depicts the three components of VI support:

- VI Provider Library (VIPL) – Provides interface routines as specified in the VI Architecture Specification 1.0.[5] An overview of this library is provided in Section 3.
- Kernel Agent Intermediate Driver – Provides entry points to service VIPL requests.
- PEGNIC Device Driver – Provides the Kernel Agent Driver with access to the NIC for sending and receiving data.

Both the Kernel Agent and PEGNIC drivers interact with the standard Windows NT NDIS Library. The PEGNIC driver is an NDIS Miniport NIC driver, which means it takes advantage of the functionality and performance characteristics that NDIS provides for NICs. The NDIS library is meant to lessen the burden of coding multiprocessor support and other complex operating system programming issues. This allows developers to focus on writing only hardware-specific code.

![Diagram of Layered Driver Design]

Figure 5.1 Layered Driver Design
The Kernel Agent and PEGNIC drivers have the shared inter-driver communication structure shown in Figure 5.2. This structure contains pointers to routines in the Kernel Agent driver and in the PEGNIC driver that are setup when the drivers are initialized. Using these pointers, requests are passed between these drivers without requiring the use of a standard I/O Request Packet (IRP). Avoiding the overhead of creating and passing IRPs between the PEGNIC and Kernel Agent reduces communication overhead significantly. This enhancement is a major factor in sending and receiving packets, since most communication between the Kernel Agent and PEGNIC drivers is for these purposes alone. When the user posts a send descriptor, the Kernel Agent services the send request using the pointer to the SendPacket routine within the communication structure to call the SendPacket routine in the PEGNIC driver and send the data onto the network. Similarly, when the PEGNIC receives a VI message it passes it to the Kernel Agent driver by invoking the ViaReceive routine through the pointer in the communication structure, which starts a receive operation in the Kernel Agent.

![Common Structure in non-paged pool](image)

**Figure 5.2 Inter-driver Communication Via a Common Structure**

5.2. Connecting to Driver

In order to get a handle to the VI driver the user initially calls VipOpenNic(). A call to CreateFile() sends an IRP to the driver for the IRP_MJ_CREATE dispatch
function. This function obtains the process ID of the process that made the system call and allocates a structure associated with the process ID that is inserted on the tail of a driver-maintained list of the current processes that have obtained an instance of the driver. This structure stores information on the error queue for this process (more on this in Section 5.8) and the process ID which is stored with various structures in the driver that are allocated for this process so that when the process exits the driver can free all resources it was using that were not freed. After the call to CreateFile( ) that initializes an information structure for this process, another call is made that requests a Device I/O Control function in the driver to complete the initialization of the user connection. This second call is required to pass in the NIC handle that was returned to the user on the first system call to be associated with an error queue (this will be described in more detail in Section 5.8).

5.3. VI Endpoint Creation/Destruction

When the user makes a request to create a VI endpoint, a call to VipCreateVi( ) is made. Inside the VIPL, a VI structure is allocated that contains the send and receive queues and a synchronization semaphore. After this structure is initialized, a Device I/O Control request is sent to the driver, which executes a routine that sets up a structure that contains all of the information for a VI that the driver needs. The driver maintains a table of pre-allocated VI structures so that one can simply be returned from this table without allocating a new structure. The index into this table is used as the VI ID and is returned to the user. The user passes in virtual addresses corresponding to the VI send queue head and tail pointers as well as handles to user created semaphores for both the send and receive queues.
An MDL is created to obtain the physical addresses for the send queue pointers. These pointers are required by the driver in order to modify the send queue itself when sending data represented by descriptors. The handles to the send and receive queues are also mapped into the driver so that the driver can use them to signal multiple completions to the user at once and limit the user from removing descriptors from the queue that have not been completed.

In order to de-allocate a VI, the user makes a call to \textit{VipDestroyVi( )} that sends a request to the driver that de-allocates the MDLs previously created and frees the VI structure by putting it back in the VI table in the driver.

\section{VI Connection Management}

The connection procedure that is used to set up a connection between two VI endpoints is illustrated in Figure 5.3. The user that wants to act as the server for the connection makes a call to \textit{VipConnectWait( )}. On the remote end, the user that acts as a client makes a connection request by calling \textit{VipConnectRequest( )}. When the server calls \textit{VipConnectWait( )}, a Device I/O Control request is sent to the driver and the driver allocates a connection handle that is placed in a connection handle setup table. This table is hashed by the discriminator specified for the connection and each entry in the table is a linked list. In addition, the driver buffers connection requests it receives from a remote machine that are for discriminator values that the user is not currently waiting for a connection request on. Thus, when a new request to wait for a connection is made, the driver first checks the buffered connection requests for a match and if there is one it returns the request immediately to the user to decide whether or not to accept it. If there
are no buffered requests, the driver obtains a pointer to a user event passed in by the call to \texttt{VipConnectWait()} and wakes up the user using that event upon receive a request.

When the user acting as server has received a connection request it verifies that the attributes of the remote VI correspond to its own and if they do not the user calls \texttt{VipConnectReject}, which just sends a request to the driver to send a rejection message to the remote user, and the remote user is woken up from the event it was waiting on and finds the rejection message. If the attributes match the user calls \texttt{VipConnectAccept()} and the driver sends an acceptance message to the remote user.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{connection_setup_process.png}
\caption{Connection Setup Process}
\end{figure}

When the user at either VI endpoint wants to close its connection, a call to \texttt{VipDisconnect()} is made. The Kernel Agent driver fields a request and sets the status of
all of the descriptors on the VI's queue to VIP_STATUS_DONE, returns the VI to the
IDLE state, and sends an asynchronous message to the remote end informing the remote
driver that the VI disconnected. The remote driver fields this message and posts an error
on the error queue corresponding to the process owning that VI and wakes up a user
thread that removes the new error message from the head of the queue and informs the
user of the disconnection.

5.5. Memory Protection/Registration

In order to provide a secure manner of protecting a process' VI resources from
being accessed by other processes, the VI Architecture introduces an item called a
protection tag. This tag can be associated with a VI or a buffer that the user has
registered. On a call to VipCreatePtag( ), the driver allocates a structure that creates a tag
that can be associated with a process' memory buffer or VI in subsequent calls.

As described previously, in order to give the user direct access to the actual data
buffers that will be sent out to the network and to minimize latency, the user specifies its
own buffers and registers them. This registration process is completed through a call to
VipRegisterMem( ), which passes the virtual address of a user's buffer to the driver. The
driver creates an MDL describing that buffer in order to gain a virtual-to-physical
mapping and associates the registered buffer region with a buffer handle that is placed in
a table for later use. The driver can then simply access the user's registered memory
region information at a later time by indexing into this table using the memory handle.
When the user is finished with a region of memory, it makes a call to
VipDeRegisterMem( ), and the driver deallocates the MDL allocated for that region and
frees the buffer handle table position that was used.
5.6. Data Transfer & Completion Operations

As mentioned in Section 2, there are three types of data transfer operations that the Virtual Interface Architecture supports: traditional Send and Receive, and RDMA. These operations and how they were implemented to satisfy the user VIPL calls are explained in detail in the following sections. In addition, the mechanisms used to inform the user of data completions and to synchronize between the user and the kernel driver are described.

5.6.1 Sending Data

In order to request a send operation, the user makes a call to \textit{VipPostSend( )}. This places a descriptor on the tail of the VI's send queue, and traps to the kernel in order to perform the actual send operation. The reason for requiring a kernel trap and why this implementation was chosen will be explained at the end of the section after all of the implementation details have been explained.

I employ Neither I/O as the data buffer description type for the \textit{VipPostSend( )} entry point that handles posting of descriptors on a VI's send work queue. Neither I/O was chosen because all of the entry points that handle requests from the VIPL are executed in the same context as the caller process. As mentioned in Section 4.1.1.2, this implies that any virtual address in the context of a user process that initiates calls to the driver should map to the correct corresponding physical address in kernel space. Neither I/O proves to be the most efficient of the three buffer transfer types because it does not involve any overhead of translating virtual addresses to physical addresses or redundant copying of data. Therefore, its use allows me to implement zero-copy sends.
When a VI message is sent, the Kernel Agent driver routine `SendData` handles the actual dividing up of the message into NDIS packets and calling the underlying PEGNIC driver `SendPacket` routine through the shared driver inter-communication structure (see Section 5.1). Since I am using Ethernet packets for data transfer, it was necessary to use the first 18 bytes of the actual data in the packets as a VI header in order to identify the packets as VI packets. This header specifies the sender VI ID, the destination VI, the VI message ID, and the number of packets in the VI message. A VI message is split up into \( n = \frac{\text{Message Size}}{\text{Effective Frame Size}} \) packets, where the Effective Frame Size is the amount of space left in the Ethernet packet after subtracting the size of the Ethernet header and the extra data bytes used for the VI header.

Sending data does not require any memory-to-memory copying because the packet descriptors that are allocated point directly to the registered memory user buffers containing the data. The Packet DMA Engine on the NIC uses these descriptors to transfer the data directly to SRAM located on the NIC. Although the PDMA engine queues the packet on the NIC transmit queue immediately, the packet is not necessarily transferred at that moment. Performance is improved by interleaving the sending of packets, instead of preparing all of the packets for a VI message and starting the DMA engine to transfer them all at once. While a single packet is being prepared, the previous packet is in the process of being sent by the NIC. A study was done on the relationship between message size and the number of packets from the message to DMA at a time, and it was found that for message sizes less than 4K, a DMA per packet gave the lowest message latency but for larger message sizes, a larger DMA size of 2 or 4 packets resulted in a lower message latency. This can be seen in Figure 5.4.
Figure 5.4 Measured Latency with Varying DMA Size

In order to determine the driver implementation that would provide the lowest latency in sending a message, several different implementations were developed and compared. Initially, the sending was implemented by trapping to the kernel, performing the send immediately by passing the packets to the PEGNIC, and returning. This implementation delivered fine performance for a single VI, but when multiple VI send operations are outstanding the user processes posting send requests to their VI queues can be blocked. This is because drivers in Windows NT are serialized, or in other words when a driver is fielding a request for a particular entry point, it cannot field another such request until the first is complete, and the user requesting process is blocked on the system call for the request. In order to alleviate this situation, a separate thread was created in the driver to handle the sending of data, and a kernel trap was still required for each call to *VipPostSend*(), but the Send thread handled the sending of all descriptors
posted. Thus, the \textit{VipPostSend}() entry point would place an entry on a global queue that the Send thread would service. The send thread would sleep when the global queue was empty, and in this event the user process would wake up the thread when it placed the first entry on the global queue. This was detrimental to the performance, especially for small messages, because the latency of waking up the kernel thread was not small. The final implementation that delivered the highest performance was using a \textit{Deferred Procedure Call} (DPC). A DPC is basically a callback that is scheduled in Windows NT by placing an entry on a queue (associated per processor). This queue is serviced as soon as a processor’s \textit{IRQL} drops below \textit{IRQL\_DISPATCH\_LEVEL}, and all callbacks on this queue run at \textit{DISPATCH\_LEVEL}. The implementation for this method was done just as with the thread, except instead of using a thread to service outstanding send operations a DPC was scheduled. This DPC runs as long as there are outstanding entries on the global send queue. An additional optimization is that when the user requests a message to be sent that fits in a single packet, the request is serviced immediately and a DPC is not scheduled. This is because the time to service such a request in the Dispatch entry point is so minimal that other processes requesting data to be sent will not be blocked for long, and we want the latency of small messages to be as little as possible.

5.6.2 Receiving Data

The \textit{VipPostRecv}() user VIPL routine is called to post a descriptor on the receive queue for a VI. The driver handles this call using the same dispatch handler as for the send operation, in which \textit{Neither I/O} is used as the buffer description type. In the average case, there is no trap to the kernel required for posting a descriptor, the user simply puts the descriptor on the tail of the receive queue in user space.
Upon receiving a packet, the PEGNIC driver must determine whether the message is for the VI or WinSock driver. This is accomplished by using the NDIS call *GetFirstBufferFromPacket*() to search the header. If it is a VI packet, the inter-driver communication structure is used to pass the data to the Kernel Agent driver’s *ViaReceive*() routine. As mentioned in Section 5.1, this does not involve the overhead of creating an IRP and allows a much tighter coupling between drivers. Upon entering the *ViaReceive*() routine invoked by the underlying device driver, the Kernel Agent copies the data from kernel space to registered memory in user space, resulting in a single-copy receive. It is not clear how one might implement a zero-copy receive mechanism because it is not possible to preview the headers of incoming packets on the NIC. Thus, the DMA engine does not know where to send the data. To allow for zero-copy receives, the NIC would have to provide in hardware a mechanism for directing packets to different drivers.

The *ViaReceive*() routine in the Kernel Agent is set up to handle various messages differently. There are several different message types that I employed for connection setup and tear-down that are handled differently from normal data packets. *ViaReceive*() simply routes connection management packets to the proper connection handler routine depending on the message type and if the packet is a data packet it performs the necessary operations to place that data into a descriptor on the destination VI’s receive work queue, as mentioned above.

In addition, this routine buffers incoming messages so that if any packets in the message are not received after a certain timeout period, the message can be dropped without using a descriptor and holding up the rest of the queue. Descriptors are only used when an entire message has been received. The driver maintains a pointer to the next
unused descriptor in the queue, and never traverses from the head of the queue on its own, rather the user points this next pointer to the correct place when the first descriptor is posted or the queue is full of complete descriptors and a new unused descriptor is posted. This requires a kernel trap on the VipPostRecv() call, but only when the queue is initially setup and if the queue becomes completely filled up, which should rarely happen if an application is written correctly and posts enough descriptors.

An interrupt control vector table, located on the NIC, contains a counter used to enable a hardware interrupt to occur after a certain number of packets have been received on the NIC. This allows the number of interrupts to be reduced by combining the receiving of many packets into one interrupt event. However, this may be inefficient for small packets by delaying their delivery to the user application.

A study was done on the relationship between the message size and the number of packets to receive per interrupt, and it was found that in most cases 1 packet per interrupt provided the lowest message latency because it insures that the driver processes the receive immediately when the data arrives. Only message sizes of 32K and 64K obtained lower latencies with 2 packet per receive interrupt. The results can be seen in Figure 5.5.
5.6.3 Completion Mechanisms

The *VipRecvWait()* and *VipSendWait()* VIPL routines are called by the user to perform a blocking wait for a receive or send operation to complete after posting a descriptor on a receive or send queue. These routines reset the event (or some other type of synchronization object) that is being waited on by the user thread, set a send or receive flag in the VI handle, and block while waiting for the user event to be set. When a VI message is received or a send completes, the proper event is triggered by the Kernel Agent driver, and the user thread is released. However, since the driver and user could be accessing the send or receive queues at the same time, a race condition exists.

The initial solution to this race condition was to have the user not take off descriptors from the head of the queues, but rather trap to the kernel and have the driver perform the operation, this way the driver would have to acquire and release the same
lock that is used to add things to the queue by the driver. Thus, in a call to 
_VipRecvWait( ) or _VipSendWait( ), the driver would field an I/O operation and perform 
the steps necessary. First, check the head of the queue and if it is DONE, return the 
descriptor at the head immediately. If the head is not found to be DONE, then within the 
driver a flag would be set and the process would sleep on an event in kernel space. This 
event would be set by the driver in _ViaReceive( ) and wake the process up so that it could 
take the descriptor off the head and return from the driver I/O operation to the user. This 
solves the race condition simply, but is a drastic hit to performance. In addition, as was 
mentioned in the send data implementation, the driver is serialized and can only handle 
one _DriverIoControl request that it receives from the I/O Manager at a time. This is a 
major problem that can cause deadlock in situations where a distributed application is 
running on two machines and both machines are sending and receiving data at the same 
time, but cannot field their sends because they are waiting in the driver.

A simpler solution, which would not require a trap to the kernel, might be to 
create a shared lock between the driver and the user. However, this method fails because 
the _ViaReceive( ) routine in the driver that is called when data is received runs at IRQL 
_DISPATCH_LEVEL, which is above the level that user code runs at and at the same 
level as the Dispatcher (hence DISPATCH level). Thus, processes running at this level 
cannot be preempted by the dispatcher and cannot spin on a lock without starving the 
user and leading to a deadlock situation if the user cannot run to release the lock.

The final solution implemented involves the user performing all of the 
manipulations to the receive and send queues of a VI (posting and removing descriptors), 
and the driver only sets the descriptors DONE when they complete, in order. Thus, the
only thing that has to be ensured when the user calls \textit{VipRecvWait}() or \textit{VipSendWait}() is that the user is awakened when waiting and only when the descriptor on the head is DONE. The outline of the situation that occurs between user and kernel space is shown in Figure 5.6.

When the user calls \textit{VipRecvWait}(), the head of the queue is first checked. If it is DONE, the head is removed immediately, otherwise the user has to inform the driver that it is going to wait to be woken up when the descriptor completes. This requires an I/O call to the driver to set a flag informing the driver that the user is waiting for the descriptor information. When the driver fields the I/O call, the process has to acquire a lock in kernel space that insures that the driver function that receives data will see the flag if it is set before it completes a descriptor. Before setting the flag, however, the process must again check if the head of the queue is DONE, since while it was spinning waiting to acquire the lock the other receive process may have already set the descriptor DONE and it is no longer necessary to set the flag. Upon returning from the driver call, the process checks the return value and if it returned DONE then the head descriptor is returned, otherwise the process waits on the event to be set by the driver upon completion.
**Figure 5.6 Example Synchronization between User and Kernel**

This entire process can reduce performance because a driver call is required every time the user has to sleep on an event. This adversely affects performance because the driver could be fielding other more important send data requests during this time. Thus it is desirable to not require calling the driver. The reason for signaling the driver that the user is waiting to receive something from the queue is because the user does not necessarily have to use *VipRecvWait()* and could poll the queue instead using *VipRecvDone()* described below. Thus, the driver cannot simply set the event to wake the user up every time it receives something, because it may end up waking up the user when it calls *VipRecvWait()* at a later time. The solution to this was to use a semaphore between the user and driver that the driver always decrements when it receives something and the user always increments when it takes a descriptor off the queue. In this manner, when the user calls *VipRecvWait()* they wait on the semaphore if it is zero, otherwise they decrement and continue.
The \texttt{VipRecvDone()} and \texttt{VipSendDone()} VIPL routines are called by the user to poll the head of the receive and send queues for a completed descriptor. Upon completion of a receive or a send, the Kernel Agent sets the corresponding descriptor's status to \texttt{VIP\_STATUS\_DONE}. The user calls \texttt{VipRecvDone()} or \texttt{VipSendDone()} repeatedly to check the value of the head descriptor's status without involving the kernel. As mentioned in the discussion above, the driver always increments a semaphore whenever it completes a descriptor on the send or receive queues. Thus, when \texttt{VipRecvDone()} or \texttt{VipSendDone()} checks the head of the queue for a completed entry and finds one \texttt{DONE}, it must also decrement the semaphore count.

5.7. Completion Queue Management

To create a Completion Queue, the user makes a call to \texttt{VipCreateCQ()} and passes the number of entries desired on the queue. The VIPL allocates a buffer according to the number of entries in user space, and registers this memory with the Kernel Agent driver so that the driver can also access the queue to place entries on it. In addition, information on the length of the queue, a handle to a semaphore for access to the queue, and pointers to the head and tail of the queue are passed to the driver. The page containing the pointers to the head and tail is locked in memory and a physical address is obtained so the driver can access the queue properly. A CQ handle is obtained to store all of this information in the driver, and these handles are maintained by the driver in a table indexed by the ID of the CQ.

The mechanism used to synchronize between the user and driver in accessing the Completion Queue is the same as was described for the send and receive queues. The driver increments a semaphore every time it places an entry on the tail of the queue, and
the user decrements the semaphore every time it removes something from the head of the queue, whether it is using \textit{VipCQWait()} or \textit{VipCQDone()}. This insures that the user will never remove an invalid entry off the queue, and provides the mechanism for waking the user up when an entry is placed on the queue.

5.8. Error Handling

The driver maintains an error queue per process per NIC Handle (call to \textit{VipOpenNic()} that is used to inform the user of asynchronous errors that occur. When an asynchronous error occurs, a descriptor is placed on the queue that contains information about whether the error was in relation to a VI or descriptor, and any handles necessary to access the appropriate item. The user call to \textit{VipOpenNic()} spawns a thread that goes to sleep on an event that is set by the driver when an asynchronous error occurs, after the driver has placed an error descriptor on the appropriate error queue. When this user thread is woken up, it immediately makes a call to remove the head entry off the error queue and informs the user of the error that occurred. The user can make a call to \textit{VipErrorCallBack()} during initialization to setup a specific handler that can process asynchronous errors that occur in a specific manner.

One specific type of asynchronous error that can occur involves an unexpected disconnection of two connected VIs. When this occurs the Kernel Agent driver on the side of the VI that disconnected sends a message to the remote driver informing it of the sudden disconnection so that the remote driver change the state of its VI back to IDLE and set its descriptors to DONE. Figure 5.7 below depicts the states of a VI, and the transition from CONNECTED to IDLE is shown for a disconnect. As mentioned in Section 2, the ERROR state is actually never reached by an unreliable VI
implementation, because the user is simply informed of the error, but a VI is left in its current state.

![State Transition Diagram](image)

**Figure 5.7 VI States**

### 5.9. Multithread Synchronization

Support for multithreading has been implemented using executive spin locks. These spin locks raise the IRQL to IRQL_DISPATCH_LEVEL upon an acquire operation and reset the IRQL to the proper level upon release. There are also support routines for acquiring spin locks at dispatch level that assume the process is already running at this level and avoid the overhead of having to raise the level. I have taken advantage of this enhancement in the receiving code and in other sections of the driver that always run at IRQL_DISPATCH_LEVEL. In addition, the Kernel Agent driver does not hold two separate spin locks simultaneously because this can lead to deadlock. I have implemented separate locks for various structures in the driver in an attempt to reduce lock contention in the system.
5.10. Hardware Support for VI on PEGNIC

Figure 5.8 depicts the Packet Engines GNIC. The control logic and PCI interfacing for the card is implemented in the Hamachi core ASIC, but the card has external synchronous RAM that it uses for storing packets when its internal FIFOs are full.

![Figure 5.8 PEGNIC Block Diagram](image)

In order to add support for VI in hardware, it would be necessary to add more RAM (approximately 8MB if we want to meet the requirements of the specification) on the card. This additional memory would hold 64K Send and Receive descriptors to handle up to 1K virtual interfaces, a Translation and Protection Table or TPT, send and receive doorbells, per-VI context info, and per-Completion Queue context info. In addition, the Hamachi Core chip would have to be modified to allow for controlling of a set of VIs and multiplexing between them and the specification for headers would have to be modified to allow for Ethernet and VIA headers so that the VIA Control could handle
the data appropriately depending on the protocol type. An overview of some of the modifications to handle VI in the Hamachi core is shown in Figure 5.9.

Figure 5.9 Modified Hamachi Core and Memory Interface

Some of the existing features of the Hamachi Core are that it has a 2.1/2.2 Compliant PCI Interface with a 64-bit Data Path and 64-bit Addressing, a PDMA or "Packet" DMA that is Descriptor based, 1KB burst FIFO storage for each channel, an external 512 KB to 2MB FIFO, checksumming hardware, MAC FIFOs (8KB transmit, 16KB receive), statistics counters, and Interrupts.

To support VI, the hardware interfaces are virtualized and associated with a VI by storing a context for each VI and the direction of transfer in the SRAM memory on the NIC. When a send or receive operation is performed for a particular VI, the VI's context
is used to control the operation of the hardware. The host memory contains its own VI queues and buffers that are registered by the user, as shown in the figure. When descriptors are posted, the DMA Engine on this chip copies data from the VI queues in host memory to the physical VI queues in the SRAM on the NIC. Each VI queue on the NIC also has an associated Doorbell memory mapped register that the user can use to inform the NIC of a new posted descriptor. A Kernel Agent driver manages the area in host memory containing VI queues that correspond to those on the NIC. This Kernel Agent implements a group of services inside the target operating system (Network device control and management, Virtual Interface resource management, Completion Queue resource management, Host memory management, and Connection management). In addition, the host memory should have a global interrupt queue to which the DMA Engine on the NIC writes VI interrupt status words to asynchronously notify the host of various events. Last, in order to implement VI in hardware efficiently, new control logic is needed to handle posting of descriptors, scheduling VI sending, issuing VI-related interrupts, and allowing for both VI and Ethernet communication simultaneously.
6. **Multicast Protocol Overview**

For years unicast transmissions proved to be sufficient for communication over the Internet. Unicast transmission is essentially one-to-one communication between one sender process and one receiver process. The sender transmits a packet and only one machine receives it, as exemplified in Figure 6.1. It was not until 1993 that the first implementation of multicast became a reality in the 4.4 BSD Release. Multicast is a mechanism used for sending packets from one machine to a group of other machines, as shown in Figure 6.2 [44]. Multicast is therefore a form of broadcast, but only to a subset of the nodes of a network. When a node on the network transmits a packet that has a multicast address as its destination, then a specific subset of the nodes will receive the packet. Each node may listen to more than one multicast address and any number of nodes may be a group of a specific multicast address.

\[\text{\textbf{Figure 6.1 Unicast}}\]

\[\text{\textbf{Figure 6.2 Multicast}}\]

Multicast is extremely useful for sending information to various hosts over the Internet, such as in distributing real time audio and video to a set of hosts which have joined a distributed conference. It is also extremely useful in certain distributed applications that need to transmit a lot of information to a subset of machines in a small cluster of machines.
Ethernet supports multicast in the following manner. The normal Ethernet destination address consists of 6 bytes (some early LANs used only 16-bit addressing and therefore their destination address was only 2 bytes, however today almost all Ethernet networks use 6-byte addressing). The first 3 bytes identify the manufacturer of the Ethernet card that is used and the last three bytes identify a specific card manufactured by the vendor. The multicast Ethernet address differs in that the least significant bit of the first of the 6 bytes is set to one, as can be seen in Figure 6.3 below [45].

<table>
<thead>
<tr>
<th>Byte 6</th>
<th>Byte 5</th>
<th>Byte 4</th>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LSB IS '1'</td>
</tr>
</tbody>
</table>

Figure 6.3 Multicast Address Format

Each machine that wants to receive messages destined to a particular multicast group must join the group by registering the group's multicast address. Ethernet card hardware may detect multicast packets, however these packets have to be filtered, so that only those destined for a registered multicast address are passed to the application. This filtering may be done either in hardware or in software.

6.1. IP Multicast

In IP Multicast there is a notion of a Multicast "group" in which packets sent to multicast addresses are sent to all members of the associated multicast group. The packets are distributed using a multicast tree. Figure 6.4 depicts a simple multicast distribution tree. The arrows in this distribution tree indicate the passing of a multicast packet to the next router servicing members of the multicast group. The main advantage of multicast is that it eliminates the need for sending multiple messages in order to send a
message to multiple hosts. This reduces the traffic substantially at the sender because if it were required to send multiple messages the links close to the sender would experience heavy traffic.

![Multicast Distribution](image)

Figure 6.4 Multicast Distribution

Group membership in an IP multicast group is dynamic and thus hosts can join and leave the group at any time. To join a group, a host needs only to notify its local multicast router that it wants to receive packets sent to a particular multicast group address. Group Management is handled by the Internet Group Management Protocol (IGMP) [52]. This protocol specifies that hosts who want to participate in multicast communication must first join the base multicast group with the address 224.0.0.1. This is because the multicast router periodically queries hosts on the network about their membership by multicasting the request to all hosts (IP-multicast address 224.0.0.1). The multicast hosts report their membership after waiting a random delay by multicasting to each of the multicast addresses that correspond to its group membership. A host will cancel its report related to a particular group if another hosts’s report for this group is received during the delay period.
IP Multicast is a technique that allows multicast distribution to be extended to multiple independent subnets through an internet. These subnets can be and often are implemented using different technologies such as Ethernet or token ring and the members of a multicast group can be dispersed over different link layer technologies.
7. Multicast VI Implementation

Many parallel applications have communication patterns that could take advantage of one-to-many communication, however the Virtual Interface Architecture currently does not support multicast. By providing this functionality in the VI Architecture, the performance of applications that use one-to-many communication can be improved. Systems that already use multicast, such as the Brazos parallel programming environment [12], can make immediate use of this feature. In order to evaluate the potential value of a multicast facility within the VI Architecture, I constructed a prototype implementation.

In IP Multicast, as was described in Section 6, there is no connection required between the node that wants to send and the remote nodes that will receive. Rather, the data is simply sent to a multicast address and any remote node that has joined that multicast group (specified by the IP multicast address) will receive the data. Furthermore, a machine does not even have to join the multicast group in order to send data to it. The VI Architecture, on the other hand, is a connection-oriented communication protocol. Every VI must have a connection to a remote VI in order to send or receive data. To keep this model consistent when adding multicast support, it was necessary to have the notion of a “one-to-many” VI, that is a VI that sets up connections to multiple remote VI endpoints. The user is then able to post a single descriptor on its queue whose data is sent to all of the connected VIs. Thus, there is no longer the notion of a multicast group, but rather a multicast connection because the user should not have any knowledge of multicast addresses but should only be required to set...
up a multicast connection between VIs. The underlying details of how the connections actually enable sending to multiple users at the same time is abstracted from the user.

I have extended my software implementation of the VI Architecture to include multicast capability. This involved modifications to each part of the system described in the previous chapters. Throughout the remainder of this section, I will describe the details of my Multicast VI implementation and the changes that were necessary through each of the three levels of the system: the VI Provider Library (VIPL), the Kernel Agent NT driver, and the PEGNIC NT driver. In addition, I modified the Brazos Distributed Shared Memory System library in order to take advantage of Multicast with VI, and I will provide an overview of these modifications and the issues that were involved to perform this task.

7.1. Additional Support in VIPL

At the highest level of the VI implementation is the VIPL, which provides the user with the API to interact with the network. In order to allow the user to setup VIs with one-to-many connections, it was necessary to add two multicast-support routines to the VIPL. The prototype for these routines are shown below.

- **VIP_RETURN**
  
  VipMCOpenNic (  
    IN const VIP_CHAR *DeviceName,  
    IN const VIP_CHAR *BaseMCIPAddr,  
    OUT VIP_NIC_HANDLE *NicHandle  
  )

  This call is used instead of the call to *VipOpenNic( )* that the user normally uses in order to obtain an interface to the NIC, if the user wishes to also initialize multicast settings. It was necessary to add this function to the VIPL in order to allow the user to specify a
BASE multicast address to be associated with the handle returned to the user for an
interface. This is the only knowledge that the user has of a multicast address, and it is
only required to insure that drivers on different nodes in a local network do not use the
same multicast addresses when setting up VI connections.

- **VIP_RETURN**
  
  VipMCConnectAccept ( 
  IN VIP_VI_HANDLE ConnHandle, 
  IN VIP_VI_HANDLE ViHandle 
  )

This call is used instead of the call to *VipConnectAccept* that is normally used for
accepting a connection to a remote VI, to allow for multiple connections to be accepted
for the same VI. Multiple remote VIs can be connected to the same local VI by calling
this routine multiple times.

7.2. **Connecting to Driver**

As mentioned in the previous section, the VIPL call *VipMCOpenNic* has been
added to allow the user to initialize multicast settings for their one-to-many VI
connections. The user is not required to use this call to get a handle to the NIC if they
plan on using multicast, they can still use the normal *VipOpenNic* call, but the Kernel
Agent driver will use the default base multicast address. The problem is that without
calling *VipMCOpenNic*, multiple machines in the network will use the same default
multicast addresses for setting up their connections and although the VIs will not actually
consume descriptors for messages that were meant for other VIs, there will be
unnecessary interruption of the processor in receiving a message meant for another node
and finding no VIs locally registered for that address. The call to *VipMCOpenNic*
allows the user to specify a base IP multicast address that is stored in the driver for future use when setting up one-to-many VI connections.

The format of a Class D user IP multicast address is 224.0.0.0 – 239.255.255.255, which translates into 4 bytes. A physical Ethernet address is 6 bytes, however, and thus an IP multicast address has to be mapped into a physical Ethernet address. As specified in Figure 6.3, the LSB of the first byte of the Ethernet address is always a 1 to specify multicast. In my implementation the user actually specifies a 5 byte IP Multicast address; the normal 4 byte address plus an additional byte that is used by the driver to complete the mapping and to allow the user a greater number of bits when specifying the base address. This not only insures the same machines do not use the same multicast addresses, but also allows for proper distribution of addresses in a network, which will be discussed in further detail in Section 7.5. The mapping of the IP multicast address to the Ethernet multicast address can be seen in Figure 7.1. The first three bytes of the base IP address specified by the user are stored as the BaseMCAddr and the last two bytes specified are stored as the NextMCAddr offset. The NextMCAddr offset is incremented each time a new connection request is accepted to create a new Multicast address for that connection. To create the Ethernet Address, the first byte is set to 1, the next three bytes of the address are set to the first three bytes of the multicast IP address, and the last two bytes are the last byte of the IP address plus an additional byte.
7.3. **VI Multicast Connection Management**

7.3.1 **Setting Up a VI Multicast Connection**

The connection setup procedure for setting up each connection in a one-to-many VI connection is basically the same as the procedure for setting up a point-to-point connection between two VI endpoints except for the call to `VipMCConnectAccept()` made by the user acting as server to accept a connection. Thus, when a user wants to setup multiple connections to the same VI, they must call `VipConnectWait()` to wait for each connection request, and then call `VipMCConnectAccept()` to accept each connection if the remote VI’s attributes match.

Upon calling `VipMCConnectAccept()`, an I/O control request is sent to the Kernel Agent driver and a driver entry point corresponding to this function enters. This entry point does the following: First, it checks the VI’s attributes and immediately returns with an error if they do not match. Second, if the remote attributes matched and this is the first connection to be accepted for this VI, it increments the NextMCAddr offset and obtains a new Multicast Address to send to the remote endpoint. This Multicast Address is stored in the VI structure on the server-side as the sending address for this VI. Third, a connection acceptance message is sent to the remote user containing the Multicast
Address associated with this connection that the remote VI endpoint will receive from. Note that if this was the first connection to be accepted for this VI, the VI will transition to the *Connected* state, but will still be able to continue accepting further connections at any point in the future. In addition, since the client VI's attributes must match the server VI's attributes in order for the connection to be accepted, and the user has no knowledge of the multicast address being used for a connection, this connection procedure provides some security for multicast.

On the client-side, when a connection acceptance is received from the server the following steps are taken: the Ethernet address of the remote VI's computer is added to the client's VI structure as the sending address for this connection, and the routine *GbitMCAAddLeaf( )* is invoked to register the VI for the multicast address received from the server and to add this multicast address to a hardware multicast address table (called a CAM Register) in the PEGNIC. The details involved in adding a multicast address to the CAM Register are described in Section 7.3.3. The Kernel Agent driver has a Multicast Address Registration table that is used for registering VIs to receive data for a multicast address associated with a connection. This table is hashed using the least significant 10 bits of the Multicast Address, and 1K entries are allocated to minimize the chance of multiple multicast addresses hashing to the same entry in the table. In addition, this particular number of entries was chosen to avoid a situation in Brazos where a node has VIs registered for different multicast addresses that were received from other nodes that hash to the same location in the table because their least significant byte is the same. This situation and the solution used in Brazos will be described in Section 7.6.
7.3.2 Tearing Down a VI Multicast Connection

When *VipDisconnect()* is called by the user on the client side of a one-to-many VI connection or when the server disconnects asynchronously, the routine *GbitMCRemoveLeaf()* is executed on the client in order to de-register the VI associated with a multicast address and to remove the multicast address it was associated with from the CAM structure if there are no additional associated VIs.

7.3.3 Multicast Address Receive Filtering

Upon entering DriverEntry, the Kernel Agent driver makes a call to the PEGNIC driver to set up a multicast address filter register on the card. This register specifies which types of packets the card will accept (all multicast packets, all multicast packets for which the NIC is registered in the CAM Register, no multicast packets, etc.).

Several modifications had to be made to the PEGNIC driver to program the NIC to accept certain multicast packets. Multicast Ethernet addresses had to be programmed by the hardware-level driver (Miniport) into an internal memory structure (CAM Register) on the PEGNIC to provide receive frame filtering. The structure of the CAM Register can be seen in Figure 7.2. As can be seen, there are six bytes in each entry to allow for the multicast address (bits 0 – 47), bit 48 for setting the type of entry (can be either VLAN or Multicast, but we only care about Multicast), and bit 49 for enabling that entry to be compared against an incoming multicast packet’s destination address.
To provide the Kernel Agent driver with an interface to add/remove Multicast addresses to the CAM Register on the PEGNIC, the *SetInformation( )* routine in the Miniport driver was modified to allow for two additional request types that are shown in Table 7.1 below. The insertion routine keeps a pointer to the last position in the hardware table that a multicast address was written to, and searches from there for the next available entry and places the specified multicast address in that position. The removal routine starts at the first position of the table and searches until it finds the specified multicast address, then removes it from the table and updates the pointer to index this position in the table if the index value is less than the current value (closer to the beginning of the table).

**Table 7.1 PEGNIC Driver Add/Remove Multicast Address Handlers**

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID_CUSTOM_INSERT_MULTICAST_ADDRESS</td>
<td><code>InsertMulticastAddress( )</code></td>
</tr>
<tr>
<td>OID_CUSTOM_REMOTE_MULTICAST_ADDRESS</td>
<td><code>RemoveMulticastAddress( )</code></td>
</tr>
</tbody>
</table>

The PEGNIC is currently limited to 64 entries in the multicast address memory table. If
a larger number of multicast addresses are required (as was the case in my implementation) the NIC can be set up to accept all multicast packets. In this case, the driver contains its own multicast group table in memory, and will refuse or accept packets using this table. There is a significant increase in overhead using this approach for two reasons: 1) the NIC is interrupted for every packet sent on the wire, and 2) the address registration verification is done through software. However, this is the only known solution to joining a large number of multicast groups without increasing the size of the memory on the NIC. In my implementation, I am using a hybrid approach that uses the hardware memory table as long as the number of multicast addresses that are required is less than or equal to 64. Once this number reaches the limit, the entire contents of the multicast address table are copied to the host memory and the multicast address filter is set to accept all multicast packets. This implementation maintains high performance for the average case of less than 64 addresses, but still functions if the number of addresses is higher. In order to insure that thrashing does not occur, the multicast address table is not copied back to the hardware table from host memory unless the number of addresses drops to several less than 64.

7.4. Data Transfer

After a user has a one-to-many connection for a VI, sending data to the connected VIs is just as it is with single connections. A call to VipPostSend() posts a VI on the send queue and uses the sending address specified in the VI structure to send the data, which should be a multicast address. As depicted in Figure 7.3, the client VIs that are connected should have added this multicast address to their hardware table (as mentioned in the previous sections) upon connection acceptance and they will receive any data sent
to that address. If the client VIs need to respond to the Server VI, they can simply send on their send address, which is the remote address of the server.

![Diagram of One-to-Many VI Communication]

**Figure 7.3 One-to-Many VI Communication**

When a multicast packet is received and is found in the hardware multicast address table on the PEGNIC (or the table checking is disabled), it is passed up from the PEGNIC to the *ViaReceive( )* routine. This routine was extended to check the header of incoming packets for a multicast address, determine which VIs must receive the packets (using the VI Multicast Address Register Table), and deliver the packets to descriptor buffers for each of those VIs.

### 7.5. VI Hardware Support for Multicast

Supporting multicast in VI hardware would require minor additions to the NIC hardware structures and memory. First, a VI-compliant NIC needs a Multicast address group table similar to that on the GNIC II. This table is used for filtering which packets a
NIC should accept based on the destination multicast address. The group table should be able to hold at least 256 addresses. Second, the NIC requires increased memory for holding information for all of the remote VIs a VI is connected to, in addition to the memory for each VI typically on a VI-compliant NIC. Third, the NIC requires a multicast group register table containing an entry for each multicast group that specifies which VIs are registered for that group. This would allow for fast on-board determination of which VIs should receive an incoming VI message.

Due to the difficulties in supporting a reliable multicast protocol in hardware, support for multicast was not included in the VI specification. Supporting multicast in a software emulation of VI poses fewer technical challenges, and therefore this support might best be served as an optional addition to the existing VI specification.

7.6. Extending Brazos to Use VI Multicast

Brazos currently uses IP Multicast with TCP/IP for communicating between nodes and updating other nodes when modifications to pages occur. An 8-bit vector represents a node mask where each bit represents the corresponding node ID in the system. Each possible bit vector value is associated with a multicast address, and in the initialization of the WinSock network each node registers those multicast addresses that correspond to bit vectors that contain a ‘1’ for their node. In other words, each node registers to the multicast address that their node should receive on. It is not necessary to join a special group in order to send to a subset of other nodes, a send to a multicast address can be performed without any registration.

In order to use Multicast with VI in Brazos, the method used for IP Multicast in Brazos had to be conformed to be used with VI. As before, an 8-bit vector represents the
node mask, but the difference being since VI is connection-oriented for every node mask value where a node’s bit position is ‘0’, that node must have a VI that is the server and is connected to those nodes whose bit positions are ‘1’. In addition, for those values where a node’s bit position is ‘1’, the node must request a connection with a client VI to each possible sender. Thus, each node mask value in which a node’s bit position is ‘0’ will map directly to a VI that can be used to multicast to the nodes whose bit positions are set, but those mask values where a node’s bit is ‘1’ do not map directly to a single VI, because remote VIs on multiple other machines could have the same node mask that is connected to two different VIs on a node (example: 3 nodes, mask value is 001; both nodes 1 and 2 will have a VI multicast connection to two different VIs on node 0). For each node, there are a total of $2^{\text{NUM\_NODES}-1} - 1$ VIs that will be the servers for multicast connections to other VIs, and $2^{\text{NUM\_NODES}-2} \times \text{NUM\_NODES}-1$ that will be the clients receiving on a multicast connection.

### 7.6.1 MC Address Distribution

As mentioned in Section 7.2, VipMCOpenNic() can be used to specify a base multicast address when obtaining a handle to the NIC. In Brazos, a different base multicast address was specified for each node to insure that different connections were not using the same addresses. Just insuring that the multicast addresses used were different was not sufficient to obtain optimal performance, however. The reason is that a situation can occur where two different machines in the network accept connections to VIs on another remote machine with different multicast addresses, but the same values in the lower bytes that are used to hash into the Multicast Address VI registration table. Thus, both of these addresses hash into the same location in the table and when packets
are received for each of those addresses in the ViaReceive() routine a search is required to find the VI with the matching address instead of a direct hit immediately. Thus, the multicast addresses had to be partitioned across all nodes in the Brazos system such that each node has a set of multicast addresses whose lower byte values are unique and will hash into a unique location in the VI registration table on each node.

7.6.2 Allocating VI Buffers

When Brazos is initialized, VI buffers for descriptors are pre-allocated and pre-posted to the receive queues of VIs. Using Brazos with non-multicast VI connections requires two VIs per remote node (one for each channel) and a constant number of buffers are allocated for each of these VIs and pre-posted. For VI with multicast, the VIs that are the senders for multicast connections do not require any buffers for receiving because they are only used for sending, while the VIs that are the clients for multicast connections will require buffers for receiving. Allocating a fixed number of buffers per receiving VI imposes a limit, however, because unlike non-multicast VI communication in Brazos, the number of VIs per node increases exponentially as the number of nodes increases. Thus, simply allocating a fixed number of buffers per VI will lock down too many pages in the system when the number of nodes used increases past a certain threshold, and Windows NT specifically limits the number of pages that the user can pin down for general use to 50,000 or less depending on how much memory a computer has.

There are several possible solutions to this problem. There are two channels for communication used in Brazos, an asynchronous channel and a synchronous request channel. Thus, there are two sets of VIs required, one for the asynchronous channel for each multicast connection, and one for the synchronous request channel for each
multicast connection. So we need to optimize the number of descriptors posted on the VI queues for each of these channels.

The communication channel VIs receive requests from remote threads, and there can only be one outstanding request by the requestor thread at a time. The solution to posting descriptors to the communication channel VIs that have multicast connections then is to only post \( x \) descriptors simultaneously, where \( x \) is the number of threads on the node on the other side of the connection. This reduces the number of descriptors per communication channel VI substantially.

The difficulty comes in with the asynchronous channel VIs, because it is not known how many updates might be received on a VI at a time. The simple solution that I am currently using in my implementation is using trial-and-error on a per-application basis by posting buffers to the asynchronous VIs until there are no problems running out of VIs. The better solution that is currently being implemented in Brazos and will be complete in the next couple of weeks is a send-throttling protocol that insures that a thread doesn’t ever send on a VI for which there are not receive descriptors posted at the other end. This is what is currently being done in WSDLite.

7.6.3 Setting Up Multicast Connections

Brazos implements regular VI communication by creating two VIs on each node for each remote node (one for Asynchronous channel and one for Comm channel) and then sets up connections between those VIs by creating a separate thread for every other node in the system. These threads use a simple ordering scheme where the node with the lower ID waits for connections and the higher node makes connection requests to set up the VI connections.
In order to setup the VI connections between nodes for multicast the situation is much more complex, because each node requires connections from all other nodes, possibly at the same time, and there is no explicit order. Thus, it was necessary to implement a connection setup order using synchronization messages and a set of regular communication VIs. Initially, a single VI is created corresponding to each remote node in the system, connections are setup between these VIs, and a single descriptor is posted on each receive queue.

To keep this implementation simple and since connection setup time is not in the critical path, I use only one thread on each node to setup the connections. Each thread calls the routine shown in Figure 7.4 below.

**Figure 7.4 Procedure for setting up MC Connections**

VOID MC_VISetUpConnections ( )
{
   // If my Node ID is zero then I will act as the server and set up my multicast send
   // connections first
   if (NODE_ID == 0) {
      ConnSetupFinished = MC_VIServerWaitForConnections( );
   }

   // stay in this loop until all multicast connections are setup
   while (1) { 
      MC_VIRecvConnMsg(&connMsg);

      switch(connMsg.Type) {
         case MC_CONN_REQ:

            // make a connection request to the remote node
            ViError = VipConnectRequest(hS_AsynchVI[currConnIndex],
             LocalAddr, remote,
             INFINITE, &remoteViAttribs);
            if (ViError != VIP_SUCCESS) {
                DSM_Error("Error connecting to node \n");
            }

            // check that requirements match
if (remoteViAttribs ReliabilityLevel != localViAttribs ReliabilityLevel ||
    remoteViAttribs QoS != localViAttribs QoS) {
    DSM_Error("requested and granted reliability factors do not match\n");
}

// make a connection request to the remote node
ViError = VipConnectRequest(hS_CommVI[currConnIndex], LocalAddr,
    remote, INFINITE, &remoteViAttribs);
if (ViError != VIP_SUCCESS) {
    DSM_Error("Error connecting to node \n");
}
// check that requirements match
if (remoteViAttribs ReliabilityLevel != localViAttribs ReliabilityLevel ||
    remoteViAttribs QoS != localViAttribs QoS) {
    DSM_Error("requested and granted reliability factors do not match\n");
} break;

case MC_SERVER:
    ConnSetupFinished = MC_VIServerWaitForConnections(); break;

case MC_EXIT:
    ConnSetupFinished = TRUE; break;

default:
    DSM_Error("received an unknown message type while setting up MC
    connections.\n");
}

// If all of the connections have been setup then this thread exits
if (ConnSetupFinished) {
    break;
} // end while(1)
}

The basic protocol requires that each node take turns acting as the server for its multicast connections used for sending to other nodes, and the other nodes act as clients, waiting in a routine that polls a completion queue for any multicast connection setup messages. There are three message types that were created for setting up multicast connections:
MC_CONN_REQ, MC_SERVER, and MC_EXIT. The node that is acting as the server for setting up connections sends these messages to remote VIs depending on the action it wants that remote node to take.

Initially, Node 0 is the server, and it executes a routine that loops through all of the values of the node mask and for each value that contains a '0' for its bit mask it knows it needs to set up connections with remote client VIs. So, for each such mask value the server does the following for each bit that is set to '1': First, it sends a MC_CONN_REQ message to the a connection setup VI corresponding to that node ID, and then it calls VipConnectWait() to wait for the request. The MC_CONN_REQ message just informs the corresponding node to make a connection request to the server, and provides it with the Node ID to make the request to and the discriminator to use. The entire connection setup procedure is diagrammed in Figure 7.5. The remote node, upon receiving the message, makes the connection request to the Server node using the discriminator specified and an asynchronous VI. When the Server receives the request, it accepts the connection using VipMCConnectWait() with its communication VI and waits again for another request. The client receives this connection acceptance and then makes a connection request for its communication VI, and the server accepts that connection with its asynchronous VI. The client now goes back to waiting for any additional messages.

Once Node 0 acting as server has gone through the entire node mask and setup all necessary VI multicast connections, it now sends an MC_SERVER message to the next Node ID informing them it is their turn to become a server and setup their connections.
Node 0 then exits the server connection setup routine and becomes a client waiting for connection setup messages.

This procedure continues for every node until the last node (NodeID = NUM_NODES-1) has setup all of its connections. This node will then send an MC_EXIT message to each of the other nodes in the system which just informs the other nodes that all of the connections have been setup and they can exit the connection setup routine in Figure 7.4. There is no need for a barrier for synchronization after this point because we know that we already have connections between all of the VIs when the MC_EXIT message is received on a node, and thus we can start the application.

Figure 7.5 VI Multicast Connection Setup Procedure
8. Experimental Methodology

All data presented in Section 9 were obtained on a network of four Compaq Proliant 5500 Server systems connected by Gigabit Ethernet. Each server has four 450 MHz Xeon processors with 1Mbyte of L2 cache and 512 Mbytes of interleaved EDO DRAM. Each system has two 33 MHz, 32-bit PCI busses, and all run Windows NT 4.0, Service Pack 4. The Packet Engines Gigabit Ethernet card has a peak bandwidth of 125 Mbytes/sec, a switch cut-through latency of 500ns (64 byte, FIFO), and is repeater-based with fiber-optic links.

8.1. Performance Benchmarks

To evaluate the performance of my VI software implementation with and without multicast I have used two classes of test software: low-level network benchmarks and five scientific software distributed shared memory (DSM) applications. There were two low-level tests performed: the bandwidth and latency test which was run between two computers using only a single processor for application processing on each computer, and the time to service multiple requests test, which was run between one server computer and two to three client computers. The DSM application runs were made on a four-node configuration with one thread running per node, in the Brazos parallel programming environment [16].

8.1.1 Bandwidth and Latency

To measure observed bandwidth and latency I used a low-level benchmark that sends a varying amount between the two nodes for a large number of iterations. The message size was varied from 8 bytes to 32K bytes. This test was intended to verify that
adding multicast support did not degrade the performance of the point-to-point VI implementation already established.

8.1.2 Time to Service Multiple Requests

To compare the performance of VI with multicast support to the performance without multicast support, I ran two applications and compared their performances. The first application involved multiple Server VIs on one machine with point-to-point VI connections to multiple Client VIs on remote machines. The second application involved a Server VI on one machine with a Multicast connection to multiple client VIs on remote machines. In both of these applications, the server sends replies that are varied from 8 bytes to 32K bytes to multiple client requests of 28 bytes and the total time it takes the server to service the clients is measured over a large number of iterations.

8.2. Brazos DSM Application Benchmarks

The five applications that were used to evaluate the performance use shared memory to communicate between processes: LU Decomposition, Barnes-Hut, Water, FFT-3D, and SOR. The problem sizes of each of these applications are shown in Table 8.1.

<table>
<thead>
<tr>
<th>Application</th>
<th>Data Set Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>2K x 2K matrix, 64 element blocks</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>32K bodies</td>
</tr>
<tr>
<td>Water</td>
<td>4K molecules for 5 time steps</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>$2^7 \times 2^7 \times 2^6$ points</td>
</tr>
<tr>
<td>SOR</td>
<td>2K x 2K matrix, 64 element blocks</td>
</tr>
</tbody>
</table>
LU Decomposition

I used a version of LU Decomposition from the SPLASH 2 Benchmark suite. This application splits a matrix $A$ into upper-triangular ($U$) and lower-triangular ($L$) matrices such that $L \times U = A$. A triangular set of equations can then be trivially solved using $L$ and $U$ with forward substitution or back-substitution. The version I used blocks the matrix into $b \times b$ size blocks, where $b$ is the blocking factor. I used a blocking factor of 64 on a 2048 x 2048 element, which results in 8192 blocks. By allocating data in each block contiguously, false sharing is reduced and temporal locality is enhanced, and an even distribution of the blocks among the threads is provided.

Barnes Hut

This application taken from the SPLASH benchmark suite is a hierarchial N-body problem that simulates the interaction of a system of bodies in 3-D space over a number of time-steps. An octree data structure was used for maintaining information on each body: the leaves maintain information on each body and the internal nodes represent space cells. This program has highly unpredictable access patterns because in order to compute the forces on each body, the program must traverse part of the octree once for each body. In my experiments, I ran Barnes with a data size of 32,768 particles.

Water

Water is an application in the SPLASH benchmark suite that calculates forces and potentials over a specified number of time steps in an N-body system of water molecules contained in a cubical box. Newtonian equations of motion for water molecules have to be solved during each time step, and the inter-molecular and intra-molecular potentials summed to provide the potential of the entire system. To calculate all potentials it is
necessary to evaluate $n^2/2$ potentials at each time step. To reduce this computation, however, a notion of a spherical cutoff radius is used, which specifies a half box length as the maximum distance between particles whose interactions are evaluated. I ran a 4K molecule execution for 10 time steps.

**Fast Fourier Transform (3D-FFT)**

This application, taken from the NAS benchmarks, numerically solves a partial differential equation using forward and inverse Fast Fourier Transforms. The input to the algorithm is a complex array $A$ of size $n1 \times n2 \times n3$. A series of 1-D FFTs are performed on the input array. First, a $n3$-point FFT is performed on each $n1n2$ complex vector. Next a $n2$-point FFT is performed on each $n1n3$ vector. No communication is required to complete both of these 1-D transforms. A large amount of communication is required for the next step, which is a transposing of all the results of each thread to the next thread to use that portion of the input array, and thus the speedup of the application is limited. The last step involves performing a $n1$-point FFT on the set of $n2n3$ vectors.

**Successive Over Relaxation (SOR)**

The version of *Successive Over Relaxation* (SOR) that I used is taken from the Treadmarks [46] distribution of sample DSM applications. SOR is a nearest-neighbor algorithm used to solve partial differential equations. The implementation that I used is called a red-black SOR, since the source matrix is split up into two separate input matrices in a checkerboard pattern. The first part of an iteration involves computing each value in the black matrix by taking the average of the four values from the surrounding red matrix. Likewise, in the second part of an iteration, the red values are calculated from the black matrix. This matrix is distributed among all the available threads by
allocating $\frac{NumRows}{NumThreads}$ contiguous rows to each thread and limiting the shared values to those along the border between two threads. I used an input matrix of size 2048 x 2048 integers, and iterate over the matrix for 1000 iterations.
9. Experimental Results

9.1. Low-Level Benchmark Results

In this section, I present results from the low-level benchmarks described in Section 8.1. Figures 9.1 and 9.2 show the performance of UDP and VIA on Gigabit Ethernet. The data in each figure is plotted on a log-log scale to more clearly show the performance at small message sizes.

9.1.1 Observed Bandwidth and Latency

Figure 9.1 shows the observed throughput in Mbit/sec for WinSock and VI with Gigabit Ethernet for message sizes between 8 bytes and 32 Kbytes. We see that the VI implementation achieves a higher peak throughput (an average of 1.9 times higher) than UDP across all message sizes, in large part due to the reduced software overhead associated with the VI Architecture. In addition, we see that the VI implementation achieves an even more significant increase in throughput over TCP/IP (an average of 2.20 times higher), due to additional overhead involved in reliability and flow control features of TCP. Multiple packet copying is also a problem in the protocol stack implementations, whereas the VI driver implementation accesses the user’s registered buffer directly from kernel space in the context of the user process using Neither I/O. We can also attribute the increase in throughput over WinSock to the use of the inter-driver communication model described previously, since this avoids the I/O request creation and completion overhead inherent in the WinSock Protocol drivers. We notice that at a message size of 4096 bytes, the Ethernet/UDP throughput increases substantially. This
can be attributed to a feature of the Packet Engines Gigabit Ethernet Card that issues an interrupt after a certain number of packets have been received. A vector on the card specifies the number of packets per interrupt and this is by default 3 packets in Ethernet. If this number of packets is not received within a certain timeout, an interrupt is issued immediately. Since a 4096 byte message requires three packets, the throughput increases because the card will issue an interrupt as soon as the message has arrived, whereas for smaller messages the timeout will occur before an interrupt is issued, incurring a larger receive time at the receiver. The throughput of TCP increases at 2048 bytes due to overlapping of DMA and packet setup time that is more significant with reliable transmission. At 4096 bytes, the same situation occurs as with UDP, however due to the overhead involved in the reliability and flow control features of TCP, it does not maintain as high a throughput when the message size reaches 8096 bytes.
Figure 9.1 Observed Throughput with Varying Message Size

Figure 9.2 depicts the observed unidirectional message latency, measured in microseconds, relative to the message size. We see here that the VI implementation provides a significant 57% reduction in latency over UDP and TCP/IP for all message sizes. The ideal (wire-time) latency is also depicted on the graph. Both the WinSock and VI implementations deliver lackluster performance relative to wire time for small messages. We also notice the sudden decrease in the latency of Ethernet/UDP when the message size reaches 4096 bytes. This again can be attributed to the receive interrupt vector default setting of three packets per interrupt on the PEGNIC as described in the
analysis of the observed throughput. In addition, the latency of Ethernet/TCP decreases at 2048 bytes and at 4096 bytes for the same reason as with the observed throughput.

![Graph showing latency vs. message size](image)

**Figure 9.2 Observed Latency vs. Ideal Latency (wire time)**

### 9.2. VI Multicast

Figure 9.3 shows the time to service multiple requests at the server for two and three clients for VI with and without multicast. We can see here that the time (in msec) is substantially less for VI with multicast for two clients. Further performance benefits are achieved with three clients. Because the server VI in the multicast version of the application only has to send one message in response to each of the clients’ requests, as
the number of clients increases, the amount of time saved from multicasting the reply grows exponentially.

![Graph showing Time to Service Clients vs Message Size in Bytes.]

**Figure 9.3 Time to Service Multiple Requests**

The average reduction in time using multicast with two clients was 37.25%, which is definitely a significant reduction. Increasing the number of clients to three, the average reduction in service time increases to 52.10%. We notice that there is no real difference between the time to service multiple clients for two and three clients for VI with multicast in this graph. This is due to the fact that the cost for sending a multicast message to answer all of the clients at once is fixed, and the only time that should increase as the number of nodes increases is the time for all of the clients to make their requests. Since the clients use very small messages (28 Bytes) for their requests, the difference in time is negligible between two and three clients. For VI without multicast,
on the other hand, there is a noticeable increase in the time to service multiple clients because the server has to send a separate message to each client.

9.3. Brazos DSM Application Results

I used five parallel scientific programs to evaluate the application performance of the software implementation of the VI Architecture with and without multicast. I ran each application with two, three, and four nodes, one thread per node using Ethernet UDP with multicast, Ethernet VIA, and Ethernet VIA with multicast. For each application, I present a bar graph that depicts the execution time of Ethernet VIA and Ethernet VIA with multicast relative to the execution time of the Ethernet UDP version. Each bar is divided up into the time that the application spends idle, executing user code, or performing kernel calls, as reported by the operating system. The user time includes both application and runtime system execution time, and the kernel time should contain much of the time spent in drivers and the network protocol stack. The most important component of execution time for my analysis of the performance is the idle time, since this indicates the total time spent waiting on replies for requests to remote nodes, as well as the time spent synchronizing (waiting at barriers). Since the idle time includes synchronization time, however, I also measured just the amount of time that is spent waiting for requests to remote nodes (the time from just before calling VipPostSend( ) until just after receiving the reply). Note that this time is still not only the data communication time, but also includes the time it takes for the remote node to get their diff (a runlength encoding of the changes made since the last global synchronization point) for the page to the requesting process and whatever computation that involves, as well as the time in which the process that is getting the diff is interrupted by the processor
if the processor is busy computing. However, this measurement represents the best approximation of the total time spent communicating.

The limiting factor on performance of parallel computing applications is the time spent communicating between the nodes in the system relative to the total execution of the application. Since the size of messages and the rate of message transfer is directly related to the amount of communication that will occur on the network, I include the distribution of message counts and sizes for the shared memory applications I used. These numbers were obtained using ServerNet / UDP in previous experiments by the Brazos group [26]. I expect that applications that transfer a large number of messages per second will benefit more from my VI implementation than those that do not communicate as frequently, and that I should see a significant performance increase for such applications across all message sizes.

Table 9.1 Message Distributions for DSM Benchmarks

<table>
<thead>
<tr>
<th>Message Size (Bytes)</th>
<th>Barnes</th>
<th>3D-FFT</th>
<th>LU</th>
<th>Water</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2125</td>
<td>2068</td>
<td>1119</td>
<td>727</td>
</tr>
<tr>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.10%</td>
<td>54.90%</td>
</tr>
<tr>
<td>25.40%</td>
<td>42.90%</td>
<td>58.20%</td>
<td>12.10%</td>
<td></td>
</tr>
<tr>
<td>9.00%</td>
<td>0.10%</td>
<td>0.20%</td>
<td>0.10%</td>
<td></td>
</tr>
<tr>
<td>14.00%</td>
<td>0.10%</td>
<td>0.20%</td>
<td>20.60%</td>
<td></td>
</tr>
<tr>
<td>4.60%</td>
<td>0.00%</td>
<td>0.20%</td>
<td>3.10%</td>
<td></td>
</tr>
<tr>
<td>12.20%</td>
<td>0.00%</td>
<td>0.10%</td>
<td>0.70%</td>
<td></td>
</tr>
<tr>
<td>29.00%</td>
<td>0.00%</td>
<td>0.10%</td>
<td>0.10%</td>
<td></td>
</tr>
<tr>
<td>3.70%</td>
<td>0.20%</td>
<td>0.10%</td>
<td>3.10%</td>
<td></td>
</tr>
<tr>
<td>2.10%</td>
<td>53.30%</td>
<td>22.80%</td>
<td>5.30%</td>
<td></td>
</tr>
<tr>
<td>0.00%</td>
<td>3.30%</td>
<td>18.10%</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td></td>
</tr>
</tbody>
</table>

9.3.1 3D-FFT

Figure 9.4 below depicts the execution times of 3D-FFT relative to UDP. As can
Figure 9.4 3D-FFT Relative Execution Times

be seen in Table 9.1, 3D-FFT exhibits the second-highest communication rate of the applications studied and approximately 50% of the messages are fairly large. As expected for such communication, this application displays one of the higher performance improvements using VIA over UDP. Table 9.2 shows the reduction in total execution time for each application and node configuration for VIA and VIA with multicast relative to UDP. As shown, there is an 8.26% reduction in execution time with VIA over UDP. At first glance, this does not seem like a large performance improvement. The main thing to remember here though is that if the communication to computation ratio is smaller than even if the communication time has been reduced significantly, the reduction in the total execution time may not be as significant. If we look at the bar graph in Figure 9.4, we see that the idle time has decreased by over 10% and the kernel time has also decreased for 2 nodes with VIA. This still is not as significant a performance increase as is shown with the low-level benchmark. As
mentioned earlier, the idle time also contains the synchronization time at barriers, so to better evaluate the amount of time spent communicating, I also measured the time spent waiting for replies after making requests to other nodes. This data can be seen for each application in Table 9.3. In this table, we see that the actual percentage of time spent waiting for data is only approximately 7.56 seconds (30% of total execution time) with UPD for 2 nodes, and the improvement with VIA actually provides a 22% reduction in this time. Furthermore, the performance improvement of the actual data communication time should be higher than this because the data waiting time includes the time spent while the remote node is servicing a request for a diff. As the number of nodes increases the percentage of time spent waiting for data increases, while the benefit of VIA decreases because of a lack of multicast.

Table 9.2 Reduction in Total Execution Times

<table>
<thead>
<tr>
<th>Application</th>
<th>Total Exec UDP</th>
<th>% Red. VIA</th>
<th>% Red. VIA Mult.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-FFT (2)</td>
<td>25.41</td>
<td>8.26%</td>
<td>9.09%</td>
</tr>
<tr>
<td>3D-FFT (3)</td>
<td>21.66</td>
<td>9.93%</td>
<td>14.42%</td>
</tr>
<tr>
<td>3D-FFT (4)</td>
<td>17.48</td>
<td>4.41%</td>
<td>12.76%</td>
</tr>
<tr>
<td>Barnes (2)</td>
<td>54.20</td>
<td>11.77%</td>
<td>11.38%</td>
</tr>
<tr>
<td>Barnes (3)</td>
<td>39.11</td>
<td>6.55%</td>
<td>10.61%</td>
</tr>
<tr>
<td>Barnes (4)</td>
<td>31.71</td>
<td>-2.11%</td>
<td>12.46%</td>
</tr>
<tr>
<td>LU (2)</td>
<td>34.19</td>
<td>4.15%</td>
<td>3.83%</td>
</tr>
<tr>
<td>LU (3)</td>
<td>25.02</td>
<td>6.00%</td>
<td>6.63%</td>
</tr>
<tr>
<td>LU (4)</td>
<td>19.62</td>
<td>7.24%</td>
<td>7.49%</td>
</tr>
<tr>
<td>SOR (2)</td>
<td>134</td>
<td>1.59%</td>
<td>1.47%</td>
</tr>
<tr>
<td>SOR (3)</td>
<td>93.58</td>
<td>2.74%</td>
<td>3.10%</td>
</tr>
<tr>
<td>SOR (4)</td>
<td>72.72</td>
<td>3.82%</td>
<td>4.44%</td>
</tr>
<tr>
<td>Water (2)</td>
<td>96.56</td>
<td>4.27%</td>
<td>4.59%</td>
</tr>
<tr>
<td>Water (3)</td>
<td>68.88</td>
<td>6.05%</td>
<td>6.20%</td>
</tr>
<tr>
<td>Water (4)</td>
<td>54.65</td>
<td>7.43%</td>
<td>7.80%</td>
</tr>
</tbody>
</table>

As expected, the improvement using multicast is approximately the same as VIA without multicast for only two nodes. With two nodes there was no overhead inherent in
the multicast implementation of any application that would detract from the base performance. However, with three nodes we see a slightly higher performance improvement with VIA with multicast because of the additional communication.

Table 9.3 Reduction in Time Spent Waiting for Data

<table>
<thead>
<tr>
<th>Application</th>
<th>UDP Data Wait Time</th>
<th>% Red. VIA</th>
<th>% Red. VIA Mult.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-FFT (2)</td>
<td>7.56</td>
<td>22.01%</td>
<td>21.77%</td>
</tr>
<tr>
<td>3D-FFT (3)</td>
<td>8.44</td>
<td>19.43%</td>
<td>27.25%</td>
</tr>
<tr>
<td>3D-FFT (4)</td>
<td>7.44</td>
<td>5.24%</td>
<td>24.73%</td>
</tr>
<tr>
<td>Barnes (2)</td>
<td>11.75</td>
<td>49.96%</td>
<td>50.55%</td>
</tr>
<tr>
<td>Barnes (3)</td>
<td>8.69</td>
<td>26.01%</td>
<td>44.53%</td>
</tr>
<tr>
<td>Barnes (4)</td>
<td>7.68</td>
<td>-16.80%</td>
<td>46.48%</td>
</tr>
<tr>
<td>LU (2)</td>
<td>1.58</td>
<td>48.10%</td>
<td>48.10%</td>
</tr>
<tr>
<td>LU (3)</td>
<td>2.20</td>
<td>42.73%</td>
<td>48.18%</td>
</tr>
<tr>
<td>LU (4)</td>
<td>1.60</td>
<td>53.13%</td>
<td>47.50%</td>
</tr>
<tr>
<td>SOR (2)</td>
<td>2.46</td>
<td>50.00%</td>
<td>48.78%</td>
</tr>
<tr>
<td>SOR (3)</td>
<td>3.32</td>
<td>43.07%</td>
<td>50.30%</td>
</tr>
<tr>
<td>SOR (4)</td>
<td>3.77</td>
<td>36.29%</td>
<td>48.81%</td>
</tr>
<tr>
<td>Water (2)</td>
<td>2.63</td>
<td>42.59%</td>
<td>43.73%</td>
</tr>
<tr>
<td>Water (3)</td>
<td>2.15</td>
<td>38.14%</td>
<td>44.19%</td>
</tr>
<tr>
<td>Water (4)</td>
<td>1.96</td>
<td>30.61%</td>
<td>37.76%</td>
</tr>
</tbody>
</table>

With 4 nodes the performance benefits with VIA are not very high because of the advantage that Ethernet has in being able to use multicast. Table 9.4 shows the reduction in messages sent due to multicast, and we can see that the number of messages that had to be sent was reduced by 22% using multicast. If we look at the performance with VIA with multicast this conclusion is confirmed, as this implementation is also able to take advantage of multicast and thus has a much higher benefit over UDP.
### Table 9.4 Reduction in Number of Sent Messages

<table>
<thead>
<tr>
<th>Application</th>
<th>Total Rcv Msqs</th>
<th>Total Sent Msqs</th>
<th>% Red. In Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-FFT (3)</td>
<td>61018</td>
<td>51719</td>
<td>15.24%</td>
</tr>
<tr>
<td>3D-FFT (4)</td>
<td>74321</td>
<td>57954</td>
<td>22.02%</td>
</tr>
<tr>
<td>Barnes (3)</td>
<td>146258</td>
<td>85354</td>
<td>41.64%</td>
</tr>
<tr>
<td>Barnes (4)</td>
<td>239312</td>
<td>100881</td>
<td>57.84%</td>
</tr>
<tr>
<td>LU (3)</td>
<td>21298</td>
<td>17096</td>
<td>19.73%</td>
</tr>
<tr>
<td>LU (4)</td>
<td>17515</td>
<td>17260</td>
<td>1.46%</td>
</tr>
<tr>
<td>SOR (3)</td>
<td>31484</td>
<td>25985</td>
<td>17.47%</td>
</tr>
<tr>
<td>SOR (4)</td>
<td>49978</td>
<td>38978</td>
<td>22.01%</td>
</tr>
<tr>
<td>Water (3)</td>
<td>62724</td>
<td>58569</td>
<td>6.62%</td>
</tr>
<tr>
<td>Water (4)</td>
<td>88872</td>
<td>79784</td>
<td>10.23%</td>
</tr>
</tbody>
</table>

### 9.3.2 Barnes Hut

Barnes Hut uses multicast the most out of all the applications that I studied, reducing the number of sent messages by 58% with four nodes. It also has the highest message rate of all the applications studied, at 2100 messages per second as shown in Table 9.1. The fact that Barnes uses multicast extensively and has a high communication rate implies that we should expect to see a large benefit with VIA with only two nodes, but a higher benefit due to VIA with multicast for three and four nodes.

The performance results with Barnes were as expected, as shown in Figure 9.5. For a two node configuration, VIA outperforms UDP by 11.77% (reduction in execution time). For three and four nodes, however, the performance improvement with VIA is less, and VIA actually performs worse for 4 nodes because it does not take advantage of multicast. Looking at VIA with multicast, we see that indeed the ability to use one-to-many communication greatly improves the performance in Barnes, achieving approximately a 12.46% reduction in execution time with 4 nodes. Again, as was mentioned in the previous section the reduction in execution time is not totally representative of the reduction in communication time relative to UDP. Table 9.3 shows
a 44.53% reduction in the time spent waiting for data, which is a better approximation of the communication time reduction. In addition, Table 9.4 depicts a 60% reduction in the number of messages sent with 4 nodes, which supports the fact that VIA actually performs worse than UDP because of its lack of multicast.

![Figure 9.5 Barnes Hut Relative Execution Times](image)

9.3.3 LU

LU exhibits a moderate communication rate as the result of a situation that occurs in Brazos with this particular application that reduces the number of messages sent and decreases the usefulness of multicast. Due to the moderate communication rate, we did not expect to see a huge impact on the overall execution time of LU with VIA. Furthermore, since multicast is not that beneficial to LU we also did not expect to see a high increase in performance due to multicast with VIA. Figure 9.6 below demonstrates this to be the case, with a 7.24% reduction in execution time due to VIA and a slightly
higher reduction with multicast. Table 9.4 confirms the low use of multicast showing only a 1.46% reduction in the number of send messages. The reduction in data wait time shown in Table 9.3 confirms that we are impacting the communication time significantly with approximately a 50% reduction in data wait time with four nodes.

**Figure 9.6 LU Relative Execution Times**

### 9.3.4 Water

Water exhibits a low communication rate compared to other applications studied, and the majority of these messages are small, point-to-point synchronization messages. We therefore expected that VIA would have some impact on the execution time, but that multicast would not increase the benefits by much. This can be seen in Figure 9.7, where the execution time is decreased up to 7.4% with four nodes, but the increase due to
multicast is only slightly higher. Thus, the benefits due to VIA appear to increase as the number of nodes increase and the amount of communication in the application increases.

![Diagram of Water Relative Execution Times]

**Figure 9.7 Water Relative Execution Times**

### 9.3.5 SOR

In the version of SOR used, the "red" and "black" matrices are divided into contiguous rows among available threads, and only data along the boundary between two threads is shared. The result is a high computation to communication ratio, and thus the requirements for communication in SOR are low. I therefore did not expect a large performance increase using VIA. In addition, multicast offers no performance benefit with SOR because of pair-wise sharing [17], so it was not expected that VIA with multicast would make a difference. This basically proved to be the case, as can be seen
in Figure 9.8. There was only a 3.30% reduction in execution time due to VIA with four nodes and a slightly higher reduction at 4.44% with multicast.

![Figure 9.8 SOR Relative Execution Times](image)

9.4. Summary of Results

The low-level benchmark results in this section clearly show an improvement on message latency across all message sizes for VI over regular UDP and TCP/IP stack protocols. The tests done to measure the time to service multiple clients also definitely show that multicast can improve the performance of VI with messages that are destined for multiple remote nodes. For "real" applications, the improvement in low-level communication performance did not translate into similar observed improvements in application execution time. This is because the relative communication time was low. The applications that had the highest communication rates proved to benefit the most
from the VI Architecture. Those that made extensive use of multicast actually lowered the benefit of the normal VI Architecture, however the multicast version of VI proved to take advantage of the communication in such applications and still deliver the lower execution times over UDP. In some applications where the distribution of data was shared in pairs, multicast did not offer any significant performance benefits with VI (or Ethernet).

The multicast VI protocol that I have designed does not scale well as the number of nodes increases in the Brazos system. Since VI is connection-oriented (hence its multicast protocol has to also be connection-oriented), many-to-many communication such as is provided in IP multicast is not directly possible using the VI Architecture. Thus, in order to provide many-to-many communication all nodes require one-to-many connections to all subsets of the nodes in the system. This requirement grows exponentially with the number of nodes. In addition, Brazos uses multiple channels (asynchronous and communication) which doubles the number of VIs required. In order to provide better scalability for VI with multicast in general, another approach is needed. As mentioned in Section 10.2.1, another approach to multicast is to have one VI be a server for a many-to-many connection that has a one-to-many connection to all the client VIs, which have point-to-point connections to the server VI. Then, to send data from one of the clients to the rest of the VIs connected, a message can first be sent to the server VI which then multicasts it to all the client VIs. This reduces the number of VIs required substantially, but provides slower performance in that it requires a message being sent to a server node, and in addition the server node can be heavily loaded if many client VIs
wish to send at the same time. A hybrid approach could provide better scalability while not sacrificing performance. This approach has not been evaluated experimentally.
10. Related Work

10.1. User-Level Network Access Techniques

The Virtual Interface Architecture incorporates techniques for user-level communication and low latency that were used previously in various academic projects. These previous projects and how they relate to VI are discussed in this section. I also discuss other VI software implementations.

10.1.1 U-Net

Von Eiken et al. at Cornell implemented U-Net [8] (User-Level Network Interface), which provides a virtualized network interface to the user using a combination of hardware and operating system mechanisms. They argued that the entire protocol stack should be placed at the user-level and the operating system and hardware should provide the user with protected access directly to the network. This was the first full system to provide user-level network interfaces without any custom hardware or OS modification and which supports traditional networking protocols and higher performing parallel language implementations. Their central idea was to remove the kernel from the critical path of sending and receiving messages, eliminating system call overhead as well as providing the ability to streamline buffer management at user-level. This requires a multiplexing and demultiplexing device in order to enforce protection boundaries in the send and receive paths, and U-Net incorporates this directly in the network interface, essentially providing a virtual view of the interface at user-level.
The Virtual Interface (VI) Architecture is based largely upon the U-Net system. The U-Net architecture is composed of three main components: endpoints, communication segments, and message queues. This is almost the same as VI with the exception being that queues contain circular buffers for descriptors instead of linked lists and there is a limit on the number of descriptors. As can be seen in Figure 10.1, an endpoint is the user application's handle to the network interface and the message queues contain descriptors pointing to the data in the communication segment.

![Diagram of U-Net Interface](image)

**Figure 10.1 U-Net Interface Diagram**

Similar to VI, in order to send data the user places descriptors on the send queue and the NI flags the descriptor when it is done sending the data. Likewise, received data is copied to the appropriate segment and a descriptor is pushed onto the corresponding queue, and U-Net provides a polling or event-driven mechanism for the user to check the status of the queue. U-Net also supports emulated endpoints that go through the kernel in order to provide access to the NI and consume no network interface resources but do not
offer the same level of performance. These emulated endpoints are very similar to the VI interface provided by the software emulation that I have designed. The driver provides a kernel VI that the VI in user-space is associated with, and this kernel VI does the actual interfacing with the network card to use its resources for sending and receiving data.

![Diagram](image)

**Figure 10.2 U-Net Emulated Endpoints**

The multiplexing and demultiplexing of messages in U-Net uses a tag in each message to determine the destination endpoint and communication segment, just as with VI, which allows communication between arbitrary processes. The message tag also provides a protection boundary because the operating system has to determine the
appropriate tag based on the destination process and the route between two nodes, and any authorization checks necessary to ensure an application is allowed access to the network resources. This is similar to the concept of a protection tag in VI, that is associated per-process with descriptors and VIs. U-Net attempts to support true zero-copies through what is called Direct-Access U-Net architecture. This allows communication segments to span an entire process’ address space and lets the sender specify the offset within that segment to deposit the data directly by the NIC. This avoids the copy that an application might have to do from a buffer to the actual data structures it is working with. The main problem with this architecture, however, is that it is difficult to implement in hardware because the NI must have an MMU consistent with the main processors’ as well as handle incoming messages destined to an unmapped virtual memory page. This requires the NI to have memory mapping hardware, access to all of physical memory, and ability to handle page faults properly. The Virtual Interface Architecture does not support this feature, probably because of the hardware limitations that were discovered with trying to use this feature in U-Net.

The U-Net architecture was implemented on SPARCstations running SunOS 4.1.3 and a Fore SBA-100 ATM interface. This hardware interface was very simple and did not include payload CRC calculation and no reassembly of multi-cell packets, so the U-Net architecture had to be implemented in the kernel. Using this U-Net implementation they achieved a 65 us single ATM-cell round-trip latency and 15 Mbytes/sec bandwidth on an 8-node ATM cluster of standard workstations, and account 33% of the send overhead (14 us) and 40% of the receive overhead (10 us) to CRC computation.
10.1.2 Virtual Memory-Mapped Communication (VMMC)

Dubnicki et al. at Princeton University have designed a Virtual Memory-Mapped Communication (VMMC) model that provides direct data transfer between a sender’s and a receiver’s virtual address space [42]. The features of this model are as follows: it eliminates operating system involvement in communication, provides full protection, allows the user to manage buffers, provides zero copy protocols, and minimizes software overhead. VMMC essentially provides a mechanism for propagating data from a sender virtual address space to a receiver virtual address space, almost eliminating any CPU overhead for this to take place. The protection is fulfilled because communication can only take place if a receiver gives a sender permission to send data to a given area of the receiver’s address space. The receiver exports areas of its virtual address space where it wants to receive incoming data and the sender must import remote buffers that will be destinations for data. The approach that VMMC takes to communication is basically removing the explicit “send” and “receive” operations in many standard protocols and instead allowing the user direct access to the network interface while managing their own buffers. This frees the CPU to do other work and have these operations taking place transparently with minimal CPU interruption. This is basically where VI gets its RDMA operations, which are similar in that in order to do an RDMA Write the user just has to make a call to place an item on a queue and on the remote end it happens transparently. Likewise, RDMA Reads also occur transparently on the remote end and only require a few operations on the initiator end.

There are two data transfer modes supported by VMMC: deliberate update and automatic update. Deliberate update is an explicit user-initiated transfer of data from a
virtual address of the calling process to a remote address, similar to a normal network protocol send operation. With automatic update, the system implicitly performs an update on each write to a certain local memory address. In order to do this, a sender must first create an automatic update mapping which sets aside a region of the sender's virtual memory and maps it to a remote receive buffer (as seen in the Figure below).

![Figure 10.3 Automatic Update Mapping](image)

The CPU overhead to send data as a result of these modes can be very small; either one local write for an automatic update or a few user-level instructions for a deliberate update. There is no CPU overhead to receive data, because there is no explicit receive.

The VMMC Model also has the notion of a destination proxy space, which is a separate address space in each sender process that is used for addressing imported receive buffers. When a sender imports a receive buffer, a representation of that buffer is mapped into its destination proxy space. VMMC also provides notifications to support transfer of control and to notify receiving processes about external events. Similarly to VI, notifications can be associated with each exported buffer for a receiving process.
Figure 10.4 Mapping Receive Buffer into Sender's Destination Space

The software implementation of VMMC in SHRIMP-II consists of five parts: a VMMC daemon thread running on each node, a user-level network interface driver, VMMC-specific kernel extensions, the VMMC basic library, and compatibility libraries. This is similar to VI in that the VMMC library has direct access to the network (SHRIMP-I required a system call) and it also provides the user with a library of calls to abstract the user from the details. In addition, kernel extensions are required to lock receive buffer memory and to create destination space mappings, which is similar to registering descriptor buffers for VIs.

Virtual Memory-Mapped Communication has been implemented on a Myrinet network of PCI-based PCs [8]. Myrinet is a high-speed local-area network or system-area network for computer systems. It is composed of point-to-point links that connect hosts and switches and it can deliver up to 1.28 Gbits/sec bandwidth in each direction [43]. The Myrinet NIC has a 32-bit, 33MHz LANai control processor with 256 KB of SRAM. The SRAM is used for network buffers as well as for storing code and data for
the LANai processor. The interface also includes three DMA engines, two that are used for moving data between the SRAM and the network, and one that is used for moving data between the SRAM and the host memory over the PCI bus. The LANai processor executes a control program which manages the operation of the DMA engines and implements a low-level communication protocol. The host memory cannot be accessed directly by the processor, so it must use the host-to-LANai DMA engine to transfer data to and from host memory.

The software implementation of VMMC in Myrinet now consists of four parts: a VMMC daemon thread running on each node, a kernel-loadable VMMC driver, a network mapping LANai control program, and a VMMC LANai control program that actually implements VMMC. When the system boots, each VMMC daemon loads a specific LANai control program that was derived from software provided by Myrinet and sets up the mapping of the network. Once this has completed, each VMMC daemon extracts the routing information and replaces the LCP with an LCP that implements VMMC. For sending data, each process has a separate send queue allocated in LANai SRAM. To submit a send request, the user process must now write the next entry in the send queue with the length of the data to be sent and a proxy address specifying the destination. This has more resemblance to VI than the SHRIMP implementation did in that the requests are queued for transmission by the user. On receipt of a message, the data is scattered appropriately according to two physical address in the header.

The Myrinet implementation was run in a system of four P166 PCs with 512KB L2 caches and 64MB of EDO memory, as well as Myrinet PCI NICs, connected to a Myrinet switch. The Myrinet network provides a network bandwidth of about 160MB/s,
however the total user-to-user bandwidth is limited by the PCI-to-SRAM DMA bandwidth. The PCI bus maximum bandwidth is almost 128MB/s for 64KB transfer units, but since there is no hardware support for scatter-gather, transfer units larger than a page (4KB) cannot be used because consecutive pages in virtual memory are not usually consecutive in physical address space. Thus, the bandwidth is limited by host-to-LANai bandwidth with a 4KB transfer unit, which is 110MB/s. The highest bandwidth achieved with VMMC using a ping-pong benchmark is 108.4MB/s, which is about 98% of the available bandwidth.

10.1.3 Active Messages

Von Eicken et al. at Berkeley argued that existing message passing multiprocessors have unnecessarily high communication costs and research prototypes of message driven machines demonstrate low communication overhead, but poor processor cost / performance. They implemented Active Messages, a simple communication mechanism that allows cost effective use of the hardware and offers tremendous flexibility.

They show that active messages are sufficient to implement the dynamically scheduled languages for which message driven machines were designed. They also outline a range of enhancements to mainstream processors to add hardware support for active messages. Implementations were developed on nCUBE/2 and CM-5 and evaluated using a split-phase shared-memory extension to C called Split-C.
10.1.4 OS Mechanisms

Druschel and Peterson have done extensive research on optimizing network performance using an OSIRIS ATM card built for the Aurora Gigabit testbed, which was designed specifically to support software experimentation. Only the most critical, high-speed functions are implemented in hardware to allow more control through software. Druschel et al. devised techniques for making the OS more effective in delivering network data to application programs, two such techniques are fast buffers (fbufs) [21] and application device channels (ADCs) [20]. These mechanisms are designed to improve the user-to-user throughput and latency.

The intent of the fbuf mechanism is to allow data to move across domain boundaries (device drivers, network protocols, applications) without sacrificing the bandwidth delivered by the network to the user-level process. It combines two well-known techniques for transferring data across protection domains: page remapping and shared memory. The general idea is that data is generated by some source module, passed through one or multiple software layers, and consumed by a sink module. The source domain is called the originator domain and the other modules are considered receiver domains. A message that arrives at the source domain is stored in a protocol data unit (PDU), where each arriving PDU is received into a buffer. On the output side, the PDU is fragmented into a smaller set of PDUs by simply representing each fragment by an offset and length into the original buffer. The path that the data takes from the source domain through multiple other domains to the final receiver domain is called an I/O data path. Now, in order for the data to transfer between these domains they use either move or copy semantics, which present the same performance but a copy is
required if the passing domain needs the data in the future. To enforce protection and security a buffer is made immutable, to prevent a situation where the data is received from an untrusted domain and modified by a malicious application. In addition, the cross-domain transfer facility must operate on pageable, rather than pinned-down, buffers, because a buffer could be passed to an untrusted application that keeps a reference to it for a long time. This is unlike VI, which requires its data buffers to be locked down so that the NIC knows where the buffers are at all times. To analyze the performance of their implementation, one experiment they ran was a loopback test to measure end-to-end throughput of UDP/IP using a null modem connection of the Osiris NICs. Using this test, the maximal throughput achieved is 285 Mbps, 55% of the network bandwidth supported by the network link.

Application device channels take the approach that fbufs had one step further to allow the application program restricted, but direct, access to the OSIRIS network adapter. This approach is most like U-Net and VI, in that it avoids the kernel altogether and provides minimal app-to-app message latencies. In addition, similarly to the mapped buffers and associated queues in VI, an ADC consists of a pair of pages from the transmit dual-port memory of the OSIRIS card, one transmit and one receive page. Upon opening a network connection, an application can map one pair of these pages into its address space. This application is linked to an ADC channel driver and a replicated network protocol stack. The OS assigns a set of VCI, a priority, and a list of physical pages to the ADC. Thus, incoming data is queued on the receive queue of an ADC if its VCI matches that of one in the set of the ADC, and the priority is used to determine the order of transmission from ADCs' transmit queues. The difference between this
implemenation and older mapped device driver implementations is that the OS remains in control of the device and the device can be shared among a number of untrusted domains. The round-trip latencies obtained using a pair of DEC workstations back-to-back, between two programs configured directly on top of the OSIRIS device driver (as is the case with my VI driver) was 417 us for a 1K message. They also run a test with one side receiving and the other side generating PDUs as fast as the host can receive them and they measure a maximum throughput of 379 Mbps with double-cell DMA and 250 Mbps with single-cell DMA (when the data cache is invalidated after each DMA transfer). They found the cache invalidations to have a significant impact on the throughput.

10.1.5 VI Implementations

A number of different organizations have implemented software emulations of the VI Architecture. These implementations and performance analyses of them are presented in this section.

10.1.5.1 VI Proof-of-Concept Tests

Berry et al. [2] developed prototype software implementations of the VI Architecture on 100 Mbit/sec Ethernet and Myrinet. The first project’s intent was just to test the soundness of the VI Architecture concepts and validate that the architecture could in fact be implemented. The Ethernet VIA implementation was compared with that of UDP using an NT NDIS driver using an application to application latency test and a bandwidth test. The results showed that VI did provide a latency reduction of 30% and that it was a feasible architecture.
The purpose of the second project was to emulate a hardware implementation and determine what kind of performance benefits could be gained by moving the VI functionality from software into hardware. The Myrinet implementation involved programming of the LANai network interface controller on the Myrinet NIC to perform the VI emulation, a significantly less costly approach from my driver implementation in that the Myrinet NIC could be programmed with the knowledge of VI in sending and receiving data and did not use an Ethernet-packet based system. Message latencies of approximately 55 microseconds were achieved for 32 byte messages and approximately 240 microseconds for 8 Kbyte messages.

10.1.5.2 BobNet

Csanady and Wyckoff implemented Bobnet [5], a low-latency message-passing protocol designed to operate as efficiently on 10 or 100Mb/s Ethernet as it does on faster, more expensive network interface devices such as Myrinet. This implementation differed from my layered NT driver in that it was implemented using a single driver to support VI for several network cards under the Linux OS. As with my driver, it features zero-copy sends directly from user-space data, and one-copy receives. In addition, Bobnet also serves as the medium of converting the user's VI requests to the Ethernet-wire in order to preserve the simple message passing interface. Csanady and Wyckoff presented overall one-way latencies for 1KB messages for three different network interface cards: the SMC EtherPower 100 Mb/s (30 usec), the EtherExpress 100 Mb/s (39 usec), and a prototype Gigabit Ethernet Packet Engines card (27 usec).
10.1.5.3 Other VI Work

Speight et al. performed an analysis of the potential of the Virtual Interface Architecture in which they examined the low-level performance of two VI implementations, one implemented in hardware on GigaNet cLAN and the other in device driver software on Tandem ServerNet. They observed that both VI implementations offered a significant performance advantage relative to the corresponding UDP implementation on the same hardware. In addition, they investigated the problems associated with delivering performance to distributed applications on clusters of workstations by exploring performance and implementation issues that arise when adapting an MPI library implemented using UDP to VI. The issues they analyze are memory registration costs, polling vs. blocking, reliability of delivery, and memory-to-memory copying. They achieve a 55% reduction in message latency seen by user applications across all message sizes by eliminating explicit ACKs, reducing memory-to-memory copying, and finding the best synchronization primitives.

10.2. Multicast

10.2.1 IP Multicast in ATM Environment

Multicast in the Virtual Interface Architecture has not been previously explored, but work that is very relevant is in the area of implementing IP multicast over ATM. ATM is also a connection-oriented network hardware that provides virtual channels for connections between machines in a network. Similarly to VI, ATM does not provide the capabilities required by IP-multicast for operation as a subnetwork. This is due to the fact that ATM is not a shared medium technology such as Ethernet or FDDI.
Broadcasting and Multicasting in shared medium networks simply involves having multiple hosts monitor the medium for packets with a specific multicast address. Note that this is also the case in my software implementation, since I am actually running on top of Ethernet, but in order to specify multicast over VI hardware this shared medium does not exist. Similarly to the one-to-many connections that I have specified for VI, ATM has a point-to-multipoint connection defined in the UNI 3.0 spec. [53] This is useful only for the sender to transmit packets to multiple known receivers, but there is no way for a receiver to send data over this connection, and the sender has complete control over membership in the connection.

G. Armitage has explored true IP multicast communication (multipoint to multipoint or MTM) [53] in two different ways. The first MTM connection scheme involves setting up N point-to-multipoint connections to connect N members in a complete mesh. This topology provides the required MTM connection, but does not scale well when the number of hosts in the mesh becomes large. With N hosts, N * (N-1) virtual channel pairs have to be set aside for the MTM connection. In addition, each host's interface has to provide reassembly for N-1 virtual circuits. The second MTM connection scheme is to use a server as the root for an MTM tree [53, 54]. In this method, the server sets up a point-to-multipoint connection with all the hosts to be included in the MTM connection, and the leaves then form point-to-point connections back to the server (root). Any member can then send cells to the root and have them sent to the rest of the members through the point-to-multipoint tree. This only requires one host to store information regarding membership in the multipoint-to-multipoint connection and the server could be designed to eliminate the scaling problems by having
extra reassembly engines and/or multiple ATM interfaces. The one problem with this method is the server needs to both receive and retransmit every cell in the MTM connection.

Bagwell and Marlow [48] have examined the properties of IP-multicast and how ATM networks can achieve IP-multicast functionality, and they expect that a hybrid approach of the two previously mentioned techniques will provide the most effective means of mapping IP-multicast subnet functionality to ATM.

10.2.2 Reliable Multicast Protocols

An efficient, reliable Multicast implementation has been designed for Illinois Fast Messages (FM/MC) using a spanning tree forwarding procedure that is completely handled by software running on the LanAI processor on Myrinet. Myrinet only supports point-to-point communication without multicast, and they implement multicast through software to observe the performance. They state that they achieve low latency for multicast by not involving the host processors in the forwarding of multicast fragments. In order to specify group members, their application must tell the FM/MC library about the structure of a multicast group and each node maintains a group membership table on its network adapter. A handle is returned to the user for multicasts to the group. They present that flow control is a major problem with multicast that is difficult to solve because messages that are sent are destined for multiple nodes and receive buffers where overflow can occur. In addition, without flow control, messages may be dropped, resulting in unreliable communication.

In comparison, my implementation is unreliable and thus does not include any flow control. In running with Brazos I avoid the problem of buffer overruns by posting
more descriptors on VI receive queues and if the queue is full any new messages received will be dropped. Brazos includes retries and thus will resend the message. This implementation could be extended to include flow control for reliability by not sending to a VI unless its queue has available descriptors posted for receiving data.

Their normal FM 1.1 code achieves a minimal round-trip latency of 58 us for 16-byte messages and a maximum throughput of 13.3 Mbytes/sec for messages of 128 bytes. After they added multicast related changes to the LCP and FM library software, their roundtrip unicast time for 16 byte messages increased with another 12 us to a total of 82 us. They achieve a sustainable throughput of 10.1 Mbyte/sec for 16Kbyte messages.

Group communication has also been explored in the Amoeba Distributed Operating System [14]. They explore primitives that make multicast communication available to the user that have been integrated within their OS. Their primitives guarantee global ordering of multicast messages and can be performed in slightly more than two messages, comparable to an RPC.

Totem is a Fault-tolerant multicast group communication system that was developed by Moser et al. at the University of California [56]. Totem invokes operations in the same total order throughout the distributed system using a message ordering strategy that employs timestamps.

10.2.3 Vulnerabilities of Reliable Multicast Protocols

Parks et al. performed an examination of the vulnerabilities of several reliable multicast protocols [50]. They noted that various mechanisms employed by many protocols to provide reliability can present vulnerabilities. Sender-initiated reliability protocols place the burden of loss detection on the sender by requiring a positive
acknowledgement from every receiver for every packet sent. Such protocols suffer from ACK implosion: increasing amounts of both network bandwidth and processing time are consumed as the number of receivers in the group increases. Denial-of-service attacks have been reported that produce such implosions. Receiver-initiated reliability protocols place the burden of loss detection on the individual receivers. Receivers generate a NAK when they detect a lost packet, which can cause a NAK implosion if a large number of receivers lose the same packet, and this protocol is susceptible to flooding attacks. Other attacks that are described are the forged data attack, premature ACK, and NAK and retransmission suppression. To defend against such attacks, they present that the first line of defense is to give receivers the capability to filter out packets received from an attacker, which prevents the attacker from interfering with the operation of the reliable multicast protocol. This solution does not reduce the network bandwidth consumed, however, and thus they say it is necessary to block the transmission of packets close to the source. They discuss cryptographic and non-cryptographic defense measures to come up with such solutions to attacks.

10.2.4 Secure Agreement Protocols

SecureRing group communication involves protocols that provide reliable ordered message delivery and group membership services [56]. In these protocols, messages are multicast to groups of processors within an asynchronous distributed system and are delivered in a consistent total order to all members of the group through the use of a Byzantine fault detector.

Pessi [59] provides a presentation of various IP multicast systems and the applicability of the IP Security Architecture to these systems. The IP security
architecture (IPSEC) was standardized in August 1995 and defines two types of security
headers that hold security information for authentication and/or encryption algorithms
and keys, lifetime of keys, lifetime of security association, etc. There are two encryption
key management systems proposed for use with this security architecture, SKIP [57] and
Photuris. Pessi basically states that asymmetric cryptosystems are ideal for multicast, but
the cost is prohibitive because it is not practical to encrypt bulk data. He also states that
some key distribution systems or public key encryption algorithms are ideal for multicast
but do not provide widespread usability.

Caronni et al. [58] propose a series of approaches for achieving scalable security
in IP multicast that provide privacy and authentication on a group-wide basis. They
support dynamic groups which implies that newly joining members must not be able to
understand past group communications, and that leaving members should not understand
future communications. As a result, this requires a key change for all group members
when a leave or a join occurs, which poses a problem if the groups are large.
11. Conclusions and Future Work

I have described the design and implementation of a tightly-coupled intermediate device driver that emulates the VI Architecture. Layered drivers typically use I/O request packets to pass requests to lower drivers on the stack, which can incur unnecessary overhead if significant inter-driver communication exists. Alternative methods such as the inter-driver communication facility presented in my implementation can be a much more effective means of servicing requests. In addition, multiple packet copying that occurs in protocol stacks as well as user-kernel context switching incurs additional latency in message transfer that limits realization of the actual network interface hardware performance. I also have successfully implemented a VI multicast connection scheme over Ethernet hardware that demonstrates the usefulness of multicast communication in the VI Architecture and provides a significant reduction in the time to service multiple clients in the low-level test while not incurring any significant additional overhead.

I have used my VI implementation in a distributed shared memory system and measured the observed performance benefits in real scientific applications. The performance benefits in terms of reduction in execution time in these applications were approximately 10-14% for applications that had a high rate of communication and took advantage of multicast such as 3D-FFT and Barnes Hut, but were less significant or did not provide any improvement at all for applications that had a low rate of communication and did not use multicast. The observed performance benefits of the VI Architecture with multicast were 5-10% higher than those observed without multicast, and in one case the VI architecture actually performed worse than Ethernet because of its lack of
multicast capability. This definitely suggests that multicast capability would be useful in
the VI architecture for use in distributed computing applications. In addition, the
protocol that I have designed is a simple solution to providing multicast that does not
introduce any additional software overhead and can be easily added to the VI
Specification. Other DSM systems such as Orca [1] and Amoeba [14] use group
communication to implement sequentially consistent software and could benefit from the
use of VI with Multicast.

Scalability of the VI multicast protocol that I have designed is an issue that has to
be dealt with if IP-multicast functionality using VI is desired in a network such as in the
Brazos system. The number of VIs required in order to provide a full mapping of the
nodes in a system grows exponentially as the number of nodes increases using many one-
to-many VI connections, and more than 1K VIs can be required in a moderate to large
system. A hybrid approach such as that suggested for ATM by Bagwell and Marlow [48]
is needed in order to provide better scalability in large systems and to provide IP-
Multicast like functionality (many-to-many) with VI.

Flow control is a difficult problem that plagues multicast because there are
multiple destination nodes involved and buffer overrun on each of these remote nodes is a
concern. In order to implement a reliable version of VI with multicast, flow control
mechanisms such as those designed by Verstoep et al [49] should be designed in order to
include mechanisms to handle situations where VI queues are empty or all of the
descriptors are completed. In Brazos, we are implementing a throttling mechanism that
ensures that data is never sent to a VI whose queue does not have available descriptors,
but this functionality can be specified in the architecture as a property of reliable VIs.
If VI is to be used in a network area larger than a locally maintained network, security is also an issue that is a concern for multicast. Currently VI specifies attributes that must match in order to connect two VIs, but there are no real mechanisms specified that could stop a user with harmful intent from blasting into a VI and attacking a system in manners that we know to be problems on the internet (denial-of-service attacks, flooding). Cryptographic and non-cryptographic mechanisms can be devised for implementing security [50, 56, 57, 58, 59] but many of these mechanisms cannot protect against all attacks and introduce other problems such as NAK implosion as well as high overhead. A standard for security is needed if VI with multicast is to be implemented in large networks in order to protect against malicious attacks. Since VI is intended for System Area Networks, however, we suspect that the protocol will not be used in large-scale networks and security is not a real issue.
References


d110.501/1995/multicast.html](http://www.tcm.hut.fi/Opinnot/Tik
110.501/1995/multicast.html)