To the Graduate Council:

I am submitting herewith a thesis written by Kara Ann Hall entitled "The Implementation and Evaluation of Reliable IP Multicast." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. Tom Dunigan, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Kara A. Hall

Date 11/30/94

The Implementation and Evaluation of Reliable IP Multicast

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Kara Ann Hall

December 1994

Acknowledgments

The author would like to express her appreciation to Dr. Tom Dunigan for serving as director of this work. The author is indebted to him for his help, patience, and kindness. She also wishes to thank Dr. David Straight and Dr. Jeffrey D. Case for their time and participation on the thesis committee. In addition, the author wishes to thank G. A. Geist for his contribution in making this work possible.

The author would like to thank Ms. Wallace Mayo, fellow TA's, labstaff and professors for their help and kindness over the past two and one-half years. She wishes to express special thanks to her parents Mr. Glenn L. Cantrell and Mrs. Hazel V. Cantrell. For their love, training, *never-give-up* philosophy and prayers, the author is ever indebted. Finally, the author wishes to especially thank her husband Paul Douglas Hall for his infinite love, patience, encouragement and prayers. Above all, she thanks God for guiding her in life and for answering prayers.

ABSTRACT

This research investigates multicast communications on both local area networks (LANs) and wide area networks (WANs). Developing and evaluating reliable protocols for multicast communication is the focus of the study. Today, most multicast applications rely only on *best effort* delivery. This research measures the performance of reliable Internet Protocol (IP) multicast protocols and develops analytical models.

This research further evaluates reliable IP multicast by implementing the protocol in the Parallel Virtual Machine (PVM). Currently, PVM's **pvm_mcast()** uses looping and unicast messages to send one message to many hosts. On a LAN, broadcast has been examined as perhaps a more efficient method in sending a multi-destination PVM message. IP multicast permits a single message to reach multiple hosts on both the LAN or the WAN. The IP multicast protocol which was developed for this study is implemented reliably to satisfy PVM's design criteria. The the design and performance of the IP multicast version is compared with PVM's present multiple unicast implementation.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Parallel computing is a way in which the natural limitations of a single processor can be overcome. In parallel computing, a problem is divided into many smaller independent tasks. Similarly, multicasting (or broadcasting) data is a way in which some of the limitations of the transport media are overcome. In multicasting as the number of copies of a message being transmitted from a single source are decreased, usually speed-up increases. According to Deering[1], multicasting in a LAN establishes more robust and efficient communication for distributed applications.

Early forms of network communication were circuit-based like most telephone calls. In circuit switching, an end-to-end path (or dedicated line) is set up before any data is transferred. Protocols for CSMA/CD[1], token-bus, and token-ring networks were developed in which dedicated lines were not necessary. All hosts in these networks have a single communication channel that is shared. Source and destination addresses are used to determine which host receives the message. In these networks, special addressing makes group communication possible. Instead of sending N copies of a message to N hosts, these networks provide multicasting (and broadcasting) where one copy of a message is sent to N hosts.

[1]Carrier Sense Multiple Access, with Collision Detection

Early forms of group communication used ground and satellite microwave systems[2]. This type of communication has *driven* the television and radio industries and even telephone where cables do not exist. To prevent microwave information from being pirated, data can be encrypted (or scrambled). To increase the number of channels and to reduce communication piracy, coaxial cable is now being used in cable television (CATV) networks.

Group communication is also used in many multiprocessor systems. Multiprocessors contain numerous identical processing units which are closely arranged and connected by minimal length high-speed transmission links. In order to decrease run-time in these machines, multicasting or parallel unicasting of identical messages is often used. Developments such as inexpensive, fast and readily available transmission media (e.g., twisted-pair, coaxial cable) and CPU's (e.g., personal computers, workstations) in LANs add a new dimension to parallel computing.

As computer networks such as local area networks (LANs) have become more economical, faster and more convenient to use, distributed systems have developed. *Distributed operating systems* can provide fault tolerance and reliable communication in networks. Such systems can provide a desk-top interface for architecturally different hosts and transport devices. Although distributed systems may tie together mainframes and/or multiprocessors, many distributed systems connect modern desk-top computers to form an inexpensive parallel machine.

Since the communication links in a distributed system are much longer than in a multiprocessor, saving bandwidth and time are critical. Multicasting and broadcasting algorithms have been used within LANs as well as multiprocessors for this purpose. Since the early 1980's, multicasting and broadcasting have existed in the CSMA/CD, token-bus, and token-ring networks. Broadcasting is the process of sending one copy of a message to all hosts in a network, and multicasting is the process of sending one copy

of a message to a group of hosts in a network. Today, multicast capabilities across the Internet[2] exist. Steve Deering[3], who developed the recommended Internet standard for Internet Protocol (IP) multicast, states,"IP multicasting is the transmission of an IP datagram to a *host group*, a set of zero or more hosts identified by a single IP destination address."

IP multicast is attractive for several reasons. It can provide group communication for both a LAN and a WAN. Group communication over the Internet is possible due to the development of the Multicast BackbONE (MBONE)[4], which is a virtual network that uses IP multicast addressing at the network level. Another benefit is the savings in bandwidth since theoretically only one copy of a message needs to be sent in order for hosts in the group to receive it. Both a good routing algorithm and a time-to-live (*ttl*) set just high enough to reach the most distant machine in the IP multicast group add to the bandwidth savings. Within most LANs, the bandwidth remains the same whether there is one receiver or hundreds. Also, to save bandwidth, IP multicast routers (*mrouters*) only multicast a session if there is at least one active listener. In addition, IP multicast can be ported to numerous types of workstations by making a few changes to the kernel.

One application which could possibly take advantage of IP multicast is the Parallel Virtual Machine[3] (PVM). PVM is a message passing library which works like a distributed operating system except that unlike most distributed operating systems, PVM can run on top of the existing operating system; the kernel does not have to be reconfigured. PVM is very versatile and can be used in a WAN as well as a LAN. PVM hides from the user the differences in the architecture of the hosts and of the transport media contained within the virtual machine. Since PVM is used as a parallel machine, speed is critical. Provided that all PVM hosts are IP multicast-capable, the speed of PVM multicast should increase if IP multicast is used.

[2]Internet - a loose confederation of networks and of internets (networks of LANs and WANs)
[3]Parallel Virtual Machine - developed at ORNL[5,6,7,8]

1.1 Objectives and Significance

This research investigated multicast communications on LANs and WANs. Today, most IP multicast applications rely on *best effort* delivery. This study focused on developing and evaluating a reliable protocol for IP multicast communication. Also, the performance of the reliable multicast protocol (RMP) for IP developed as part of this research is measured and compared to the performance of *best effort* and unicast protocols.

This research addressed several questions. First, what is required in order to use IP multicast? Second, what are the performance limitations of IP multicast? Third, how is reliable IP multicast achieved? Finally, what is the performance of reliable IP multicast? In this study, a reliable IP multicast protocol is used in order to answer these questions. In addition, an analytical model of multicast reliability was developed. Finally, test suites for measuring performance were developed.

Further evaluation of reliable IP multicast was accomplished by implementing the RMP in PVM. Currently, PVM's multicast function (**pvm_mcast()**) uses *one-to-all* fanout to send one message to many hosts[7]. This results in a duplication of the message and contention as the packets are sent onto the network.

Kaashoek and Tannenbaum[9] state that the major design issues for group communication are: addressing, reliability, ordering, delivery semantics, response semantics, and group structure. Several forms of group *addressing* exist. Two of the most common methods of addressing are to allow the sender to specify all of the destination addresses and to use a single address for each group. In the performance tests described in Chapter 3, a single IP multicast address was used to send a message to a group. The reliable IP multicast implementation in PVM used a combination of these strategies. In the implementation, the IP multicast address was used to transfer the message from the sender's

pvmd to the receiver's *pvmd*. Task identifiers are then used to transfer the message to the appropriate task.

Reliability deals with recovering from communication errors. For reliable protocols, most often positive acknowledgments (ACKs), to inform that the message was successful, or negative acknowledgments (NAKs), to inform that the message was unsuccessful, are used. In the protocol developed as part of this research, ACKs were used. Since PVM uses ACKs, the RMP will fit into PVM with out major changes to PVM. The sender also uses *timeouts* and retransmission to improve reliability.

Ordering pertains to the ordering of messages sent to a group. The protocol fits the FIFO ordering style. PVM also uses FIFO ordering for packets. In PVM, the sequence numbers are used to insure that remote *pvmds* receive packets in order.

Delivery semantics determines how many processes must receive the message successfully. In the RMP, a simple percent of the number of receivers was used to determine whether a multicast message was successful or not. This strategy was also used in the PVM implementation. This is similar to the strategy used in the Ameoba system[9] (for more detail see section 3.1.1).

Response semantics defines how to respond to a multicast (or broadcast) message. In the RMP and in PVM, no response was expected. However, VMTP[10] which is discussed in sections 2.3 and 3.1.1 uses the *client-request–server-response* style with acknowledgments expected for both the request and the response.

Group structure pertains to whether a group is *closed*, *open*, *static* or *dynamic*. In a *closed group*, only group members can communicate with the group. In an *open group* non-members may send messages to the group. In *static groups*, members join the group at the start and remain in the group until the process is finished. In *dynamic groups*, membership may change during the life of a group process. In the performance tests

described in Chapter 3, *closed, static groups* were used. PVM groups are *closed.* Also, PVM forms new task groups for each multicast message.

1.2 Methods and Assumptions

Many issues were involved in developing, testing and implementing the reliable IP multicast protocol. One of the most important issues was deciding whether to use positive acknowledgments or negative acknowledgments to insure reliability. In this study, the protocol uses ACKs (positive acknowledgments returned from the receivers of the multicast to the sender). Since PVM uses ACKs, the protocol is readily applicable to the study of IP multicast in PVM. Using ACKs allows us to handle losses immediately and to prevent the message from being removed from the sender's *history buffer* before all receivers have a correct copy.

In addition, several other decisions on how to implement the protocol had to be made. For instance in determining the number of multicast retries, a simple approach was used. A pre-set percentage of the number of expected ACKs was chosen. Also, the time at which a receive was *timedout* was pre-set at just a bit more than the value of the maximum time for successful ACKs.

Another issue involves the IP group members. Only one group is used for all multicasting in the performance tests. In implementing IP multicast in PVM, all hosts in the virtual machine receive the IP multicast packet. All hosts join the multicast group when their local PVM daemon (*pvmd*) is started and remain part of the group until the daemon is halted.

Another issue involves varying several parameters for testing purposes. In this study, the test configuration consists of a LAN of ORNL[4] or UTK/CS[5] Unix[6] work-

[4]Oak Ridge National Laboratories
[5]The University of Tennessee, Knoxville's Computer Science Department
[6]Unix is a trademark of Unix Systems Laboratories, Inc.

stations and a WAN of ORNL and UTK/CS Unix workstations. The pool of IP multicast-capable workstations includes seventeen machines at ORNL and fourteen machines at UTK/CS. The number of machines, the load of the machines (i.e., the time of day), and the message size were all varied during testing.

Finally, the *loopback* option for IP multicast sockets was set to zero. In this case, the sender's host never looks for packets coming from itself. Although PVM version 3.2.6 used in this research allows a multicast packet to be sent to tasks on the sender's machine, later versions do not (PVM version 3.3 FAQ[7][11]). This is because few PVM parallel programs have slave tasks running on the same machine as the master task.

1.3 Thesis Overview

The next chapter surveys related literature in the areas of multicast and broadcast on LANs, IP multicast, and reliable multicast. In Chapter 3, several reliable multicast protocols are discussed in addition to this study's reliable multicast protocol (RMP) developed for IP multicast and its performance. Also, the development and analysis of the analytical model of reliable IP multicast are presented. In Chapter 4, the implementation and performance of reliable IP multicast in PVM is described. Finally, the concluding chapter summarizes this research as well as some of the outstanding issues related to reliable IP multicast.

[7]Frequently Asked Questions

Chapter 2

Survey of Related Work

The first of the following three sections surveys multicasting and broadcasting in LANs. The second section looks at IP multicast (IP multicast) literature. The last section reviews work involving reliable multicast.

2.1 Multicast and Broadcast on LANs and Multiprocessors

2.1.1 Multicast and Broadcast schemes.

Many LANs support some form of multicasting. The CSMA/CD, token-bus, token-ring, FDDI and ATM LANs, and even ground and satellite radio systems have multicast capabilities[2].

The IEEE 802.2 LAN addressing standards provide multicast protocol as well as broadcast protocol for CSMA/CD, token-bus and token-ring LANs[2]. On a CSMA/CD LAN (e.g., an Ethernet LAN), the sender of a multicast packet uses the group address for the destination address. This address is determined by the high-order bit of the destination address. If the high-order bit is a one, then the address is a multicast address.

The rest of the packet is filled with the multicast address for the group. The broadcast address consists of all one bits. If the high-order bit is a zero, the address is unicast and only the machine with that address will accept the message.

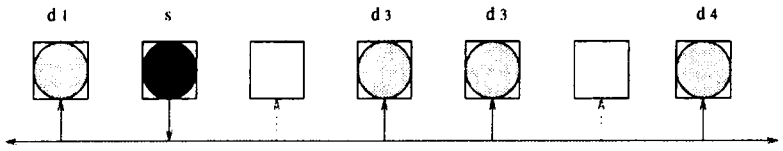

Figure 2.1: Multicasting in a CSMA/CD LAN.
All nodes will see the packet and accept or reject the data based on the destination address. The source node is labeled S and destination nodes labeled $d_{\#}$.

Once the sender's packet is placed on the network, all hosts in the same LAN can *see* the packet (see Figure 2.1). Each station will check the packet's address against their own and then against that of each multicast group which it belongs. If there is a match, the packet is accepted by the receiver and processed similarly to that of a unicast message. The matching is usually all done in hardware.

In a token-ring LAN, the process of addressing and accepting a multicast packet is similar to that of a CSMA/CD LAN. However in the token-ring, the sender must possess a token to transmit data (see Figure 2.2). Packets are passed in order from one machine to the next until the packet is returned to the sender. When the data is copied from the packet to the node, each receiver marks the packet as received. When the sender receives the packet, it checks to see if all members of the group have received the data.

The token-bus is physically similar to the CSMA/CS LAN in that all hosts can *see* the packet. However, it is logically similar to the token-ring in that a sender on a token-bus must possess the token in order to transmit a message. Addressing and acceptance of a multicast packet remains the same as that of the other two LANs.

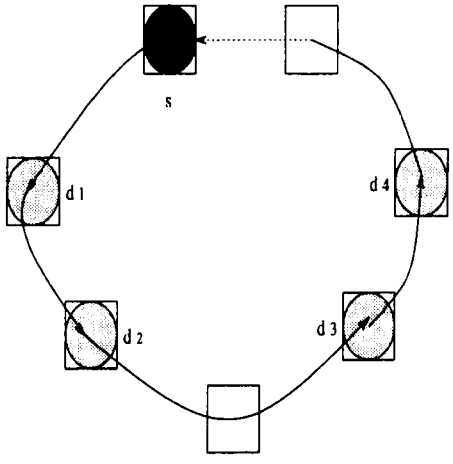

Figure 2.2: Multicasting in a token-ring.
All nodes will eventually see the packet and accept or reject the data based on the destination address. The source node is labeled S and destination nodes labeled $d_{\#}$.

By the mid 1980's, the FDDI (Fiber Distributed Data Interface) LAN was developed[2]. This LAN is a very high performance token-ring LAN which uses the same addressing and packet acceptance method as those above. Due to the large number of hosts and the large amount of cable which may be involved, the amount of time needed for a packet to complete one cycle through the FDDI LAN may take quite a bit of time which would add latency. However, unlike a token-ring LAN, a new token may be placed on the FDDI LAN once the frames from a previous message are all transmitted instead of waiting for the frames to return.

Asynchronous Transfer Mode (ATM) LANs provide fast switching of small packets containing a 5-byte header and 48 bytes of information[12]. The ATM protocol was developed for transporting multi-media information in telecommunications networks, but is also being used in LANs connecting workstations. ATM switches can forward packets simultaneously from one switch to all hosts and switches to which it is directly connected. This parallelism of an ATM switch is used to multicast packets in the ATM network.

2.1.2 Uses of Multicast and Broadcast in LANs

The following works describe multicast and broadcast applications for LANs. Satyanarayanan, et al.[13] have developed a file system, Coda, for large-scale distributed computing systems using workstations. Their test environment consisted of three servers with Ethernet multicast enabled and not based on IP. When measuring bytes transferred, they found that the use of multicast reduced the network load. Also, they found a small improvement in longest packet transmission time due to multicast, and they indicate that it would be even better if even larger files were used. When measuring transmitted packets, the improvement does not appear to be so dramatic. However, this is due to the necessary use of non-multicast control packets.

Ahamad and Belkeir[14] present a multicast-based process for load balancing in an Ethernet type LAN using the multicast capabilities of the local network. In their LAN, a host can join or leave a multicast group at anytime with virtually no message overhead in maintaining the groups. This is done by telling the software to receive or not receive messages from a particular multicast group. Hosts of similar work loads are placed in the same group. This allows a host to easily change membership based on workload and makes it possible to send a request via multicast for a *work transfer* to only those hosts which have a lower workload than the host that wants to transfer the task. If a host is in need of transferring an incoming task, it sends a request for transfer only to groups of lower workloads via a multicast message. This process is said to reduce the overhead of load balancing because a host does not waste computational resources when it cannot accept a task. Ahamad and Belkeir's multicast algorithm studies show that the average response time for tasks is quite good.

Satyanarayanan and Siegel[15] describe a parallel remote procedure call *Multi-RPC*. This application was implemented on Unix workstations and uses multicasting for parallelism. It was developed as a multicast version of the RPC used in the distributed

computer environment, Andrew, at Carnegie Mellon University. The researchers derive an analytic model of the system and validate it. Trial runs show that MultiRPC can contact up to 100 servers in parallel. They note that MultiRPC does not use true multicast. Since most of the workstations used in this study were not multicast capable, software filtering of broadcast packets simulated the multicast.

2.2 Internet Protocol (IP) Multicast

Numerous works contributed to the development of IP multicast and the Internet MBONE. In the 1984 RFC[1]919[16], Mogul proposed broadcast protocols for *best-effort* IP datagram broadcasting. Internet gateways were used to protect against any unwanted broadcasts. His work includes information on how to broadcast IP datagrams on local networks that support broadcast, how to address broadcasts, and how gateways manage them.

In the 1985 RFC947[17], Lebowitz and Mankins presented a problem of multi-network broadcasting and the motivation for solving the problem. They stated that a shortcoming of broadcasting was that broadcast packets are only received by hosts on the physical network on which the packet was broadcast. Their solution was to use a *broadcast repeater* transparently relaying broadcast packets from one LAN to another and to forward broadcast packets to hosts on networks which do not support broadcasting at the link-level. The forwarder and repeater were implemented separately. Lebowitz and Mankins stated that a large amount of overhead was involved.

Other works more recent and closely related to IP multicast are the 1988 RFC1075[18] *Distance Vector Routing Protocol* which provides a routing protocol for IP multicast that most MBONE routers use and the 1989 RFC1112[3] *Host Extensions for*

[1]Request for Comments

IP Multicasting which is the recommended standard for IP multicasting in the Internet. RFC 1112 provides protocols for addressing, sending and receiving IP multicast packets. These and other works will be described in more detail in the following sections.

IP multicasting often involves audio and video data which is bandwidth intense. The MBONE tools offer a non-traditional interactive teleconferencing media such as Network Video (*nv*), Visual Audio Tool (*vat*) and a shared drawing *whiteboard* (*wb*). As the number of IP multicast-capable machines grows, the need for applications which take advantage of IP multicast will also grow, and this can be a problem. According to Jacobson[19], "The MBONE has a total of 500kb/s of bandwidth that has to be shared between 1200 networks and 10,000 hosts. To put that in perspective, 500kb/s is a *total* of 6 *vat* conversations or 4 *nv* video sessions."

IP multicast is still in its infancy. Currently, it works quite well without a great deal of support, privacy, control and reliability measures (see Chapter 5 for further details). As the number of users grows, these problems will need to be solved.

2.2.1 Multicast Addressing

IP multicast addressing is a key to multicasting in the Internet. IP multicast supports group communication in the Internet by using class D IP addresses. To travel within a LAN, the multicast address must be mapped to the local multicast address. Multicast routers control the forwarding of IP multicast datagrams between any two subnets supporting IP multicast.

As mentioned in section 2.1.1, IEEE 802.2 LANs support multicast. Deering[3] details the procedure by which an IP host group address is mapped to LAN multicast addresses in order to send IP multicast datagrams into the LAN. In order to map an IP group address to an Ethernet multicast address, the low-order 23-bits of the IP address are placed into the low-order 23 bits of the Ethernet multicast address. Since an IP group

address contains 28 significant bits, it is possible that more than one IP group could map to the same Ethernet multicast address. In store-and-forward networks, the IP multicast address may map to the local address of an IP-multicast router. The router would then handle the delivery within and among such networks. In addition, Pusateri[20] specifies another method for the transmission of IP multicast datagrams over token-ring LANs. Pusateri tries to provide a more efficient means of mapping an IP multicast address to an assigned token-ring functional address.

2.2.2 Routing Algorithms

Many IP multicast routing algorithms exist. Currently DVMRP is used to route multicast datagrams through the MBONE. IP multicast routers (*mrouters*) are necessary for routing packets through non-IP multicast gateways; this is termed tunneling. Unlike unicast datagrams which could pass through any gateway in its path, an IP multicast packet must be encapsulated using the address of another mrouting host as the destination. The IP multicast packet can only travel from one subnet to another subnet if both subnets contain a working *mrouter* where both are directly connected or are connected via the MBONE. Being directly connected means that the packet is forwarded from the *mrouter* of one subnet to the *mrouter* of an other subnet. Being connected through the MBONE means that there are one or more *mrouters* through which the packet passes other than the *mrouters* in the sender's and the receiver's subnet. All IP multicast routes are manually built. The *mrouter*'s host uses information contained in the */etc/mrouted.conf* file to forward packets outside its subnet. This file contains information such as the local host's address and the address of one or more *mrouting* capable hosts.

An *mrouter* can send (or receive) a message to (from) one or more *mrouters* outside its subnet. To decrease the traffic, the *mrouter* only forwards a message if there

is an active listener. The Internet Group Management Protocol (IGMP) takes care of reporting active listeners. Pruning reduces the fanout for events which there are no listeners since the message is not forwarded. According to Mark Handley[2], the fanout problem will not be completely solved until all routers have multicast capabilities.

Waitzman, et al.[18] describes the Distance Vector Multicast Routing Protocol (DVMRP) which is based on the Routing Information Protocol (RIP) and the Truncated Reverse Path Broadcasting (TRPB) algorithm. DVMRP is used by IP multicast routers for routing datagrams through the Internet. As part of this, tunneling was developed to route IP multicast datagrams between two IP multicast routers separated by a non-multicast gateway. This is accomplished by encapsulating a normal multicast datagram with the IP information necessary for passing through the gateway.

Deering[1] proposes extensions to common internetwork routing algorithms to support low-delay datagram multicasting. He provides information for modifying the single-spanning-tree routing algorithm in order to reduce costs of multicasting in extended LANs. Also, he combines link-layer and network-layer multicast routing algorithms hierarchically to support multicasting across large, heterogeneous internetworks. Some of the algorithms Deering reviews are:

- Single-spanning Tree Multicast Routing
- DVMRP
- Reverse Path Forwarding
- Reverse Path Broadcast
- Truncated Reverse Path Broadcast
- Reverse Path Multicasting
- Link-state Multicast Routing
- Hierarchical Multicast Routing

His work examines the functionality, reliability, and performance of these algorithms.

[2]M.Handley@cs.ucl.ac.uk via mbone@ISIS.EDU mailing list Sat Aug. 6, 1994.

Ballardie, et al.[21] presents an IP multicast protocol, Core Based Tree (CBT), which is said to be low-cost, relatively simple, efficient and scalable due to the small amount of information needed with respect to a multicast group. The protocol is separate from unicast routing as opposed to being closely bound to a unicast algorithm and is easy to install in an internet that contains many different unicast routing algorithms. This multicast routing protocol for IP builds core-based multicast trees, one per group. Routers sending to groups of which they are not members need to know nothing about the groups. These routers route packets from non-member senders to the tree as they would a unicast. Also, routers on a multicast tree need only know their nearest up and down tree neighbor routers. Since a minimal amount of information about a group is needed, the architecture is low-cost, simple, efficient and scalable. Ballardie[3] lists the following features of CBT,

" * CBT routers use their own IGMP[4] message type to exchange routing messages, therefore the CBT-BONE is a separate virtual network from the MBONE used for CBT style multicast.

* Multicast packets are sent **only** over CBT tree links (tunnels) which are set up as part of an explicit join process (hence no need for pruning).

* Multicasts are tunneled over CBT tree links for the corresponding group and contain a CBT header which allow for enriched semantics. Fields in this header include an additional TTL field to allow finer grained scope control. Depending on the value in this field, CBT border routers would know what to allow in and out. Also, in the CBT header is a flow-id field which should allow for a lightweight RSVP mechanism to operate (optionally) at join time. Security fields are in there too."

[3] A.Ballardie@cs.ucl.ac.uk via mbone@ISI.EDU, Thursday., June 16, 1994
[4] Internet Group Management Protocol

2.2.3 MBONE tools

Twenty-one years ago, trials with sending packets containing video data were conducted, but this was limited to the ARPANET[5][22]. Today, teleconferencing (many-to-many communication) across the MBONE has tools based on IP multicast available for the use of video, audio and shared drawing boards. As the number of IP multicast-capable machines grows, the need for applications which take advantage of IP multicast also grows.

Some of the better known MBONE tools are *vat* (visual audio tool), *wb* (whiteboard), *nv* (network video), and *sd* (session directory). Designed by Steve McCanne and Van Jacobson of LBL[6], *vat* is used for speaker and microphone control. It provides telephone quality audio of 64kbps. The second tool, *wb* also designed by McCanne and Jacobson, is a computer version of a whiteboard. As a user draws or writes on the *whiteboard*, the data is multicast and the boards are updated for all users in the group. Also, text and postscript can be imported to *wb*. The third tool, *nv* was designed by Ron Frederick at Xerox PARC. It displays and transmits video. No special hardware is needed to receive video. The video display size, contrast and brightness can be altered. *Nv* supports some video capture boards and has a transmission default rate of 128kbps. As mentioned earlier, *nv* uses a large amount of bandwidth; it tries to decrease the amount of bandwidth used by sending only the parts of an image which change. Finally, *sd* was also designed by McCanne and Jacobson. From *sd* one can start *vat, nv,* or *wb* sessions. Via *sd* session listings, a user may join other sessions. To decrease bandwidth usage, *sd* limits the scope of session advertisements. *Sd* allows a user to start a new session, and provides a randomly selected, unique, non-conflicting IP multicast address, a port number, and application discriminator[23]. The *ttl* and viewing times for new sessions have defaults

[5]ARPANET - an early network created by ARPA (now DARPA), the (Defense) Advanced Research Projects Agency of the U.S. Dept. of Defense[2].

[6]Lawrence Berkeley Laboratory, University of California, Berkeley

which can be over-ridden. *Nv*, *vat*, and *sd* use unreliable IP multicast; *wb* provides its own reliable IP multicast protocol.

2.3 Reliable Multicast

Deering[3] states, "A multicast datagram is delivered to all members of its destination host group with the same *best-efforts* reliability as regular unicast IP datagrams." In building fault-tolerant software such as PVM, *best-effort* reliability will not suffice especially if one multicasts beyond the subnet where the frequency of packet loss increases. In this section, literature about reliable multicast is reviewed.

Pingali, et al.[24] compare three reliable multicast protocols. The first protocol is a non-optimized positive acknowledgment-based protocol. The second protocol is a non-optimized negative acknowledgment-based protocol which uses unicast NAKs. The third protocol is a NAK-style protocol which Ramakrishnan and Jain developed for LANs and which Jacobson developed for WANs. This third protocol uses a multicast (or broadcast) NAK to reduce the amount of reverse-traffic.

In the ACK algorithm, the sender initiates the retransmission by *timing-out* while waiting for acknowledgments. As the number of receivers increases, the number of ACKs increases. Not only does this cause contention in the network, but the sender must take time to process each ACK. This protocol was found to have a lower throughput than the two negative acknowledgment protocols. However, it is much less complicated to implement.

In the NAK versions, the receiver initiates error control. As mentioned, the NAK protocols differ in the manner in which a NAK is returned. In the non-optimal unicast NAK protocol when a host receiving multicast packets detects an error (e.g., a lost packet), a NAK is immediately sent to the transmitting host for which the error was

detected. Using a NAK decreases the reverse traffic since a NAK is only sent when an error is detected.

In the multicast NAK protocol, a random time is set between discovering an error and sending a NAK. Before sending a NAK, the receiver checks whether or not the NAK has been multicast by another receiver. If so, it sets a timer and waits for the retransmission, else it sends the NAK. The NAK is sent via multicast to all members of the group. This decreases the number of NAKs being sent by receivers from as many as N down to at most one per multicast. According to Pingali et al., Jacobson has tried using the multicast NAK-style algorithm in a shared *whiteboard.* Of these three protocols, the multicast NAK protocol was found to have the highest throughput; this protocol will be discussed in more detail in Chapter 3.

The Versatile Message Transaction Protocol (VMTP) by Cheriton[10] can provide reliable multicast communication. VMTP has been implemented in Unix systems[25]; the use of the protocol involves reconfiguring the kernel. VMTP is a transaction protocol which supports RPC. *Transaction-oriented communication* is described as *client-request-server-response* communication where transactions are initiated by the client's request and terminated by the server's response. The protocol can handle error detection, retransmission, duplicate suppression, and security required for transport-level end-to-end reliability. It operates on top of a simple unreliable datagram service. Checksums are placed in packet trailers where they can be calculated quickly. Although it does not directly provide for reliable multicast transport, VMTP enables the user to provide reliable multicast communication by using the acknowledgment protocol it establishes. The VMTP protocol will be discussed further in Chapter 3.

Xpress Transfer Protocol (*XTP*) provides reliable multicast services and is based on the *XTP* unicast transmission protocols[26]. No current implementation of *XTP* exists. Since it does not have to build the multicast service from *scratch*, XTP

multicast benefits from the extension and modification of unicast *association properties.* XTP multicast and XTP unicast are assumed to operate identically except for the multicast differences which Strayer, et al. term the XTP *Definition.* One major difference between an XTP multicast and unicast is that the multicast communication is not duplex. XTP unicast communication has duplex endpoints containing a sending side and receiving side. XTP multicast endpoints are one-sided meaning multicast endpoints are either for sending or receiving not both. The XTP *Definition* suggests using *cloning* for constructing a reverse path for a XTP multicast reply. In cloning, the multicast transmitter spawns a slave to receive responses from the multicast receivers. The slaves in return send the responses to the transmitter establishing full-duplex communication.

As part of the ISIS[7] project, Birman and Joseph[27] developed prototype protocols for establishing reliable multicasting in a distributed computer system. IP multicast was not used. The testing environment was viewed as a large area network with clusters of sites interconnected by high speed devices. Since *best-effort* reliability does not address problems such as site failure, recoveries, or process migration from site to site, Birman and Joseph try to provide a system for handling these problems. Also, they address delivery ordering issues that arise when multiple clients communicate with a process group concurrently or when a single client transmits many multicast messages to a group without pausing to wait for acknowledgments.

The first of the three protocols, the GBCAST protocol, is used for group multicast. It has the most constraints and is the most costly. It is used to transmit information about failures and recoveries to group members. The ABCAST protocol is less constrained and may be used where total order on multicast messages is not necessary. The CBCAST protocol is the weakest protocol of the three in that it uses less distributed synchronization than GBCAST or ABCAST. CBCAST was considered the

[7]ISIS is a distributed operating system developed at Cornell University.

most likely candidate for use in the ISIS system. Birman and Joseph's research found implementation costs to be low.

Armstrong, et al.[28] describes the multicast transport protocol (MTP) for reliable transport. Bormann, et al.[29] have implemented this protocol to be used with the *X window sharing* tool *Xy*. The protocol utilizes the multicast capability of certain lower layer networking architectures. This transport protocol allows any number of transport providers to work together in realtime without networking applications needing to possess detailed knowledge of the number or location of members.

MTP2 by Bormann, et al.[29] is an optimized form of MTP; it avoids some of the problems of MTP. In MTP, no recovery from master failure exists. In MTP2 if the master goes down, the group members check to ensure it is no longer available, elect a new master, and form a new group identification. The new MTP2 master accumulates the status of active messages from all members. Also, unlike a MTP master, the MTP2 master may migrate from host to host depending on the load of the hosts in the group. In addition, MTP2 allows time parameters to be adjusted dynamically. MTP does not provide for priority messages, but the MTP2 master processes incoming packets in order of their priority. The priority controls the sequence in which the tokens are granted. Another difference is that MTP receivers must receive at least one packet per message to know the senders address. Otherwise, it can not determine to whom to send the NAK. On the other hand, MTP2 receivers may send a NAK to the master or any sender (which may forward it to the master or the correct sender) if the sender is unknown. MTP2 supports unicast data, whereas MTP provides only multicast support. MTP will be discussed further in Chapter 3.

Braudes and Zabele[30] present the reliable, adaptive multicast protocol (RAMP). This protocol provides group address and membership authority, modifications to existing routing protocols, and can provide reliable multicast on top of IP. The quality of service

can be modified based on the number of NAKs received. RAMP uses a NAK-style protocol with multicasts or unicasts retransmissions of the multicast packet based on the number of NAKs. This protocol will also be discussed further in Chapter 3.

Huang, et al.[12] developed a reliable multicast transport service for an Asynchronous Transfer Mode (ATM) network. This service sits atop the unreliable ATM Adaption Layer 5 (AAL5) and uses the vendor supported Application Programming Interface (API)'s socket-like services, not IP. ATM relies on fast packet switching for speed. ATM multicasting is based on parallel transfers of N copies of the packet to be multicast. Since this method is somewhat unreliable, Chang, et al.[31] investigate reliable message-passing protocols; PVM was chosen for their tests. An ATM LAN with ATM switches connecting workstations was used in their tests. One performance test involved a normal implementation of PVM over an Ethernet network using BSD sockets. A second test involved implementing PVM in an ATM network using BSD sockets. The third and fourth tests involved implementing PVM in an ATM network using different versions of the Fore Systems' ATM API library (FSAA) with the necessary changes to PVM. FSAA which provides the socket-like interface for ATM messages is a lower-layer protocol than BSD sockets. FSAA is a connection-oriented service in which the connection must be made before data is exchanged.

Although the multicast process was altered for the tests, PVM's acknowledgment protocol remained the same. In phase I of each test, a task ID list is sent to each remote *pvmd*. The list contains the IDs for all tasks on the same host as the remote *pvmd* which are to receive the multicast packet; this is normal for PVM. In phase II, first normal PVM procedures were used to multicast the data; N serial sends were sent to N remote *pvmds*. Then PVM was re-implemented such that N simultaneous sends were made to N remote *pvmds*. In the serial version of phase II, all ATM/PVM tests had a throughput of about 7Mbps; the Ethernet/PVM test had a throughput of about 4Mbps. In the parallel

versions of phase II, the ATM/FSAA/PVM tests provided the highest throughput of about 27Mbps which is about half the maximum throughput of the ATM/FSAA system alone. The ATM/BSD/PVM test had a throughput of approximately 20Mbps. The Ethernet/PVM test had a throughput of about 8Mbps.

Finally, Kaashoek and Tannenbaum[9] look at reliable group communication in distributed systems. Their work deals with broadcast messages, but can be applied to multicast situations. They identify and discuss group communication design issues. Also, they introduce an algorithm for totally-ordered group communication which is reliable. Plus, they provide algorithms for implementation in any distributed system.

Kaashoek and Tannenbaum's work is applied to the Ameoba distributed operating system. As mentioned in section 1.1, the Ameoba system has to be built into the kernel of its hosts and uses positive acknowledgments (ACKs) and is not based on IP. The *sequencer* can multicast the message for a sender but does not have to manage this. It does supply the sequence number in either case. The *sequencer* also manages retransmissions for the group. This protocol will be discussed in more detail in Chapter 3.

Chapter 3

Measurement, Analysis and Models

3.1 Algorithms for multicast reliability on LANs and WANs

IP multicast as defined in RFC 1112 uses the User Datagram Protocol (UDP), and therefore, it has the same reliability as UDP. UDP is a connectionless transport-layer protocol which provides an interface to IP for user processes. UDP does not guarantee that the datagrams will reach the intended receiver. UDP provides a port number and an optional checksum to verify contents of a datagram. UDP datagram packets can be 1) lost 2) out of sequence 3) duplicated 4) corrupt or 5) delivered correctly to the wrong place. In any case, it is the user's responsibility to provide the necessary acknowledgments and *timeouts* to make UDP reliable.

Most of the IP multicast applications to date have used unreliable multicast. For instance, *nv* and *vat* do not use reliable multicast. Although there may be large losses of data at times, perhaps causing a *blackout* of service, the small losses of such audio/visual information are quite tolerable. On the other hand in some applications, even small losses can not be tolerated. Applications such as whiteboards need reliable multicast to ensure that all updates are made for all receivers from all senders [24].

In order to make a datagram service reliable, some form of acknowledgment takes place. The acknowledgment may be in the form of a positive acknowledgment (ACK) where the receiver acknowledges all properly received packets. For example in PVM, the UDP daemon-to-daemon communication relies on ACKs for error control. On the other hand, the acknowledgment may be negative (NAK) where the receiver only acknowledges detected errors.

In the ACK-style protocols, which are reviewed again in section 3.1.1, the receiver must acknowledge all multicast messages at least once. The sender may retransmit either by unicast or multicast. In VMTP[10], the sender waits for an acknowledgment before transmitting a new message. This means that the *history buffer* need only contain the current multicast message or multiple-multicast message. On the other hand, in the Ameoba system[9], it is the receiver which discovers lost packets. The Ameoba's *sequencer* keeps all messages long enough for every live receiver to ACK. For each IP multicast retry, both VMTP and Amoeba must wait for at least the ACKs from those receivers which did not acknowledge the original message. In comparison with a *one-to-n* fanout as is used in PVM, the use of VMTP or Ameoba with IP multicast theoretically decreases the out-going multicast traffic from N sends to one. Since the Ameoba system's receivers are responsible for requesting retransmissions as well as ACKing, the Ameoba system could incur more reverse traffic than VMTP and PVM. VMTP's reverse traffic (the ACKs) would be no more than that of PVM.

In the NAK-style protocols, which are reviewed again in section 3.1.2, the receiver only acknowledges packet errors. Since the sender does not have to wait for NAKs, it may send several messages before a NAK is returned for any one of them. Therefore, it usually retains a message for a particular time period which will vary depending on the environment. If the sender deletes a message from the buffer prior to receiving a NAK for it, the data may have to be reproduced at a higher-level or an error message may

be sent in place of it. Also, if the sender has not sent a multicast message for a while, state information may need to be multicast in order for the receivers information to be updated. The sender may opt to retransmit by multicasting or if the number of NAKs is *small* by unicasting.

In addition, the manner in which a NAK is returned may vary. NAKs may be sent either by unicast or multicast. Since NAKs are only sent when an error such as a lost message is detected by the receiver, the amount of reverse traffic is much reduced in comparison with the ACK-style protocol. If the NAKs are unicast, then only the receivers which detect an error would return an acknowledgment. On the other hand, if the NAKs are multicast, then other receivers seeing the NAK may wait for the sender to retransmit the message. This option ensures that at most one NAK per multicast will be transmitted and further reduces the amount of reverse traffic.

In this section, reliable multicast protocols are examined. The following protocols are discussed as well as the one developed as part of this research:

- ACK-style protocols
  - VMTP by Cheriton[10]
  - Amoeba systems group communication by Kaashoek and Tanenbaum[9]
- NAK-style protocols
  - A Multicast-NAK-style protocol for LANs by Ramakrishnan and Jain and for WANs by Jacobson[24].
  - MTP Unicast-NAK-style protocol by Armstrong, Freier, and Marzullo[28]
  - RAMP NAK-style selective retransmission protocol by Braudes and Zabele[30]

3.1.1 ACK-style Protocols

VMTP use in reliable multicast

Cheriton's RFC 1045 *Versatile Message Transaction Protocol* (VMTP)[10] was not made explicitly for obtaining reliable multicast, but can be used by a client to aid in

making multicast reliable. As mentioned section 2.3, VMTP is a transaction protocol consisting of a *client-request–server-response*. This is useful in remote procedure calls (RPC).

VMTP provides a message identifier, a checksum, and ACKs with *timeouts* and retransmissions. The 32-bit identifier contains the client and transaction information. This is used for filtering duplicate *requests*. The identifier and its checksum allow for better protection against corrupt packets than the 16-bit IP checksum. Also, the checksum is placed in the trailer a more logical place than IP's leading checksum. The checksum may optionally just check the packet header.

The ACKs are dealt with in one of two manners. One way is for the server to return an ACK containing the same identifier as the client's *request*. Another way to ACK is by using a cumulative acknowledgment. In this case, the ACK also acknowledges all unACKed *requests* which have lower identifiers values. This is mainly used in situations where the *requests* are streamed. The *request* may be retransmitted up to a maximum number of retries. Cheriton suggests a maximum of five retries. However, lower or higher values may be used depending on the environment. The protocol also provides for selective retransmission of parts of the data.

The client and the server both have timers. The client's timer is set based on whether it is waiting on a first response or a retransmission, on a multi-packet response, or on responses to a group request. The server's timer is also set depending on what the current conditions are, such as waiting for a request, for the ACK to a response, or for a terminating-request if the client goes down. Figure 3.1 lists the steps for using VMTP in reliable multicasting.

VMTP does not directly support reliable multicast, but Cheriton suggests that a client could maintain a list of members for a group which would contain pertinent information, such as group and transaction identification and acknowledgments[10].

Reliable Multicast with VMTP	
1.	Client prepares information table for a new multicast request in order to keep track of ACKs.
2.	Client (re)transmits multicast request.
3.	Servers receive the request.
	3.1. Check for errors.
	3.2. ACK the request or times out waiting for a request.
	3.3. Prepare or updates the information table for the response
	3.4. (Re)transmit response.
	3.5. Wait for an ACK of the response or times out.
4.	Client receives the ACK for a request or *times out* and retransmits.
5.	Client waits on the responses or *times out.*
6.	When a response is received, the client checks for errors.

Figure 3.1: VMTP

A description of how VMTP's ACK-style protocol can be used for multicasting.

Depending on the number of servers which do not ACK, the client could re-issue the request via multicast until all are ACKed, or it may retransmit a single request to specific servers from whom no ACK was received.

Ameoba System's Reliable Multicast

The Ameoba distributed operating system provides for reliable group communication in a LAN. This is accomplished by the *sequencer* which is built into the kernel of each host in the system. When a group is created on one machine, that machines *sequencer* is enabled and manages the sequence numbers for messages in that group. If that particular *sequencer* fails, the members of the group *elect* a new one. There is no guarantee that all unACKed messages up to the failure will be preserved. Figure 3.2 lists the steps for reliable multicasting in the Ameoba system.

Reliable Group Communication in the Amoeba system	
1.	The sender transmits the multicast (or broadcast) message:
	1.1. either by point-to-point to the *sequencer* - which incorporates the sequence number, then multicasts the message.
	1.2. or by multicast to the group - the *sequencer* will then multicast the sequence number (the *accept* message).
2.	The *sequencer* stores the message in a buffer until all ACKs are in.
3.	The receivers wait for the message (and seq. number).
4.	If the new seq. number is higher than expected, the receivers send a message to the *sequencer* requesting the missing message.

Figure 3.2: Group Communication in the Amoeba Distributed OS
Uses an ACK-style protocol.

Two methods are used for acknowledgments by the system. One is a *point-to-point* method called the *PB method* where the sender sends the multicast (or broadcast) message to the *sequencer* which in turn multicasts the message. The other is the *BB method* where the sender sends the multicast message. In this case when the *sequencer* sees the multicast, it issues the sequence number in a short *accept* message. The BB packet is only valid if the *accept* message is sent. In both cases, clients must ACK the *sequencer*. The *sequencer* keeps a *history buffer* in order to re-send any lost messages. When the kernel of a client notices a gap in the sequence numbers. It sends a message directly to the *sequencer*, which replies with a copy of the message.

Since in the *PB method* the message goes onto the network twice, it uses about $2n$ more bandwidth than the *BB method.* However, with the first method, the receivers are interrupted only once. With the second method, the receivers are interrupted twice once for the multicast and once for the accept message. Kaashoek, et al.[9] suggest that PB be used for small messages and BB be used for large messages. The user determines what will be considered *small* or *large*.

3.1.2 NAK-style Protocols

Multicast NAK-style protocol

According to Pingali, et al.[24], the NAK-style protocols by Ramakrishnan and Jain for LANs and by Jacobson for WANs are very similar. Both protocols multicast (or broadcast) the negative acknowledgment and can provide reliable multicast on top of IP. The Pingali analysis of these protocols found them to have a higher rate of throughput than a NAK-style protocol using unicast NAKs and a non-optimal ACK-style protocol. In these multicast NAK protocols, a receiver sets timers to a random value upon detection of a an error. If no NAK has been sent before the timer expires, that receiver NAKs the message via a multicast. If a NAK has been sent before the timer expires, then the timer is reset. The receiver waits for either the retransmission, another NAK, or its timer to expire. The receivers which have no losses incur a small cost for the multicast interrupts of the NAKs and retries. Figure 3.3 lists the steps for this NAK-style protocol.

Multicasting NAKs	
1.	The sender multicasts messages.
2.	The sender sets a timer for for the removal of the message from the buffer.
3.	When a receiver detects a lost packet, it waits a random period of time.
	3.2. If no NAK was multicast during this time period, it multicasts the NAK to the sender and receivers and waits for the multicast message.
	3.3. Else if a NAK was sent it resets its timer and waits for the multicast message, a NAK, or its timer to expire.
4.	Retransmissions have priority over sending a new multicast message once the NAK is received.

Figure 3.3: Multicast NAK-style Protocol for LANs and WANs

Unless the sender has gone down, all receivers will eventually receive the message. In addition, the sender sets a timer for each multicast message held in its *history buffer*. When a timer expires, its message is removed.

Unicast NAK-style protocol

Multicast Transport Protocol (MTP) by Armstrong, et al.[28] and MTP2 by Bormann, et al.[30] are NAK-style protocols that differ from the previous protocols in that the NAKs are unicast to the sender of the multicast message instead of multicast. The MTP2 optimizations were listed in section 2.3. As for acknowledgments, the major difference between MTP and MTP2 is that MTP2 does not require the receiver to know to whom the NAK should be sent.

MTP is built with protocols which provide group address and network service access. When a process joins, several options are set, such as whether it will be a master, a producer (sender) or a consumer (receiver), whether the service will be *best effort* or reliable, and whether it will receive from one or many senders.

A master server controls group communication. Unlike the MTP2 master, if the MTP master fails, all can fail. The master maintains a *message acceptance* record for the twelve most recent messages. The record contains information as to whether the message was accepted, rejected or still in progress. This record is *piggy-backed* with every message the master sends. Also, the master is responsible for assigning global message sequence numbers for all messages. This is accomplished through the use of a token.

A sender must receive a token from the master in order to multicast. This token contains the global sequence number. When the sender reaches the end of the message, the token is automatically released. Tokens have their own sequence numbers separate from other messages. The senders maintain the message data for a limited time period for retransmissions. If the sender *times out* and removes a message before the NAK is

received, it issues a NAK indicating that the message no longer exists. If this places the receiver in a state of failure, it withdraws from the group.

Only those receivers which did not receive a particular message will send a NAK for the message. The data field of the NAK contains a list in ascending order of message and sequence number pairs for missing packets. When the sender retransmits the multicast message, the receivers which did not have errors are capable of dealing with duplicates. By decreasing the number of NAKs to at most one per multicast message, the multicast NAK can save bandwidth as compared to the unicast NAK.

NAK-style protocol with unicast and multicast retransmissions

Reliable, Adaptive Multicast Protocol (RAMP)[30] is a proposed transport-level protocol which can provide reliable multicast service on top of IP multicast. It is adaptive allowing for different levels of reliability. The quality of service can be modified usually based on the number of retransmission requests or router-generated squelch requests by sending a modification request to the Multicast Group Authority (MGA).

RAMP uses negative acknowledgments in error control. A major difference between this and the other NAK-based protocols is that it can use either unicast or multicast for retransmissions. Deciding which to use depends on the number of NAKs which the sender receives within a particular time span. The sender multicasts messages. The receivers receive the messages or detect errors, such as missing one or more packets. The receiver which detects an error sends a NAK to the sender requesting all missing packets, sets a timer, and waits for the retransmission.

When the sender receives a NAK, it sets a NAK timer. When this NAK timer expires, the NAKs are counted, and the sender will send copies of a message via unicast or multicast depending on the number of requests for the message. If the message no longer exists, the sender replies with a NAK for the NAK. Since it only retransmits via

multicast when the number of NAKs is *large* enough, this protocol optimizes the number of receiver interrupts.

3.1.3 Protocol Used in This Study

The goal is to evaluate the reliable multicast protocol (*RMP*) by implementing it in a real application; PVM was chosen. Currently PVM's multicast function, **pvm_mcast()**, is only task-initiated. Although the local PVM daemon (*pvmd*) does the actual multicasting, it never initiates a multicast. A positive acknowledgment system already exists in PVM's *pvmd*-to-*pvmd* communications. Therefore, instead of duplicating the acknowledgments as would be required in using the RMP, PVM's ACK process is altered slightly in order to reliably implement IP multicast in PVM (see section 4.2.3 for more details).

Cheriton's VMTP is a positive acknowledgment protocol. VMTP could process PVM messages and provide reliability for the underlying multicast capabilities. However, according to Geist, et al.[6], it has already been rejected because it is not widely available and would require modifying the kernel of each host in the system. The Ameoba systems multicast protocol is also a positive acknowledgment protocol. However, it is embedded in the operating system too, and one would have to modify the kernel of each host in the *parallel virtual machine*. Neither of these protocols would easily work in PVM, and both would decrease PVM's portability.

The negative acknowledgment protocols may decrease the amount of reverse traffic. However, the MTP and MTP2 schemes have a master controller. Linking such a master and the *pvmd* could require major changes to PVM. Since the RAMP scheme provides a multicast group authority, the same sort of problem would exist in trying to use it with PVM. Finally, the works by Ramakrishnan and Jain and Jocobson are negative acknowledgment protocols which multicast the NAKs. These multicast NAK-

style protocols would not be simple to implement in PVM either. Currently, PVM does not allow *pvmd*-initiated multicasting, and it uses an ACK-based protocol.

The protocols mentioned above require reconfiguring kernels and/or some re-structuring of PVM. The RMP is an ACK-based protocol. This allows the RMP to be transferred to PVM with only modest changes to the PVM logic.

3.2 Multicast Analysis Methodology

To evaluate the performance of unreliable and reliable IP multicast, a test environment was established, a set of test programs and scripts. The set of test machines included about thirty Sun workstations at both Oak Ridge National Laboratories (ORNL) and the University of Tennessee at Knoxville (UTK) providing both a local and wide area test environment. The test environment also included protocol for detailed analysis of multicast packet flow. The wide area multicast route was also alterable from a few hops to many hops by changing routing information for both *mrouters* at ORNL and at UTK. The following sections present a description of IP multicast socket extensions and of an unreliable program and its development into the RMP.

3.2.1 IP multicast socket extensions

Only a few vendors (e.g., Silicon Graphics and Sun Solaris) presently provide IP multicast support. Therefore, preparing the test environment required much effort. The kernel of the machines used in the work had to be re-built using IP multicast software, only then could the UDP sockets be used for multicasting as is described in the paper by Deering [23]. In order to do wide-area multicasting, MBONE routes had to be configured at the local and remote sites.

The programming extensions required to support IP multicast included additional **setsockopt()** calls. In order to send or receive IP multicast, AF_INET sockets with type SOCK_DGRAM were used. Any IP multicast address from 224.0.1.0 to 239.255.255.255 could be used as the address in a **sendto()** function call. Since the delivery of a datagram to a particular socket is dependent upon the destination port number, the port number had to be included in the sockaddr_in information for the **sendto()**. Also, the senders message was not received by the senders host; the loop-back option was set to zero (the default). This is accomplished by using the following option.

```
unsigned char  loop = 0;  /* if 0, then host mach. doesn't hear */
setsockopt(socketfd,IPPROTO_IP,IP_MULTICAST_LOOP,&loop,sizeof(loop));
```

The time-to-live (*ttl*) parameter depends on the *distance* (or number of hops through the MBONE) between a sender and a receiver. The *ttl*'s value can be any number from 0 to 255.

- A ttl of 0 reaches only the host of the sender.
- A ttl less than or equal to 31 stays within the sender's *subnet.*
- A ttl less than or equal to 64 stays within the sender's *site.*
- A ttl less than or equal to 127 stays within the sender's *region.*
- A ttl less than or equal to 254 stays within the sender's *continent.*
- A ttl of 255 is unrestricted.

Sites and *regions* may be defined differently among internets. In this study, the *ttl* for the LAN was set at one. For the UTK/CS-ORNL WAN, it was set at about 76 for the MBONE connection via SURAnet and at 18 for a direct tunnel between ORNL and UTK via a T1 line. In order to set the *ttl*, the following socket option was used:

```
unsigned char   ttl=1;   /* reaches the hosts in the LAN */
setsockopt(socketfd,IPPROTO_IP,IP_MULTICAST_TTL,&ttl,sizeof(ttl));
```

To receive datagrams sent to a particular port, that local port was used. Then to join the IP multicast group the following was used:

```
struct   ip_mreq    imr;
struct   hostent    *hp;
char     host[HOSTNAMESIZE], *send_to_addr;
hp = gethostbyname(host));
memcpy( ( char * )&imr.imr_interface.s_addr, hp->h_addr, hp->h_length );
imr.imr_multiaddr.s_addr = inet_addr(send_to_addr );
setsocketopt(socketfd,IPPROTO_IP,IP_ADD_MEMBERSHIP,
&imr,sizeof(struct ip_mreq));
```

The *leave* option was not used since the memberships are dropped when the socket is closed or the process holding the socket is killed.

3.2.2 Testing Unreliable Multicast

A simple *echo*-type program was used to test unreliable IP multicast in a LAN and a WAN. To characterize packet loss, an effort was made to determine if the losses were due to the multicast packets or due to the ACKs, and if the losses correlated with message length, time of day, number of processors, network load or host load.

In this program, the client was designated as the *sender* (or master) and the servers as *receivers* (or slaves). Also, only the sender sends multicast messages, and only the receiver receives multicast messages. Both the sender and the receiver can communicate via unicast messages. In the *echo* portion of the program, the sender sends an l-byte multicast packet containing the sequence number; it waits to receive the four-byte ACKs containing the sequence number unicast from all of the receivers before repeating any *send-receive* steps. When the sender does not receive an ACK from a receiver due to *timing out*, it is termed a message loss. The client does not check for lost messages; only the sender does. In limited testing on a LAN and a WAN ($ttl = 76$), the losses were nearly all multicast packets and rarely the unicast ACKs (see section 3.3.1 for more detail).

The master program begins with a *start-up* routine. One at a time, the master starts the receivers program via **rsh**. Each receiver sends a four-byte message containing its PID number to the sender. Upon receipt of the PID, the sender stores the receiver's address and PID number which are later used for identifying or killing the slave program. The sender returns to the receiver a four-byte unicast message containing $-1*$PID. The client checks for this message, returns its PID to the sender, and waits for a multicast message containing $-$PID. Again, the client will reply with its PID. Once the sender receives both replies from the appropriate receiver, the receiver is assumed to be ready. Since **rsh** failures were common when large numbers of machines were being used and since MBONE multicast router (*mrouter*) failures were possible, this *start_up* routine was used to assure that receivers were indeed working before the *echo* testing was begun. After the *start-up*, the number of slave programs remained constant until one died or the master killed them.

The master and slave(s) should now be ready to start the *echo* routine. First, the sender stores timestamps for calculating the overall time to do r repetitions and for calculating the time to call a **sendto()**. Also, time was stored for calculating the round-trip-times (*rtt* – sending the IP multicast message and receiving a particular ACK) for each slave per ACK received which are used to calculate maximum, minimum, and average receive times.

An l-byte packet is sent via multicast to the receivers in the group. The receivers return the sequence number which is the first 4-bytes of the packet. The sender waits for the ACKs or *times out.* As each ACK returns, a time value is stored; this value is the time between sending the multicast and receiving the ACK from a particular slave. Also, a packet counter is incremented for the slave. If the incoming packets sequence number is less than expected, the packet time stamp for that slave is *zeroed-out*, and its packet counter is incremented for that particular sequence number. This indicates

an out of sequence packet. In testing, these steps are repeated r times for an l-byte message. At the end of the r repetitions, the slave information is checked and duplicates, out of sequence, and lost packets are counted. The total time for r repetitions and the maximum, minimum, and average times for the receives are also calculated.

3.2.3 Testing Reliable Multicast

The reliable IP multicast program evolved from the unreliable *echo* routine. The RMP used in this study contains the same *start-up* procedures as the unreliable program (see Figure 3.4). Although these *echo* routines are basically the same, the the reliable routine provides multicast and unicast retransmissions. Data was checked when all of the ACKs for one IP multicast had been received.

Multicast Reliability Algorithm	
1.	Set counter $M = 0$ (multicast retry counter).
2.	Set p = Percentage of receivers representing a successful multicast.
3.	Set R = Maximum number of retries of the multicast send.
4.	Send multicast packet m. Increment counter M.
5.	Start loop (from 0 to *num_o_rcvrs*) to recv ACKs:
	5.1. set timer for rtt,
	5.2. recv ACK,
	5.3. turn alarm off,
	5.4. repeat steps 5.1 through 5.4 until loop is finished,
6.	Count the number of messages ACKed:
	6.1. If ((num_ACKed $> p$)) or ($M > R$)), then issue no more than u unicast sends for receiving the ACK.
	6.2. Else, repeat steps 3 through 6.

Figure 3.4: The Reliable IP Multicast Protocol

For error control, ACKed packets were counted. When all ACKs for a send were received or *timed out*, a check was made to determine if a packet was lost. If the

number of multicast retries (M) was less than the maximum number of retries (R) or the $num\_ACKed < p\%$ of the number of receivers, then the multicast packet was retransmitted. Otherwise, up to u retries of a unicast copy were sent to the receiver(s) who had not ACKed. This is similar to the scheme used by Kaashoek and Tannenbaum in the Ameoba system[9]. These retries took precedence over any new multicast sends. A receiver which lost 100% of *all* packets after r repetitions was considered dead. It was removed from the receivers list, and the number of receivers was decreased by one.

Since UDP packet corruption is very unlikely on a local network, the UDP checksum is usually disabled. Corruption would more likely occur on a wide area network, but the probability of corruption on the WAN is still very small. Packets were not checked for corruption. In order to do so, some type of checksum would have to be used by the application, or the UDP checksum would have to be enabled in the kernel of each host. Therefore, the receivers immediately returned an ACK containing the sequence number of the multicast message. The receiver could have been required to check the packet for corruption before replying.

If a receiver is down, then the reliable method takes much longer than unreliable since it must *time out* for zero to M multicast sends and for u unicast retries for the r repetitions before being deleted from the list. However, if the hosts, receivers and *mrouters* were running properly, then the time for reliable multicast would not be much more than the unreliable method. Often only one multicast or one unicast retry was needed to recover from a *time out* which adds only milliseconds more.

3.3 Multicast Performance

3.3.1 Losses on a LAN for a *best-effort* program

In the LAN using a master on a SPARC 10 workstation at ORNL with *best-effort*, the losses for 1000 messages of l length were very small. The failure rates range from a low of 0% to a high of only 0.056%. Table 3.1 lists the percent of losses on the ORNL LAN for unreliable IP multicast. Losses occurred most often for workstations which are inherently slower or busier than others. At particular times of the day, around noon or 3:00 am EDT, losses were higher. In gathering the data for the LAN, the unreliable *echo* program was run approximately every hour for about 3 days by using a script. A sender and 16 receivers at ORNL were used during these tests. Each test consisted of $r = 1000$ repetitions for $8-, 512-, 1024-$, and $2048-$byte packets.

Table 3.1: Failure Rate of *best-effort* IP multicast on a LAN using 16 receivers, $r = 1000$ repetitions and $ttl = 1$.

Packet Length	%losses
8	0 - 0.056%
512	0 - 0.050%
1024	0 - 0.043%
2048	0 - 0.038%

The losses on the LAN can be attributed to several things. A host may drop packets simply because it is too busy to catch every packet or its buffers may be too full for incoming packets. Also, a host may be slow in transmitting an ACK because it is very busy or because the network is very busy. This could cause the master to time out before the packet arrives. If the master is on a busy host or slow host and a large number of receivers are returning ACKs, then even the unicast acknowledgments could be dropped.

3.3.2 Losses on a WAN for a *best-effort* program

The WAN from ORNL to UTK/CS[1] had some problems. First, the multicast route takes several hops through the MBONE before reaching Knoxville with a *ttl* of seventy-six, whereas a unicast packet has the nicety of a *T1* line connecting Oak Ridge and Knoxville. Second, as was later discovered, receiving-end *mrouters* were not functioning properly causing as much as 100% losses for UTK/CS receivers. Tests in the reverse direction (UTK/CS to ORNL) were not affected by this problem and did not have many losses. Finally, the hops through the MBONE decreased the speed of multicast packets. When using a *ttl* of seventy-six, a multicast packet takes approximately 180 ms to travel between UTK/CS and ORNL, whereas a unicast ACK takes only 4 ms.

In early tests, the *start-up* routine did not include a multicast send. At times the test results were not too disappointing. Losses appeared to be a function of packet length, number of remote machines, and time of day. However, the remote machines at UTK/CS had 100% losses much of the time for all packet sizes. The initial multicast was added to make sure that *mrouters* were running at the start of the test. However, many times it was only the longer length packets which had complete losses. So, another 8-byte test was added after the 2K-byte test to check whether it was just that large packets could not make it through or whether a receiver or *mrouter* had gone down. Usually but not always, if the 2k packets had 100% losses, the second 8-byte tests also did.

Later while running a multicast talk (*mtalk*) program manually between ORNL and UTK/CS and watching both the sender and receiver, it was observed that for long time periods multicasting into UTK/CS resulted in 100% losses. Then for a few minutes there would be virtually no losses, whereas when multicasting from UTK/CS into ORNL, the losses were continuously quite low. Also, *mtalk* was run between UTK/UTCC [2] and

[1] University of Tennessee, Knoxville's Computer Science Department.
[2] UTK's, University of Tennessee Computing Center

UTK/CS machines. A looping effect was noticed which seemed to be somehow related to the *ttl* value. Part of the problem may be due to the MBONE router's use of DVMRP. Stephen Deering[3] reports that *mrouters* when running in different subnets which are connected by a bridge can cause routing loops on the MBONE. This is the configuration of the UTK *mrouters*. Presently, Deering and others at Xerox PARC are working on this problem.

To eliminate the possibility of the problem being within SURAnet, the MBONE links were changed to tunnel directly between ORNL and Knoxville with a *ttl* of eighteen. Occasionally, 100% losses for the distant machines still occurred. It appeared that if the program could be run within a particular window of time, then losses would be much less. Outside that window, nothing would be received at UTK from ORNL.

Table 3.2 shows test results for the WAN using unreliable multicast. In these tests, a *ttl* of 76 and $r = 100$ repetitions were used. Both sets of data involve two receivers at ORNL and two receivers at UTK/CS. For set one, the sender was on a SPARC 10 at ORNL. For set two, the sender was on a SPARC 5 at UTK/CS. Since the number of losses were so high for the ORNL to UTK/CS tests, the number of receivers and repetitions was kept low. The losses for the WAN originating at UTK were on the order of ten percent less than the losses for the WAN originating at ORNL. This difference is due to the UTK *mrouter* looping problem mentioned earlier in this section. The number of hops through the MBONE combined with the normal UDP problems also affect losses.

[3]deering@parc.xerox.com via mbone@ISI.EDU mailing list, Thursday, Oct. 20, 1994

Table 3.2: Failure Rate of *best-effort* IP multicast on a WAN with a total of 4 receivers, $r = 100$ repetitions and $ttl = 76$

Packet Length $r = 100$	%losses sender at ORNL 2 receivers each subnet	%losses sender at UTK/CS 2 receivers each subnet
8	26 - 50%	0.8 - 2%
512	0 - 50%	0.5 - 3.5%
1024	35 - 50%	0.5 - 3.5%
2048	26 - 51%	2.0 - 6.25%

3.3.3 Reliable LAN Multicast Performance

In measuring the performance of the reliable IP multicast protocol, several parameters were varied, such as packet size and receiver set size. The reliable IP multicast program which was discussed in section 3.2.3 was used to examine time verses packet size for various receiver set sizes in both the LAN and the WAN. For comparison, this program was altered to form a multiple unicast version. This new program uses a loop to send multiple unicast packets instead of IP multicast, and it only retransmits via unicast; also, no multicasting is performed in its *start-up* routine. The following subsections summarize the results of the performance tests for both programs.

Using 1, 2, 4, 8, and 16 receivers in a LAN, the reliable IP multicast program was run with a 1000 repetitions of each message size, 8-, 512-, 1024- and 2048-bytes. The results of the test are based on the average round-trip time (*rtt*) per repetition for each message size. As the number of receivers and/or the message size increased, the average *rtt* per repetition increased only slightly (see Figure 3.5). For instance, as the number of receivers increased from 1 to 16, the time increased by only about 12 ms for each packet size. Also, as the packet size increased from 8-bytes to 2k-bytes, the time increased by at

most 0.6 ms. For IP multicast on the LAN, the data shows that with a large message size or a large number of receivers the increase in *rtt* is relatively small.

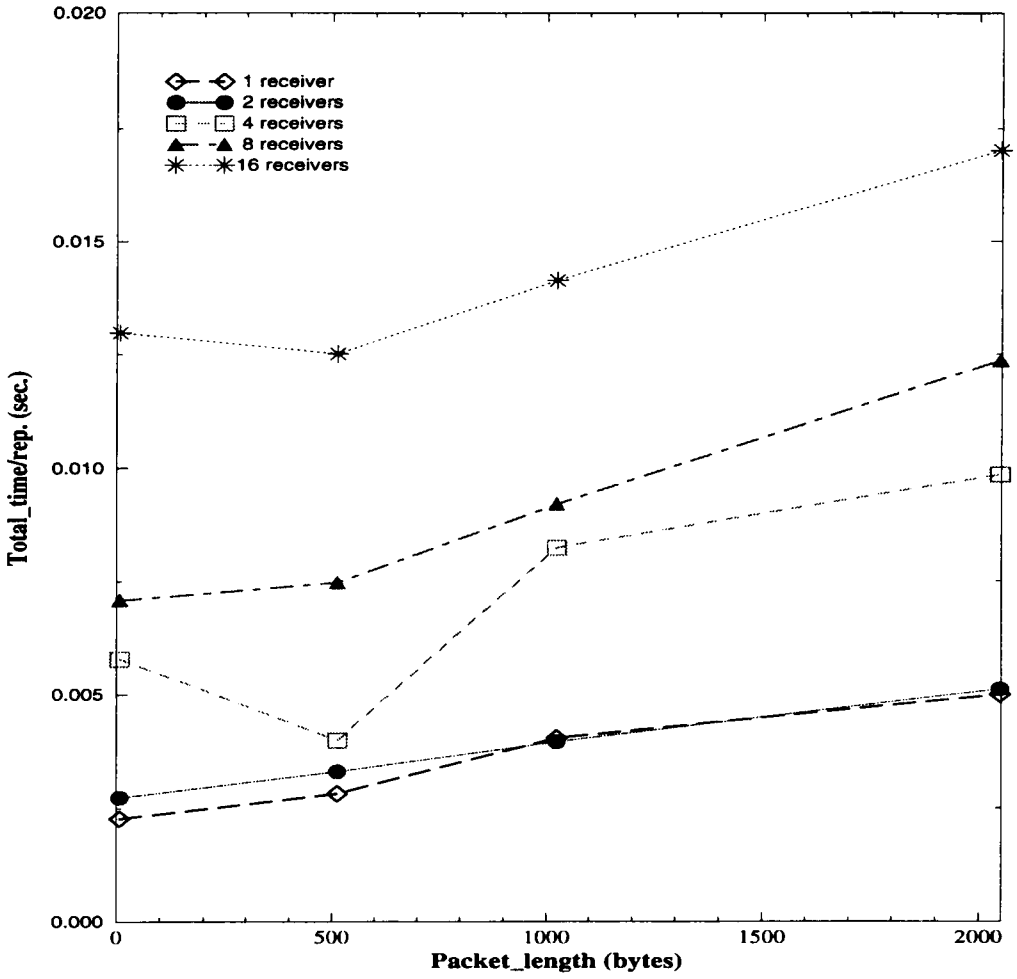

Figure 3.5: Reliable IP multicast LAN Performance.
Total Time/rep (sec.) vs Message Length using 1, 2, 4, 8, and 16 receivers.

Using 1, 2, 4, 8, and 16 receivers in a LAN, the reliable multiple unicast program was run with a 1000 repetitions for each message size of 8-, 512-, 1024- and 2048-bytes. The results of these tests are also based on the average *rtt* per repetition for each message size. As the number of receivers and/or the message size increased, the *rtt* per repetition

dramatically increased (see Figure 3.6). For instance, as the number of receivers increased from 1 to 16, the time for a 2k-byte packet increased by 150 ms on average. The data shows that with a large message size or a large number of receivers the increase in time is quite large in comparison with IP multicast.

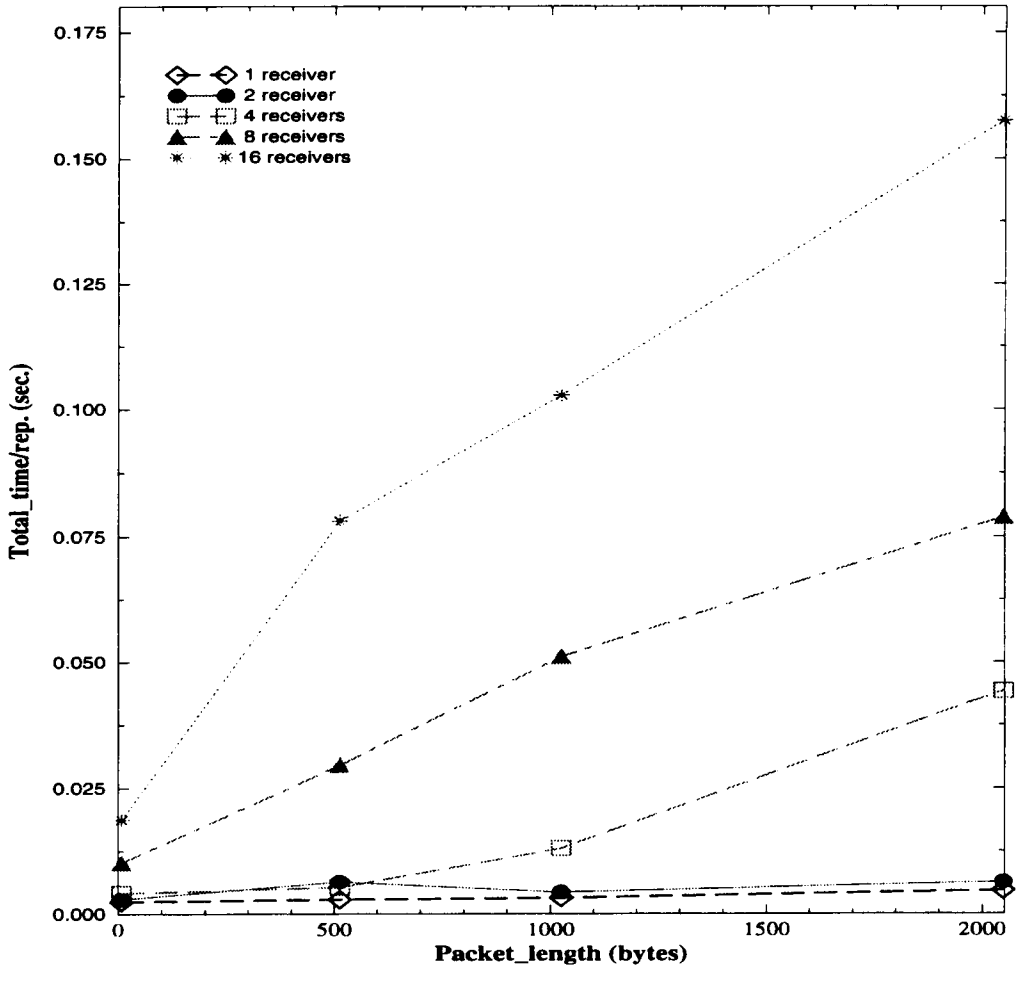

Figure 3.6: Reliable Multiple Unicast LAN Performance.
Total Time/rep (sec.) vs Message Length using 1, 2, 4, 8, and 16 receivers.

The results indicate that in a LAN the use of IP multicast for larger message and receiver set sizes would be an asset to a program such as PVM which uses multiple

unicasts to implement multicast. Using the same data as above, Figure 3.7 represents the average *rtt* per repetition for IP multicast and multiple unicasts for 8 and 16 receivers. This allows us to compare the IP multicast and a 1-to-n fanout approach for large message sizes and for large numbers of receivers. As can be seen in this case, the IP multicast *rtt* is much less than the multiple unicasts for larger packets. These results are consistent with the model in section 3.4.

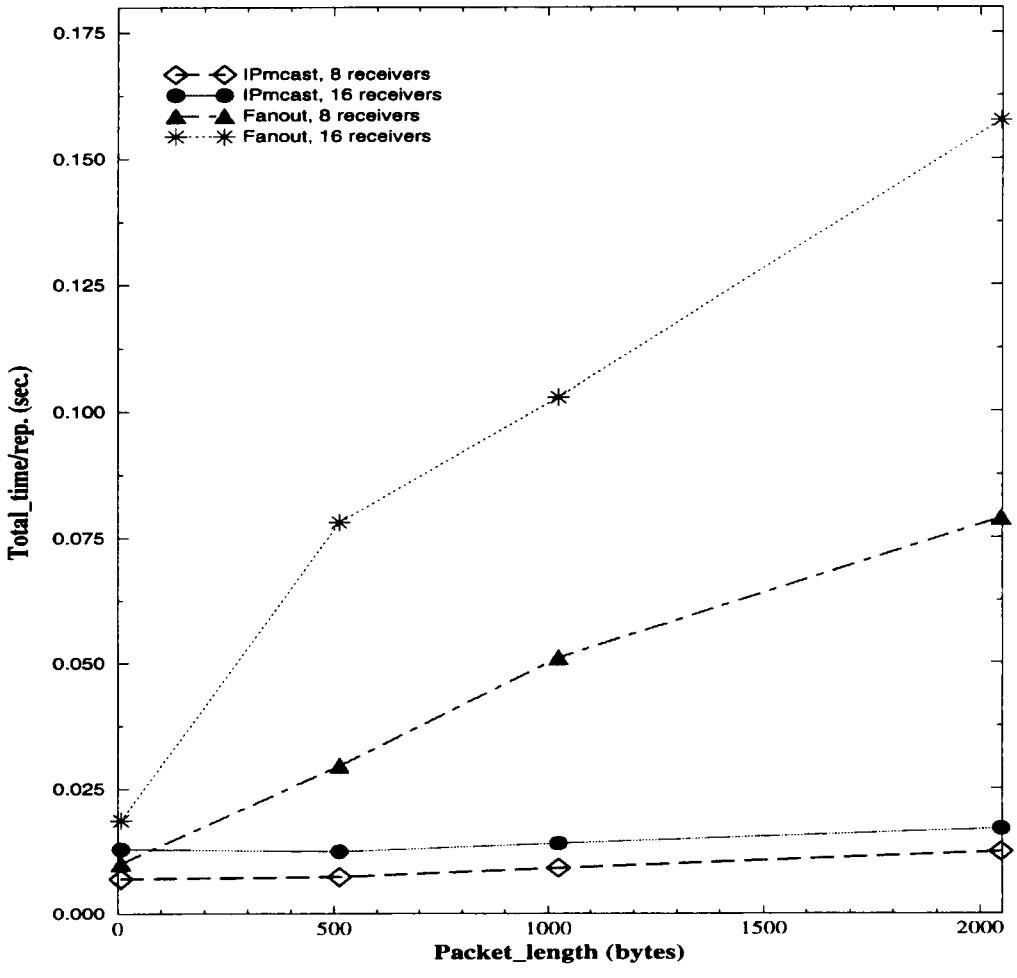

Figure 3.7: Reliable IP multicast vs Multiple Unicasts LAN Performance. Total Time/rep (sec.) vs Message Length using 1, 2, 4, 8, and 16 receivers.

3.3.4 Reliable WAN Multicast Performance

In the *directly-connected* WAN with the sender at UTK/CS, the reliable IP multicast and multiple unicast programs were run using 5 receivers on each subnet with 8-, 512-, 1024, and 2048-byte messages for 1000 repetitions each. The following results are based on the average *rtt* per repetition for each message length in the WAN with a *ttl* of 18.

The performance of the reliable IP multicast program was better than the multiple unicast performance for the WAN especially for larger packet sizes (see Figure 3.8). On average, the IP multicast *rtt* time per repetition was a little more than 100 ms. The multiple unicast results for *rtt* time per repetition ranged from a little more than 100 ms to more than 400 ms. The multiple unicasts increase sharply for the 2k-byte messages. This could be due to the fragmentation of the message as it is sent out of the subnet or due to improperly tuned timers for receiving messages. Other possible reasons will be discussed in section 3.5.

Also, in the WAN using a *ttl* of 76 with the sender at UTK/CS, the reliable IP multicast and multiple unicast programs were run using five receivers on each subnet with 8-, 512-, 1024, and 2048-byte messages for 100 repetitions each. The number of repetitions was decreased since the IP multicast could have taken an extremely large amount of time due to *timing out* and retransmissions. The following results are based on the average *rtt* per repetition for each message length with a *ttl* of 76. The performance of the reliable IP multicast program for this WAN was poor in comparison with the multiple unicasts over the WAN (see Figure 3.9) except for the 2k-byte packets where both perform about the same. Recall from section 3.3.2 that the unicast path is much shorter (4 ms) than the multicast path through the MBONE (180 ms).

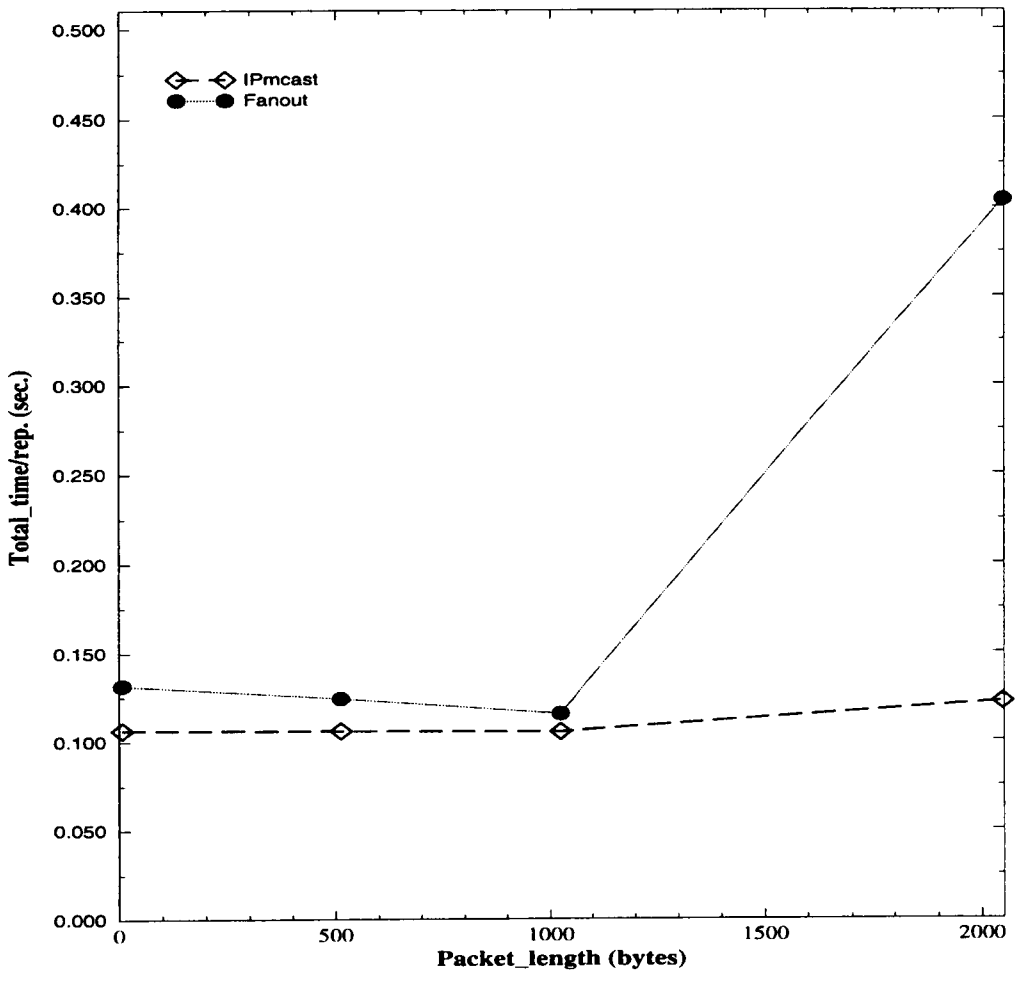

Figure 3.8: Reliable IP multicast vs Multiple Unicast WAN Performance ($ttl = 18$) Total Time/rep (sec.) vs Message Length with the sender at UTK/CS, receivers: 5 at ORNL and 5 at UTK/CS.

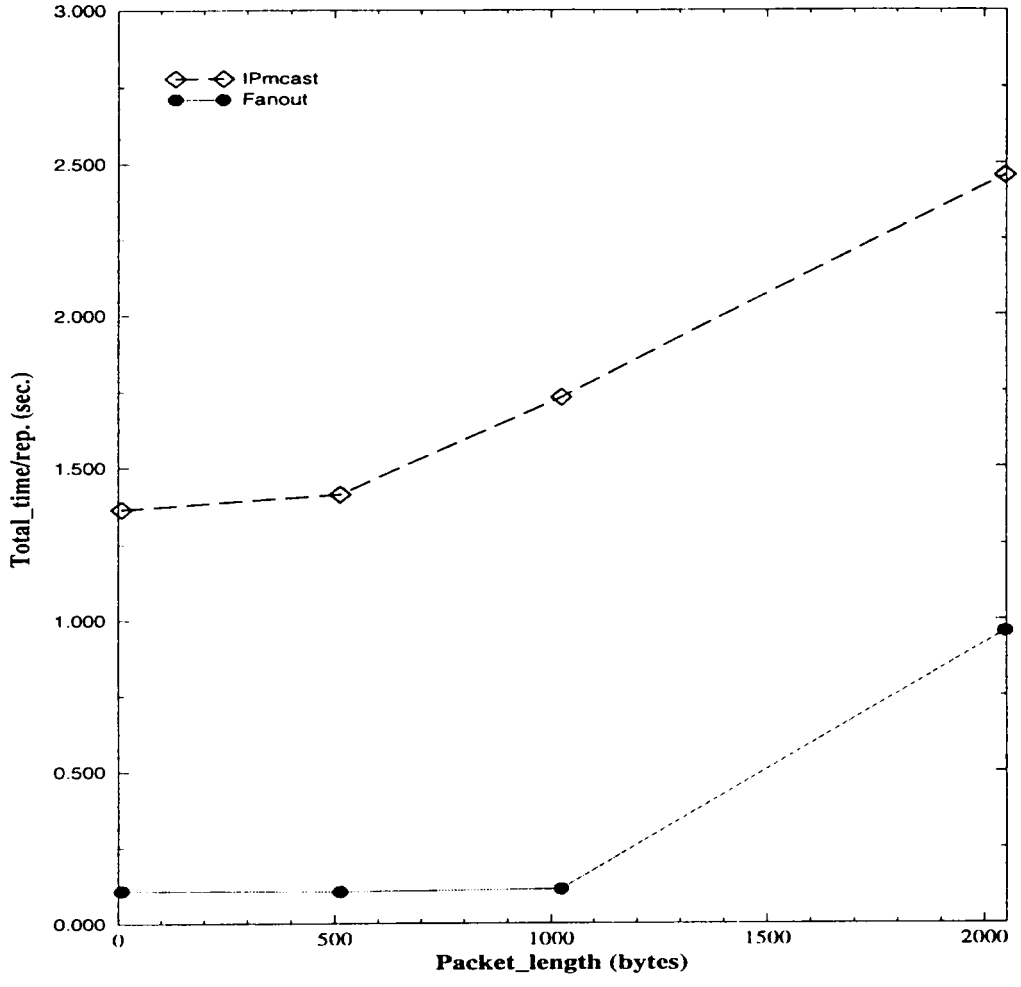

Figure 3.9: Reliable IP multicast vs Multiple Unicast WAN Performance ($ttl = 76$). Total Time/rep (sec.) vs Message Length with the sender at UTK/CS, receivers: 5 at ORNL and 5 at UTK/CS.

The time per repetition for IP multicast was more than ten times that of the multiple unicast sends. Both sets of data show an increase in the *rtt* per repetition as the packet size increases from 8-bytes to 2k-bytes. The increase in the number of hops through the MBONE causes a decrease in the performance of IP multicast. This and other performance issues will be discussed further in section 3.5.

Both programs used in the performance tests could have been optimized further. For instance, a more dynamic method for determining timer values might decrease unnecessary waiting. PVM contains such a method for its UDP messages. Also, better algorithms for determining whether to retransmit via multicast or unicast may help reduce reverse traffic.

3.4 Performance Model of Reliable Multicast

The following model explores the theoretical speed-up of using IP multicast instead of multiple unicast messages. The *send-receive* time is divided into relevant parts. The s value represents the time to call the **sendto()** routine which was assumed to be approximately the same for the IP multicast as for the unicast. The *out* parameter is the time it takes a message to travel from the sender's OS to the receiver's OS. The a variable represents the time for the message to be processed by the receiver and the ACK to be sent from the receiver. The in is the amount of time the ACK takes to travel from the slave host to the master's OS. Finally, r is the time for the ACK to travel from the master's OS to be be received by the master program. Most of these time components could be broken further depending on the particular computing environment. the ACK time values a, in and r are assumed to be equal for multicasting and unicasting.

Let:

s_u = unicast sendto() process time
out_u = unicast message from OS out to receiver time
s_m = multicast sendto() process time
out_m = multicast message from OS out to receiver time

a = time for receiver to process the message & to send the ACK
in = ACK from receiver to sender's OS time
r = ACK from OS to application time

n = number of receivers
U_t = time for n unicast sends and n receives
M_t = time for a multicast send and n receives

Time to send a message to all n machines and to receive all n ACKs.

$$U_t = n * (s_u + out_u + a + in + r) \tag{3.1}$$

$$M_t = (s_m + out_m) + n * (a + in + r) \tag{3.2}$$

Speedup, S, is:

$$S = \frac{U_t}{M_t} \tag{3.3}$$

Or,

$$S = \frac{n * (s_u + out_u) + n * (a + in + r)}{(s_m + out_m) + n * (a + in + r)} \tag{3.4}$$

Reduce the fraction by $\frac{1}{n*(a+in+r)}$,

$$S = \frac{\frac{(s_u+out_u)}{(a+in+r)} + 1}{\frac{(s_m+out_m)}{n*(a+in+r)} + 1} \tag{3.5}$$

Let:

$$A = (a + in + r) \tag{3.6}$$

Assuming, $s_u \simeq s_m$, then let s_u and $s_m \simeq T$. Then speedup, S^*, is:

$$S^* = \frac{\frac{(T+out_u)}{A} + 1}{\frac{(T+out_m)}{n*A} + 1} \tag{3.7}$$

On a LAN, both out_u and out_m increase as packet size increases. However, out_u not only increases because it takes longer to process n sends through the OS, but also increases due to the contention between some out going unicasts and the incoming unicast ACKs from earlier sends. This is less of a problem on a WAN where the ACK naturally takes longer to return. With IP multicast, no contention exists between an ACK and the IP multicast, unless there are duplicates or late arriving ACKs from earlier multicast messages. Unusual traffic and timer interrupts can affect timings for

either a unicast or a multicast send. Also, as the number of receivers (n) increase, the number of unicast packets being processed and sent by the sender increases using more bandwidth, whereas the number of multicast sends remains the same, unless there is a retry. However, on a WAN where an IP multicast packet must make many hops through the MBONE, speed up can decrease if the MBONE route is greater than the underlying Internet route for the unicast. Even when the multicast route is the same, multicast packets incur slightly more overhead due to the need to be encapsulated and to pass through the tunnels from *mrouter* to *mrouter*. The model suggests that IP multicast should out-perform multiple unicast as the number of receivers (n) increases.

3.5 Model Validation

The model does not allow for the dynamics of a network. In a LAN, the time for doing one or more sends can be affected by many things, such as message and receiver set size. Also, as the number of sends increases, the more opportunity there is for interruptions, such as context switches, timers, priority messages. In the work by Clark, et al.[32], it was found when processing a TCP packet on a Sun-3/60 workstation that the operating system overhead took more than twice as much time as did the actual TCP processing time. Such operating system costs are also factors in the cost of sending a UDP packet. Machine load can affect OS overhead. Also, the number of hops between hosts can affect the round-trip time.

In this study, a program was written to determine the timer interrupt costs. In this program, a time function was placed within a loop, and for each loop, the time value was stored in a vector. The vector's values were then analyzed and it was found that the minimum time value was $26\mu s$, the average value was $28\mu s$ and the maximum was $3077\mu s$; the total time was 0.29 seconds. The extremely large spike of $3077\mu s$ which appears in the data, most likely was caused by a timer interrupt (usually every 10ms). Excluding such an extreme case, the maximum time was about $400\mu s$.

The send time for a unicast send was evaluated. In testing, eight unicast packets were sent to the discard port of 8 machines in the LAN. *Tcpdump*, a network analysis tool, was used to watch for messages appearing on the network from the sender. The time differences between the first and second, second and third, etc. packets as reported by *tcpdump* were examined. The data showed that approximately every 2.5 milliseconds a new packet appeared on the network. This gives us an idea of how much time is involved in processing the packet. However, this time value does include the amount of time a host must wait before sending the next packet onto the line. One must also consider that the *tcpdump* timestamps may not be totally accurate.

As the ttl increased, the performance decreased substantially. The local area network IP multicast studies achieved very good results as compared to the wide area network IP multicast studies. As seen in section 3.3.3, the reliable IP multicast outperformed the reliable multiple unicast program. However, as the number of hops between hosts increased, the IP multicast performance deteriorated.

The model shows that speed-up is directly related to the process time for a send and the time for a packet to travel from the sender to the receivers. The process time can depend on packet size; as packet size increases, *rtt* increases. Also, the number of ACKs returning before all packets are sent as well as machine and network load can affect travel time. In running the performance tests at different times during the day, a slow down could be seen for busy times of day such as between 10am and 3pm EDT with the noon hour being the worst time of day for performance. Also, speed-up is directly related to the number of receivers. As the number of receivers increases, the round-trip time for the multiple unicast sends increases due to to the increase in the *multicast* packets and ACKs. The model and the *echo* tests suggest that a reliable IP multicast should provide better performance than a unicast-based multicast.

Chapter 4

Implementation of reliable IP multicast in PVM.

4.1 The Parallel Virtual Machine

The desire to solve scientific problems quickly has led to the development of *parallel computing*[33]. Many parallel computers use multiple processors connected by very short communication links all within one frame. Parallel computers allow parts of a problem (tasks) which can be executed independently of each other to be distributed over many processors. *Message-passing* routines are used for communication among the processors. In general, parallel computers are expensive. Due to the demand for a less expensive parallel computer and the development of efficient, inexpensive, high speed computers (e.g., workstations) and networks, *distributed operating* systems were developed to run on LANs. Within a *distributed operating* system hosts use *message-passing* routines to communicate. A group of computers connected by a *distributed operating* system work together to form a *multiprocessor*. In addition to workstations, a *distributed operating* system may be used to connect mainframes and/or multiprocessors.

The Parallel Virtual Machine (PVM) is a *message-passing* system which works like a *distributed operating* system, but usually it runs on top of the existing operating

systems (e.g. Unix)[5]. PVM does not provide a file system or memory manager. Since PVM *joins* physically separate and architecturally different machines over a LAN or a WAN, it is called a *virtual machine*. The hosts in the PVM do not have to be in the same LAN, but there does have to be some type of internetwork between them in order to pass messages.

The PVM library (*libpvm*) allows user applications (tasks) written in C or FORTRAN to communicate with the local PVM daemon (*pvmd*) or remote *pvmds* and remote tasks easily. Most communication between tasks is relayed via *pvmd*-to-*pvmd* communication. The *libpvm* provides an interface between the task and the *pvmd*. In addition to routing and controlling messages, the *pvmd* provides authentication, process control and fault detection[5]. The first *pvmd* which is started manually by the user is the master. The master can add, delete, or reconfigure slaves; other than these jobs, the slaves function like the master.

PVM *version 3.2.6* provides a multicasting routine, **pvm_mcast()**. This routine forwards the receivers' task identifiers, TIDs, to the local *pvmd* and then receives the group identification, GID. The function then forwards the multicast message with the GID to the local *pvmd*. This *pvmd* first sends the TIDs and the GID to the appropriate *pvmd*, and then sends the message via unicast messages those *pvmds* (see section 4.2.2 for further details). Also, PVM provides a special library, *libgpvm*, for group communication. This allows the user to identify groups by a number from 0 to $(p-1)$ where p is the number of tasks. To send to a group, the user calls **pvm_bcast()** using the group identification, GID. The **pvm_bcast()** relays the message to the local *pvmd*. This *pvmd* in turn processes the incoming request in the same manner it would a **pvm_mcast()** request.

Several mechanisms for delivering multi-daemon-destination messages have been considered by the PVM development group. Broadcast has been examined as perhaps a more efficient method in sending a multi-destination PVM message than a *one-to-all*

unicast method. The results of the broadcast study were very good especially for larger message and receiver set sizes. [8]. However, with a broadcast, all machines on the LAN even those not using PVM receive the packet. Broadcast is limited to hosts on a LAN and does not work over a WAN. Some PVM *version 2's* use a spanning-tree algorithm much like that used by the hypercube in order to perform multicasting. However, later versions (3.0-3.3) returned to the *one-to-all* fanout approach to perform more robustly. IP multicast permits a single message to reach a subset of hosts on both a LAN or a WAN. If the host is not a member of the multicast group, it ignores the message. So, IP multicast should be a suitable delivery mechanism for PVM.

To further evaluate the RMP, IP multicast was implemented reliably in PVM based on the RMP and was compared to the unicast-based function **pvm_mcast()** which is currently used for multicasting in PVM. The objective was to understand the issues involved in using the RMP in a real application and to measure and analyze any performance improvements in reliable IP multicast.

4.2 Multicast in PVM

4.2.1 PVM message-passing

At the application level, sending a PVM version 3.2.6 message consists of three steps. The task must first initialize a send buffer by a call to **pvm_initsend()** or **pvm_mkbuf()**. Then, the task must *pack* the message into the buffer using one or more PVM packing routines (i.e., **pvm_pk*()**). Finally, the buffered message is sent to another task by either calling the **pvm_send()** routine for unicasting or the **pvm_mcast()** routine for multicasting.

Receiving a PVM message consists of at least two steps. Optionally, one may use **pvm_bufinfo()** to receive information about a particular message. To actually receive the message, either **pvm_recv()** or **pvm_nrecv()** (a non-blocking receive) can be called. Then, the received message must be unpacked by using **pvm_upk*()**.

Both the **pvm_send()** and **pvm_mcast()** routines communicate with remote *pvmds* and tasks by first sending the message to the local *pvmd* via Unix domain datagram socket (see Figure 4.1). The message is then forwarded to the appropriate receiving *pvmd*(s) via a UDP socket. The message is then sent to the remote *pvmd* and finally to the appropriate task(s) on the same host. The sending *pvmd* retransmits the message until it receives an ACK or until the number of retries reaches the maximum. Upon receipt of the ACK the message is removed from the output queue. If the message is not ACKed by the last re-try, the message is dequeued, and the remote *pvmd* is assumed to be *dead*. The sending *pvmd* informs all other *pvmds* of the problem, and the downed *pvmd* is deleted from the PVM host tables.

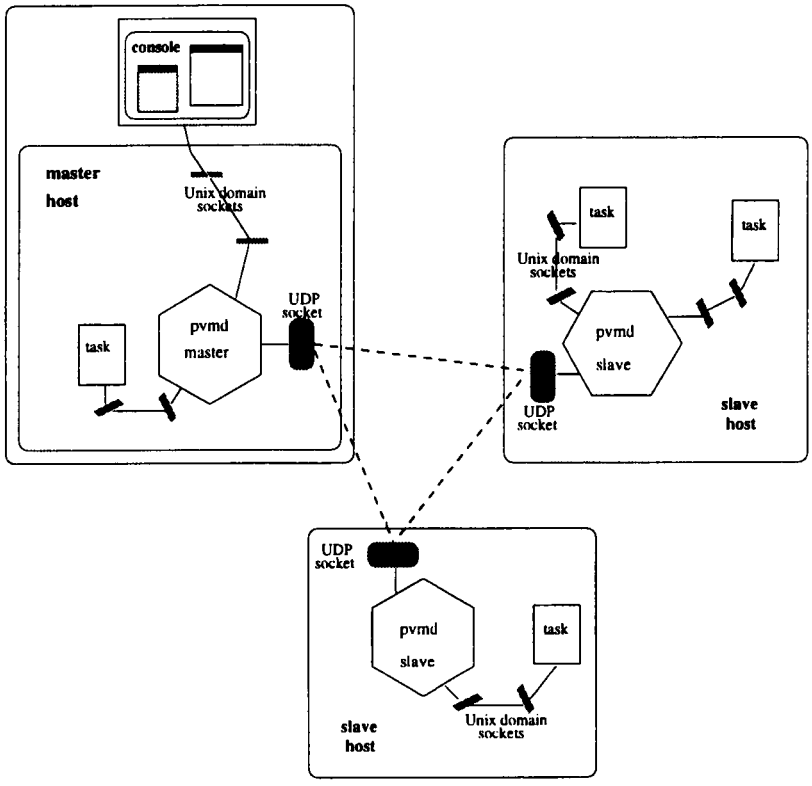

Figure 4.1: A Partial Anatomy of PVM.

The receiving *pvmd* replaces the ACK number with the sequence number and returns an ACK for the packet to the sending *pvmd*. The receiving *pvmd* will then forward any messages destined for tasks on its host. A task may receive a message by calling a receive routine and then *unpacking* each of the packed items. The message may be received from a specific source with a specific message tag, or from any source with a specific message tag, or from a specific source with any message tag.

4.2.2 PVM's pvm_mcast() protocol

The function pvm_mcast(), a part of the *libpvm*, is used by a task for multicasting a message. PVM version 3's daemon uses multi-unicasts (a *one-to-n* fanout multicasting algorithm) to reliably send a task-initiated message to multiple destinations. Some earlier versions use a spanning-tree algorithm. In this section, the following are discussed forming a multicast address, initiating a multicast message, and sending a multicast message in PVM. The decision for choosing the *one-to-n* fanout algorithm is also discussed.

PVM multicast address

Within the PVM system, a *task identifier* (TID) is used to address tasks, groups of tasks, and *pvmds* (see Figure 4.2 from [6]). With its *G* bit set to one, the TID repre-

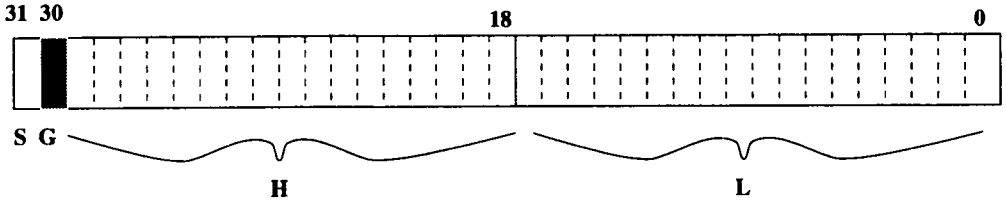

Figure 4.2: Representation of a Group Identifier (GID).

sents a group address or GID. The GID is formed by the *pvmd*. A new GID is formed for each multicast message; any discarded GIDs may eventually be reused. The multicast descriptor, **struct mca**, stores information about all active multicasts. As with all TIDs,

the *H* field is set to the value of the host index for the local host. The *L* field is set to the value of a counter which is incremented for each multicast. The main difference between a non-multicast task identifier and a GID is the setting of the G bit.

Initiating multicast

The multicast message is initiated by a request with a code of TM_MCA sent from the task via *libpvm*'s pvm_mcast() to the local *pvmd*. The request contains a list of task identifiers, the multicast group, for all tasks (*recipient-tasks*) which are to receive the multicast message. Thus, PVM uses a *closed group* scheme. This list can change each time a new multicast message is sent making the groups dynamic. However, once the TM_MCA message is formed, no other task may join the group. The message is passed to the *pvmd*'s tm_mca() function which forms the GID and a new **mca**. Also, the list is sorted according to the host index and checked for duplicate or erroneous TIDs. The information is stored in the **mca**. The *pvmd* replies with the GID to the requesting task, and then sends a message with a code of DM_MCA to remote *pvmds* (*recipient-pvmds*) which have tasks in the group. The DM_MCA message contains the GID and the TIDs of tasks in the multicast group which are on the same host as the *recipient-pvmd*. The receiving *pvmd*'s dm_mca() function processes the information and each forms a local **mca** in order to know which tasks are to receive the multicast message (see Step I of Figure 4.3).

Sending multicast

Upon receipt of the GID, pvm_mcast() then sends the multicast message to the local *pvmd*, using the GID (see Step II of Figure 4.3). When the local *pvmd* receives the message from the sending task, the routing layer of the *pvmd* prepares a packet descriptor, **struct pkt**, one for each local *recipient-task* and one for each remote *recipient-pvmd*.

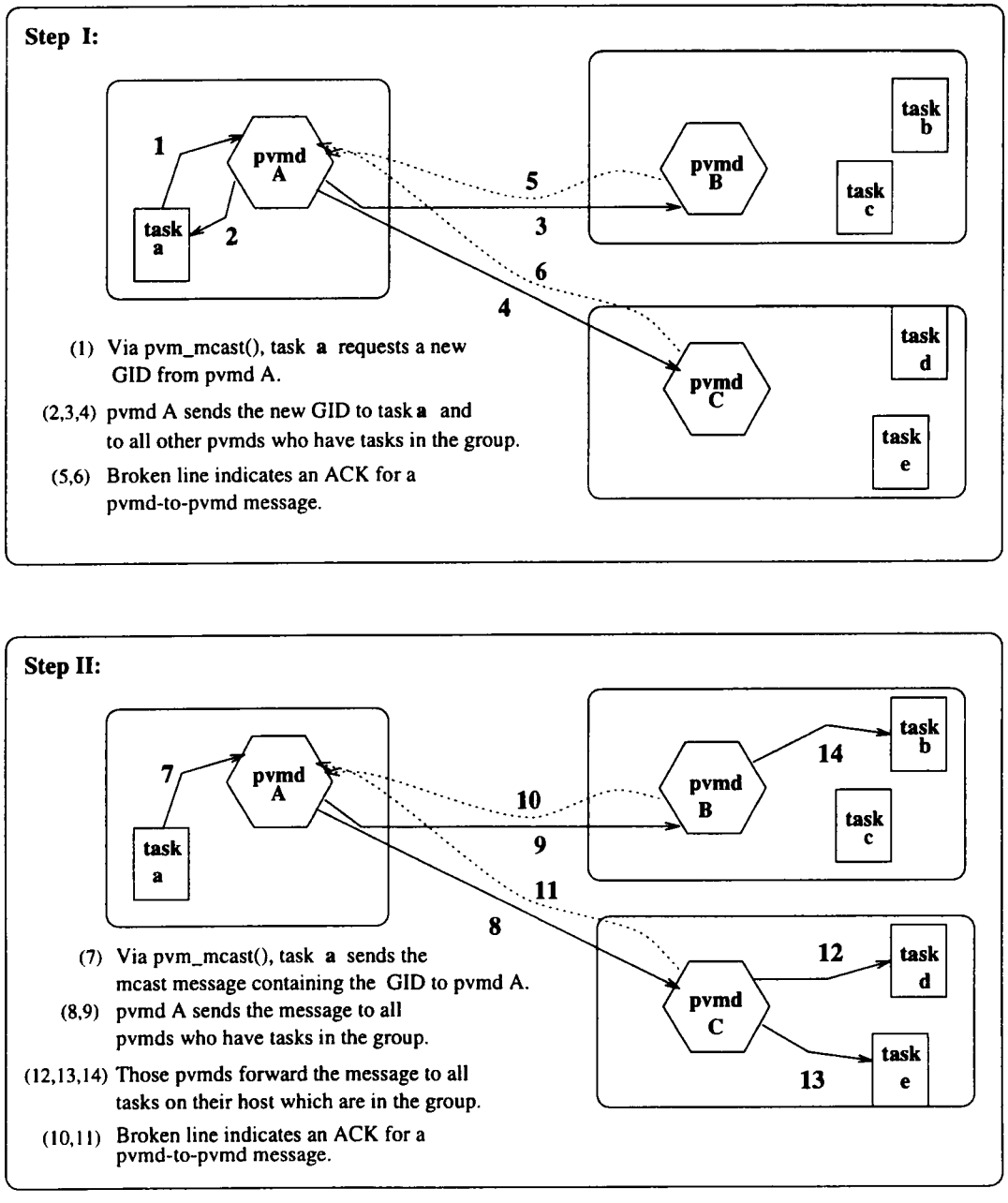

Figure 4.3: Diagram of PVM's pvm_mcast() function.

The **pkt** stores information, for use in *timing out*, sequencing, and acknowledging the message as well as a copy of the message. Each message is placed on a queue. PVM then unicasts each message to the proper *recipient.* When a multicast packet arrives at a *recipient-pvmd*, it is ACKed by the *pvmd*, and it is then duplicated and sent to each *recipient-task* on the same host (see Step II of Figure 4.3). When these *pvmds* detect the packet with the EOM bit set, they remove the **mca**. Because of PVM's reliability measures, the multicast address and data packets will arrive in proper order at each destination.

PVM Multicast routing

A spanning-tree algorithm was used in some *Version 2* packages to deliver multicast messages because it decreases the contention on the Ethernet for a particular host. The spanning-tree divides the sends among the hosts in the PVM. Therefore, while one host may be waiting to send, another host can send. To implement the spanning-tree algorithm, each *pvmd* is identified by a node number from 0 to $(n-1)$, where n represents the number of nodes. A bit-wise operation on the node number determines the children of each node as is done in the Hypercube. The ACKs were sent directly to the root of the tree.

There were several drawbacks to using a spanning-tree in PVM. For instance, numerous measures had be taken to resolve problems when a machine on the interior of the tree goes down. Extra work is involved in determining who did not receive the multicast message. One must determine whether the node failed before or after a multicast was sent. One must determine whether all, none, or part of the children received the message. Also, a recovery process involving eliminating the failed machine and eventually re-sending the message to those who lost it must be completed.

Due to its robustness, the *one-to-n* fanout was re-implemented in PVM version 3. This ensures that the failure of one host will not cause the loss of messages except for the ones to that host. Also, fault-tolerance is simplified.

4.2.3 The pvm_ipmcast() protocol

To further test the RMP described in Chapter 3, developed as part of this study was the function **pvm_ipmcast()** which provides reliable IP multicast to PVM. It was hoped that using IP multicast would decrease out-going traffic and contention, and improve PVM's performance for multicasting. Just like **pvm_mcast()**, **pvm_ipmcast()** is part of *libpvm*. As well, it also works with the *pvmd* to send a single message to multiple destinations. Only a slight difference between the two functions exists in the *libpvm*. Most of the changes were made to the *pvmd*. In the following section, implementation issues are discussed, such as making the IP multicast socket, initiating an IP multicast message, sending an IP multicast message, and acknowledging the message.

IP multicast socket

In PVM's **mksocs()** routine in **startup.c**, several sockets are made. The *netsock* socket is used for *pvmd*-to-*pvmd* communication. The *ppnetsock* socket is used for *pvmd*-to-*pvmd'* communication where *pvmd'* is a daemon process spawned by the master *pvmd* in order to start slave *pvmds*. The *loclsock* socket is used for task-to-*pvmd* communication. The *mnetsock* socket was added for *pvmd*-to-*pvmd* IP multicast communication. Now, the *pvmds* not only have to listen to the usual sockets, but also, must listen to the IP multicast socket.

The IP multicast *loop-back* option was set to zero which means the sender will not receive its own multicast messages. The sender must know what address and port number on which the receiver will be listening. In this study, the IP multicast address

and the port numbers were static. For testing purposes, this is good since one will always know what address and port number to watch while using network debugging tools. However, for general use, this will cause problems especially if two users on the same network choose the same values (see Chapter 5 for further discussion).

Initiating IP multicast

Pvm_mcast() was not removed from the *libpvm*. Most of the changes were made to the *pvmd*. The *pvmd* can distinguish between those packets to be sent via *one-to-n* fanout from those to be sent via IP multicast. As for the *libpvm*, pvm_ipmcast() is virtually the same as pvm_mcast(). However, instead of sending the TM_MCA code to the local *pvmd* to request a GID, pvm_ipmcast() sends the TM_IP_MCA code. This allows the *pvmd* to differentiate between the two types of multicast messages. The task identifiers contained in the request are sorted and checked as before. Also, the GID is returned to the initiating task and the DM_MCA message is sent to *recipient-pvmds* as before.

Sending IP multicast

In order to satisfy PVM's design criteria, IP multicast was implemented reliably. In this implementation, IP multicast was inserted in such a way that few changes were made to PVM's logic. Most of the changes were made to the PVM routines loclinpkt(), netoutput() and netinput() found in pvmd.c. Instead of duplicating the ACK process, the ACK protocol of PVM was changed slightly to be used for the IP multicast acknowledgments needed by the RMP. This kept the changes to PVM modest which was preferable.

In the loclinpkt() function in addition to the normal **pkt** descriptors for unicast copies (subsequently denoted by $\mathbf{pkt}_u$) a **pkt** (subsequently denoted by $\mathbf{pkt}_m$) was formed whose packet GID *H* field was set to the local host index. This $\mathbf{pkt}_m$ was used as the IP

multicast packet descriptor. The local *pvmd* controlled the removal of the IP multicast packet from the queue by setting the appropriate ACK information for this $\mathbf{pkt}_m$ when necessary.

While the other $\mathbf{pkt}_u$s were being formed a descriptor (**struct ip_info**) stored a pointer to the sequence number for each remote *pvmd*. As well, when the IP multicast $\mathbf{pkt}_m$ was formed, information (e.g., the GID, an IP multicast counter, and an ACK counter) was recorded in the **ip_info**. The $\mathbf{pkt}_u$s containing unicast copies were formed as they would be in **pvm_mcast()** with a couple of changes. In case more than one task was multicasting from a particular host, each of these $\mathbf{pkt}_u$s contained an identifying number, *pk_mca_num*. This allowed the **ip_info** to be updated correctly when a multicast packet was ACKed. As well, an IP multicast marker, *pk_ipmca* was set to one so that the **netinput()** function could determine if the ACK was for an IP multicast packet.

These $\mathbf{pkt}_u$s were not only used to store the ACK information, but were used as *back-up* $\mathbf{pkt}_u$s to the IP multicast $\mathbf{pkt}_m$ If the $\mathbf{pkt}_m$ packet was removed from the queue before a *back-up* $\mathbf{pkt}_u$ was ACKed, this *back-up* $\mathbf{pkt}_u$ packet was sent as it would be in **pvm_mcast()** via unicast to the intended receiver.

Presently due to the way in which the unicast *back-up* $\mathbf{pkt}_u$**s** were allowed to collect the ACK data, the appropriate sequence number had to be sent to the *recipient-pvmd* in order for the ACK to be formed properly. This information was sent in the multicast packet. In the **netoutput()** function of the *pvmd* when the IP multicast packet was encountered, the sequence numbers for the unicast packets were copied from **ip_info** to the tail of the packet header. The header size had to be increased for this, and this limited the number of receivers which could be used. Then the packet was sent using the IP multicast address and port numbers on the *mnetsock* socket (see Figure 4.4). If an ACK was received for a multicast packet, then a zero was placed in the **ip_info**'s

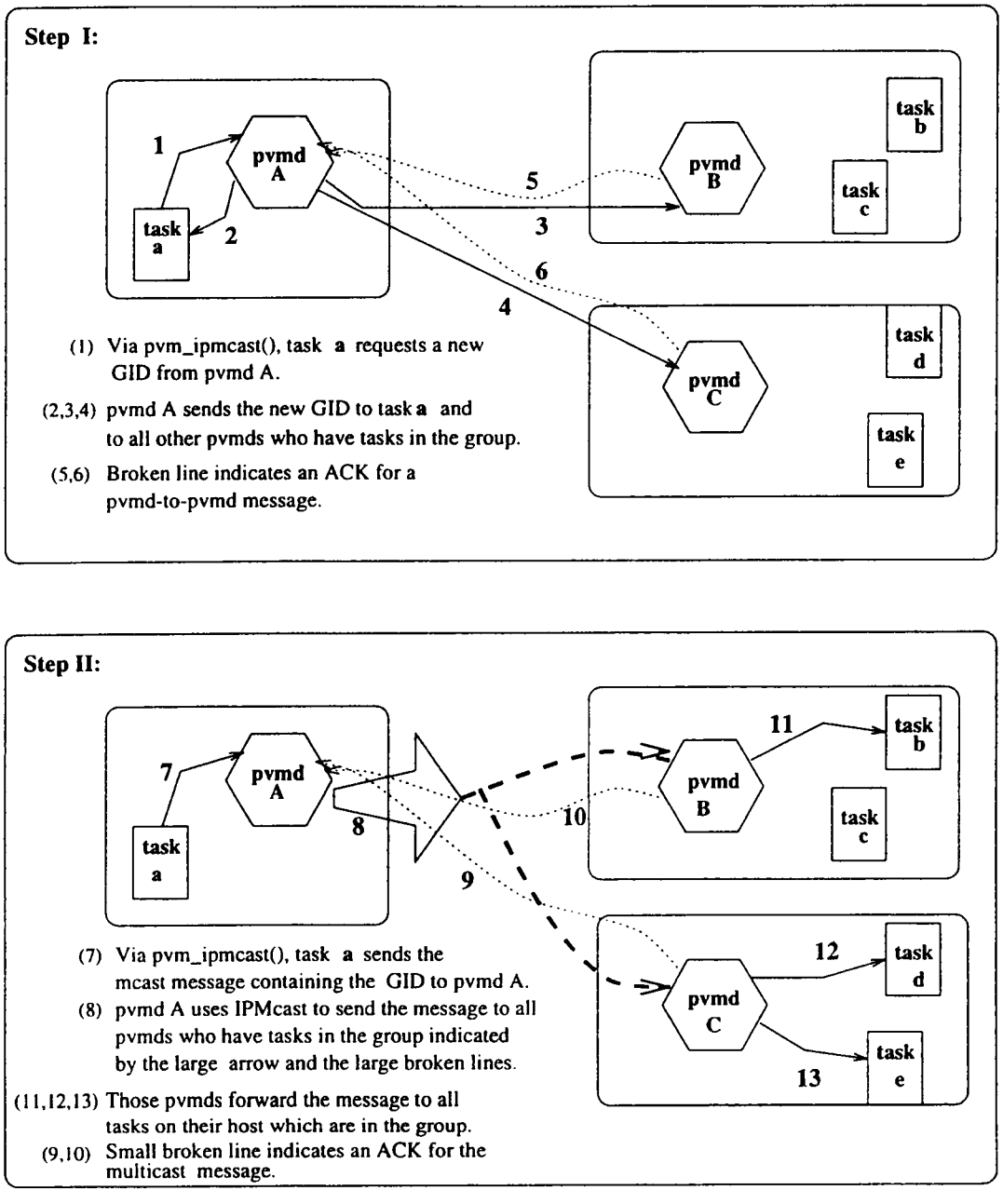

Figure 4.4: Diagram of the pvm_ipmcast() function.

sequence number slot for the ACK's sequence number. The unicast copies held in the $\mathbf{pkt}_u$s were only sent if the IP multicast packet *timed out.*

The *recipient-pvmds* listening for incoming packets would receive the IP multicast packet on their *mnetsock.* This was how the recipient was able to tell that the incoming packet was an IP multicast packet. When the **netinput()** function was called, the sequence numbers were first removed from the tail of the header of the packet. The number belonging to the particular *pvmd* was stored as the sequence number for the incoming packet. If the new sequence number was zero, the function returned to its calling function decreasing the amount of reverse traffic, else the packet was processed as it would be in the non-IP multicast routine.

If the IP multicast message was still in the queue when **netoutput()** was called again, the function first checked the **ip_info** to determine whether or not the multicast packet had been *successful*; then it checked to see if the number of multicast retries (M) was greater than the maximum number of retries (R). If both were false, the IP multicast packet was re-sent. However, if either was true, the descriptor's information for that packet was flushed, and the packet was *ACKed* by the local *pvmd.* Any remaining unACKed *back-up* packets were sent via unicast at this point.

Acknowledging IP multicast

Upon receipt of an ACK, the *pvmd* checked to see if the ACK was for an un-ACKed packet in the queue. If so, then the $\mathbf{pkt}_u$ which was to receive that ACKs information was checked to determine whether or not it was an IP multicast packet. If it was, then the **ip_info** ACK information for this particular IP multicast packet was incremented in order to keep track of the ACKs. This was how the IP multicast packet could detect how many ACKs had been received.

4.3 Comparative Performance of the RMP in PVM

The desire in reliably implementing IP multicast in PVM was to determine whether it would improve performance, particularly for larger message sizes and a larger number of receivers. However, the performance of the IP multicast, **pvm_ipmcast()**, function which was developed as part of this study was slightly worse on average for the LAN when compared to PVM's multiple unicast version, **pvm_mcast()**, and that for the WAN the IP multicast version was much worse. As the number of hops through the Internet increased, the performance of the reliable IP multicast in PVM deteriorated greatly.

The performance problems mentioned in section 3.3 as well as several factors mentioned below may have led to decreased performance of the RMP in PVM. For instance, the DM_MCA message which was mentioned in section 4.2.2 must be accepted by a receiver before the IP multicast packet can be accepted. Therefore, the IP multicast packets are hampered by these initial unicast control packets. Also, the IP multicast process incurs overhead in addition to the normal PVM packet processing. For instance, the **ip_info** IP multicast descriptor must be properly filled for each multicast message, and the ACKs for multicast packets must be counted so that necessary retransmissions occur properly. In addition, the process which was used for determining a retransmission method may have caused unnecessary waiting. As well, the receivers of a multicast packet must check the packet header for the appropriate sequence number to determine whether to continue processing the packet. Due to this, IP multicast in PVM does not conform to our model except that as the number of receivers increases time increases at a slower rate for IP multicast than for multiple unicasts.

LAN Performance of pvm_ipmcast()

On the LAN using 2, 4, 6 or 8 receivers each at UTK/CS and at ORNL and a 1024-byte message, the **pvm_mcast()** messages are faster than the **pvm_ipmcast()** messages

on average by 2 to 20 ms (see Figure 4.5). Also, comparing this figure to Figures 3.5 and 3.6 above, both IP multicast and multiple unicast methods incur overhead due to PVM. The additional IP multicast overhead and restrictions appear to decrease the IP multicast performance further.

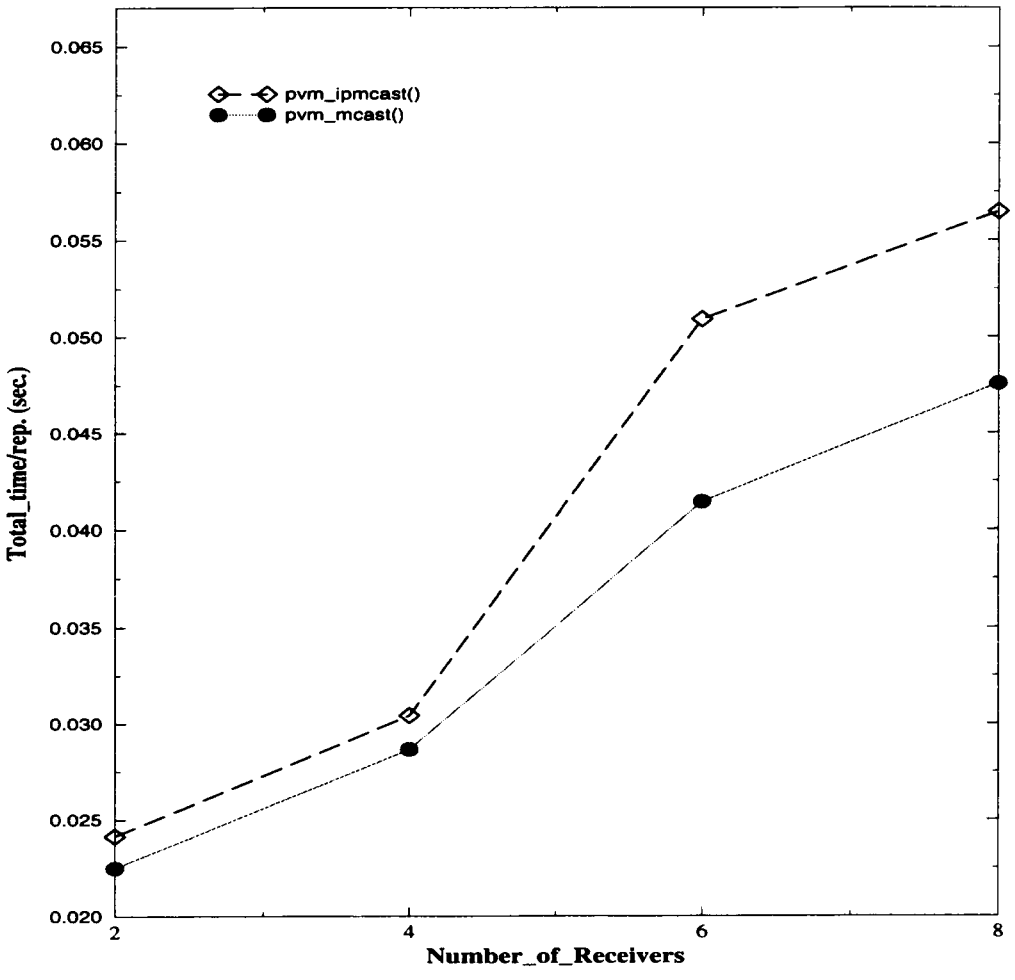

Figure 4.5: pvm_ipmcast() vs pvm_mcast() LAN Performance
Total Time/rep (sec.) vs Number of Receivers for 1024-byte packets.

WAN Performance of pvm_ipmcast()

On the *directly-connected* WAN using 1, 2, 3 or 4 receivers each at UTK/CS and at ORNL and a 1024-byte message, PVM's **pvm_mcast()** out-performed the **pvm_ipmcast()** by 5 to 10 ms for a *ttl* of 18 (see Figure 4.6) and by about 400 ms for a *ttl* of 76 (see Figure 4.7), except in the case of eight receivers where they perform about the same. For the IP multicast version the increase in time as the number of receivers increases is a slower rate than that of the non-IP multicast version. In these PVM WAN studies, IP multicast began much worse than multiple unicast, and as the number of clients increased the difference in their performance decreased. For the WAN, not only do the problems mentioned in the subsection above exist, but also the numerous hops through the MBONE and it's *mrouters* add to the latency for large *ttl*s. Due to this at the present time comparisons between IP multicast via the MBONE and multiple unicast in a WAN may not always be fair.

Since PVM is to be used as a parallel machine, speed is a major factor. The implementation of the reliable IP multicast protocol in PVM does not provide the speed-up which was desired since only the multiple unicasts were replaced with IP multicast and, not much else was altered. Some of the reasons for limited IP multicast performance are examined in the next chapter.

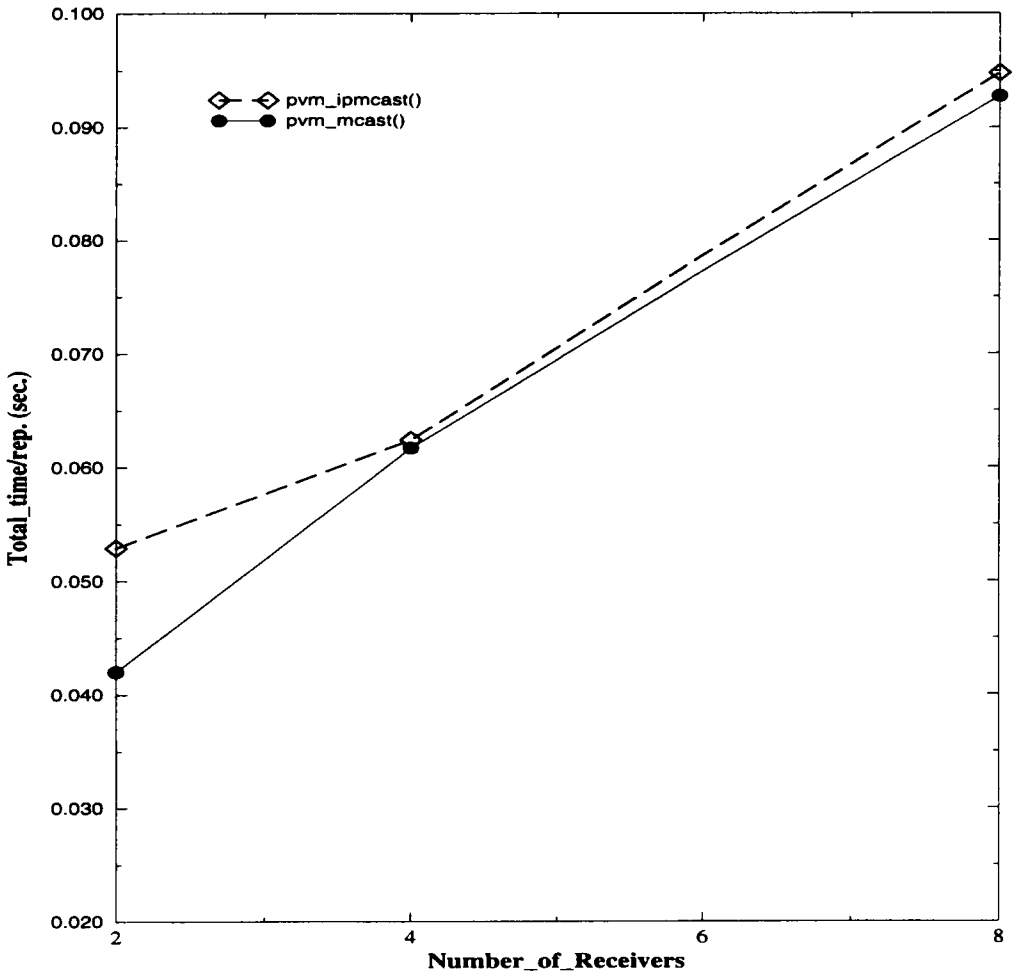

Figure 4.6: pvm_ipmcast() vs pvm_mcast() WAN Performance ($ttl = 18$)
Total Time/rep (sec.) vs Number of Receivers for 1024-byte packets.

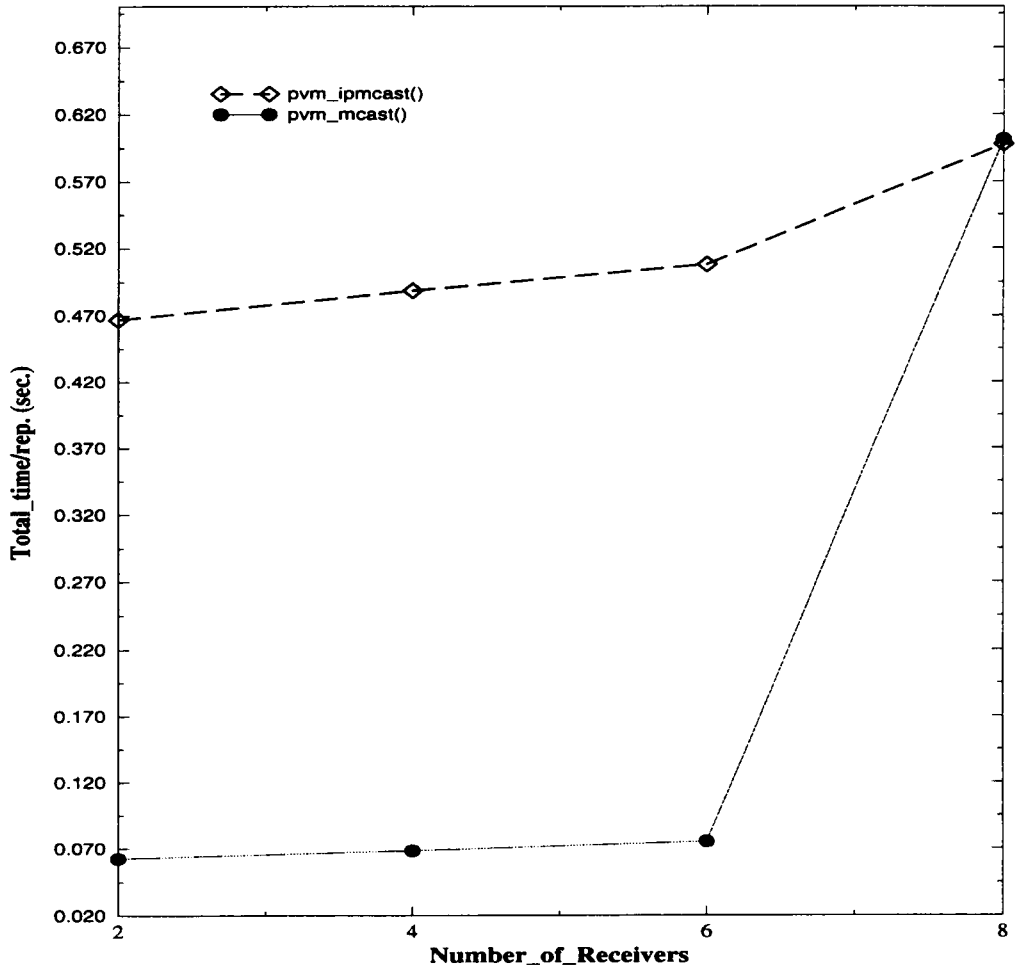

Figure 4.7: pvm_ipmcast() vs pvm_mcast() WAN Performance($ttl = 76$)
Total Time/rep (sec.) vs Number of Receivers for 1024-byte packets.

Chapter 5

Summary and Recommendations

5.1 Summary

Although many view the MBONE as a production model, there are still many problems to be solved and new uses to be found. In this research, several questions about reliable IP multicast were addressed. Those findings are reviewed below as well as limitations and future work in IP multicast.

5.1.1 Research Summary

The requirements for the use of IP multicast were discussed in section 3.2. The machines used in the test environment had to be IP multicast-capable. Since none of the machines in the test configuration came with kernel support in place, each machine had to have its kernel rebuilt using the IP multicast software. The use of the IP multicast socket options were also described for setting the *ttl* and for joining the multicast group. In addition, each subnet had to have an *mrouter* for forwarding packets. Multiple *mrouter*'s on the same LAN caused long periods of packet loss. Currently, the routes are implemented manually via the */etc/mrouted.conf* file.

The performance limitations of IP multicast were described in Chapter 3. Problems relating to the WAN used in the performance tests were discovered. Part of the

problem is believed to be caused by DVMRP multicast routers located on hosts at UTK/UTCC and UTK/CS which are connected by a bridge. Also, since IP multicast is fairly new, *bugs* still exist in some of the software. Another problem had to do with the number of hops through the MBONE which decreased performance. Currently, the speed-up of IP multicast over a WAN is limited by the number of hops. For each hop, there is an increased probability that a packet may be lost. An IP multicast packet traveling from one subnet to another may have to travel many times further than a unicast packet would between the same subnets. As the number of *mrouters* increases and the IP multicast path is shortened, this should not be as much of a problem.

The development of the reliable IP multicast protocol which was used in this study was discussed in section 3.2.3. Many reliable multicast protocols exist. However, most protocols have been developed for LANs where the kernels of the machines were altered or for specific multicast programs, such as Jacobson's *whiteboard* program. Therefore, the RMP in this study was written keeping in mind that it would be implemented in PVM. Several items which exist in PVM were not duplicated during the implementation of the RMP, such as special addressing for multicast packets, ordering of packets, and task protocol. Also, PVM's positive acknowledgment protocol was used as the basis of the ACKs for the RMP.

Finally, the performance of reliable IP multicast was examined. Reliable IP multicast worked well in the LAN used in this study. For larger packet and receiver set sizes, IP multicast was much better than multiple unicasts. However, as the *ttl* is increased on the WAN, IP multicast performance deteriorated. The performance of IP multicast in PVM was also disappointing. In order to implement the RMP in PVM and to obtain an improvement over what already exists, major changes to PVM's logic would need to be made, such as providing an IP multicast sequence number which would allow only the IP multicast packet's **pkt** to receive all the multicast ACK information. When

the IP multicast **pkt** is dequeued, only the unicast *back-up* **pkt**s which are needed for unicast retransmissions would need to be formed. In PVM, there also needs to be a process for discovering *mrouter* problems. In the case where an *mrouter* is down, all **pvm_ipmcast()** requests could automatically be processed as a **pvm_mcast()** request. In addition, when the number of IP multicasts to a virtually static group is large, providing a less dynamic multicast group would decrease the number of DM_MCA messages and could decrease any delays due to a receiver discarding an IP multicast packet which could arrive before the DM_MCA message. As the single *echo* tests showed, IP multicast improves over multiple unicasts as the number of receivers increase. It is possible PVM *pvm_ipmcast()* performance would have surpassed *pvm_mcast()* with more than eight receivers, but the resources to conduct such tests were not available.

5.1.2 Limitations

There are several drawbacks to using IP multicast. Many of the problems are related to management, privacy, and reliability. Another issue is that IP multicast is not widely available. Also, no easy manner exists in which IP multicast addresses and port numbers can be generated through the application. In addition, there are major steps yet to be made in the routing of IP multicast and in increasing its bandwidth. These issues are discussed further in the following sections.

Control

IP multicast is very new, and there are many problems which have not yet been solved. Since there is a lack of vendor support, most problems are being solved through a collective effort. Usually problems, concerns, questions, etc. are posted to mailing lists (e.g., mbone@ISI.EDU) where other MBONE users may help. Such lists are a source of information on conference scheduling or the latest tools. One problem is the lack of

control on the MBONE. Other than peer pressure, nothing automatically manages the operation of the MBONE. The use of low *ttl* values at the application level does help in keeping local multicast traffic from flooding the Internet. An additional problem is that IP multicast does not currently provide privacy; this is left for the user to provide. Plus, IP multicast sockets are UDP-based which means that the MBONE only provides a *best-effort* service. Presently, most MBONE tools are *best-effort.* The user is responsible for any necessary reliability, privacy and control measures as well as scheduling an event.

Availability

Currently, there are very few machines which are IP multicast capable. Mosedale[19] lists locations for obtaining software for reconfiguring some machines, information on connecting to the MBONE, and information about obtaining and using MBONE tools. According to Mosedale, the following come with kernel support in place: Solaris (2.1+), BSD/386 (1.1+), DEC OSF/1 (2.0+), and IRIX (4.0+). For some systems which do not support IP multicast, software for reconfiguring the kernel exists. All of the machines used in this research had to be reconfigured. Also, one IP multicast-capable machine per subnet had to run the *mrouted* program to route multicast packets since routers in general are not capable of routing IP multicast packets.

Address, Port, and *ttl* assignments

In this work, the address and port number were static. For testing purposes this is good since one will always know what address and port number to watch while using network debugging tools. For general use, this will cause problems especially if two users on the same network choose the same values.

A better way to choose these numbers is through random selection, but there will be clashes all too soon according to the *birthday effect.* To use McCanne and Jacobson's

sd[22] program which greatly reduces the occurrence of an address/port clash would be a better way of selecting these numbers. *Sd* randomly generates the numbers, and it avoids selecting those numbers which it hears in advertisements from other sites. However, *sd* does not currently have an application interface. To use *sd* with an application, one would need to enter the data as command line arguments, or the application would have to prompt the user for the information. According to Stephen Casner[1], the authors of *sd* plan to make a program interface. This would mean that a program would not have to generate its own address and port number or input the data manually. Also, although there may be no clashes between the host from which *sd* was run, there possibly could be clashes for other hosts in the multicast group if those hosts are not within the same subnet. In this case, one may wish to test the address/port numbers for all hosts in a particular multicast group when using a WAN.

Routing and Bandwidth

Currently, the MBONE's bandwidth is only about 500kbps. As IP multicast becomes more widely used, the need for more bandwidth will increase. Also, multicasting a packet across the Internet may involve many more hops between *mrouters* than a unicast packet would take to travel to the same destination. Researchers are already looking into making IP multicast routers more widely available which should decrease the number of hops and increase the available bandwidth.

If speed-up is the objective, reliable IP multicast over a WAN is not ready for production. Since IP multicast relies on specialized routers which are few in number compared to ordinary routers, an IP multicast packet incurs a large amount of overhead on a WAN. For instance, encapsulating a packet to be forwarded from one *mrouter* to another adds a small cost which unicast packets do not have. In addition, this cost is

[1]CASNER@ISI.EDU via email Wed., Sept. 14, 1994

increased for each hop through the MBONE. In many cases, the IP multicast packet may have to travel hundreds of miles through the MBONE while a unicast packet with the same source and destination may travel only tens of miles. As well, for each hop, the probability of a packet being lost may increase.

5.2 Future Work

The MBONE's use is increasing rapidly. Internet users are discovering the uses or new uses for the MBONE. However, for it to grow, problems need to be solved and limitations overcome. To provide reliable IP multicast to the entire MBONE community, several things need to happen. An RFC for a reliable IP multicast protocol is needed such that reliability can be easily provided to MBONE applications and in order to standardize the method. More reliable IP multicast software and tools are needed before the MBONE can move into main-stream use. A dynamic address and port server with which applications can interface is also needed. More IP multicast routers are needed in order to increase the bandwidth and decrease the number of hops through the MBONE making reliable IP multicast more competitive. Changes such as these would prepare IP multicast for *prime-time*.

BIBLIOGRAPHY

Bibliography

[1] S. E. Deering. Multicast Routing in Internetworks and Extended LANs. *Proc. ACM SIGCOMM '88 Symposium. ACM*, August 1988.

[2] A. S. Tanenbaum. *Computer Networks Second Edition*, chapter 1. Prentice Hall, 1989.

[3] S. Deering. Host Extensions for IP Multicasting. Request for Comments 1112, Stanford University, 1989.

[4] S. Casner. Frequently Asked Questions (FAQ) on the Multicast BackbONE (MBONE). May 1993. file://venera.isi.edu:mbone/faq.txt.

[5] R. Manchek. Design and Implementation of PVM Version 3. Master's thesis, University of Tennessee, Knoxville, 1994.

[6] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[7] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. 1994. file://netlib2.cs.utk.edu:pvm3/pvmug.

[8] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 1992.

[9] M. Kaashoek and A. S. Tanenbaum. Efficient Reliable Group Communication for Distributed Systems. *(Submitted for publication 1994).*

[10] D. Cheriton. VMTP: Versatile Message Transaction Protocol. Request for Comments 1045, Stanford University, 1988.

[11] PVM version 3.3 Frequently Asked Questions, rev. 5. 1994. file://netlib2.cs.utk.edu:pvm3/techfaq.

[12] C. Huang, E. P. Kasten, and P. K. McKinley. Design and Implementation of Multicast Operations for ATM-Based High Performance Computing. In *Proc. Supercomputing '94*, Washington D.C., November 1994.

[13] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Communications*, 39(4):447–458, April 1990.

[14] M. Ahamad and N. E. Belkeir. Using Multicast Communication for Dynamic Load Balancing in Local Area Networks. *by personal communication.*

[15] M. Satyanarayanan, E. H. Siegel. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Communications*, 39(3):328–348, March 1990.

[16] J. Mogul. Broadcasting Internet Datagrams. Request for Comments 919, Stanford University, 1984.

[17] K. Lebowitz and D. Mankins. Multi-network Broadcasting within the Internet. Request for Comments 947, BBN Laboratories, 1985.

[18] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. Request for Comments 1075, BBN STC, and Stanford University, 1988.

[19] D. Mosedale. Dan's Quick and Dirty Guide to Getting Connected to the MBONE. 1994. file://genome-ftp.stanford.edu/pub/mbone/mbone-connect.

[20] T. Pusateri. IP Multicast over Token-Ring Local Area Networks. Request for Comments 1469, Consultant, 1993.

[21] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT) An Architecture for Scalable Inter-Domain Multicast Routing. *SIGCOMM '93, San Francisco*, September 1993.

[22] M. Macedonia and D. Brutzman. MBONE, the Multicast BackbONE. *IEEE Computer*, January 1994. (draft article accepted for publication).

[23] S. Deering. IP Multicast Extensions for 4.3BSD UNIX and Related Systems MULTICAST 1.2 Release. 1989. file://gregorio.stanford.edu/vmtp-ip/ipmulticast.README.

[24] S. Pingali, D. Towsley, and J. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. *ACM, SIGMETRICS 94*, May 1994.

[25] C. Jepeway. The Design, Implementation, and Evaluation of RCALC. Master's thesis, University of Tennessee, Knoxville, 1994.

[26] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. *XTP: The Xpress Transfer Protocol*, chapter 8. Addison-Wesley Publishing Company, Inc, 1992.

[27] K. P. Birman and T. A. Joseph. On Communication Support for Fault Tolerant Process Groups. Request for Comments 992, Cornell University, 1986.

[28] S. Armstrong, A. Freier, and K. Marzullo. Multicast Transport Protocol. Request for Comments 1301, Xerox, Apple, and Cornell University, 1992.

[29] C. Bormann, J. Ott, H.-C. Gehrcke, T. Kershat, and N. Seifert. MTP-2: Towards Achieving the S.E.R.O Properties for Multicast Transport. In *Presented at the ICCCN '94*, San Francisco, September 1994.

[30] R. Braudes and S. Zabele. Requirements for Multicast Protocols. Request for Comments 1458, TASC, May 1993.

[31] S. Chang, D. Du, J. Hsieh, M. Lin, R. Tsang. Parallel Computing Over a Cluster of Workstations Interconnected via a Local ATM Network. Technical report, University of Minnesota Twin Cities, September 1994.

[32] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, June 1989.

[33] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, chapter 7. MIT Press and McGraw-Hill, Inc., 1993.

VITA

Kara Ann (Cantrell) Hall was born in Sevierville, Tennessee on December 10, 1964. She attended Pittman Center Elementary School and Gatlinburg-Pittman High School. She graduated high school in June 1983. The following fall she entered the University of Tennessee, Knoxville. In August 1989, following marriage and a change of schools, she received her Bachelor of Science degree from the college of Natural Sciences in Mathematics Teacher Education at the University of Houston. The next month she began teaching mathematics to 6-12 grade students in the Greenwood Independent School District in Midland, Texas. In August 1990, she began teaching mathematics at Roane County High School in Kingston, Tennessee. In August 1992, she returned to the University of Tennessee, Knoxville and in December 1994 received a Masters of Science degree in Computer Science. After graduation, she will be employed as a computer programmer at International Paper in Natchez, Mississippi.