ORNL/TM-13030

Computer Science and Mathematics Division

Mathematical Sciences Section

PVM AND IP MULTICAST

Thomas H. Dunigan and Kara A. Hall

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
thd@ornl.gov hall@cs.utk.edu

Date Published: December 1996

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Lockheed Martin Energy Research Corp.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-96OR22464

Contents

PVM AND IP MULTICAST

Thomas H. Dunigan and Kara A. Hall

Abstract

This report describes a 1994 demonstration implementation of PVM that uses IP multicast. PVM's one-to-many unicast implementation of its *pvm_mcast()* function is replaced with reliable IP multicast. Performance of PVM using IP multicast over local and wide-area networks is measured and compared with the original unicast implementation. Current limitations of IP multicast are noted.

1. Introduction

PVM, Parallel Virtual Machine [7], is a message-passing system that allows a collection of heterogeneous computers on a network to function as a virtual parallel computer. PVM supplies the functions to automatically start up tasks on the logical distributed-memory computer and allows the tasks to communicate and synchronize with each other. PVM is used by researchers to speed up computing applications by harnessing the power of workstations and supercomputers connected by a network. This report describes describes extensions made to PVM to support IP multicast as part of PVM's one-to-many communications protocol.

This report is a subset of a larger research effort by Hall [9] that looked at many of the research issues in reliable multicast protocols. Research issues studied in adding IP multicast support to PVM include:

- multicast reliability
- delivery and response semantics
- group structure
- reliable multicast methodologies
- performance implications

This report summarizes the choices made within these research issues. The following section describes multicast protocols. Section 3 describes the use of multicast in PVM, the extensions made to support IP multicast, and the effects on performance. The final section summarizes the current (1994) limitations of IP multicast in PVM and IP multicast on the Internet.

2. Multicast

Multicast communication is a mode of communication where one transmitter can be heard by many receivers. It is common in radio or satellite-based communication, but is even supported in digital communication systems such as local area networks (LANs). Though less common on wide area networks (WANs), recent multicast extensions to the Internet's protocol suite (IP) have spawned many research efforts to examine potential uses for wide-area multicast services.

2.1. Multicast and Broadcast on LANs and Multiprocessors

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[1].

The IEEE 802.2 LAN addressing standards provide multicast protocol as well as broadcast protocol for CSMA/CD, token-bus and token-ring LANs[1]. 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.

Once the sender's packet is placed on the network, all hosts in the same LAN can *see* the packet. Each station will check the packet's address against their own and then against that of each multicast group to 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 done in hardware.

Multicast and broadcast are used in a variety of applications. Satyanarayanan, et al.[17] have developed a file system, Coda, for large-scale distributed computing systems using workstations. When measuring bytes transferred, they found that the use of multicast reduced the network load. Ahamad and Belkeir[13] present a multicast-based process for load balancing in an Ethernet-type LAN using the multicast capabilities of the local network. Satyanarayanan and Siegel[16] describe a parallel remote procedure call *MultiRPC*. MultiRPC does not use true multicast; software filtering of broadcast packets simulates multicast.

2.2. Internet Protocol (IP) Multicast

Numerous works contributed to the development of IP multicast and the Internet MBONE.[1] In the 1984 RFC[2]919[8], 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[11], Lebowitz and Mankins presented a problem of multi-

[1]MBONE is the Multicast backBONE of the Internet.
[2]Request for Comments

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[6] *Distance Vector Routing Protocol* which provides a routing protocol for IP multicast that most MBONE routers use and the 1989 RFC1112[22] *Host Extensions for 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. IP multicast provides unreliable (best-effort) delivery using UDP datagrams. 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. Deering[22] 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. Multicast routers control the forwarding of IP multicast datagrams between any two subnets supporting IP multicast.

Many IP multicast routing algorithms exist [24]. Currently DVMRP [6] is used to route multicast datagrams through the MBONE. IP multicast routers (*mrouters*) are necessary for routing packets through non-IP multicast routers; this is termed tunneling. Unlike unicast datagrams which could pass through any router 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 another 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 built manually.

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/vic*), Visual Audio Tool (*vat*) and a shared drawing *whiteboard* (*wb*). *Nv*, *vat*, and *sd* use unreliable IP multicast; *wb* provides its own reliable IP multicast protocol. 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[5], "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."

2.3. Reliable Multicast

Deering[22] 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.[23] 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;

Hall [9] summarizes a number of applications that include multicast services, including VMTP [4], XTP [26], ISIS [12], MTP [20], MTP2 [2], RAMP [18], and Ameoba [14]. Of particular relevance, Huang, et al.[3] 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.[21] 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.

3. Reliable IP multicast in PVM.

The need to solve scientific problems quickly has led to the development of parallel computing[10]. Many parallel computers use multiple processors connected by very short communication links all within one frame. Parallel computers allow parts of a problem (tasks) that 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)[19]. 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[19]. The first *pvmd* that 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 back 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 to 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 [25]. However, with a broadcast, all machines on the LAN, even those not using PVM, receive the packet. Broadcast is also 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.

IP multicast was implemented reliably in PVM 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 reliable multicast protocol in a real application and to measure and analyze any performance improvements in reliable IP multicast.

3.1. Multicast in PVM

3.1.1. 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 3.1). The message is then sent 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.

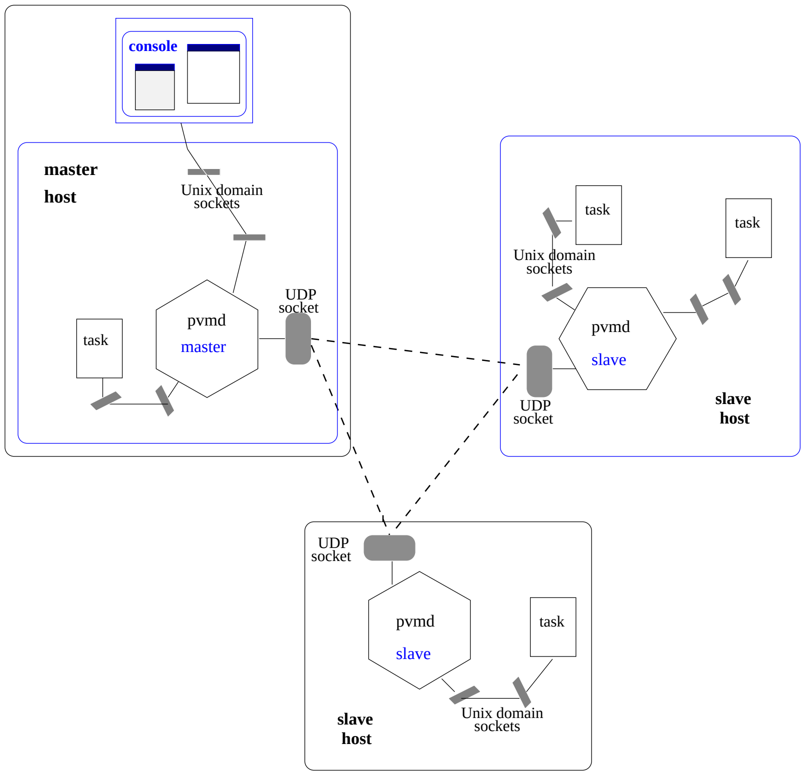

Figure 3.1: *A Partial Anatomy of PVM.*

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.

3.1.2. 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.

3.1.3. PVM multicast address

Within the PVM system, a *task identifier* (TID) is used to address tasks, groups of tasks, and *pvmds*. With its G bit set to one, the TID represents 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.

3.1.4. 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 3.2).

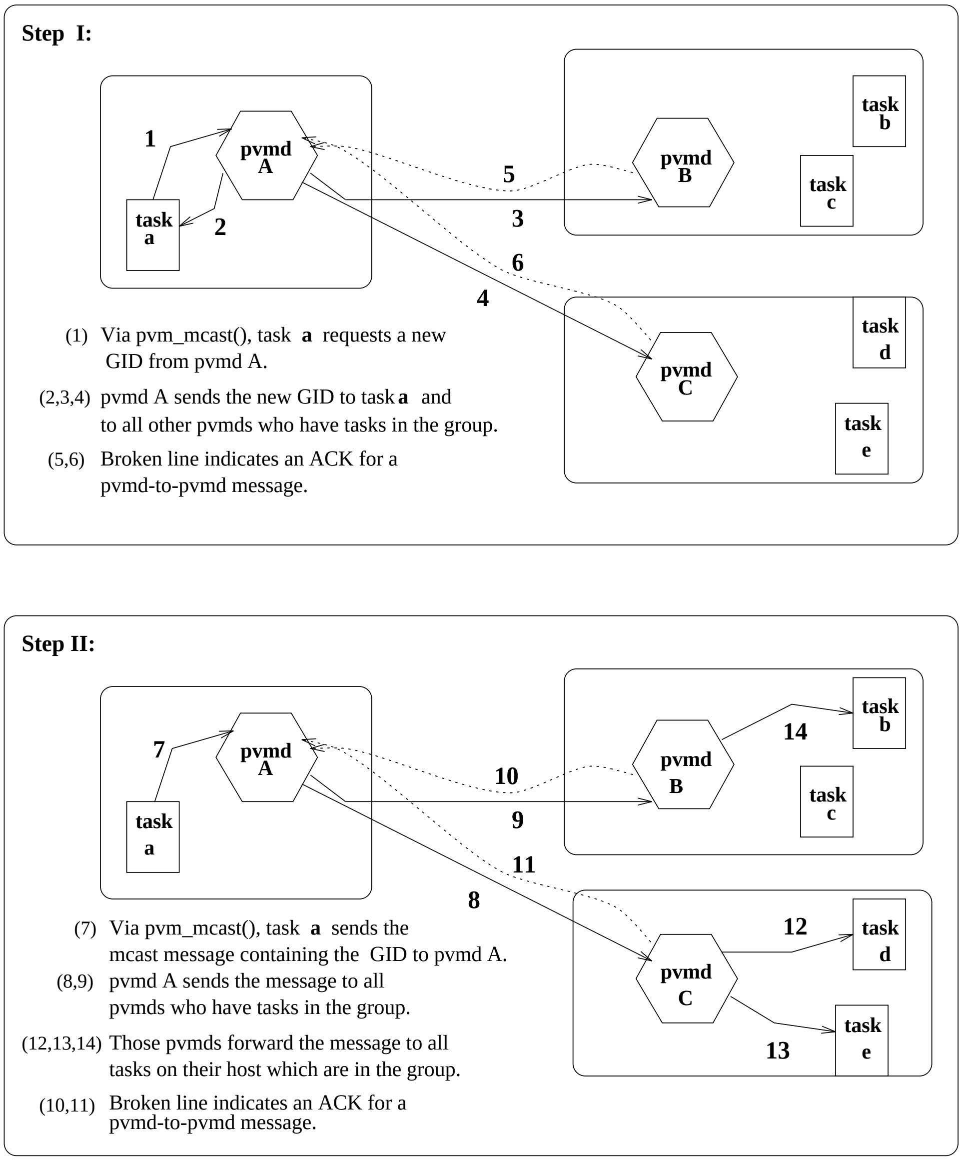

Figure 3.2: *Diagram of PVM's pvm_mcast() function.*

3.1.5. 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 3.2). 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*. 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 3.2). When these *pvmds* detect the packet with the end-of-message (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.

3.1.6. 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, extra logic is required to handle the case 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.

3.2. Adding IP multicast to PVM

As noted in section 2, efficiently implementing reliable multicast is a matter of open research. There are advantages and disadvantages to both NAK-based and ACK-based reliablity models. Since a positive acknowledgement system already exists in PVM's *pvmd*-to-*pvmd* communications, it was decided to use that system to implement reliable IP multicast in PVM.

3.2.1. The pvm_ipmcast() protocol

Reliable IP multicast was added to PVM with pvm_ipmcast(). 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*, and 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.

3.2.2. 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 was helpful, since one would 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.

3.2.3. 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.

3.2.4. 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 a reliable multicast protocol. 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 pkt_u), a **pkt** (subsequently denoted by pkt_m) was formed whose packet GID *H* field was set to the local host index. This 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 pkt_m when necessary.

While the other pkt_us 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 pkt_m was formed, information (e.g., the GID, an IP multicast counter, and an ACK counter) was recorded in the **ip_info**. The pkt_us 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 pkt_us 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 pkt_us were used not only to store the ACK information, but were also used as *back-up* pkt_us to the IP multicast pkt_m. If the pkt_m packet was removed from the queue before a *back-up* pkt_u was ACKed, this *back-up* 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* pkt_us 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 3.3). If an ACK was received for a multicast packet, then a zero was placed in the **ip_info**'s *sequence number slot* for the ACK's sequence number. The unicast copies held in the pkt_us 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.

3.2.5. Acknowledging IP multicast

Upon receipt of an ACK, the *pvmd* checked to see if the ACK was for an unACKed packet in the queue. If so, then the pkt_u that 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.

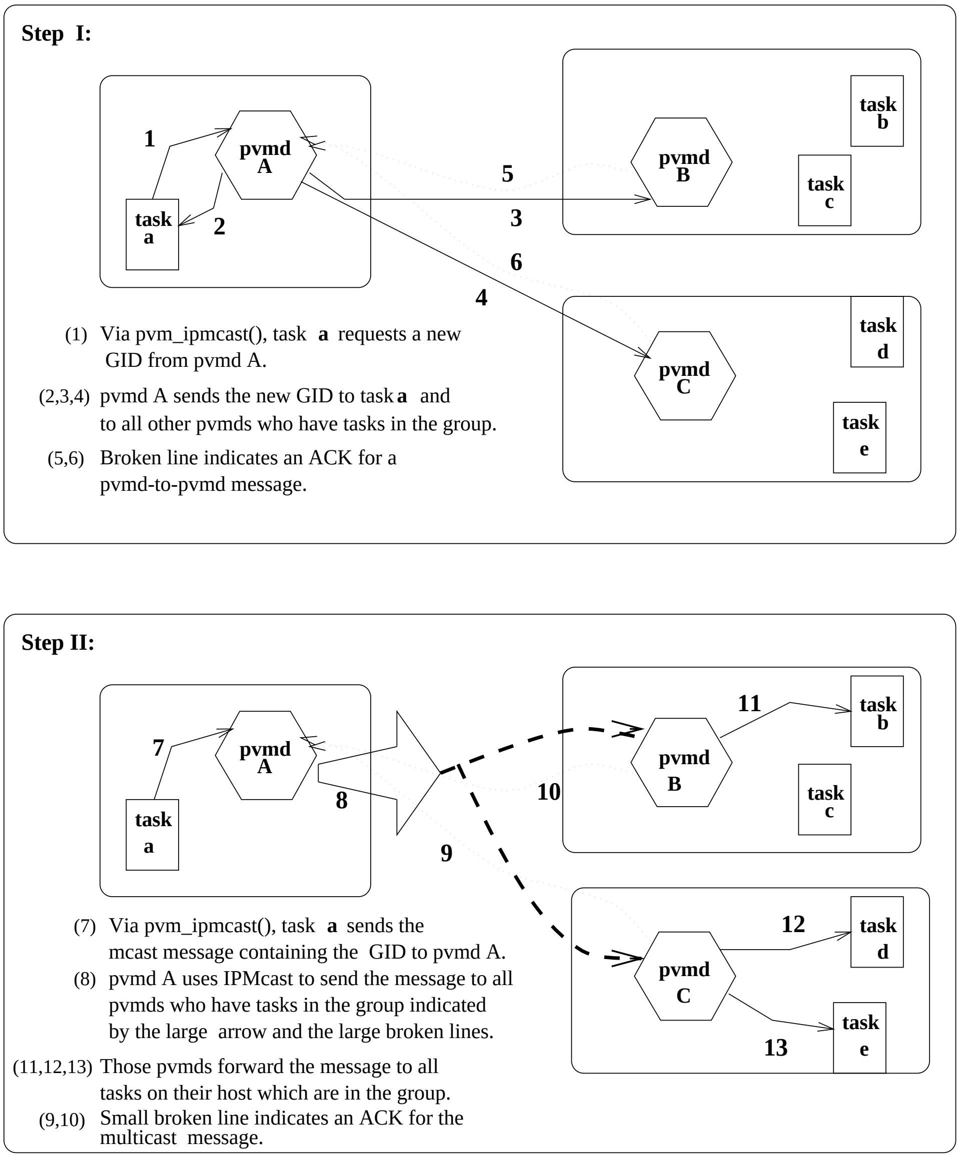

Figure 3.3: *Diagram of the pvm_ipmcast() function.*

3.3. PVM IP multicast Performance

The goal for implementing reliable IP multicast in PVM was to determine whether it would improve performance, particularly for larger message sizes and a larger number of receivers. The ACK-based protocol was tested without PVM on both local and wide-area networks [9], and a performance model was developed to predict speedup of multicast communication over unicast. The testing illustrated that performance was sensitive to packet size, network load, and multicast-routing hops (*ttl*) [9]. The tests of the reliable multicast protocol and the model suggested that reliable IP multicast should provide better communication performance than a unicast-based multicast. However, the performance of the IP multicast, pvm_ipmcast(), function that 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 the IP multicast version for the WAN 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 above and several factors mentioned below may have led to decreased performance of the reliable multicast protocol in PVM. For instance, the DM_MCA message 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 that 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.

3.3.1. 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 3.4). Also, comparing these results to the standalone reliable multicast protocol test [9], 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.

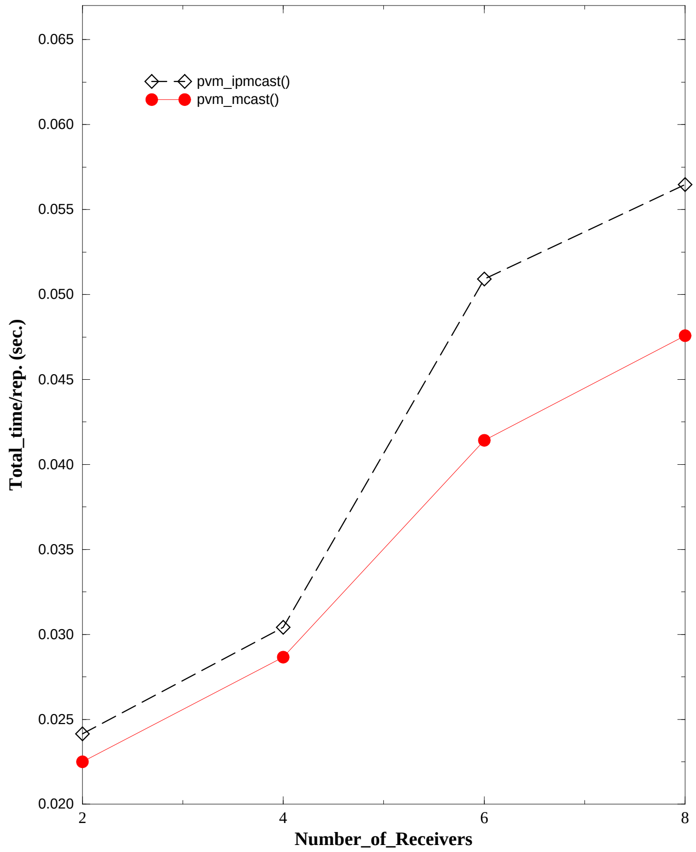

Figure 3.4: pvm_ipmcast() *vs* pvm_mcast() *LAN Performance*
Total Time/rep (sec.) vs Number of Receivers for 1024-byte packets.

3.3.2. 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() outperformed the pvm_ipmcast() by 5 to 10 ms for a *ttl* of 18 (see Figure 3.5) and by about 400 ms for a *ttl* of 76 (see Figure 3.6), 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, but 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 *ttls*. Due to this, comparisons between IP multicast via the MBONE and multiple unicast in a WAN may not always be fair at the present time.

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 because 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 section.

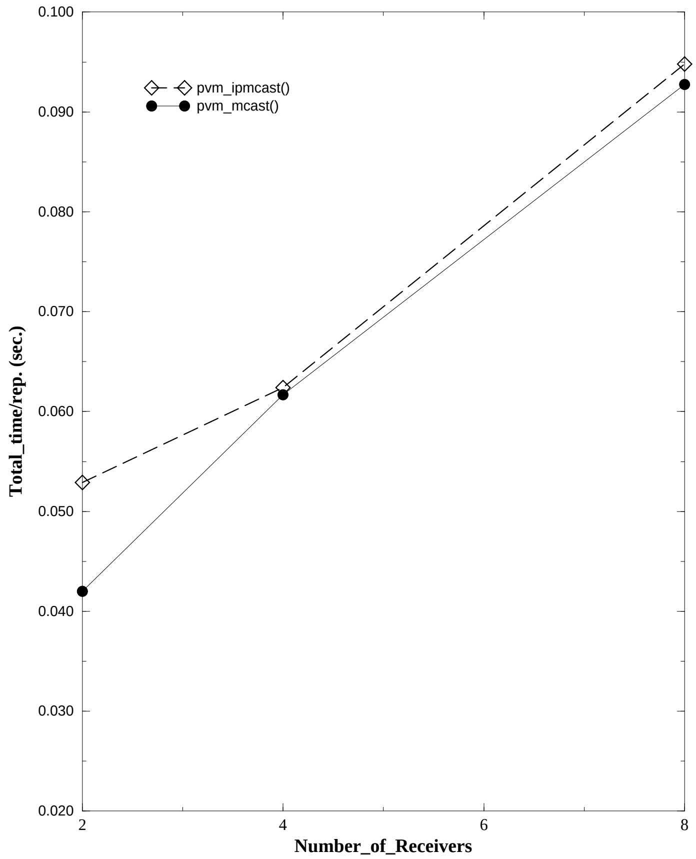

Figure 3.5: pvm_ipmcast() *vs* pvm_mcast() *WAN Performance (ttl = 18)*
Total Time/rep (sec.) vs Number of Receivers for 1024-byte packets.

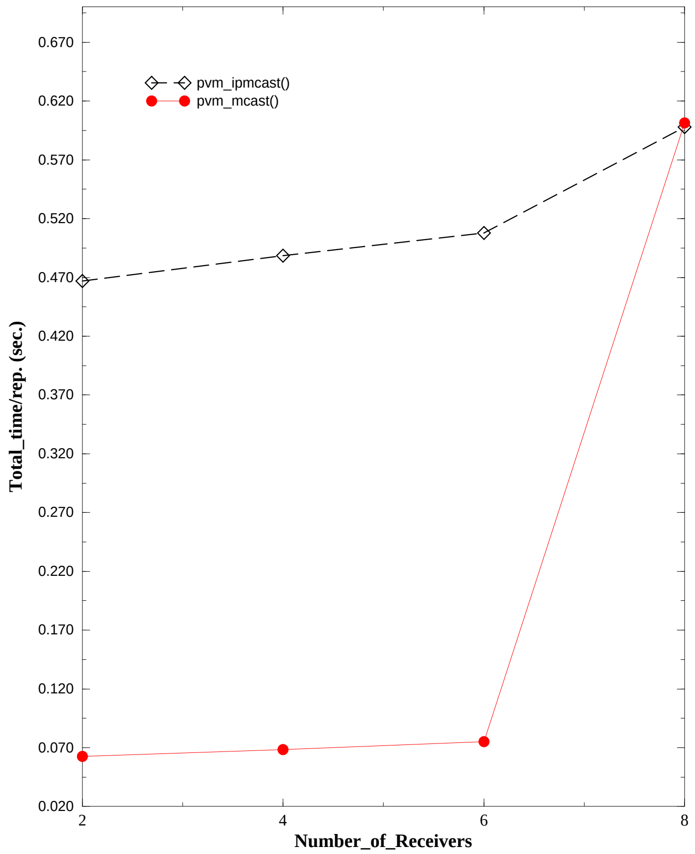

Figure 3.6: pvm_ipmcast() *vs* pvm_mcast() *WAN Performance(ttl = 76)*
Total Time/rep (sec.) vs Number of Receivers for 1024-byte packets.

4. 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.

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 described for setting the maximum hop count *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.

Problems relating to the WAN used in the performance tests were discovered during these experiments. Part of the problem is believed to be caused by DVMRP multicast routers located on hosts at UTK/UTCC and UTK/CS that 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 IP multicast performance 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.

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 reliable multicast protocol in this study was written keeping in mind that it would be implemented in PVM. Several items that exist in PVM were not duplicated during the implementation of the reliable multicast protocol, 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 reliable multicast protocol.

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 number of multicast-router hops increased on the WAN, IP multicast performance deteriorated. The performance of IP multicast in PVM was also disappointing. In order to implement the reliable multicast protocol 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 that 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 that 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 pvm_mcast() requests. 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 [9], 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.

4.1. 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.

4.1.1. Control

IP multicast is very new, and there are many problems that 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.

4.1.2. Availability

Currently, there are very few machines that are IP multicast capable. Mosedale[5] 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 that 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.

4.1.3. Address and Port assignments

In this work, the address and port number were static. For testing purposes this was helpful because one would 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*[15] 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 that 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[3], 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

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

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.

4.1.4. Bandwidth and Routing

Currently, the MBONE's bandwidth is only about 500kbps. As IP multicast and the multimedia applications that use it becomes more widely used, the need for more bandwidth will increase. Smarter multicast routing algorithms can prune back multicast traffic associated with inactive sessions and thus make better use of existing bandwidth. 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 that 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 that unicast packets do not have. In addition, this cost is 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.

4.2. 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 commercial 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 production use.

For the current status and availability of IP multicast in PVM, the reader is invited to visit *http://www.epm.ornl.gov/pvm.*

5. References

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

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

[9] K. Hall. The Implementation and Evaluation of Reliable IP Multicast. Master's thesis, University of Tennessee, Knoxville, 1994.

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

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

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

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

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

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

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

[17] 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.

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

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

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

[21] 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.

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

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

[24] 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.

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

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