THE MESSAGE BUS II AND THE RELIABLE MULTICAST PROTOCOL

BY

BRIAN DAVID WHETTEN

B.S., University of Illinois at Urbana-Champaign, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

Dedicated to Version 2 of the Reliable Multicast Protocol

ACKNOWLEDGMENTS

I have received enormous assistance and encouragement on this thesis, from people too numerous to name. I gratefully give special thanks to the following, however.

Thank you to Simon Kaplan, for advising me through this process. Your honesty and intelligence are evident to all who work with you. Thank you for your advice, your funding of this research, and the freedom you gave me to make my own decisions.

Thank you to Doug Bogia, Bill Tolone, Rocco Martin, Elsa Bignoli, Ted Phelps, Alan Carroll, and the rest of the Delta group, for your friendship, advice, encouragement, and shining examples. A special thanks to Doug and Bill for patiently answering all of the millions of questions I have had.

Thank you to my parents, family and friends, for your love, your constant encouragement, and your prodding. This could not have been done without you. A particular thanks to Armando Mendoza, Stuart Malles, Slim Simpson, Simeon Michaels, Crystal Hanscom, Kimberly Wragge, and Nikki Slocum, for helping me to balance work with a healthy dose of pleasure and laughter.

Thank you to Domenico Ferrari, Bruce Mah, and everyone in the Tenet group at Berkeley. You have helped me to really become part of this research field.

Thank you to Todd Montgomery and Jack Callahan at West Virginia University. Your support of version 2 of RMP has been fantastic, and the results are astounding. Todd, your programming expertise and speed is amazing, and I am glad to have you both as my co-worker and my friend. Thanks also to Charles Herring and Jeff Wallace at ARMY CERL in Illinois for their support of version 2.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF ILLUSTRATIONS

Figure Page

CHAPTER 1

INTRODUCTION

Group communication is becoming an increasingly important class of networking. Applications in this class--such as shared white boards, shared editors, and video conferencing--involve sending one message to multiple destinations. This type of communication is called *multicasting*. Recent developments such as the IP Multicasting[Deering89] standard for the Internet now allow efficient multicast datagrams to be sent over internetworks. These unreliable datagrams get processed at each router where they are copied only as necessary to get the packet to each of the hosts in the destination group. This allows multiple senders to receive a packet more efficiently than if the sender had to send a separate datagram to each of them. In the best case where all destinations are on the same LAN, one IP Multicast packet to all of them costs the same as a non-multicast packet to just one. This thesis describes the Message Bus II, a high level tool for group communication, and the Reliable Multicast Protocol, a protocol that provides reliable and efficient multicast communication on top of unreliable datagram services such as IP Multicasting or IP.

The Message Bus (MBus) is a high level communication tool that easily allows processes to communicate with each other without having to have explicit knowledge of the other processes' names, locations, or even existence. It is a client-server program, where one or more servers are run on different hosts, and they allow a wide range of clients to easily communicate with each other, including those written in C, C++, LISP, and CLOS. It uses a publisher-subscriber model of communication, where each process declares what *domains* it wishes to subscribe to, and all messages that are published to a domain are automatically delivered to all of the clients that are currently connected to a MBus and are currently subscribing to that domain. A domain is an arbitrary hierarchical text string, and can be used to represent topics of conversation, collaborative groups, computer services to be provided by other servers, or many other things.

The original Message Bus (MBusI) [Carroll93] is based on a single central server which routes messages between a number of clients connected to it with TCP-IP streams. It has demonstrated the usefulness and ease of use of the communication model, but it has very low scalability, allowing no more than 16 clients to communicate with it at one time. This motivated the design and implementation of the Message Bus II (MBusII). The MBusII uses the same user interface that the MBusI does, but it is based on a new reliable delivery system, the Reliable Multicast Protocol (RMP), which provides both a fully distributed multicast transport mechanism and a fully distributed group naming service. It allows the new MBusII to support up to hundreds of simultaneous clients across the Internet.

The MBusII achieves greater scalability over the MBusI by being fully distributed and by using a datagram protocol instead of a virtual circuit protocol. It is fully distributed in that multiple copies of the server can be run on different hosts, and communication can happen between any subset of them. Not having a central server reduces the importance of any one host, and therefore reduces the possibility of a single host being the bottleneck of communication. In addition to the efficiency bottleneck a single server causes, a typical UNIX process can only have up to 16 TCP/IP streams open at a time, which means that only 16 processes can be communicating to it at the same time. The MBusII does not have this limitation on scalability because it uses a datagram protocol for communication between servers instead of a virtual circuit protocol. This allows many MBusII servers to be in communication at the same time without dedicating a file descriptor to each of those connections.

RMP is a fully self-contained protocol that can be adapted to be used with a range of tools, or linked in to a C program and run directly. RMP is based on the token ring protocol first proposed by J. M. Chang and N. F. Maxemchuk in [ChMa83] and [ChMa84]. This protocol, which we name TRP, provides what can be thought of as a single N-way virtual circuit between a single group of processes connected by a broadcast medium. TRP is fully distributed, so that all processes play the same role in communication. The protocol provides atomic, reliable message delivery that is both totally and causally ordered, using causal ordering as defined in [Lamp78] and [BiJo87]. Finally, the protocol provides K-resilient fault tolerance, so that up to K processes can crash or partition away simultaneously without disrupting the message delivery between the rest of the processes. K is a user-configurable parameter that must be the same across all communicating processes. Higher values of K provide higher fault tolerance, but at the cost of higher latency of message delivery.

The Reliable Multicast Protocol (RMP) improves on the limited scalability of TRP by using multicast instead of broadcast, by allowing multiple simultaneous token rings, by providing a fully distributed group naming service, by improving the reformation algorithm for group membership changes that are not due to site crashes, and by implementing more efficient and robust flow and congestion control. The new reformation algorithm allows RMP to provide *virtual synchrony*, as defined by the ISIS project [BSS91]. This means that when new members join or leave a group these operations appear to be atomic, so that the sets of messages delivered before and after each membership change are consistent across all sites.

A common belief in the research community is that totally ordered reliable multicast protocols are inherently slow. This belief has come about in large part due to the experiences researchers have had with the early versions of ISIS, which for a long time was the only system of this type available. ISIS has since become much faster [BSS91, BiCl94], but the misconception remains. Experience with RMP belies this concept. RMP was tested on 8 SparcStation10's on a 10 Mb/sec (1250 KB/sec) Ethernet. In this environment, the throughput to a single destination is 842 KB/sec, or 67.4% of the network capacity. For group communication to any group of two or more destinations on a LAN, RMP exceeds not only the maximum throughput of TCP/IP, but any other possible non-multicast and non-broadcast algorithm. This is because both the packet latency and throughput of RMP stay roughly constant as the number of destinations increase, whereas the performance of other algorithms decreases linearly. For a group with 8 destinations, RMP has a 6.8 MB/sec aggregate throughput, which is 5.4 times the bandwidth of the supporting Ethernet. RMP pays a moderate latency penalty for its total ordering, but this is only a factor in short messages to a single destination. In groups of two or more, RMP latency is lower than most other protocols. Therefore RMP demonstrates that a fault tolerant, reliable, atomic, totally ordered multicast protocol can actually achieve much better performance in group communication than systems that don't provide these features. In addition, if the application does not need total ordering and atomicity of messages, RMP can provide roughly the same latency as other protocols by delivering packets immediately at the destination.

Unlike other systems such as ISIS and Transis that provide the same guarantees, RMP achieves this with only a moderate latency penalty. A minimum length RPC to one destination in our testing environment takes around 1 ms, and increases with the number of destinations. RMP has a roughly constant per packet latency of 3 ms, regardless of the number of destinations. In larger networks where the link latency is more of a factor, we expect RMP to take the same amount of time (the round trip latency of a packet) to send an acknowledged packet to the destination as other approaches such as RPC. However, because of the atomicity and total ordering of messages, RMP will on average delay a packet at the destination for between 1.5 and 2 times as long as other methods before delivering it. The factor of the delay depends on the group size, and is a fundamental limit on the speed with which the given ordering guarantees can be provided.

Chapter 2 presents some background information needed to understand RMP and other reliable multicast protocols. Chapter 3 of this thesis describes the motivation for RMP and the MBusII. Chapter 4 discusses previous work and compares it to RMP. Chapter 5 defines the terms used in the descriptions of RMP and the MBusII, and describes the system assumptions that were made. Chapter 6 explains the new aspects of the MBusII that are needed for a programmer to use the MBusII. Chapter 7 describes the architecture of the whole system, and how all of the parts work together. Chapter 8 describes RMP. Chapter 9 details the system optimizations that were done to achieve the current level of performance, and Chapter 10 quantifies that performance over a 10 Mb/sec Ethernet. Finally, Chapter 11 draws some conclusions about the project, and explains what future work has been done since the conclusion of the MBusII project in December, 1993.

CHAPTER 2

BACKGROUND

IP Multicasting

IP Multicast is a new Internet protocol standard that is being supported by most major vendors. It provides an efficient unreliable datagram service to multiple destinations. There are now ports for most architectures, and its usage and support is constantly increasing. One packet sent to an IP Multicast group gets sent to all of the hosts in the group. Instead of sending a separate packet to each host, IP Multicast routers maintain routing information about the group that allows a single packet to be sent to all hosts, getting copied at the routers when necessary. Ideally, the routers maintain a minimum spanning tree connecting all of the hosts in the group. This is not feasible in practice, but it has been shown that close approximations can efficiently be achieved [BaFrCr93]. In addition, IP Multicasting makes use of hardware multicasting on links, such as Ethernets, that support it. Hardware multicasting allows one packet to be sent to multiple destinations on a link for the same cost as a unicast to one of them. This combination of multicast routing and hardware multicast support makes IP Multicasting much more efficient for group communication than other alternatives such as IP. This efficiency is explored in more detail in the performance analysis section, where it is shown that for a group of N destinations, it can be up to N times more efficient in terms of network resources than a series of N unicasts. If there are an average of K sites collocated on the same LAN, IP Multicasting is typically K times more efficient than IP. For sites that are widely separated across the country or world, IP Multicast is at least S times more efficient, where S is equal to the average number of hosts per site instead of per LAN.

Ordering Guarantees

The second main criteria for evaluating multicast protocols is the level of delivery guarantees--both ordering guarantees and reliability guarantees--that they provide. As explained earlier, programming asynchronous distributed systems is very difficult. To help alleviate this, a multicast tool can provide source ordering, causal ordering, and/or total ordering guarantees on the delivery of messages. Usually, each of these ordering guarantees includes the lower ones, so that Source Ordering $\subseteq$ Causal Ordering $\subseteq$ Total Ordering.

Source ordering is similar to the ordering guarantees provided by virtual circuits. All of the packets that are generated by a source are given monotonically increasing sequence numbers, and the packets from that source are delivered at the destination in the order of the sequence numbers. Note that source ordering only provides guarantees with respect to a single source, and does not say anything about the ordering of packets from different sources. This is handled by causal and total ordering.

Causal ordering provides a partial ordering of messages sent to a group from multiple senders. Intuitively, causal ordering can be thought of as ordering messages so that for any two messages Ma and Mb that are both delivered at a site S, if Ma could have affected the contents of Mb (by being delivered to the process that produces Mb before Mb is generated) then Ma will be delivered at S before Mb is delivered at S.

The basic causal ordering relation *precedes* has two requirements:

1) For any two messages Ma and Mb, where Ma and Mb are both sent from the same source, if Ma is sent before Mb then Ma *precedes* Mb.

2) For any two messages Ma and Mb, if Ma is received at some site S before Mb is sent from site S, then Ma *precedes* Mb.

The precedes relation is transitive, so if Ma *precedes* Mb and Mb *precedes* Mc, Ma also *precedes* Mc. A system that provides causally ordered multicast guarantees that for any two messages Ma and Mb that are both delivered at a site S, if Ma *precedes* Mb, then Ma will be delivered at site S before Mb.

For a group of destinations that receive the same set of messages, total ordering imposes some total order on all messages, so that they are delivered in the same order at all destinations. More formally, for any two messages Ma and Mb, if Ma is delivered at some site S before Mb, then it will also be delivered before Mb at all other sites where they are both delivered. Total ordering does not guarantee that Ma was actually produced before Mb, for this is impossible to determine without globally synchronized clocks. Neither does total ordering imply causal ordering, although most systems (including RMP) that provide a total ordering on messages also order them causally as well.

Virtual Synchrony

The advantages of causal and total ordering in distributed systems programming has been discussed extensively in the literature [BSS91, DKM93]. A valuable extension to these ordering guarantees is the notion of *virtual synchrony* first introduced in [BiJo87]. Virtual synchrony extends the ordering of messages to include group membership changes as well. For any set of group membership changes, it imposes a total ordering on those changes so that they are observed in the same order at all sites. Furthermore, it guarantees that all sites see the same set of messages delivered before the group membership change. This is easy to provide with a causally, totally ordered delivery system, for it simply involves making each group membership change a message that is sent out and ordered along with all of the other messages. For systems that only provide causal ordering, it usually involves flushing the in-transit messages out of the system and disabling further messages, then performing the group membership change at all sites, and then re-enabling message delivery.

While both classes of virtual synchrony (causal and total) are very useful, the stronger class of totally ordered virtual synchrony is particularly valuable. We will call this guarantee *total virtual synchrony*. It allows the programmer to think of the system as a synchronous system while still getting the efficiency of asynchronous delivery. As an example, think of a shared editor application. In this application, multiple users edit their own local copy of the same document, and the consistency of this document must be maintained at all sites. In a system that does not provide total virtual synchrony the program must implement some type of locking scheme so only one user can edit a section of the file at a time. Otherwise the copies could become inconsistent when two users edit the same section at the same time, and these changes are processed in different orders at different sites. Using a system that provides total virtual synchrony, the program can guarantee that all sites see a consistent order of actions. It does this by sending out a message for each edit action, and only acting upon each edit action when the corresponding message for it is received. Because all sites see the same order of messages, they will perform the edit actions in the same order as well. This greatly simplifies the programmers task. In general, synchronous programming is much easier than asynchronous programming, and total virtual synchrony gives the advantages of synchronous programming with the speed of asynchronous communication.

Reliability and Guarantees

The other main class of guarantees that a multicast service can provide are reliability guarantees in the face of faults. In our system model, a fault can consist of a transient fault or a system fault. A transient fault is usually due to transmission errors or congestion overflow, and prevents a packet from being delivered to some or all of its destinations. A transient fault may also correspond to a process that has not received service for an extended period of time, but which has not crashed. A system fault is either a network partition or a process failure. In general, it is impossible to differentiate between one or more transient faults and a system fault in general asynchronous networks.

In a reliable communication protocol, a message is said to be *stable* when the sender of a packet knows that all destinations of that packet have received it. This is the point at which the packet no longer needs to be held for retransmission. The sooner each packet becomes stable, the less buffer space is needed at the sender.

Given our model of faults, the main reliability guarantees that a service can provide are unreliable delivery, reliable delivery, N-resilient fault tolerant delivery, safe delivery, and agreed delivery. Unreliable delivery is the standard delivery service provided by most basic multicasting services such as IP Multicasting and ST-II. This is also the same level of reliability that traditional unicast datagram protocols such as IP and IPX provide. A packet may be delivered 0, 1, or more times.

Reliable delivery is the level of reliability provided by virtual circuit protocols such as TCP/IP and SPX. These protocols guarantee that in the absence of system faults (such as site failures or network partitions) all packets will be delivered exactly once to each destination. In the case of a system fault that prevents delivery of a packet (or enough transient faults that it appears to be a system fault) the connection is terminated and an error reported.

Fault Tolerance

In many situations, it is desirable to not have to terminate a group connection if one or more sites fail or partition away. However, in the face of these system faults, it is more difficult to preserve ordering and reliability guarantees. There are four primary levels of fault-tolerant guarantees: atomic delivery within partitions, K-resilient atomic between partitions[ChMa84], agreed delivery between partitions, and safe delivery between partitions. The exact semantics of agreed and safe delivery are first defined in [DKM93].

Is has long been recognized that it is impossible in an arbitrary network to differentiate between a crashed site and an arbitrary delay in message delivery due to network congestion or other factors. In fact, it is usually desirable to assume the site is dead in order to prevent the other sites from being blocked indefinitely by it [ADKM91]. Because of this, fault tolerant applications have to use a time-out system to determine when a site is crashed. This may cause active sites to be removed from a group by the other sites, but this is a necessity and is rare if the time-outs are chosen appropriately.

The first level of fault tolerance is basic atomic delivery within partitions. This guarantee states that for totally ordered packets, if any member in a partition delivers a packet, all of the other members of that partition will deliver that packet if they were in the group membership view when the packet was sent and if they remain in the group for a sufficient period of time. This can be guaranteed by not allowing failed members to rejoin a group.

Basic atomicity does not provide any guarantees about delivery or ordering of packets between partitions. K-resilient atomicity between partitions is the first level of guarantee that addresses this. K is the minimum number of sites that must fail or partition away from a group over a short period of time in order to violate atomicity guarantees. This is particularly useful on a single LAN, where partitions are not possible, and site failures are the only the only type of system failures.

The next level of atomicity is called agreed ordering, or majority resilience. Agreed ordering guarantees that no matter how many partitions or failures occur, any members of a group that deliver any two messages will agree on the same ordering of the messages. This level guarantees total ordering across partitions, but not atomicity. One way of guaranteeing this is by not allowing minimum partitions to continue and by making sure the majority of the members of a group have a message before any member delivers it. Both of these tests require a possibly conservative calculation of how many members are in the group, which we call MaxN. MaxN is equal to the maximum number of members that are in any membership view for any packet which is not yet stable.

The final level of fault tolerance is safe delivery, also called total resilient delivery. This level requires that a packet be stable before it can be delivered. For example, in RMP, this occurs after the token has been passed once around the ring after a packet was received. At this point, the member knows that all other RMP destinations have received it. In most reliable multicast protocols, including RMP, delivery of a packet to the user happens separately from acknowledgment of the packet by the transport level. Because of this, a packet that is safely delivered to the communication level of all of the participating members may still not be delivered to all of the members of the group, because one or more sites could fail after acknowledgment of a packet but before delivering it. However, this is the highest level of atomicity that any system such as RMP can reasonably provide.

CHAPTER 3

MOTIVATION

The motivation for building a messaging tool such as the MBusI and the MBusII is that building distributed systems and multi-user applications is very difficult. These applications need reliable messages to multiple destinations, knowledge of which destinations to send to, ordering guarantees on the delivery of messages, efficient delivery of messages (high throughput) and scalability to hundreds of users. Standard networking facilities don't provide these features and so application writers have had to program these features themselves on top of lower level protocols.

Some of the multi-user applications with these requirements are shared tools, such as shared whiteboards and shared editors, CSCW applications, and groupware applications. These applications are predicted to become very widely used in the near future. Some distributed systems, such as some distributed data bases, are already very important, and have the same requirements.

An example of a multi-user application is a shared whiteboard. In this application, multiple users share the same computerized drawing area, and any change made by one user must be encapsulated into a message and sent to all of the other users as well. Once there, it is used to update the other users' screens. These displays must be kept consistent across the users. The first of the application's networking needs is to be able to send reliable messages to multiple destinations (the users who are looking at the same display.) If a message gets dropped, the different displays will get in inconsistent states and the users will see different things. The second need is that the list of users must be maintained, in order to know who these messages must go to. The third need is that some ordering on the messages should be imposed. Programming an application such as this with fully asynchronous (unordered) messages is very difficult, because inconsistencies can appear in the users' displays. For example, if one user erases an area of the screen at the same time that another user writes something on it, it is possible that one person will see the text (because his write operation happened after the erase) while another won't (because her write operation happened before the erase). The traditional method for solving this problem is to impose some type of locking scheme on the display so that only one user at a time can modify a given section. A better solution is to make every action of the user a message, and then impose a total order on the messages. Combined with reliable delivery, this causes each user to see the same series of messages, and hence the same series of actions. It is very difficult and time consuming to implement a system with these features on top of TCP/IP, RPC, IP, or any of the other common network protocols. However, now that these features have been implemented in the MBusI and MBusII, they can easily be used by other programs. By using a tool such as the MBusI and the MBusII, the communication aspect of an application becomes very simple, and programmers can concentrate on the other aspects of their program.

The motivation for building RMP is that by putting the reliable communication facilities into a separate protocol, multiple tools such as the MBusII can be built or ported easily on top of it. By encapsulating most of the communication functionality in the transport protocol, these tools can instead concentrate on providing useful interfaces to programmers in multiple languages. In addition to the MBusI, many similar tools have proven very useful, including Polylith[Purtilo85], and a recent bus from WVU [CaMo94]. Most of these tools have been built as a single server, which limits scalability and efficiency. By allowing them to be ported easily to RMP, they will all get the same benefits that the MBusII gets.

Performance Metrics

In comparing RMP and the MBusII to other systems, the most important performance metrics are packet latency, throughput, and buffer space. In addition, it is important for a protocol to be both efficient and scaleable as well. We define packet latency as the amount of time it takes to send a single packet of one byte to one or more destinations. Throughput is the amount of user data (not including packet headers and system overhead) that can be transferred per second. Latency is measured in milliseconds and throughput is measured in bytes per second. Buffer space is the amount of memory that has to be set aside to store packets that may not yet have been delivered at all sites. Memory has to be dedicated at both the sender (for sent but not yet acknowledged packets) and at the receiver (for received but not yet delivered packets).

In addition, a communication tool such as the MBus has to be scaleable and efficient as well. Scalability is a measurement of the number of clients that can communicate through the tool without having any of the clients suffer from seriously degraded performance. Efficiency is defined as the amount of networking resources required to send a given message. Like the other metrics, scalability and efficiency are affected by topology. It is usually easier to have more clients communicate efficiently together if they are on the same or adjoining LANs than if they are widely separated on a large internet.

Some applications send very few bytes but need them to get through very quickly, and for these packet latency is most important. For applications that send more than a few bytes at a time, high throughput is usually more important than low latency. This is because if packet latency is low but so is throughput, then the latency won't matter because the sender will quickly overrun the pipe and its packets will queue up, such as when you try to send a file over a modem. If throughput is very high then almost all packets will have the same latency. An example of this is satellite communication, where it takes 200 milliseconds (ms) to get a single packet through, but many megabytes (MB) of data are transferred each second, each with this 200 ms latency. 200 ms is excessive for most distributed applications, but as long as the minimum packet latency is reasonable, throughput is more important than packet latency in most applications, particularly of the type that are usually serviced by the MBusI and MBusII. The third metric, buffer space, is only important if it gets to be very large. Small amounts of buffer space are very cheap given the cost of memory today.

Scalability is important for most group applications and distributed systems, which would like to be able to support lots of groups of clients all communicating at the same time. Finally, efficiency is also important for this class of applications, for three reasons. First, using less network resources allows more applications to be run on a given network. Second, bandwidth is money. For now the Internet does not charge on a per-packet basis, but soon it will be commercialized and charges will probably be a function of bandwidth used. Third, the network is usually a bottleneck in communication between processes. More efficient use of the network decreases or removes this bottleneck, and allows higher throughput in communication.

History and Advantages of the MBusI

The original MBusI was built in 1990 and 1991 at the University of Illinois by Simon Kaplan, Alan Carrol, and Doug Bogia. It was designed to support a Computer Supported Cooperative Work (CSCW) application called Conversation Builder. It has been in active use with this and other applications since then, and has clearly demonstrated its utility and convenience. Three aspects of it in particular have proven to be very valuable: the publisher/subscriber model of communication, the user interface, and the LISP to and from C parsing and translation facilities. It was decided that all these features should be included in any successor to the MBusI.

The first of these valuable features is the publisher/subscriber model of communication. The MBusI has demonstrated that this model is more convenient to program with than other models where explicit knowledge of the destinations is required. In those models, some type of name service has to provided so that the sender knows who to send to. The publisher/subscriber model has this built in. For the class of applications where the sender's behavior will not change due to the number or type of destinations it is sending to, the senders can just publish the information and not worry about group membership. For applications that need to know who they are sending to, they can use the paradigm as a name service, and can query the membership of a domain at any time.

Therefore by providing this built in name service, this model makes it easier to program both classes of applications.

Second, the user interface and supporting tools have worked very well. The ASCII based interface has been very powerful, for it allows applications written in a wide variety of languages to communicate with the MBusI. The multiple tools that exist for connecting to the MBus in different situations has also proven useful. There is one that starts up a process and automatically connects a stream to the MBus, there is a C library that can be linked in to an application and provides both connection and parsing functions, a utility for sending text files of commands and/or data to the MBus, and one for connecting a tty terminal to the MBus which by default takes input from the keyboard and outputs messages to the console.

Third, the LISP translation routines have made it very easy to intercommunicate between programs written in different languages. The actual form used to transmit messages in the MBusI is LISP S-expressions (in ASCII form). The LISP translation routines form a library of C functions which allow C programs to easily parse a received ASCII string into an S-expression and then manipulate it using standard LISP commands like car and cdr.

Problems With the MBusI

There are two main problems with the MBusI: it is not as efficient as it could be and it is not very scaleable. Its inefficiencies are three-fold, and affect it by decreasing its throughput. As mentioned above, the S-expression form of transmission is inefficient for communication. In addition, because no advantage is taken of the new multicasting capabilities of networks, it is not as efficient at group communication as it could be. Finally, communication is inefficient because it must all pass through a single server. If clients are widely separated over a WAN, communication between clients on the same LAN will be needlessly slowed down if the server is located much farther away. Clients should only have high latency in their communication if at least one of the clients in the domain is separated from the others by a high latency link.

The second main problem is that the MBusI does not scale very well due to the number of sockets that can be connected to one host and because it does not take advantage of IP Multicasting. Because there is only a single server and this server communicates with clients via TCP/IP virtual circuits, the number of clients that can be connected to the bus and in communication with each other is limited by the number of TCP/IP sockets that can be opened at one time by a single process. With most UNIX processes, this is 16. This number can be increased somewhat, but it remains a fundamental problem. Another scalability problem is the topology is prone to bottlenecks in communication. Since all of the communication goes through a single process, the host that process is on can become a bottleneck. If too much traffic goes through that host, either its CPU or the network it is connected to become a bottleneck to communication. This is compounded by the fact that IP Multicasting is not used in any way, so that multiple copies of the same packet have to be sent over a link. This multiplies the load on critical segments of the network and the host CPUs.

Additional Requirements For the MBusII

The applications the MBusI was originally designed for, and which are also the most important for the MBusII to support, are CSCW applications such as shared whiteboards, shared editors, and in particular a collaborative system called Conversation Builder. In addition to the demands that the MBusII be more efficient and scaleable than the MBusI while keeping its primary advantages, these applications place certain demands on the MBusI which must also be met by the MBusII. These are:

- Message delivery must be reliable and atomic. This means that any message sent to a group must get to all members of the group, and if for some reason a message can't be delivered (say, due to the sender crashing before the message gets through) then either all of the destinations will receive the message or none of them will.
- Servers should be fault tolerant, so that one server crash doesn't bring down all the rest of the servers. A server crash also should not disrupt the ordering or delivery of message traffic between other servers.
- Some messages between clients must be totally ordered. In other words, for any two messages Ma and Mb received by two different clients, Ma and Mb must be received in the same order at both clients. Otherwise the states of collaborative tools can get out of sync. As an example, consider what happens if one user of a shared editor deletes some text and another simultaneously inserts some text in the same location. If the messages for these actions are received at different sites in different orders, the text of the document can become inconsistent.

- Clients need high throughputs with large messages, because the messaging tool often acts as a FTP program to transfer large files.
- The clients need to be able to send a high number of small messages a second, but the latency on each message isn't as critical. The reasons for this are twofold. First, the clients tend to be relatively asynchronous in their messaging. In other words, when a sender has many messages to send, it typically sends them continuously, instead of blocking like Remote Procedure Calls (RPC) calls do. Also, most messages are the result of direct human input, and therefore response times need to be fast enough that users don't perceive much of a delay, but this is at the very least 100ms. Conceivably this could become a problem when operating over a WAN, but not over a LAN.
- The system needs to be scaleable to hundreds of users. These users won't necessarily all be in the same group, in fact typical CSCW and shared applications may only have 2-8 users, but new users should be able to join at any time to any group with any set of clients that are connected to any of the known servers.
- Communication must be efficient over both LANs and WANs. Many user topologies are possible, but it is assumed that the most common uses will be having a number of users at one site, or else a number of users at each of a number of multiple sites.
- Each user must be able to subscribe to multiple domains simultaneously, and this must be handled efficiently. If a user is part of both a local group and a group spread across the country, the throughput and latency penalties paid for the long distance communication in the one group should not affect the performance of the other.

Design Decisions

As with any design, in making RMP and the MBusII tradeoffs had to be made in terms of functionality, complexity, and performance. The MBusII trades complexity and some packet latency for higher throughput to groups, high scalability, and extensive functionality. It provides fault tolerance, atomicity of messages, total ordering of messages, and a high level easy to use interface. It meets all of the requirements stated above for CSCW applications. These choices make it well suited not only for CSCW applications but for any applications that need reliable messages to a group of destinations. Although the latency is higher than a standard RPC, it stays roughly constant as the number of destinations increases. This is different than most other systems of this type which increase their latency as a linear factor of the number of destinations.

For these reasons, the new MBusII is fully distributed, uses a more efficient syntax for communication between clients, and uses the new Reliable Multicast Protocol for communication between servers. These improvements solve the major problems of the MBusI.

In the implementation of RMP, a decision had to be made as to whether or not it should be put in the kernel or in the user space. While most protocols have been implemented in the kernel in the past, this has primarily been for the sake of code reuse and security. Recent work [JHC94, TNML93] has shown that there are many advantages to implementing a protocol in user space instead, including flexibility, ease of development and debugging, and (suprisingly) performance. In fact, recent work [WhKa94] has shown that a very fast UDP layer could be implemented in most kernels, providing a path through the kernel to the user space in 10-50 microseconds. This would allow user space protocols to be implemented on top of UDP that have latencies an order of magnitude smaller than current protocols.

CHAPTER 4

PREVIOUS WORK

Classification of Approaches

The main purpose of both the MBusI and the MBusII is to provide an easy to use reliable multicast service to clients. Numerous other tools and protocols have been developed to provide the same type of service. These systems can be classified on two main dimensions: whether the service is centralized or distributed and the type of transport mechanism used. The transport mechanism can be based either on virtual circuits (TCP/IP, SPX, etc.) on unicast datagrams (IP, IPX, etc.) or on multicast datagrams (IPM). Figure 1 shows how the previous work is classified on these two axis.

Two of the most important factors for judging the utility of a reliable multicast protocol are scalability and efficiency, where efficiency is measured by the maximum throughput achieved and the amount of data that is delivered for a given amount of network traffic. These factors are influenced heavily by where the protocol falls in the above classification. A centralized protocol will scale much more poorly than a fully distributed one. A stream based protocol is also going to be less scaleable than a datagram based one, because the virtual circuits have to be kept established, and thus consume system resources, even if the communication path in question is inactive. Use of multicast datagrams may make a protocol more scaleable than one that uses unicast datagrams because multicast datagrams reduce the amount of network traffic, which is often a bottleneck in the communication process.

Efficiency follows the same trends. Single server protocols are usually less efficient than distributed protocols, because the single server becomes a bottleneck. Unicast datagrams aren't necessarily any more efficient than virtual circuits, but multicast datagrams are up to an order of magnitude more efficient in terms of network traffic than either of the other approaches.

Summary of Previous Approaches

MBusI

As previously explained, the MBusI uses virtual circuits and a single server to achieve reliable multicasting. Each client connects to the server with a TCP/IP connection, and

	Stream Based (TCP, SPX)	Unicast Based (IP, IPX, RPC)	Multicast Based (IPM)
Centralized (Single Server)	MBusI OS/2 Conferencing Most PC Solutions	None	MTP
Distributed (Multiple Servers)	T.122 (CCITT)	ISIS ToolTalk	RMP ISIS Transis xAmp Totem

Figure 1. Classification of Reliable Multicast Approaches

	Stream Based (TCP, SPX)	Unicast Based (IP, IPX, RPC)	Multicast Based (IPM)
Centralized	**Scalability: Very Low** **Efficiency: Very Low**	**Scalability: Low** **Efficiency: Very Low**	**Scalability: Low** **Efficiency: High**
Distributed	**Scalability: Moderate** **Efficiency: Moderate**	**Scalability: Very High** **Efficiency: Low**	**Scalability: High** **Efficiency: Very High**

Figure 2. Scalability and Efficiency of Approaches

can send both commands and messages to it. The commands allow it to subscribe to a domain and query its membership, among other things. The client attaches the destination domain to each message it sends, and the server then sends it to each of the clients that have subscribed to that domain. Due to its position in the above classification, it has inherent problems with scalability and efficiency of group communication. These are the primary reasons a successor was deemed necessary.

ISIS

ISIS is one of the most well known tools in this area, and it has been turned into a successful commercial product. It is a distributed toolkit based on the CBCAST and ABCAST protocols. It concentrates on the ordering aspect of reliable multicasting, requiring that a reliable multicast service already exist which provides lossless, uncorrupted, sequenced (with respect only to the sender) message delivery. It also includes one implementation of this reliable multicast service, which repeatedly sends a unicast or multicast packet to the destinations, until it gets an ACK from each destination.

An implementation on top of IP Multicasting now exists as well, with much better performance.

ISIS allows the user to select the level of ordering guarantee needed, for their work has shown that many applications only need causal ordering of messages (CBCAST) instead of total ordering (ABCAST). This is important because the ABCAST protocol is much more expensive than the CBCAST protocol. The protocol is well tested and very scaleable. A number of applications have been built on top of it which can support up to hundreds of simultaneous users.

CBCAST is the core protocol of ISIS, and is used to provide causal ordering of messages given a reliable multicast transport. It takes the incoming messages and delays delivering them to the application as necessary to guarantee that causal ordering is maintained. It is based on the notion of *vector timestamps.* Each sender maintains a timestamp counter as per [Lamp78] which is incremented every time a message is sent. A vector timestamp consists of an array of these counters, one for each sender in the system. A copy of the source's current vector timestamp is included in each message sent out by a source. Each time a site delivers a message it sets each of the elements of its timestamp to the maximum of the current value of that element and the value for that element in the timestamp of the message just delivered. These timestamps are used to determine the causal ordering, and are used to delay the delivery of messages (if necessary) until all of the causally preceding messages have been received. Please see [BiJo87, BSS91, Birman93] for more details.

The ABCAST protocol uses multiple CBCASTS to globally order messages. One of the sites in the group is at all times designated as the token site, and this token can move between sites. The token site sends out a *sets-order* message to the group each time it receives an ABCAST message. This sets-order message assigns a global sequence number to the ABCAST message. For efficiency, it is possible to delay this message and batch the ordering for multiple messages into the same sets-order message. A message that uses the ABCAST protocol is delayed from being delivered at each site until the sets-order message for it is received and all of the previous messages (as determined by their global sequence numbers) have been delivered.

ISIS uses a single centralized name server to keep track of group information, and uses a flush protocol to insure that virtual synchrony is maintained when group membership changes. As previously stated, virtual synchrony requires that group membership changes appear to be atomic across all of the sites. This is achieved in ISIS by flushing all of the in-transit messages out of the system and delaying the generation of any new messages until after the group membership change is done.

ISIS has a number of advantages, most notably being its scalability, its formal proofs of its correctness, its clearly defined interfaces, the wide body of services built on top of it, and its proven robustness.

One drawback of CBCAST is of the overhead of the vector timestamps. Each timestamp is typically four bytes long, and there is one timestamp for each member of the group. To get around this, they have introduced compressed vector timestamps, which only include the timestamps of currently active members. Another drawback is that in general, the ISIS system was designed for multicast communication over unicast communication links, and so the reliability and ordering functionality is separated into completely different layers. The reliability layer is forced to use positive acknowledgments from each destination. Although acknowledgments may be delayed so that one ACK covers multiple packets, this is still inefficient in many cases, particularly under low load.

In addition, the current centralized name server is an acknowledged problem with the system's scalability and robustness in the face of failures. They have shown how to get around this, but the algorithm has not yet been implemented, to our knowledge.

A final limitation is that because all messages must be flushed from the system each time a group membership change occurs, these changes are very slow and inefficient. In addition to the waiting done while the messages are completed, the flush protocol requires an additional 2N messages to guarantee that the system has been flushed and confirm the group membership change. However, group changes in the applications the system supports are infrequent, which decreases the importance of this limitation.

Transis

Transis is a new system for distributed computing from the Hebrew University of Jerusalem, Israel [ADKM91, DKM93]. It is similar to ISIS in that it provides both causally and totally ordered multicast. Transis provides reliable, virtually synchronous, atomic multicast within a broadcast domain, which is typically a single LAN. Each broadcast domain contains a single server which is designated as the bridge to communicate with other broadcast domains. Each of the bridges communicate between each other through a reliable datagram protocol such as RPC. Causal multicast is maintained over messages that span multiple broadcast domains, but it is not currently possible to send totally ordered messages across broadcast domains.

Basic reliable communication in Transis comes from a protocol similar to the Psync protocol[PBS89]. A dependency graph is constructed at each site of messages that could causally precede it, and messages are delayed until their predecessors have arrived. This dependency graph provides both causal ordering and reliability.

A number of different total ordering schemes have been implemented on top of their causal ordering scheme. The most commonly cited one is a *pre-ordering rotating token*, similar to that used in Totem (described below). This scheme requires that a token be rotated to the sender before it is allowed to send. By only allowing one sender at a time to proceed, each sender can put a global timestamp on its packets.

The Transis system has a number of advantages. First, the protocol is relatively efficient in terms of computation required. Second, it can achieve very high throughputs even for totally ordered multicasts. If all of the sites are sending at the same rates, the system can achieve an aggregate throughput that is close to the bandwidth of the LAN multiplied by the number of sites in a domain. This is an order of magnitude higher than any non-hardware multicast system can produce. Finally, the hierarchical system of domains will also improve the system's scalability in environments where the traffic between domains is not heavy and does not need to be totally ordered.

The Transis system also has a number of disadvantages. First, a sender can not tell for sure if a message has been received by all domains, unless each of the domains happens to send a message out that has this confirmation piggybacked onto it. This means that buffer space must be extremely high in order to provide reasonably high guarantees on reliability. This is compounded by the fact that each of the destinations must retain each message received until the point in time where the system believes it is safe to drop it, so that any of them can respond to a NACK. This problem of knowing when to drop buffers becomes much more difficult in an internetwork, where congestion can frequently block packets for 5-10 seconds at a time. The authors recognize that the protocol can not permit a broadcast domain to span any type of complicated internetwork. It also casts doubts on whether the hierarchical communication can actually provide fault tolerant message delivery.

In addition, the hierarchical structure has some weaknesses. Transis can not provide totally ordered multicasts across multiple broadcast domains. The designated servers that handles communication between domains will form a bottleneck for heavy communication between broadcast domains, and will add an extra latency delay as well. This external server must stay up as well, or all communication to the rest of the broadcast domains is lost. This hampers its fault tolerance.

Finally, the latency of a pre-ordering rotating token is very high. On average, a site must wait for half the members of the group to have received the token and passed it on before it can send its packet. For large groups, this may be impractical.

Totem

The Totem protocol [AMSM92, MMA90] is perhaps closest to RMP in its approach, and has reported similar throughput levels to RMP under heavy load. It also uses a rotating token ring approach, but only provides for a single ring for each broadcast domain. Totem avoids using any ACKs by only allowing the current token holder to send data. This is called a pre-ordering rotating token ring, and is very similar to what Transis uses, the two systems having been developed in collaboration with each other. The primary difference between Transis and Totem is that Totem only allows totally ordered messages, where Transis provides causal ordering and layers total ordering on top of this. This provides high throughput under high load over a low latency network, but has the same problems that Totem does, including lower throughput and longer latency under low and asymmetrical loads. In addition, because it only allows a single sender to transmit at a time it will provide lower throughput over longer latency networks. To alleviate this problem they have proposed, but not implemented, gateways to link multiple broadcast domains together.

ToolTalk

ToolTalk is the new reliable multicast tool from Sun which is included as part of their Solaris operating system. It has a distributed architecture, where multiple clients connect into each server, and the servers communicate between themselves. The inter-server communication consists of both group membership information and messages between clients attached to different servers. The servers use Remote Procedure Calls (RPC) between each other. This is a very straightforward approach that is relatively easy to implement, but does not (and can not) take advantage of hardware multicasting support. ToolTalk does not provide causal or total ordering of messages, nor does it provide fault tolerance and atomicity.

Multicast Transport Protocol (MTP)

MTP [AFM92] is a totally ordered, reliable multicast protocol. The ordering can also be disabled and it can be used as an unordered reliable multicast protocol. It uses a central server or "master" to provide serialization of messages and handling of global state information. When a given process wants to send a message, it requests a "token" from the master. This token grants that process the right to send one message of arbitrary length, and assigns a sequence number for that message. After being granted a token, that process then multicasts out its message to all processes that are currently communicating. This group is called the "web".

Flow control is provided through a "leaky bucket" algorithm, where each process that has a token is allowed to send up to W packets of a predefined maximum length (currently 1500 bytes) every H milliseconds. This period of time is known as the *heartbeat* of the protocol. Negative acknowledgments are used on all data transmissions. Each packet of each message is numbered, and both the last packet in a window and the last packet in a message are flagged as such. Therefore, at the end of every heartbeat, each member of the web can check to make sure it has no holes in the list of packets it has just received. It can tell this by seeing if it has missed any packets in a message, if it doesn't receive a packet with an end of window or end of message flag, or if it receives packets from a message which has a higher message sequence number than expected. Each packet sent carries status information on the last 12 messages, set by the master, which allows messages to be aborted if the sender fails before it finishes sending a message.

If a process does detect a missing packet (or packets) it sends out a NACK packet to the sending process, requesting retransmission of the missing packet(s). The sender then multicasts these packets again, which count as part of its allocation or Window for that next heartbeat. Because a sender must be capable of handling retransmission requests, it must hold on to a copy of its packets for Retention heartbeats. Therefore it must have buffer space of Retention * Window * Maximum Length allocated. In order to help insure detection of dropped messages, each token holder must send at least one packet each heartbeat, even if it is an empty packet, and the shortest possible message must be at least Retention packets long.

These three parameters, Window (W), Heartbeat (H), and Retention (R), govern both the reliability and the efficiency of the protocol. They are all constantly negotiated by the participating processes and the master. The larger the Window and the shorter the Heartbeat, the higher the throughput. The higher the Window and the higher the Retention, the more buffer space that will be required. Finally, the higher the Retention and the longer the Heartbeat, the higher the reliability in the face of dropped packets.

MTP has a number of advantages. Because the master handles all synchronization and membership issues, as well as holding all of the state of the protocol, the protocol is relatively easy to implement. In general, distributed algorithms are more difficult to implement and get correct. As long as the master doesn't fail, token losses and site failures are easy to recover from.

Because the flow control is negotiated explicitly, the usable bandwidth (as measured by a percentage of the available bandwidth) can be limited so that multicast applications don't encroach on the bandwidth used by other applications.

The protocol is transport independent, so can be adapted to a wide range of networks. Finally, if large numbers of packets are lost at once, they can all be requested at once in a single NACK.

However, MTP has quite a few disadvantages. The protocol does not give guaranteed reliable delivery. Because the senders do not know the cardinality of the receivers and do not get any type of acknowledgments back from them, they can never really be sure that the message was received. Not only may the receiver miss messages, but the sender can never even guarantee that it will detect that the receiver has missed one or partitioned away. This makes it impossible to know the group membership at any given time. The protocol also claims to provide atomic delivery, but if reliability can not be guaranteed this is not possible. It does detect if a sender has failed before finishing a message, but it will commit a message for delivery at all sites before it knows that all of the sites have received it. If the sender fails or if one of the destinations does not detect and successfully rerequest a missing packet before the sender throws away the message, both atomicity and reliability will be violated.

Because of the uncertainty of delivery, senders must hang on to their packets for a length of time considered sufficient to provide a reasonable guarantee that all of the destinations received the packet. This requires a large amount of buffer space or a low throughput. The suggested figures for operation over an internet specify a maximum throughput of only 180 KB / second, and a retention period of up to 1 second. For this set of parameters, the sender must have 180 KB of buffer space. Both of these parameters are low, however. It is not uncommon to see a site that can not get any traffic through to it for 5-10 seconds or more on the Internet. To force a connection to fail due to a loss of communication of only a second is overly restrictive. Also, TCP/IP can achieve throughputs of 1100 KB/sec over an Ethernet. If MTP is to support both this level of throughput and a reasonable level of reliability for the Internet, a sender needs to have 10 MB or more of buffer space.

MTP is also not very scaleable since it uses a central server to grant all of the transmit tokens. This will limit the size of a web, and since there is no provision for allowing sites to exist in multiple webs at the same time, this limits the number of processes that can communicate together at one time. It is moderately fault tolerant, in that crashed clients are automatically removed from the group, but it can not handle the failure of the master.

The MTP protocol is inefficient in terms of sending small packets. A one byte message requires at least seven packets (of a minimum Ethernet length of 64 bytes) to be sent, and it can not be delivered at any site until at least three of them have been sent. This is both wasteful and has a higher latency than other approaches. It is also slow in terms of sending large packets. The suggested figures only allow for 180 KB/sec. While this may be increased in the future, it is the current maximum figure that the authors of the protocol believe it can support.

CCITT T.122 -- Multipoint Communication Service for Audiographics and Audiovisual Conferencing

This is a CCITT proposed standard for implementing reliable multicast for use by such services as shared whiteboards, etc. It is fully distributed, allowing multiple servers to be started that communicate with each other through a set of virtual circuits. Multiple clients can connect to each server. The connections between the servers can be hierarchical, fully connected, or of some other topology. Communication domains (multicast groups) can then be defined over the connected network. No ordering guarantees on message delivery are given, but it does provide a token for each domain which can be used to serialize messages or give exclusive access to a resource. This will allow applications to implement totally ordered multicast.

The biggest advantage of this approach is that it is being standardized. In addition, by using well developed and tested virtual circuit protocols such as TCP/IP and SPX, the transport part of the implementation is simplified, and it can be generalized to a wide range of environments. Having communication exist in a hierarchy has both good and bad points. If the hierarchy happens to emulate the network topology well, some network traffic may be saved over a reliable datagram approach such as ToolTalk. In effect, the hierarchy can serve some of the same purpose of eliminating duplicate identical packets from being sent over a link that IP Multicast routing does. However, this is achieved at the cost of having to process the packet at the user level at each of these sites instead of at the router. This will use up extra computation cycles at these computers, and will be slower and less efficient.

In addition, it does not make any use of broadcast or multicast communication where available, which makes it much slower and less efficient in these networks than protocols that do take advantage of hardware multicasting. Finally, if total ordering is needed of messages, using a single token to achieve this will be very slow and inefficient, because it will only allow one site to send a message at a time, and each site will also have to pay the cost of getting and releasing this token for each send.

The IP Multicasting White Board

There is currently a program, wb, written by Steve McCanne, that provides reliable multicasting on top of IP Multicasting. There is no formal separate protocol. It works by using straight negative acknowledgments to a multicast group. Each sender is required to keep its entire history of sent messages so that it can respond to a NACK at any time. There is no guarantee on the number of sites that have received the information, and there is no ordering guarantee on the messages received at any site. However, because a group member can request missed packets at any time it can achieve a high level of reliability in a very simple fashion. This simplicity has allowed it to be quickly implemented and it is now in use by many sites.

The drawbacks of this approach are that it is not atomic, it requires an unbounded amount of buffer space, it provides no ordering guarantees, and a missed packet could stay missing indefinitely if it is not followed up by another packet from the same sender.

Other Protocols

The protocol by Crowcroft and Paliwoda [CrPa88] is one of the first protocols to propose reliable multicast over an internetwork which supports hardware multicast. The protocol provides different levels of reliability guarantees, and uses positive acknowledgments from all destinations for reliability. The paper analyzes the flooding problems that occur with simultaneous ACKs from many destinations and proposes a windowed flow control system, in some ways similar to that used in RMP, to alleviate these problems.

The xAmp protocol [Verissimo92] is distributed but also waits for ACKs from all destinations. However, these ACKs are implemented just above the link layer of an Ethernet[MeBo76], so it pays little CPU overhead for these ACKs. This protocol is the state of the art for providing real time guarantees to a totally ordered multicast protocol. Because xAmp actually uses the broadcast link itself to provide total ordering, it will not scale to a WAN.

The broadcast protocol proposed by Kaashoek et. al. [KTHB88] uses a central token site to serialize messages and NACKs for retransmissions. It piggybacks ACKs onto sent messages and has the token site regularly contact silent sites in order to limit buffer space. This protocol has reported very good latency (as low as 1.3 ms for a NULL packet) because it has been implemented on top of bare hardware. However, because each message must be transmitted twice it will fundamentally achieve lower throughput than RMP -- 600 KB/sec is a rough upper bound for a 1250 KB/sec Ethernet, as compared to 842 KB/sec for RMP. This will also limit the latency for larger messages; as a 8KB packet in their protocol will spend a minimum of 13.1 ms on the Ethernet, as opposed to 6.7 ms for the message and ACK of RMP.

CHAPTER 5

DEFINITION OF TERMS AND ASSUMPTIONS

System Model

The system model RMP adopts is very similar to that which most similar distributed systems [ADKM91, BSS91] adopt. In these systems, each group consists of a set of receivers. Packets are addressed to a group, instead of to an individual process. In our model, a process must be a member of a group before it can send to it, although some other systems do not impose this restriction.

Each communicating process is divided into two pieces, the user application and the communication layer. The communication layer handles all of the ordering and reliability needs of the application, and delivers packets to the application in a sequence that meets these needs. When a packet is delivered to an application, it is put on a queue, and the application pulls packets off of the queue as needed.

In RMP, there is no need for any type of global clock synchronization. The system must be able to measure time in order to detect time-outs of packets, but these clocks do not have to be synchronized across systems.

In our system model, a fault can consist of a transient fault or a system fault. A transient fault is usually due to transmission errors or congestion overflow, and prevents a packet from being delivered to all of its destinations. Because RMP is built over UDP, it is assumed that all transmission errors are caught by UDP, so only omission errors, delayed packets, duplicate packets, and reordered packets are possible. A transient fault may also correspond to a process that has not received service for an extended period of time, but which has not crashed. A system fault is either a network partition or a process failure. We do not consider processes that fail in byzantine ways. If a process fails, it is assumed to fail in a fail-stop manner, where it simply stops processing.

In general, it is impossible to differentiate between a transient fault and a system fault in general asynchronous networks. Because of this, time-outs are used to detect a site that hasn't responded within a certain period of time, and the membership algorithm then removes those sites.

CHAPTER 6

SYSTEM USAGE

In seeking to explain how the MBusII operates, it is best to start first with how it is used. The MBusII allows a set of clients to connect to a set of servers that are in communication with each other. They then use a publisher / subscriber model of communication for their message passing. The servers communicate among themselves using *token rings*, which are the fundamental unit of delivery and ordering of messages between them. For more details, please see the user's manual in Appendix A.

The MBusII is a fully distributed system, in that it has multiple servers, all of which play the same role in the communication. Multiple clients can connect into a MBusII server, each with their own TCP/IP socket. If the client and the server are on the same host, the socket realizes this and no network traffic is needed in order for them to communicate together. One or more servers all communicate together using RMP. Typically, there will be one server per host that has a client on it, and these servers will handle all of the actual network communication. Alternatively, the system is flexible enough that clients can run on different machines than the servers if so desired. Figure 3 shows an example topology of clients and servers.

The clients use a publisher / subscriber model of communication, similar to that described in [Birman93, BLNS82]. Communication is centered around domains, where each domain is a hierarchical text stream similar to the textual representation of an Internet IP address. A domain consists of an arbitrary number of text strings connected together by periods, such as "page1.whiteboard.kaplan.uiuc". Domains in the MBusI and MBusII have the same role as groups in other schemes such as IP Multicasting[Deering89] and the V operating system[ChZw85]. Clients subscribe to domains, and when other clients publish a message to the same domain, it is automatically delivered to all of the clients who have subscribed to it. Clients can control whether they wish to have their own messages delivered to themselves or not. This is called loop-back, and is useful when used in conjunction with the total ordering of messages to provide consistent ordering of events at multiple sites. This model of communication greatly simplifies some applications, both because of the loop-back feature and because it does not require explicit knowledge of who is a member of a group. Group membership can be queried at any time, so applications that do need explicit knowledge can use the tool as well. Finally, this system can also be used as a point to point virtual circuit such as a TCP/IP connection.

There are two different ways to connect to the MBusII. The MBlib is a C library that can be linked into a client written in C or a similar language. It automates the establishment of a connection to the MBusII, and also provides routines for handling of messages. Messages are represented in an internal linked list structure that emulates LISP S-expressions. Routines are provided for traversing messages, converting part or all of them to text form, and parsing text strings into S-expressions. This library makes it very easy to communicate with other clients written in C or C++ as well as LISP or CLOS. The other main way to communicate with the MBusII is to open a TCP/IP socket with it

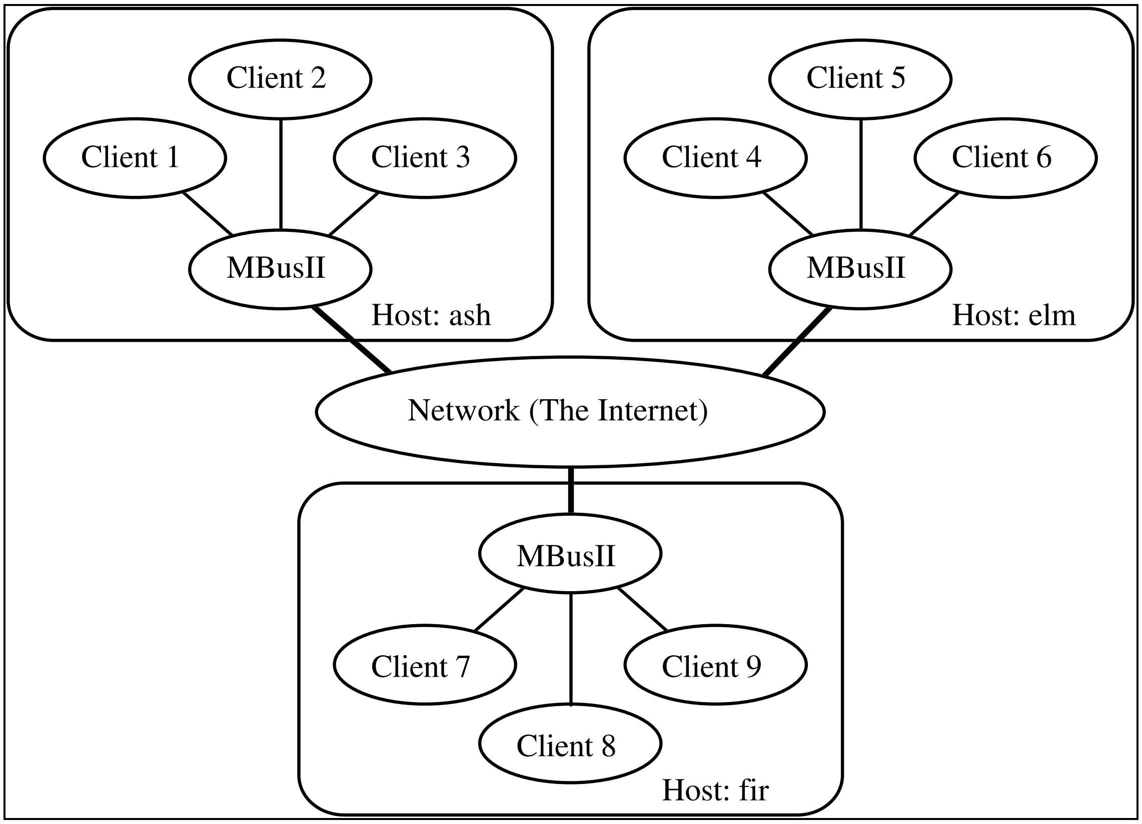

Figure 3. Interconnection of Clients and Servers

and then send the appropriate ASCII commands and messages directly to the bus through this socket. There are utilities that allow you to open a channel from the console and keyboard directly to the MBusII, for you to connect a LISP client to the MBusII, or to send a file of commands and/or messages to the MBusII.

There are two different syntax's that can be used to communicate with the MBusII. The MBusI used S-expressions for all of its messages, but this proved to be inefficient. Because of this, the MBusII internally uses a new CSTREAM format which is easier to parse by C clients and which is more efficient. The LISP Message Bus Library (LMBL) is used to convert back and forth from the CSTREAM to/from the S-expression form. It automatically detects which form a given client is using, and returns messages to that client in the same syntax. The CSTREAM syntax is described in Appendix A.

Token Rings

The single biggest difference between the MBusI and the MBusII that the user has to be aware of is the new use of *token rings*. Each token-ring is a virtual link between a number of MBuses. Each token-ring corresponds to one or more domains, and will have one or more users which have subscribed to it. When a message is sent by a user to a particular domain, RMP uses the NS library to find out which token-ring the message is to go to, and reliably multicasts it to all those (and only those) MBuses. In the example shown in figure 4, there are 6 RMP servers at two sites, all connected through the Internet. There are four token rings that have been set up, the global ring (primarily used for NS control information), one between all the Illinois servers, one between two of the Berkeley servers, and one between two physically separated servers (one at each site.) Multiple token rings are important because messages over WANs are much more expensive than those sent only over LANs, and also because you don't want to load down servers with messages that no client to that server wants. Token rings are the unit which message delivery and ordering get done at--all messages in a token ring, no matter which domain, are totally ordered and sent to all members of that ring. Token rings are moderately expensive in terms of memory, which is why a seperate ring is not just automatically created for each domain. Also, it is often important to have ordering guarantees maintained across multiple domains, and this can only be done if the domains are all part of the same token ring.

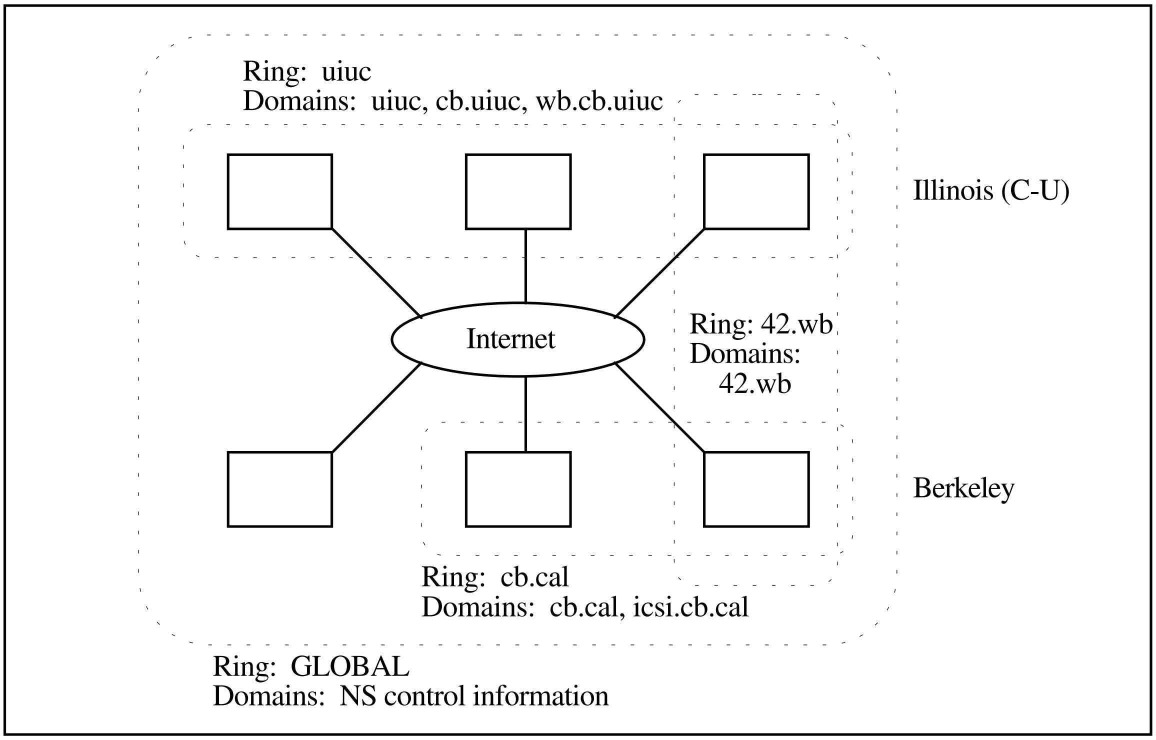

Figure 4. Use of Token Rings

Sample MBus Syntax

The application programs communicate with the MBus through a set of commands. The most common MBusII commands are listed here. For more details, please see Appendix A, which contains the users manual for the MBusII.

accept tag domain

This subscribes to a given **domain**, but only accepts messages whose tag match the regular expression **tag.** In practice, **tag** is usually specified as "*" to accept all tags from the domain.

msg tag domain-list msg

This publishes a message to one or more domains (**domain-list**), and attaches a **tag** for further filtering. In practice, most applications specify something arbitrary for the tag, and have the destinations accept all tags. The **msg** can be specified in a number of forms, but the most common is to give the length of the data and then the data.

id namestring

This assigns **namestring** to be the identifier for a client. This can be any textstring, but usually is in the form of a single string or a hierarchical domain name for use in point to point communication.

ping tag domain

This returns a list of the names of all clients who have subscribed to a given tag and domain. This is used to query the contents of a domain. It is primarily used to see if the group subscribed to a domain is empty or not, and to find the list of client identifiers associated with a domain for those applications that need explicit knowledge of this.

reject tag domain

This unsubscribes a given client from a set of **tags** associated with a **domain**. If there are no tags left subscribed to in this domain, the client is removed from the entire domain.

loop-back flag

This specifies whether messages should be sent to the sender or not. As explained earlier, this is useful for shared applications that need each client to perform its own events and the events of the other clients in the same order at all sites.

CHAPTER 7

SYSTEM ARCHITECTURE

There are four main parts to the MBusII: the Name Server (NS), the Reliable Multicast Protocol (RMP), the Message Bus (MBus), and the Lisp MBus Library (LMBL). In addition, there are numerous utilities and libraries for connecting clients to the MBus. Together, these parts roughly correspond to levels four through seven of the OSI networking model (see figure 5). IP and IP Multicasting perform the functionality for levels 1 to 3. This is not a perfect correlation, because TCP/IP sockets are used to connect clients to the MBusII server.

The OSI transport layer consists of RMP library and the NS library. Together, they provide a reliable, distributed, atomic, fault-tolerant, multicasting messaging facility over multiple domains.

The NS library takes care of maintaining the information about all domains. It stores the necessary information, and through messages to the GLOBAL ring, handles the addition and removal of clients to and from domains, of domains to and from token rings, and of MBusII servers to and from token-rings. It is distributed, with each site maintaining information on all of the domains and rings it is a part of, as well as information on all of the servers that it can communicate with and all of the domains that exist. Together, RMP and NS make up a library that can be linked in and used by any messaging tool or application to provide reliable, atomic, distributed, totally ordered multicast functionality. The most important thing they do not provide is an interface to multiple clients.

This is provided by the rest of the MBusII code, and could also be implemented by other servers built on top of RMP and NS. These are the Alternate Servers that are pictured in the Session Layer.

The Message Bus Layer provides a more full functioned interface than the Reliable Multicast Layer. It takes in and sends out character string messages. It also allows multiple clients to simultaneously connect to it via a TCP-IP port, and communicates to

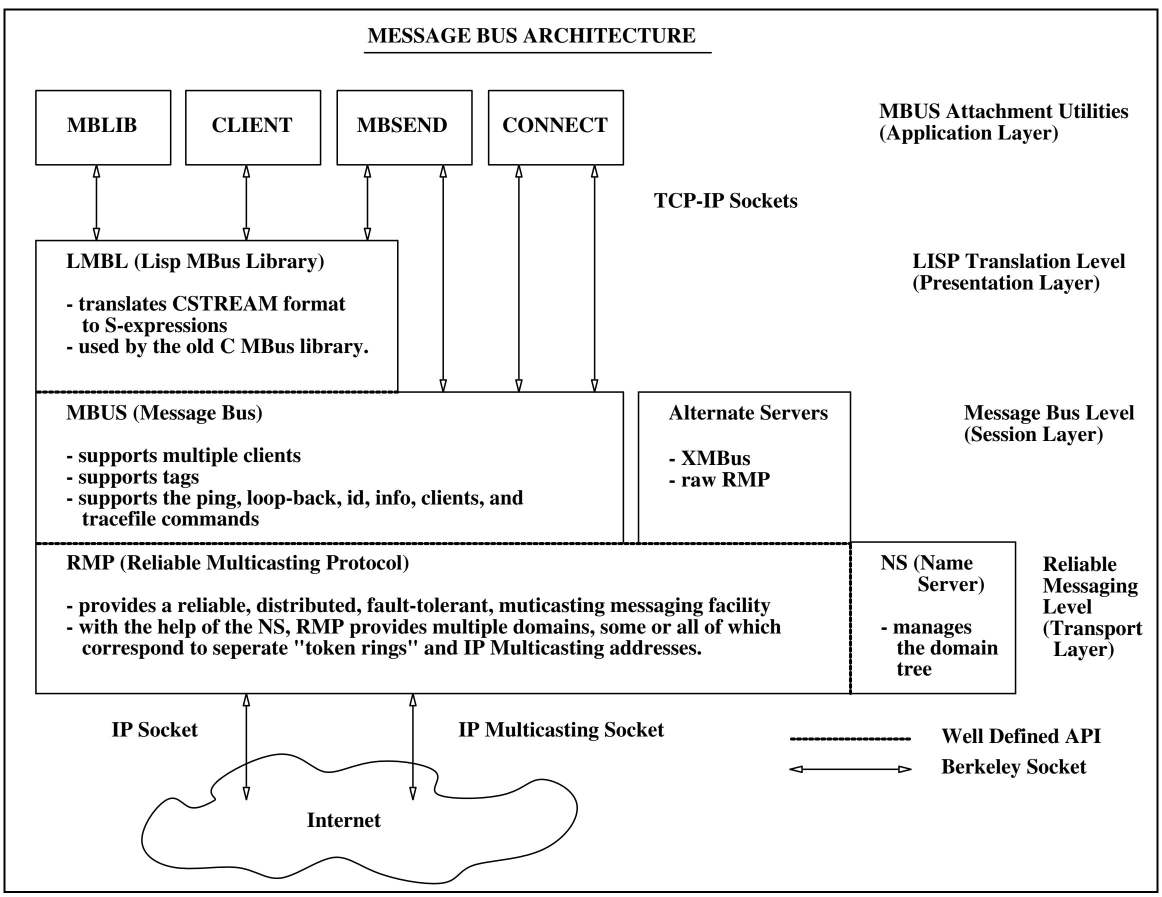

Figure 5. MBusII Architecture

the LMBL translation layer, which can parse and translate Lisp S-expressions to and from the more efficient CSTREAMS interface used by the MBus layer. It provides more features than RMP library as well, with the ping, loop-back, id, info, clients, and tracefile commands.

The Lisp MBus Library (LMBL) exists for backwards compatibility with applications that were written for the first version of the MBus. It allows both C and LISP programs to transmit and receive S-expressions over the MBus, but at a performance cost. Because new applications are moving away from this syntax, its performance is not analyzed in this paper.

The previously explained utilities for connecting in to the servers are all pictured at the application layer, and connect via TPC/IP sockets to the servers.

CHAPTER 8

RELIABLE MULTICAST PROTOCOL DESCRIPTION

The reliable multicast protocol (RMP) is the heart of the MBusII, and it handles all of the communication between MBusII servers. RMP is based on the protocol proposed by Maxemchuk and Chang in 1984 [ChMa84]. We call this the Token Ring Protocol (TRP). RMP builds on this protocol by adding many features necessary for operation in an internetwork.

Token Ring Protocol Description

The biggest decision in building a reliable multicast protocol is how to guarantee reliability. Traditional protocols use positive acknowledgments (ACKs) from the destination(s) to acknowledge successful receipt of a packet. This approach does not scale well to a multicast system, because each destination has to send an ACK for each packet. This largely defeats the advantage of using multicast packets, because it decreases both the efficiency and the performance of the protocol. Even though these acknowledgments are small, because they are all sent at the same time they can cause network congestion. In addition, having to process an ACK from each destination increases the load on the sender and decreases the performance of the protocol. To get around this, many systems use negative acknowledgments (NACKs). Negative acknowledgments shift the burden of error detection from the source to the destinbations. Packets are stamped with sequential sequence numbers which destinations use to provide reliable delivery by detecting gaps in the sequence numbers and requesting retransmission of the packets corresponding to the gaps. Because the information that a packet has been received is never propagated back to the sender, the senders in these protocols do not ever know for certain that a destination has received a packet. Because of this, senders have to indefinitely keep a copy of each packet sent if the protocol is to be considered truly reliable. In addition, a lost packet that is not followed by another packet may not be detected for quite some time.

Because of these problems, TRP uses a combination of these two approaches. This section describes the basic algorithm of TRP as originally proposed by Chang and Maxemchuk. For a more rigorous analysis, as well as a complete description and proof of the correctness of the reformation protocol, please see [ChMa84] and [ChMa83]. [MaCh84] gives a complete analysis of the number of packets sent by the protocol, and [Chang84] describes the advantages of using this protocol to design a distributed database.

In TRP, all messages are stamped with a pair (host number, sequence number for that host) which uniquely identifies each message. Each message is then broadcast to the entire group, and all senders to a group must be members of it. Messages are all handled by a primary receiver called the token site. When the token site receives a broadcast message, it broadcasts a positive ACK out to the group. Along with the (host, sequence) pair for that message, this ACK contains a sequence number, called a timestamp, which serializes the messages from all of the senders. This ACK performs a number of functions:

- It lets the sender know that the token site has received the packet. In this way it functions as a traditional positive acknowledgment.
- The timestamps in the ACKs provide a total and causal ordering on messages.
- The timestamps also provide a global basis for detection of dropped packets. The receivers can detect any missed packets, both ACKs and messages, through these global sequence numbers. With multiple simultaneous senders this provides more timely detection of missed packets than sequence numbers from each sender.

This does not solve the problem of guaranteeing the detection of dropped packets and the corresponding problem of unlimited buffer space. To solve this problem, the token site is passed among all sites in the token ring. The token is rotated (at least) once per each message sent, and each new token site is required to have all messages up to the current one before it accepts the token. Not only does rotating the token balance the load of the ACKs between the sites, it also solves the buffer problem. Given N members of a ring, once the token has been rotated N times, the token site knows that all messages with a timestamp at least N smaller than the current timestamp have been received at all destinations, and so the token site no longer needs to store these messages for retransmission.

Token passing is done as another consequence of the ACK. A field in each ACK names the new token site. When this site receives the ACK, it checks to see if it has received all of the messages and ACKs with timestamps smaller than this ACK. If not, it requests them from the previous token site. Once it has all the messages, it declares itself to be the new token site. It lets the other sites know that it is the new token site either by sending an ACK for the next message received, or if no message is received within a set period of time, by sending a unicast Confirm message to the old token site.

Each time a destination receives an ACK, it uses the ACK's timestamp to order the corresponding message. The destination can deliver a message after all of the preceding messages have been received and delivered. If the destination is missing any ACKs or messages, it requests these by sending a unicast NACK to the site it currently believes to be the token site. This site is guaranteed to have a copy of the missing message or ACK, even if it is no longer the token site, and resends the missing packet as a unicast to the site that requested it.

All three of these actions--broadcasting a message to the token site, passing the token to the next token site, and requesting retransmission of a packet--are reliable operations that use positive acknowledgments. The sender sets a timer each time one of these operations is started, and retries the given operation if the timer expires. It keeps on doing this until it receives the correct response. If an operation is tried more than a set limit of tries, the site is presumed to have failed, and a reformation protocol is run to reconstruct the list of sites in the token ring.

Passing of the token also allows the protocol to provide K-resilient fault tolerance. K is a system wide parameter corresponding to the amount of time a packet is delayed by a destination before it is delivered. By delaying the delivery of a message until the token has been passed K times, it is ensured that both the current token site and the K previous token sites have that message. This allows up to K sites to crash or partition away and still guarantee reliability of message delivery to the sites that are still active. Along with this, passing the token guarantees that site failures and dropped messages are detected within N messages. In order to bound the amount of time before a lost packet or a failed site is detected, TRP sends null ACKs if the token site does not receive a message within a given period of time. With RMP, this is currently on the order of 1 second. If the ring goes quiescent for a period of time, the token is passed all the way around the ring once and then stops. At this point, all of the sites are guaranteed to have all of the messages.

As previously mentioned, if a site repeatedly fails to receive the proper response to one of the actions that requires a positive acknowledgment, it declares the site dead and runs a reformation protocol. In this protocol, the site that first detects the failure uses multicast packets to repeatedly poll the sites listening to a given broadcast address. From the responses to the polls it constructs a new token list, and this list is atomically updated at all of the sites in the new list with a two phase protocol. If more than one site starts the reformation at the same time, this will be detected and the reformations aborted, whereupon each of these sites delays for a random period of time and then starts the reformation process again. We omit this algorithm and its proof of correctness for brevity.

In normal situations with low error rates and moderate to high traffic, TRP requires very close to 2 broadcasts per message. As the traffic rate decreases, this increases to 2 broadcasts plus a unicast due to the messages required to confirm the transfer of the token. As the error rate increases, the number of packets sent increases, but it is always lower than that required with positive acknowledgments for groups of three or more sites including the sender [ChMa84].

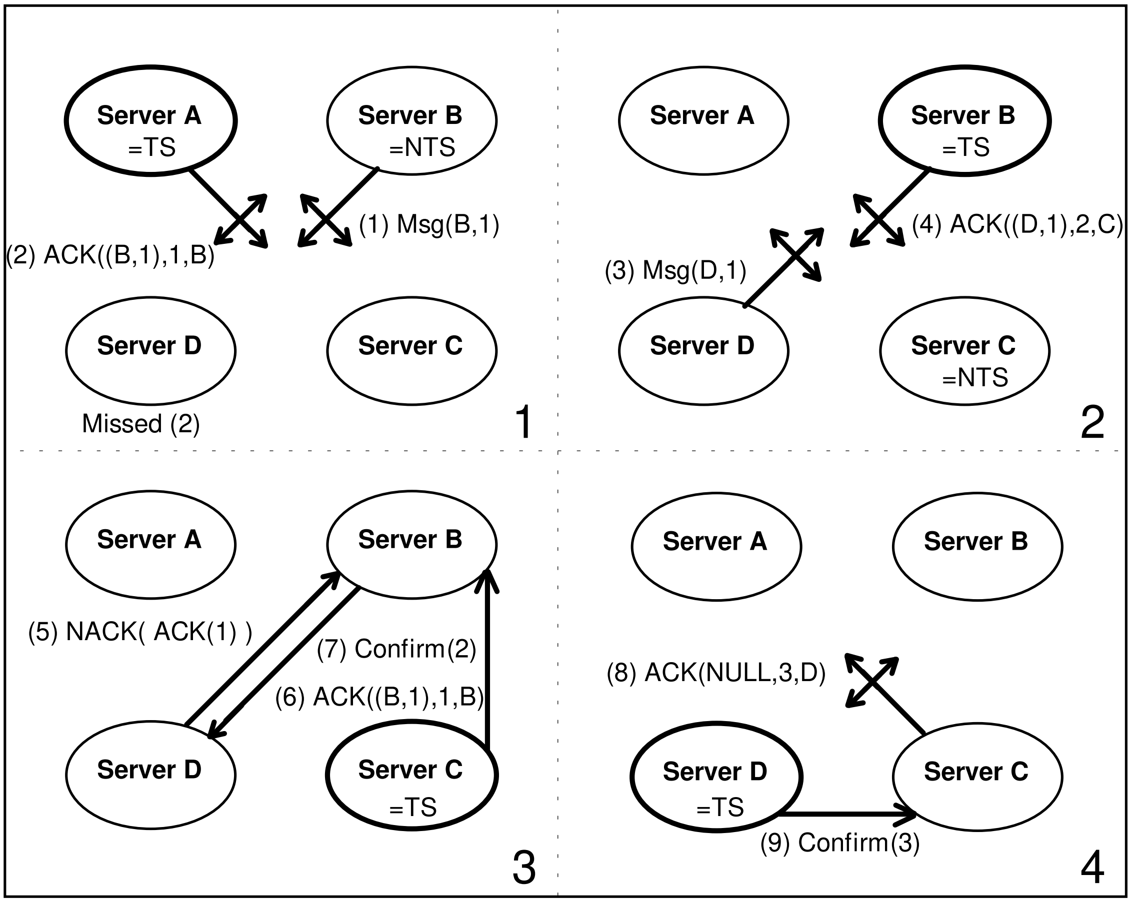

Figure 6. Operation of RMP Normal Phase

Figure 6 shows an example of how both TRP and RMP operate in the absence of failed (or apparently failed) sites. Frame 1 shows the sending of message 1 from B, and its ACK. Both messages are multicast to the whole ring, but the ACK is not delivered to D for some reason. The (server, sequence number) pair uniquely identifies each message sent and allows the time stamp to be associated with it. In this case, the message is (B,1), and it is timestamped as the global message number 1 by this first ACK. All of the sites that have received both the message and the ACK can now deliver it, since it is the first message and so there could not have been any missing messages before it. In the last field of this ACK, B is named as the new token site (NTS). When B receives this ACK, it checks that it has all of the messages up to the time stamp of this ACK, and then declares to itself that it is the NTS. Notice that site A does not yet know this, and so is waiting for either a Confirm message or an ACK message from B. If it does not receive it within a specified time-out period, it will resend its ACK. It will repeatedly do this until it decides that B is dead or until it gets an appropriate response from B.

Frame 2 is very similar to Frame 1. It shows the sending of the first message from D, and the corresponding ACK from B. After these two messages, A knows that it has successfully passed on the token, and B is now waiting for C to pick up the token. Site D, which missed the first ACK, now knows that it missed it, because it has an ACK for timestamp 2, but no ACK for timestamp 1. Because of this, it cannot deliver either of its messages yet. By default, RMP sends a message to all sites in the ring, including the sender. This costs very little extra, and these messages can easily be filtered out. So at the end of frame 2, all of the sites except for D have delivered both (B,1) and (D,1)

Frame 3 shows the NACK of the ACK((B,1),1,B). Site D, knowing that it is missing this ACK, requests it from the site it thinks is the current token site, namely B. This is a unicast message, not a multicast, and so it is sent only to B. B responds with another copy of the ACK. This retransmission request is also on a timer, so that if the retransmission fails, the ACK will be rerequested by D. Frame 3 also shows a Confirm message being sent from C to B. This occurs because none of the sites sends another message within a given period of time. This period of time is relatively short, because the new token site needs to send back this Confirm message before B times out and resends its ACK. So at the end of this frame, B knows that C is the token site, and all of the sites have delivered both message 1 and message 2.

Frame 4 shows an extra passing of the token to make the pending messages stable. In this example, site C does not receive a message for a given period of time and knows that the token needs to be passed. It multicasts a NULL ACK, with its own timestamp, to the ring. This ACK names D as the new token site, and D immediately sends back a Confirm message, since it does not have any reason to expect to see another message soon. If this happened a second time, then A would be the new token site. At this point, A would then know that all messages up to timestamp 1 had been received by all of the sites in the group, and it could discard all of the messages and ACKs up to and including timestamp 1. After another token pass, site B would know that all messages and ACKs up to 2 had been received, and would drop those. This guaranteed delivery allows us to bound the necessary buffer space for this protocol.

Extensions To The Token Ring Protocol

In order to provide better performance and to scale to multiple groups over an internetwork, RMP uses multicast instead of broadcast, allows for multiple token rings, provides a fully distributed name service, improves the reformation algorithm for membership changes that are not due to failures, and implements a more robust and efficient flow and congestion control algorithm.

Multiple Token Rings and Groups

Communication in RMP is organized around groups and token rings. Groups correspond to the set of processes that a message is to be delivered to while token rings correspond to the set of hosts that a message is to be delivered to. Both constructs are needed because multiple processes on one host may be in the same group.

A group name is a hierarchical text string, roughly modeled after the textual representation of IP addresses. There are two main differences between a group name and an IP address. First, IP addresses can only consist of up to four levels of text, whereas group names can consist of an arbitrary number of levels. Second, IP addresses are statically assigned and based roughly on the actual network topology, whereas the group names can be dynamically created or deleted at any time and can link an arbitrary group of sites together.

For every group name, there is a group number which is the binary representation of it. Group numbers are 64 bit addresses that are guaranteed to be unique across the network. They are formed by concatenating the 32 bit IP address for the first host to join a group along with a counter maintained by that host. Each host keeps the counter unique by incrementing it each time it is used and by initializing it to the host's current time (in milliseconds) every time that a RMP server is started at that host.

Token rings are the actual virtual circuit maintained between RMP servers, and it is the unit over which total ordering of messages is provided. RMP allows a single token ring to correspond to multiple groups, which allows ordering guarantees to be provided across groups and allows groups to be used at a finer level of classification than token rings. For example, if a set of hosts are engaged in a conference involving both a shared white board and a shared editor, a token ring could be used to denote the set of hosts and provide delivery to all of them, and separate groups could be set up for each of the active pages in the white board and for each of the active documents in the editor in order to make programming the effects of actions on these pages easier.

Groups are hierarchical in nature, and logically make up a forest of names. Each token ring corresponds to a subtree of this forest. When joining a group, a client requests whether or not it would like this group to correspond to a new token ring. If the client is the first member of this group, a new token ring is created for this group if possible. Token rings can only be created if they don't fragment a previously existing token ring. For example, if a token ring corresponds to the groups *uiuc.edu* and *wb.cb.uiuc.edu*, a new token ring could be created for a group *database.uiuc.edu* or *editor.cb.uiuc.edu,* but not for *cb.uiuc.edu.* If a ring for *cb.uiuc.edu* was created the members of the domain *wb.cb.uiuc.edu* would have to be moved from the old ring to the new ring. This would greatly complicate the membership algorithm, and so is not allowed.

Because the broadcast communication of TRP won't scale to an Internetwork, RMP is based on IP Multicasting. This service sends unreliable datagrams to multiple destinations. In the case where multiple destinations exist on a link that supports hardware multicasting, this packet can be delivered for the same price as a single IP packet to one of these destinations. The routing protocol sends each packet over an approximation of a minimum spanning tree between the destinations, only splitting up the packet as necessary.

In order to provide multicast service over multiple token rings, multiple IP Multicast sockets have to be opened, one per token ring. A pool of available {address, port} pairs is maintained by the Name Server (see below). Because the TRP protocol also sends some unicasts, and in order to support sites which do not run IP Multicast, the address for a token ring also contains the IP addresses and associated UDP port numbers for all the members of the ring, along with a flag as to whether or not each site supports IP Multicast. Together, we call this information the extended multicast address for a token ring. In order to send a multicast packet to a ring, a machine in the ring first sends a packet to the IP Multicast address and port associated with the ring, then sends a UDP packet to each of the {IP address, port} pairs that are flagged as not supporting multicast. Part of the header for each RMP packet contains the IP Multicast address for the token ring the packet is destined for. This allows the UDP packets from all token rings to be sent to the same UDP port.

Distributed Name Server

The name server provides consistent mapping of domain names to token rings, as well as keeping track of some mappings local to each server. Because it has been implemented with a clean API, it could be replaced by other systems, such as [PEA94], as they become standardized.

There are three levels of mappings kept in the Name Server (NS): global mappings, token ring mappings, and local mappings. A list of the extended multicast addresses for all of the current token rings and the top group name that each ring corresponds to must be maintained across some single global set of name servers. This information is sufficient for RMP servers to map a group name into the appropriate token ring and join that ring. Because the token ring membership operations are atomic, it is not necessary that all of this information be kept strongly consistent. Creations and deletions of a ring must be kept consistent across the name servers in order to prevent multiple instances of the same token ring, but other ring membership changes do not. The only consequence of the membership of the extended multicast addresses being inconsistent is that new members may fail when trying to join a ring which has non-multicast members in it. If this occurs, the operation can be retried until it succeeds. The global set of name servers all subscribe to the NameServer token ring and use it for all communication about the global mappings. These Global Name Servers (Global NSs) in turn can act as proxies for other Name Servers that want to make changes to the global token ring mappings. In the example shown earlier in figure 4 there are two Global NSs, one at each location. Each of these name servers handles the global mappings for the three sites at their group. Forwarding of Global NS changes is done through the uiuc and berkeley rings. This topology is vulnerable to crashes of either NS, but this could be remedied by including multiple Global NSs in each of the forwarding rings.

The second level consists of the mappings from token ring addresses to the token list, group names and group numbers for each ring. This information allows membership changes to a group name to be mapped to the appropriate token ring and group number, allows messages to a given group number to be mapped to and from the appropriate token ring, and provides the information that TRP needs to function. These mappings can be maintained across just each token ring instead of across the global NSs because RMP requires that a client be a member of any group (and its token ring) that it sends to. All of

Level	Information in Level	Mapping	Maintained Across
Global Level (top)	Existence of token rings Extended multicast addresses for rings	None Group name to ext. address	Global NSs (strong) Global NSs (weak)
Token Ring Level (middle)	Token list for TRP Group numbers for each ring Group numbers for each name in ring	Token ring to token list Group number to token ring Group names to numbers	NSs in each ring (strong consistency)
Local Level (bottom)	Clients subscribed to each group	Group number to client list	Individual NS

Table 1. Classification of NS Information

the information at this level has to be kept strongly consistent, which is done by the membership change protocol described below.

The third level of information maps local clients to and from group numbers. These mappings allow messages sent to a group number to be delivered to the appropriate set of clients, and are only maintained locally. This means that the total membership of a group is not maintained in any single spot in RMP unless there is only a single server in that ring. The implementation of this level is straightforward.

AcquireLock()
1. Send "lock ring" message to token ring
2. Wait until we receive our message, keeping track of lock status of ring
3. If ring is currently locked by another site
    a. Wait until "unlock ring" message is received from site holding lock
    b. Goto 1.
4. Else
    a. We now hold lock on token ring

ReleaseLock()
1. Send "unlock ring" message to token ring
2. Wait until we receive our own "unlock ring" message
3. Lock on token ring is now freed

ReformedRing(GroupName, ExtendedAddress)
1. Reformation process notifies all sites in ring that ring is locked
2. If token ring for GroupName is locked
    a. Query all members of reformed ring for lock
    b. If lock is not held by any site in ring
        i. We now hold lock
3. If we do not hold lock
    a. AcquireLock()
4. Send new ExtendedAdress to ring
5. ReleaseLock()

Figure 7. Locking and Reformation Algorithms for Global NSs

The global extended multicast address mappings are the essence of the Name Server, along with the interaction of these mappings with the new membership algorithm described in the next section. In order to provide strongly consistent global information, any time that one of the Global Name Servers creates or deletes one of the global mappings, it attains a lock on these mappings at all of the Global Name Servers. A site obtains a lock by sending a "lock ring" message to the global token ring and waiting until this message is delivered back to it. The total order of these "lock ring" messages provides the guarantee that only one site holds the lock at a time. The algorithm for the AcquireLock() and ReleaseLock() functions is shown in figure 8.

The Global NSs need to stay consistent even if one of them crashes. As in the original TRP protocol, any time that a site crashes the reformation protocol will be called on each of the rings it was a member of. A new consequence of this reformation protocol is that the site that performs the reformation calls (through a proxy NS if necessary) ReformedRing() at a Global NS. This routine frees up the lock if it was held by a crashed site, and updates the extended multicast address for that token ring with the new one. This algorithm is also shown in figure 8. The main operations that a Global NS performs are InitializeNS(), DeleteNS(), AddSiteToAddress(), and RemoveSiteFromAddress(). InitializeNS() is used when starting up a new Global NS. It is responsible for getting a copy of the current mappings from another Global NS, if one exists. The DeleteNS() algorithm is run when a RMP server is about to shut down, and simply removes all hosts from all extended addresses, removing addresses as necessary. AddSiteToAddress() adds a host to an extended address. RemoveSiteFromAddress() is the reverse of AddSiteToAddress(); it removes a site from an address, deleting it if necessary. These algorithms are presented in figure 9. All waiting done in these algorithms is done in a non-blocking way.

The AddSiteToAddress() and RemoveSiteFromAddress() functions interact with the AddMemberToRing() and RemoveMemberFromRing() routines below. AddSiteToAddress() is called to add a member to a ring, and it calls AddMemberToRIng(). This guarantees that if a ring is being created by AddMemberToRing(), then no other site can simultaneously create that ring. RemoveSiteFromAddress() is called from RemoveMemberFromRing() in order to guarantee that if a ring is being deleted then the address for it is simultaneously removed.

There is some flexibility in the implementation of the Global NSs. Because there is a clean API to the rest of RMP, a different name service could be substituted, so long as it

InitializeNS()
1. Link NS into global token ring
2. AcquireLock() on all Global Name Servers
2. Request a copy of global data from one of the NSs
3. Wait for copy of data, store it
4. ReleaseLock() for Global Name Servers

DeleteNS()
1. For each local GroupNumber and each Client which is a member of that GroupNumber
    a. RemoveSiteFromAddress(GroupName, Site)

AddSiteToAddress(GroupName, Site)
1. If Address for GroupName doesn't exist
    a. AcquireLock() on all Global NSs
    b. If Address for GroupName doesn't exist
        i. Get IP Multicast address from free pool
        ii. Create a new Address, send to other Global NSs
    c. Else
        i. Goto 2a
    d. AddMemberToRing(Address, Site) (see figure 10)
    e. ReleaseLock()
2. Else
    a. Send new Address info to all Name Servers
    b. AddMemberToRing(Address, Site)
    c. If AddMemberToRing fails, goto 1.

RemoveSiteFromAddress(GroupName, Site)
1. If Address for GroupName only contains Site
    a. AcquireLock() on all Global NSs
    b. If Address for GroupName only contains Site
        i.. Put IP Multicast address for token ring back in free pool
    c. Send new Address or notification of deleted Address to all Global NSs
    d. ReleaseLock()
2. Else
    a. Send new Address to all Global NSs

Figure 8. Locking and Reformation Algorithms for Global NSs

maintains enough consistent information on the existence of token rings to keep the creation and deletion of rings atomic. In the current implementation, each NS (one per RMP server) is considered a Global NS, and the second level information mappings for group numbers are kept globally as well. These are artifacts of implementation, and can be changed in a future version.

Membership Change Protocol

As explained above, the name server takes care of keeping the extended addresses for token rings current, but this does not take care of ordering the sites in each token ring or keeping track of the group names and numbers for that token ring. The first problem was solved in the TRP protocol with a general reformation protocol. This protocol uses multicast messages to repeatedly poll all of the sites, which respond if they are still a member of the token ring. RMP originally used this protocol to handle all token ring membership changes. Experience showed that under heavy load some sites do not always respond in time to these poll commands. Also, the protocol is too slow for normal use. Because of these reasons, a new algorithm has been designed for updating the token ring in response to membership changes that are not due to site failures.

To add a new site to the ring, the new site calls AddSiteToAddress() at a Global NS(), through a proxy if necessary. This checks to see if the address exists, and atomically creates it at all Global NSs if not. That procedure then calls AddMemberToRing() with the address, which starts listening to the IP Multicast address for this ring. If it is the only member in the new address it creates a new ring. Otherwise, it sends a multicast message to the ring informing the ring members that it wishes to join. When the current token site gets this message, it adds the member to the token list in the spot immediately after it, and sends a "new token list" message to the ring using the normal protocol. The token site sends the ACK for it as well, naming the new site as the next token site. Passing the token requires confirmation to the old token site that the token has been received, and this functions as a positive acknowledgment that the join operation has been completed successfully. The new site must request and receive the last N packets and ACKs before it can acknowledge the token. This maintains the resiliency requirements for the TRP protocol and allows the new member to handle requests for retransmissions. The algorithm for this is shown in figure 10.

To remove itself from a ring, a member reliably multicasts a "delete member" message to the group. Each member of the group stores these requests in a queue that it checks each time it accepts the token. If the queue is not empty at that time, the token site dequeues requests until it finds one from the next site in the ring or until the queue is empty. If a request from the next site in the ring is found, then the token site removes that site from the current token list and reliably multicasts the new list around. The current token site then sends out the ACK for this reliable multicast, naming the site after the removed site as the next token site. The effect of this algorithm is that a site is only removed when it is about to receive a token, which guarantees that the resiliency requirements are not violated. The removed site stops processing this token ring when it receives the message

AddMemberToRing

NEW MEMBER (called from AddSiteToAddress() with new address)
1. Open up socket to IP Multicast address
2. If we are only site in address
    a. Create new token ring
3. Else
    a. Send unreliable "new member" multicast to token ring
    b. Set timer
    c. Wait until timer expires or ACK is received naming us as new token site
    d. If timer has expired more than X times
        i. Return failure
    a. Else if timer expired
        i. Goto 3a.
    b. Else
        i.. Request past N messages and ACKs from old token site
        ii. Store token list that accompanied ACK
        iii. Return success to Global NS and then proceed as normal, confirming token pass by sending an ACK or a Confirm

CURRENT TOKEN SITE
1. Upon receipt of "new member" message
    a. Create new token list with new member in spot after current token site
    b. Reliably send "new token list" message to token ring, along with ACK for it

ALL SITES
1. Upon delivery of "new token list" message
    a. Store and start using new token list

Figure 9. Algorithm for Adding a Site to a Token Ring

RemoveMemberFromRing()
OLD MEMBER
1. Reliably multicast "delete member" message
2. Continue processing until "new token list" message is delivered
3. RemoveSiteFromList(Address, OldSite)

ALL SITES
1. Upon receipt of "delete member" message, store request in a queue
2. Upon receiving the token, scan queue for any requests from the next site in the ring.
3. If request exists, send "new token list message" removing site from list, along with ACK for it
4. Upon receipt of a "new token list" message, store it, start using it, and remove "delete member" requests from all sites in queue that aren't in the new token list

Figure 10. Algorithm for Removing a Site From a Ring

removing it from the list. When this occurs it has all of the messages up to the point where it is removed, and it knows that it will no longer be needed in the functioning of the protocol. After the "new token list" message is delivered, the site calls RemoveSiteFromList(). If the site is the last one in the token list, RemoveSiteFromList will delete the address for it.

Group name lists for a token ring and the corresponding group numbers are also updated by sending the changes as totally ordered messages. These mappings are also sent to the token ring whenever the membership of a token ring is changed, so that new sites get the current group mappings.

This algorithm provides virtual synchrony of group membership changes when considered together with the NS algorithms in figure 8. We offer only an informal argument for this. To show that the protocol is correct and provides virtual synchrony we must show that a membership change always terminates, that two rings with the same address can never be created, and that all membership changes to a group (including creation and deletion of a ring) are totally ordered with respect to each other and with respect to the other messages in a ring. A membership change always terminates because the global lock can never be held indefinitely by one site and because a failed add is retried with the latest version of the address. If necessary, the reformation protocol will be called to guarantee this. Two rings with the same address can never be created because the creations and deletions of a ring are globally serialized. Membership changes to a ring are serialized over that ring by using a totally ordered message to perform the change among the token ring members. Group changes are also done with totally ordered messages, and so are also serialized with respect to each other, with respect to the changes in the token ring membership, and with respect to the messages to the token ring. Together, these things provide strongly consistent, totally ordered membership changes and message deliveries at the token ring level, while only requiring the Global NSs to keep track of a minimum of strongly consistent information.

Flow Control and Congestion Control

Flow control is the policy used to make sure that the amount of data sent doesn't overrun the receivers. Closely connected to this is congestion control, which tries to keep the amount of data being sent from overloading the network. The simplest flow control policy is to send a packet and wait until an ACK is received back from the destination before sending the next one. This is called *stop and wait* flow control.

Sliding windows, leaky buckets, and buffering are all techniques that allow the sender to keep on sending data over a connection even if the previous packet has not been acknowledged yet. This is important in high throughput, high latency networks, where the high latency involved with waiting for an acknowledgment will leave the network empty most of the time if stop and wait flow control is used. Windowing occurs if we know that the receiver can always handle at least N bytes of data. Then the sender can send up to N bytes before getting an acknowledgment from the receiver. Each acknowledgment may acknowledge more than one packet worth of data. A basic leaky bucket allows an average of up to R bytes to be sent per unit of time. Each sender has a quota Q of data it can send, and every unit of time R credits are added to the quota, up to its limit. This limits the amount any sender can send over an interval I to Q+R*I. This policy is not as accurate as the sliding window policy, because it is not known for sure if we are overrunning the destination or not. However, this is useful in real time and multicast protocols where it is not practical to send positive acknowledgments back to the sender.

Sliding windows and leaky buckets are probably the two most common schemes used today for flow control. With modifications, these schemes are also used for congestion control. Some reliable multicast schemes have adopted a leaky bucket scheme, which enforces explicit rate controls on each sender, because it is impossible to use a sliding window with a protocol based on NACKs. Calculating the values for these rates is difficult, and they must divide up the bandwidth between the senders on a relatively static basis. This decreases the flexibility and throughput attainable by these schemes.

Because of these problems, RMP uses a modified sliding window protocol for both flow and congestion control. This algorithm is based on the algorithms proposed by Van Jacobson for TCP in [Jacobson88] but modifies them to take into account the fact that the ACKs in RMP are more expensive than those in TCP due to the rotation of the token site. A sliding window regulated by congestion and the ACKs from the token site is maintained by each sender. This provides congestion control. Because the Van Jacobson algorithms provide such good dynamic congestion control, RMP is also able to provide flow control with the same algorithms by treating NACKs as another form of congestion. The Van Jacobson algorithms for congestion control that are used by RMP include:

(1) round-trip-time variance estimation

(2) slow start

(3) dynamic window sizing on congestion

(4) exponential retransmit timer backoff

Round-trip-time variance estimation comes from the observation that when a network path becomes congested, the variance on packet latency becomes very high compared with the average. "If the network is running at 75% of capacity...one should expect the round-trip-time to vary by a factor of 16."[Jacob88] The proposed algorithm continually estimates this variance, and eliminates most of the spurious retransmissions while still maintaining time-outs small enough to detect dropped packets quickly.

The slow start algorithm (2) is used to linearly increase the window size from 1 packet to the maximum window size that the receiver allows that does not cause congestion, as calculated by algorithm (3). This is done by incrementing the window size by one packet each time that an ACK is received. Because the window size is constantly growing, slow-start actually increases the window fairly quickly. It will increase from 1 to W on a network with latency L in $L\log_2 W$ time.

With the assertion that the timer algorithm almost completely avoids retransmissions that are not due to lost packets and with the observation that most lost packets are due to congestion instead of errors, it follows that most expired timers signal congestion.

Algorithm (3) uses this to respond aggressively to this congestion by exponentially reducing the window size by a constant number (currently 50%) each time that a timer expires. The original protocol actually uses a two-level bound on this window in the face of congestion. It reduces the window size to one packet any time an error occurs, uses slow start to quickly build up to 50% of the level before the error, and then uses a slower linear increase to build up from there. RMP only reduces the current window size by 50% because RMP modifies the packet lengths according to the window size. The algorithm used for this causes slow start to reach a window size W (assuming W is less than 16K) in O(W) time instead of O(logW) time, and this makes the cost of reducing the window all the way to 1 packet too high.

Finally, the exponential retransmit timer backoff is used to double the timer each time it expires, resetting it to the value calculated by (1) when an ACK is finally received. Both this and algorithm (1) are applied to the timers for all three classes of positive acknowledgments. Along with further decreasing congestion, in RMP this allows an efficient detection method for failed sites. The maximum value for a timer is clamped at a certain value (currently 2 seconds). Then up to N retransmissions (currently 10) are allowed before a site is declared dead. This takes into account the policy that sites over long latency links should be given more time to respond than those over short paths, because there is less probability of sustained congestion over short paths. As an example, with the current values this policy detects nearby sites within 5 seconds, and distant sites within 15-20 seconds.

```
if (A < MIN_PACKET && A < W)          // Avoid "Silly Window" effect
        Delay sending packet until an ACK is received
S = min(P, W/2);                      // Send up to 1/2 of the window at a time
S = max(S, MIN_PACKET);               // Send at least MIN_PACKET bytes
S = min(S, A);                        // Can only send up to A bytes
S = min(S, MAX_PACKET);               // Reduce effect of lost packets
```

Figure 11. Packet Size Algorithm

These algorithms together allow RMP senders to quickly adapt their offered load to the available bandwidth. With modification, these algorithms also provide effective flow control. If a destination gets overrun by the senders, it will drop one or more packets. This will usually be detected by the destination very rapidly. At the most, it will be detected within N sent packets and hence N token passes. When this occurs, the TRP protocol has been changed so that the destination multicasts a NACK back to the group instead of unicasting it to the token site. In addition to requesting a retransmission of the packet from the current token site, it also informs the original sender (who is named in the NACK) that a destination has lost a packet from that sender. This is treated the same as an expired timer due to a lost packet, and causes the sender to decrease its window size by 50%. In order to make sure that multiple NACKs on the same packet do not each decrease the window size, a cache of the sequence numbers of the last three messages sent by this site that were dropped by another site is maintained by the sender. Incoming NACKs are compared against this cache, and the window size is only modified if a NACK for this message isn't in the cache.

A problem that RMP faces with flow and congestion control is that the rotating token site introduces a higher overhead per acknowledgment than traditional protocols such as TCP. This is compounded by the protocol being more complicated than TCP and thus requiring more processing per packet. To solve this problem, RMP uses larger packet sizes than does TCP. In an error free environment, having the IP or IP Multicasting layer do the fragmentation and reassembly is more efficient than having it done by RMP. If errors occur, the window size quickly drops to a single IP packet per RMP packet. Pseudo code to determine the size of the packet to be sent out (S), given the current window size (W), the available space in the window (A), and the offered packet size (P), is: shown in figure 12. The critical step in this algorithm is that up to half of the available window is sent at a time until the maximum packet size has been reached. This trades off a small amount of network utilization in some cases for higher efficiency of handling the packets.

CHAPTER 9

PROTOCOL OPTIMIZATIONS

Because RMP makes use of the new standard in Internet multicasting, IP Multicasting, it was predicted that this would allow effective aggregate throughputs to be achieved that are much in excess of the actual network bandwidth, which is the bottleneck in most communication today. The original version of the MBusII, however, performed well below expectations. The protocol was analyzed and optimized to get rid of the bottlenecks, and a 1500% speedup was achieved by this process.

At the beginning of the optimization process, point to point communication between two servers on a lightly loaded LAN over Sparc10's only achieved 30 KB / sec throughput rates, as opposed to the SunOS TCP/IP which achieves 1100 KB / sec (very close to the 1250 KB/sec bandwidth of an Ethernet). A communication tool such as the MBUS can be limited by four things: CPU speed, network throughput, network latency, and/or dropped packets. CPU speed was obviously the current bottleneck, as it performed this badly even in a relatively lossless, high bandwidth, low latency Ethernet. Network latency was predicted to be a problem with this protocol as it uses the equivalent of a stop-and-wait flow control system, so no windowing could be achieved. However, it was obviously not yet the case.

A performance study was done that consisted of two parts. First, the reasons why performance was so bad were analyzed and the MBusII was optimized for the class of loads it was designed to support. Second, a detailed performance analysis was done after better rates had been achieved to see how good the performance is compared to other known systems such as TCP, MBusI, and ISIS. This study also studied the remaining bottlenecks in the system. Both parts showed very good results. A speedup of over 1500% was achieved, and aggregate throughputs of over 540% of network capacity have been measured. A full study of performance in a LAN environment has been done, leaving performance testing and optimization in a WAN environment as a future goal.

Analysis Metrics, Factors and Goals

The first step in any performance study is to determine the metrics and factors involved with and important to that study.

Performance Metrics

There are four main possible metrics for messaging tools: average throughput, average latency, number of messages sent per second, and the amount of bandwidth used per byte transferred. Average throughput is the most critical in this environment. As explained earlier, latency doesn't matter critically for the target clients, and will have a high correlation with throughput anyway. The number of messages per second is important, but will be directly proportional to throughput because of message batching. The amount of bandwidth used per message is important because low bandwidth leaves more bandwidth for other applications and because if the system bottlenecks on the network, this factor will limit throughput. Therefore, throughput was used as the primary metric for most of the experiments. Latency was used as a secondary metric for comparison to other systems that aren't designed primarily for CSCW applications.

Performance Factors

There are quite a few factors that could affect the performance of messaging between clients, but these can be broken up into the categories of load, topology, and CPU processing. There are way too many factors to analyze, so the relative importance had to be determined. Because the performance of the MBusII was so incredibly bad at the beginning of this study, this could not be done through a partial factor analysis. There was also no time to simulate it, so this had to be done through analysis. The factors, and their believed importance, follow.

Load

- The number of message publishers (sources) at one time. **Probably Important**
- The number of message subscribers (sinks) at one time. **Probably Important**
- The sizes and types of messages sent. This is not very critical because small messages can be batched into larger ones if they start queuing up. **Not Important**

Topology

- The number of servers. **Probably Important**
- Whether they are all on a LAN or scattered over a WAN. **Probably Important**, but have not yet been able to get access to the necessary machines to test it on a WAN.
- The number of servers per LANs. If not considering WANs, this reduces to the number of total servers. **Not Important**

- The topology of the WAN. Both the average and the distribution of the latency, throughput, and error rate for each link of the network. Error rates are almost always very low on LANs and so are unimportant there. **Not Important**
- The interlinking of multiple token rings. The rings should act independently. **Not Important**

CPU Processing

- Efficiency of code execution (instructions / packet). **Critically Important**
- CPUs and architectures of the machines the servers are run on. **Very Important**

Profiling and Optimization

At the beginning of the study, the servers were totally CPU bottlenecked. Therefore, the speed of code execution was the obvious problem to be optimized, especially since no other types of optimizations seemed feasible. The goal was to minimize the amount of CPU time needed to send and receive each byte. By repeatedly finding and eliminating CPU bottlenecks, it was hoped that the system would eventually bottleneck on the capacity of the network or else on the blocking of the stop-and-wait ACK scheme used, which in fact eventually occurred.

Methodology

What factors are important to this optimization? In the comparison of RMP (and consequently the MBusII) to other messaging tools, the worst case load is when a single user connected to one server sends to another user on a different server. This load does not allow the multicasting to be used to any advantage. Again, small messages or large messages will eventually be the same through message batching, so large messages were used.

The LAN topology is the worst hit by CPU bottlenecks, which is what the optimizations were targeted at fixing. The LAN topology is also the most important case for current users. Therefore by optimizing for this one case, the most benefit for all cases could be achieved.

The main tool for this optimization was gprof. This typically showed a 10% CPU performance penalty, which was deemed small enough to give accurate results, unlike options for profiling such as generating large trace files while processing the messages. At least 1MB (5 MB after version 5) of 4K (8K after version 9) messages were sent in each test to minimize the effect of initialization costs and make the sample statistically meaningful. Each optimization round involved running the optimized code with the given test file and topology, then analyzing the data through gprof to find the CPU bottlenecks, removing those, and then repeating.

Another tool that was used was some print statements that occurred each time 10 time-outs of a given type occurred. Since this was supposed to occur very infrequently, it didn't significantly affect performance, yet because each alarm indicates a large performance hit it provided a good deal of important information and also gave some indication of when these were happening relative to the other servers and the position in the test file.

Version	Throughput	Bottlenecks Removed for This Version
1	30 KB/sec	Original version
2	56 KB/sec	Removed artificial 5% packet loss
3		Replaced custom Cpy() routine with memcpy() call
5	97 KB/sec	Replaced sprintf() call with custom Int_To_Str() call
		Changed packet size from 512 bytes to 4KB
8	180 KB/sec	Removed 80% of ualarm calls
12	335 KB/sec	Changed packet size from 4KB to 8KB
		Implemented adaptive time-outs
14	420 KB/sec	Optimized out delays in testing programs
Final	466 KB/sec	Turned off profiling

Table 2. Bottleneck Optimizations

The third tool was the utility top, which shows the amount of CPU time being used by each process. This allowed me to check the process statistics to make sure that CPU bottlenecking was still occurring in the MBusII. On two occasions (spurious alarms and inefficient testing code) it pointed out problems.

Results

Quite a few bottlenecks were found and removed. The history and effects of the major ones are shown below in table 2.

Routine	Version 1	Version 2	Version 3	Version 8	Version 12
cpy	14.0 / 3.8	17.0 / 4.1			
memcpy	1.2 / 2.3	3.5 / 1.8	6.4 / 2.9	26.2 / 16.7	21.6 / 16.6
malloc, free	3.3 / 1.6	1.9 / 2.2	2.3 / 2.6	2.4 / 4.5	2.1 / 9.7
sendto, write	12.0 / 8.9	11.4 / 1.2	17.3 / 13.8	10.3 / 5.1	18.0 / 12.3
recvfrom, read	4.1 / 6.8	7.5 / 5.8	8.6 / 5.5	9.7 / 9.1	18.5 / 9.2
sprintf	4.6 / 3.0	4.2 / 3.0			
ualarm	3.7 / 14.8	3.6 / 9.5	4.9 / 13.5	1.4 / 4.0	2.2 / 4.3
select	3.2 / 6.4	3.3 / 7.8	3.9 / 7.6	.9 / 5.8	2.1 / 6.0
sigblock, sigsetmask	4.3 / 5.3	2.6 / 3.0	1.4 / 2.9	0.9 / 3.0	0.9 / 1.6
gettimeofday	1.6 / 2.5	0.5 / 0.4	1.0 / 2.1	0.4 / 0.5	0.5 / 0.7
PROFILING	14.6 / 7.2	11.4 / 10.7	13.3 / 11.0	8.2 / 9.3	10.8 / 10.4
OTHER	33.4 / 37.4	33.1 / 40.5	43.4 / 38.1	38.6 / 42	23.3 / 29.2
Network Calls	16.1 / 15.7	18.9 / 17.0	25.9 / 19.3	20.0 / 14.2	36.5 / 21.5
Memory Calls	18.5 / 7.7	22.4 / 8.1	8.7 / 5.5	28.6 / 21.2	23.7 / 26.3
Other System Calls	17.4 / 32	14.2 / 23.7	11.2 / 26.1	4.6 / 13.3	5.7 / 12.6

Table 3. Profiling Statistics (% sender / % receiver)

Table 3 shows the 15 calls that were the top CPU users, and how they changed over the course of the optimization. Notice that memory routines took a lot of cycles at the beginning of the process, then eased up when memcpy was used instead of cpy, but as packet sizes got bigger the usage of the memory routines went back up. Also, the percentage of time spent in network calls increased the whole time (except in version 8), as packets got bigger. The percentage of time spent in other system calls in general reduced over the course of time, as did the amount of time in the non-system calls.

In the last trace, there are a couple of little bottlenecks (namely alarm and maybe the select/sigblock/sigsetmask routines) that could maybe be trimmed with some decent effort. However, almost a quarter of the program is being spent in memory routines. These should be mostly unnecessary in this tool, and were long suspected to be a problem. However, the question remained as to which part of the program was doing all the memory copies. The profiling showed that the high level MBus code was calling memcpy far more than RMP was, but many of these were for single byte transfers. The profiler could not handle memcpy correctly because it was a assembler file, and so there was no way to check where the percentage of time was being spent. Cpy() was put back in instead of memcpy(), and the time being spent on calls to copy are mostly in the MBus code, assuming that the switch from memcpy() to Cpy() didn't affect this statistic. To verify this, I wrote a testing application that uses RMP directly without all of the upper level code. It turns out, as the next section will show, that the raw RMP runs twice as fast as the MBusII program. This was actually very good news! Since a much lighter weight MBus layer could (and probably will) be built on top of RMP, we could stand to treat RMP by itself for analysis and comparison. When profiling was turned off for RMP, which removes 10% of the CPU load, the throughput remained identical, at 840 KB /sec. This indicated that the processor was probably no longer the bottleneck, and that optimization had been a success. Indeed, the analysis below validates RMP and the design decisions made.

CHAPTER 10

PERFORMANCE ANALYSIS

After the CPU bottlenecks were optimized out as much as possible, the second step in the performance study was to analyze the performance of the system in a LAN environment. Both the raw RMP package and the heavyweight MBusII package were tested. This analysis aimed to:

- Compare their performance to the performance (in terms of both throughput and latency) of other systems, notably ISIS and MBusI.
- Compare their performance (aggregate throughput only) to the theoretical maximums for non-multicast systems, and characterize the effects of the different factors on the performance of RMP.
- Determine to what extent the throughput stays constant as the number of destinations increase.
- Determine which of the factors are most important to performance.
- Determine the amount to which it is bottlenecking on the network bandwidth.

Methodology

The most important factors to messaging performance, as discussed earlier, are the number of sinks, the number or sources, the number of servers, the topology, the type of machine used, and obviously the program being run. Unfortunately, for testing I was only able to get access to 8 Sparc Station 10's and 4 Sun IPC's, all of which were on the same network. Therefore, the important variables reduced to the number of sources, number of sinks, number of servers, the machine being used, and the program being tested. As a simplification, the number of servers was set to be equal to the number of sinks because each server has to receive all the messages to a ring, and so it doesn't make sense to say a server is not a sink. The number of sources has to be less than or equal to the number of servers (and hence the number of sinks).

Once network saturation is reached, the performance of non-multicasting systems has to drop off proportional to 1/(# sinks * # sources - 1), which is the minimum number of copies needed of each packet to provide total ordering and delivery to all sinks. Under these conditions, RMP is expected to stay roughly constant, verification of which was one of the analysis goals. Exponentially increasing numbers (1, 2, 4, and 8) were used for the number of sinks and the number of sources. Both latency and throughput were measured as metrics for RMP and the MBusII. For now, only the SS10s were used, because they were fast enough to run RMP without really bottlenecking the CPU, and because we only had access to 4 IPCs.

Throughput was the primary metric being studied, and it was computed by calculating the amount of time needed to send a given large data file. Since both RMP and MBusII are tightly flow controlled, and given the 5MB data files used, this is very close to the amount of time taken to receive the data. The amount of buffering allowed added some error to this measurement, but it is less than 0.5% with a file this large.

Latency was also used as a metric for most of the tests along with throughput. To measure the latency, very small packets were sent, and because of the stop-and-wait acknowledgments, average latency is equal to one over the number of packets per second.

With all of these variables, this gives a total of 4 (# sources) * 4 (# sinks) * 2 (# of metrics) * 3 (# of packages) * 3 (replications) * 2 (platforms) = 576 trials needed for a full factorial design! This would look to be a great candidate for a partial factorial design, except that some of these trials didn't make sense (as explained above), and the number of levels for each variable was not the same. So a partial factorial design wasn't practical. Instead, I used the 10 permissible combinations of sources and sinks, the 2 metrics (except when measuring the MBusI which had built in batching already), 3 replications, and 3 packages. This allowed all but one of the analysis goals to be met--that of quantifying the relative effects of the different factors. It was decided that that could be done after the first

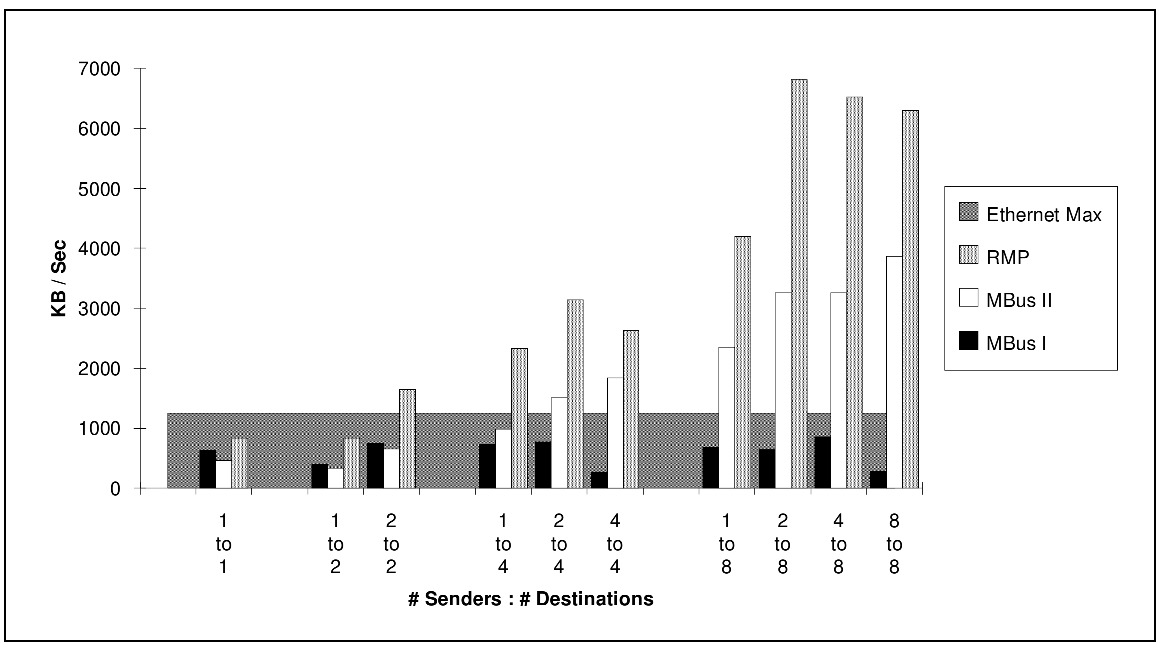

Figure 12. Aggregate Throughput (KB/sec)

study if time permitted. This gave 10 (source/sink combinations) * 2 (metrics) *5/6 (no need for MBusI latency testing) * 3 packages * 3 replications = 150 trials. Three replications were of each trial were done to increase the accuracy of each trial. While we do not show the standard deviations for each experiment, they were quite small for all of the cases except for 4 and 8 senders, where the Packet Starvation Effect [WhStFe94] was causing the Ethernet to drop a significant portion of the packets sent. Under this case, the network has an extremely high standard deviation in latency, so consistent results are not possible without extremely long runs.

Analysis of Data

Figure 12 shows the aggregate throughput of the network as a function of the number of sources and destinations. Aggregate throughput is the throughput the user sees. It is computed by taking the amount of data sent from all of the senders and multiplying it by the number of destinations. In these tests, the sender is also a destination. This is done so that it can see its own messages totally ordered with the others. This is the way that most groups use totally ordered multicast, and is actually the worst case for the throughput of the protocol because of the increased CPU load of the senders. A single-server, TCP/IP based system, the MBusI, is shown for comparison. The maximum bandwidth of the Ethernet used in this study is also shown. For the case of 2 sources and 8 sinks, RMP achieved an aggregate throughput of 6810 KB/sec. This is 5.45 times the bandwidth of the Ethernet. It is impossible for any solution that does not use either multicast or broadcast to achieve any result that breaks the Ethernet throughput boundary this way.

In figure 13, we see the single sender throughput plotted against the number of destinations. The single sender throughput is equal to the aggregate throughput divided by the number of destinations. To be compatible with other published figures, in these tests the source was separate from the destinations. Data from the MBusI is included for comparison. The performance of all applications (such as UDP ISIS, Sun ToolTalk, the MBusI, and RPC) that do not use hardware broadcast or multicast drops off as a factor of 1/N. The graph is plotted with a logarithmic axis for throughput to better show this limitation, which is a fundamental limit of the network. In contrast, RMP stays roughly constant regardless of the number of destinations. RMP breaks the fundamental unicast limit for two destinations, so has fundamentally higher throughput in this environment for all groups of more than one destination. We have not included any numbers from other reliable multicast protocols because no fair comparisons on the same platforms have yet been made.

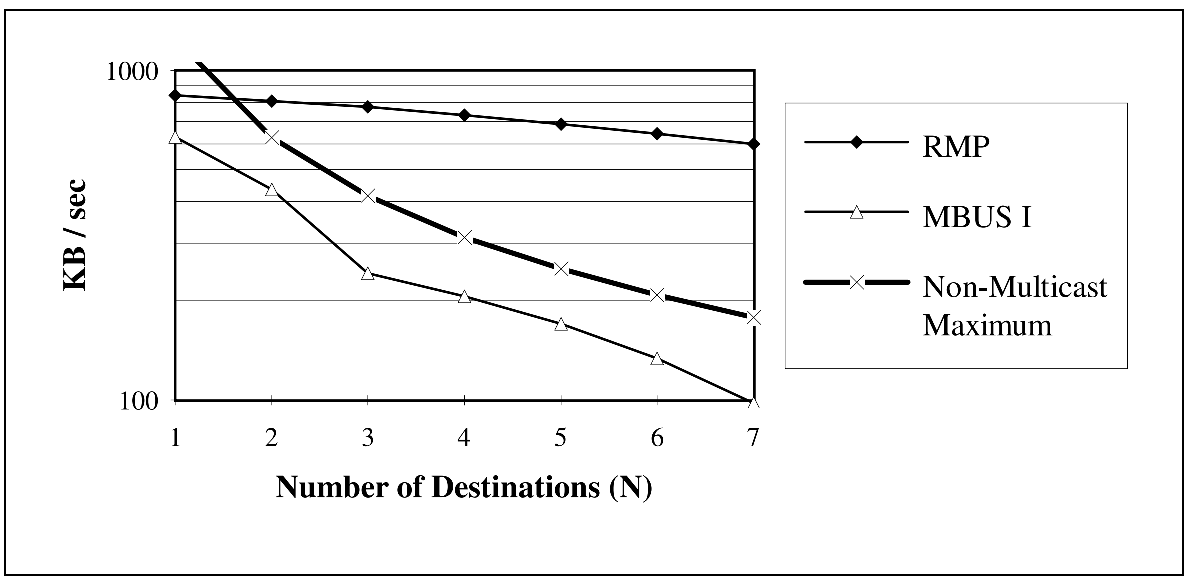

Figure 13. Single Sender Throughput (Logarithmic Scale, KB/sec)

In figure 14, we see the same factors, but with latency as the metric instead of throughput. Here again, the performance of RMP stays almost constant. While we have not yet been able to make fair comparisons to other protocols, the data that we have shows that the latency of protocols that do not take advantage of hardware multicast also scales linearly as a function of N [Clark94]. Note that this graph shows the case when fault tolerance is disabled and K=0. K does not affect throughput, but does increase latency. As K increases, the latency under heavy load will increase by a factor of roughly (K+2)/2. As the load decreases, the latency increases due to the increased time between packets and thus token passes.

The worst case for RMP latency as compared to RPC latency is when the application needs to send a short message to a single destination. In this case, the round trip RPC should take about the same amount of time as RMP, but the message can be delivered twice as fast to the destination as the RMP message because RMP must wait for the ACK in order to provide total ordering. It is a minor change to allow users to select some messages to be delivered immediately. This would not guarantee ordering or atomicity in the face of site failures, but would allow RMP to deliver messages immediately, making its latency to a single destination roughly equal to that of RPC and other low latency protocols. In this case, we expect RMP to have lower latency to multiple destinations than these protocols which scale linearly with N.

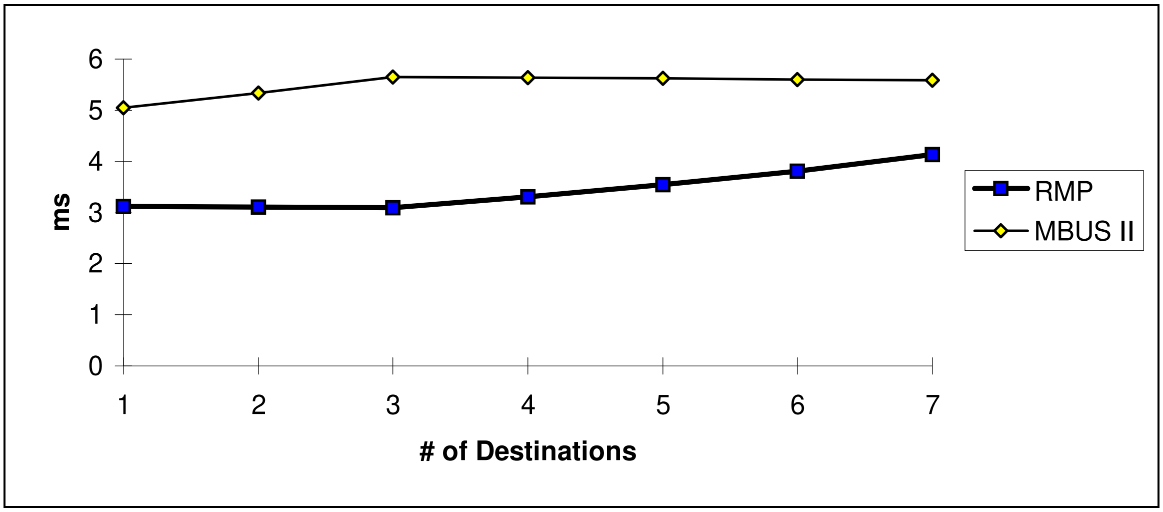

Figure 14. Single Sender Latency (ms)

In figure 15, we see a plot of the network utilization of the three systems (MBusI, MBusII, and RMP) under the 10 loads. This was computed by taking their individual throughputs, multiplying these by the amount of overhead required for that protocol, and then dividing by the total bandwidth of the Ethernet. In this case, utilization varies some across topologies, but it is clear that RMP has the highest utilization, followed by MBusI and then MBusII. Because the MBusI is based on TCP sockets, utilization should theoretically remain close to that observed for FTP (92%). However, it doesn't, so we can conclude that the CPU processing time on each packet is still a factor in performance. More analysis is needed on RMP, but it appears to still be bottlenecking on the CPU as well. This isn't that bad, however, because even if it performs slightly worse than FTP for 1-1 cases, we have seen in the previous graphs that it excels once the multicast feature can be taken advantage of. The MBusII utilization confirms the inefficiency of the code used as a front end to RMP. Combined with the processing that must be done by RMP, the burden on the CPU is enough that it is still a dominant factor in the performance of the MBusII.

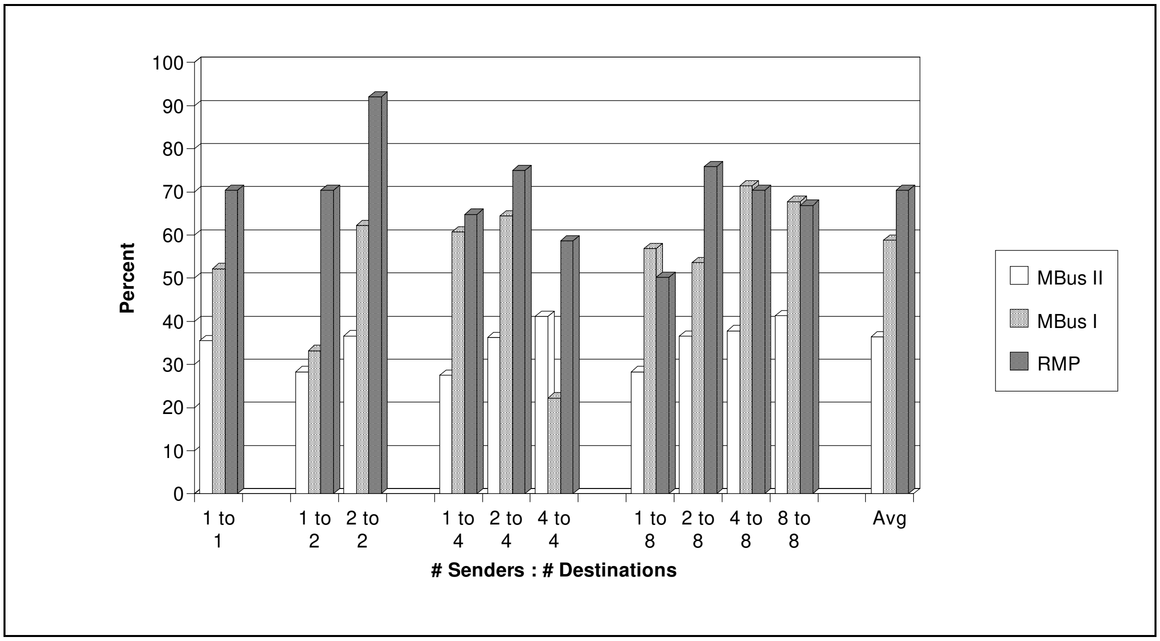

Figure 15. Network Utilization (percent)

Conclusions of Performance Measurements

The optimization and performance analysis on the MBusII and RMP are both extremely heartening. The data has shown:

- That RMP lives up to its design goals in terms of its performance in this environment. Even though it provides total ordering and atomicity of multicast messages, it can deliver better throughput rates to multiple destinations than any other solution we know of.
- Because the network utilizations are so high (median of 70%) it doesn't appear that RMP's lack of windowing will hurt it in a LAN environment.
- Both tools have long per packet latencies, but they remain roughly constant with the change in the number of destinations for each message, as opposed to most other solutions which scale linearly with the number of destinations.

- Individual throughput decreases as a roughly linear function with a shallow slope of the number of destinations.

Efficiency of IP Multicasting

As previously explained, IP Multicasting uses both link level multicasting and multicast routers to provide unreliable datagrams to multiple destinations much more efficiently than IP and other non-multicast based datagram protocols. The preceding section demonstrates that RMP and MBusII exploit this efficiency to provide reliable group communication across a LAN for less cost and with higher throughput than any solution that does not take advantage of IP Multicasting. The performance of RMP and MBusII in an internetwork situation is more complicated. The analysis of the TRP protocol[ChMa84], on which RMP is based, shows that on average, the cost of the protocol is slightly more than 2 multicasts per packet. A solution that takes advantage of sliding windows can achieve a point to point reliable stream with slightly more than 1 unicast per packet. Other solutions, such as RPC protocols, require at least 2 unicasts per packet. In practice, RMP actually provides something close to sliding windows by using packets that are up to 8KB long. Since TCP/IP packets are typically 1KB long [Postel80], a stream based approach must maintain a window of more than 8 packets order to be more efficient than RMP without windowing, and the proposed windowing scheme would remove even this advantage. The question of how much more efficient an IP Multicast packet is, as opposed to multiple IP packets, then becomes the critical question in determining how much more efficient RMP is over other, non-multicast based approaches.

Figure 16 shows some simple sample network topologies that demonstrate the efficiency of IP Multicasting. For the purposes of this analysis, we define a site as one or more interconnected networks that may span multiple buildings, but with no two hosts separated by more than 5km of cable. Examples of typical sites are university campuses and office buildings. The two most common expected types of usage of RMP are where all users are located within a site and where there are groups of users at multiple widely separated sites. Where all users are located in the same site, the factor of efficiency of IP Multicasting over IP is at least equal to the average number of members that are on the same LAN. For communication between widely separated sites, the factor of efficiency is at least equal to the average number of group members per site.

For simplicity of analysis, we show the source as always being attached to one of the LANs that has another host on it. This is actually the worst case for IP Multicasting. Moving the source to another location that connects in to one of these networks increases the advantage of IP Multicasting over IP by increasing the sharing of links that IP Multicast achieves. As an additional simplification, we assign a integral cost to each link. This cost metric may correspond to the latency of the link, the monetary cost of the link, the demand placed on the link, the throughput provided by the link, or something else. In practice, these factors all have a high correlation. In general, the longer the latency of a link, the more expensive it is, the more in demand it is (more people share the use of it), and the lower the throughput it provides.

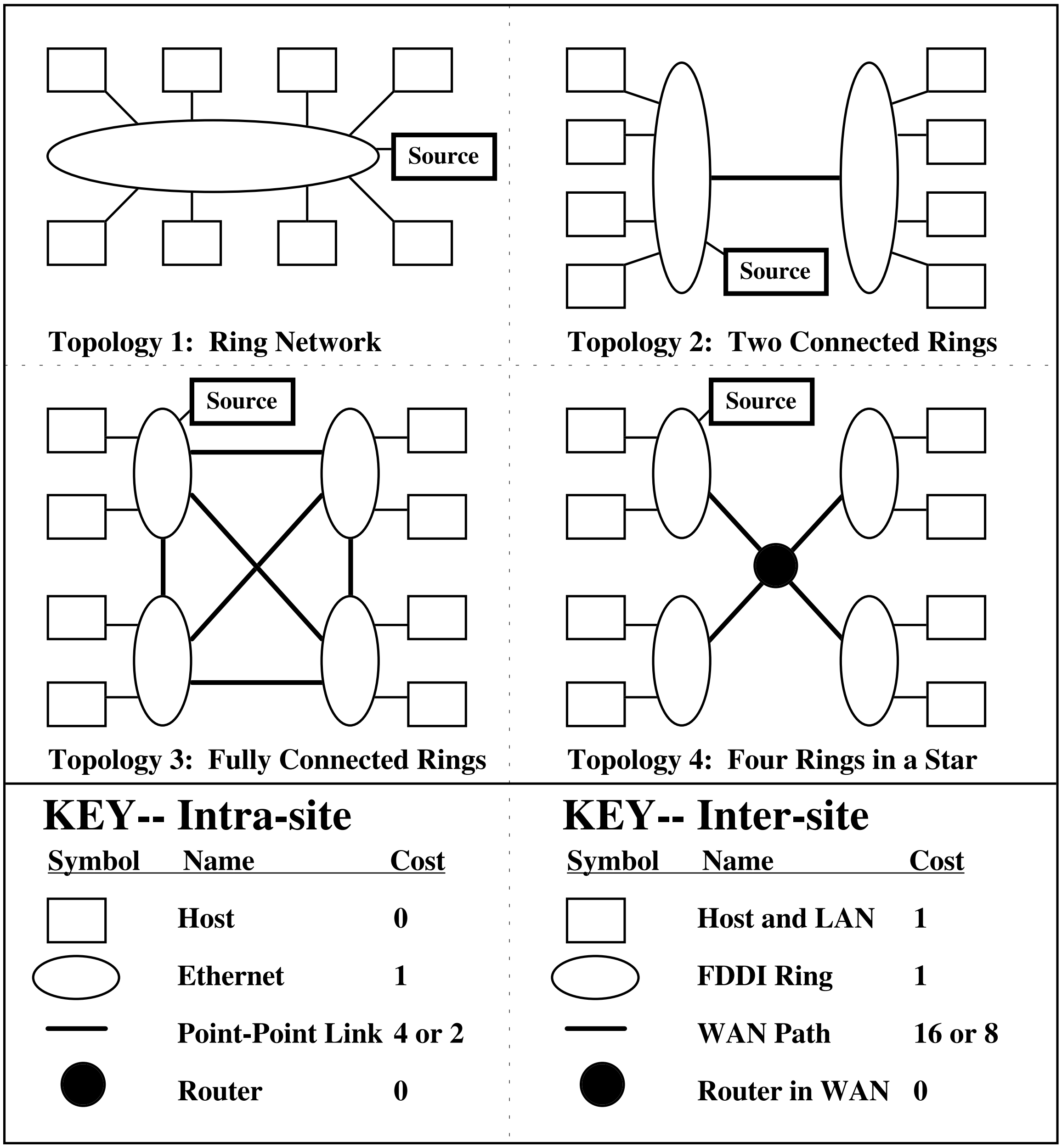

Figure 16. Sample Network Topologies

For communication within a site, Figure 16 shows four different topologies of a group of 8 destinations on 10 MB/sec Ethernets connected by point to point links (short leased lines). These point to point links could also correspond to networks of other types such as FDDI. The cost of these links is 4, except in topology 4 where the cost per link is 2 due to the central router in the middle of all of the Ethernets. This example corresponds roughly to one or more connected buildings, each with their own Ethernet.

Topology 1 consists of a single Ethernet, with all 8 of the destinations on it. Because the Ethernet provides hardware multicast support, the multicast cost for this topology is 1. In contrast, IP has to send one packet to each host, so its cost is 8. Topology 2 consists of two connected Ethernets, and so the multicast cost is the cost of sending a packet from the source to a destination on the far bus. Topology 3 consists of four fully connected Ethernets, and Topology 4 consists of four Ethernets that connect through a central router. Topology 4 is more realistic, but topology 3 is shown as a worst case for IP Multicasting. The efficiency advantages of topologies 1-3 are all attained through hardware multicasting within a single link. Topology 4 also takes advantage of multicast routing, the other main source of efficiency in IP Multicasting.

Table 4 summarizes the efficiency gains made by IP Multicasting over IP for these topologies. Topologies 1-3 which only take advantage of the hardware multicasting all show an efficiency increase at least equal to the average number of hosts per LAN. This will always be the minimum efficiency increase achieved. Topology 4, which also takes

Topology	Multiple IP Packets	IP Multicasting	Efficiency Factor
1	8 * 1 = 8	1	8x
2	4 * (4+1+1) + 4 * 1 = 28	4 + 1 + 1 = 6	4.33
3	6 * (4+1+1) + 2 * 1 = 38	1+ 3 * (4+1) = 16	2.38
4	6 * (4+1+1) + 2 * 1 = 38	1 + 4 * 2 + 3 * 1 = 12	3.17

Table 4. Intra-Site Cost Calculations

advantage of the multicast routing, shows a jump from a factor of 2.38 to 3.17. This increase is difficult to quantify in general, because it varies significantly from topology to topology. Current research is attempting to quantify this advantage better.

Table 5 shows a summary of the efficiency of IP Multicasting in three topologies that connect different sites together. For this case, figure 17 shows sets of sites connected by a large WAN such as the Internet. Each site consists of an FDDI ring with a number of Ethernets connected to it. One group member is then connected to each Ethernet. These topologies generalize the advantages of IP Multicasting to situations where the hardware

Topology	Multiple IP Packets	IP Multicasting	Efficiency Factor
2	4*(1+1+1) + 4*(16+4) = 92	1 + 2*1 + 8*1 + 16 = 27	3.41
3	2*(1+1+1) + 6*(16+4)= 126	1 + 4*1 + 8*1 + 16*3 = 61	2.07
4	2*(1+1+1) + 6*(16+4)= 126	1 + 4*1 + 8*1 + 8*4 = 45	2.8

Table 5. Inter-Site Cost Calculations

link level multicasting provides little advantage. Even so, the numbers show that IP Multicasting provides a significant improvement over unicast solutions. As a rough rule of thumb, for a set of H hosts spread over S sites that are widely separated, IP Multicasting will provide an efficiency increase of at least H/S. In the three topologies studied, the efficiency increase ranged from H/S to 1.7*(H/S), depending on the amount that the WAN links from the source are shared by the destinations.

While less than comprehensive, these calculations generalize the efficiency advantage that RMP has from just LAN environments to WAN environments as well. They show that RMP and the MBusII will be more efficient than unicast solutions for a wide range of group topologies. Since network bandwidth is typically the bottleneck in communication, this will result in superior performance for RMP as well.

CHAPTER 11

CONCLUSIONS AND FUTURE WORK

Building the MBusII and RMP was a very long but educational experience. The project was very successful. RMP was the first protocol to demonstrate that for group communication to two or more destinations, a reliable multicast protocol could provide total ordering over packets with latency similar or better than most any other solution, and with throughput clearly better than any other solution. This goes against the common wisdom, which has long held that totally ordered delivery and virtual synchrony are very slow.

While not achieving widespread use, both the MBusII and RMP were fully implemented and debugged. The final project contained well over 20,000 lines of C code, of which around 6,000 were borrowed from the original TRP and MBusI implementations.

In retrospect, the biggest mistake of the project was that RMP was built on top of some very old and buggy code from the original TRP implementation. While the code itself was crucial to understanding the TRP protocol, it would have saved a huge amount of time to have thrown it all out and rebuilt it from scratch. Even after many man-months of debugging this code, there were always a few bugs, and this was the biggest reason the code for the first version was never widely distributed.

The two biggest lessons that I personally learned were that debugging of large systems (particularly of large distributed communication protocols) is extremely difficult and time consuming, and that you really have to build a throw-away prototype of any system before you will be happy with it. While both of these lessons are taught in the classroom, it takes a large project of your own for them to really sink in.

To solve the first problem, this project really brought home the need to formally specify a protocol ahead of time, and very carefully verify it as you go along. To solve the second problem, a second version of RMP has been designed [WhMoKa94, Mont94]. The new RMP has been redesigned from the ground up, and provides a great deal more functionality and performance than version 1. At the time of this publishing, a very capable student, Todd Montgomery of West Virginia University, is finishing up the last of the implementation. In a LAN, the performance of the new protocol is 30-60% better than version 1, and looks to be one of the best solutions for WAN group communication in existence.

A partial list of the changes made for version 2 include:

1) V2 provides an implicit naming service that maps textual group names into communication groups.

2) Instead of only providing totally ordered, K-resilient packet delivery, RMP allows the user to select from the entire set of ordering and reliability guarantees described in chapter 2, including agreed and safe delivery. These are selectable on a per-packet basis.

3) For increased scalability, V2 allows processes that are not members of a group to send messages to it, and receive replies to messages, through its multi-RPC mechanism.

4) The TRP membership algorithm handles all cases, and takes an extended period of time to run. Like V1, V2 optimizes for the common membership change case with the addition of a new membership algorithm that allows non-failure membership changes to be made at the cost of only a single group message. However, this algorithm has been reworked from scratch so that no global nameservers are needed.

5) V2 uses a new membership algorithm that handles faults. It is faster, less likely to mistakenly remove processes from a group, and supports virtual synchrony and extended virtual synchrony.

6) To facilitate replicated services, V2 provides a set of mutually exclusive handlers for messages. A message can request that it be handled, and at most one process will reply to the message.

7) V2 provides a windowed flow and congestion control mechanism that allows RMP to provide high performance over both LANs and WANs, even in the face of congestion.

8) V2 seamlessly extends reliable multicast to hosts that do not support multicast.

9) The second version of the protocol was fully specified before the first line of code was written, and then built from scratch. This has allowed it to be implemented in less than 6 man-months.

10) Concurrently with its implementation, a separate team is formally verifying the specification of the protocol.

BIBLIOGRAPHY

[ADKM91] Y. Amir, D. Dolev, S. Kramer and D. Malki. "Transis: A Communication Subsystem for High Availability." *Technical Report CS9113*, Hebrew University of Jerusalem, Nov. 1991.

[AFM92] S. Armstrong, A. Freier, K. Marzullo. "Multicast Transport Protocol". *RFC1301*. (February, 1992).

[AMSM92] D. A. Agarwal, P. M. Melliar-Smith, and L. E. Moser. "Totem: A protocol for message ordering in a wide-area network." *Proceedings of the First ISMM International Conference on Computer Communications and Networks* (San Diego, CA, June 1992). pp. 1-5.

[BaFrCr93] T. Ballardie, P. Francis, J. Crowcroft. "Core Based Trees (CBT) An Architecture for Scalable Inter-Domain Mulicast Routing". *Proceedings of the 1993 SIGCOMM Conference* (San Francisco, CA, September 1993).

[BLNS82] A. Birrell, R. Levin, R. Needham, M. Schroeder. "Grapevine: An exercise in distributed computing." *Communications of the ACM*, 25, 4, April 1982. pp. 260-274.

[BiJo87] K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures." *ACM Transactions on Computing Systems.* 5, 1 (Feb. 1987). pp. 47-76.

[BSS91] K. Birman, A. Schiper, P. Stephenson. "Lightweight Causal and Atomic Group Multicast." *ACM Transactions on Computer Systems.* 9, 3 (Aug. 1991). pp. 272-314.

[BiCl94] K. Birman, T. Clark. "Performance of the Isis Distributed Computing Toolkit." *Technical Report TR-94-1432, Dept. of Computer Science, Cornell University.*

[Birman93] K. Birman. "The Process Group Approach to Reliable Distributed Computing." *Communications of the ACM.* December, 1993, 36,12. pp. 37-53.

[CaMo94] J. Callahan, T. Montgomery. "A Decentralized Software Bus based on IP Multicasting." *Proceedings of Third Workshop on Enabling Technologies: Infrastructure For Collaborative Enterprises*, Morgantown, WV, April 17-19, 1994, pp. 65-69.

[Carroll93] Alan Carroll. "ConversationBuilder: A Collaborative Erector Set." *Ph.D. Thesis*, *Department of Computer Science, University of Illinois,* 1993.

[ChMa83] J. M. Chang and N. F. Maxemchuk. "A Broadcast Protocol for Broadcast Networks." *Proceedings of GLOBCOM* (Dec. 1983).

[Chang84] J. M. Chang. "Simplifying Distributed Database Systems Design by Using a Broadcast Network." *Proceedings of SIGMOD* (June 1984). pp. 223-233.

[ChMa84] J. M. Chang and N. F. Maxemchuk. "Reliable Broadcast Protocols." *ACM Transactions on Computer Systems.* 2, 3 (Aug. 1984). pp. 251-273.

[ChZw85] D.R. Cheriton and W. Zwaenepoel. "Distributed Process Groups in the V-Kernel." *ACM Transactions on Computer Systems.* 3, 2 (May 1985). pp. 77-107.

[Clark94] Tim Clark. *ISIS work in progress*. Cornell University, Department of Computer Science, 1994.

[CrPa88] J. Crowcroft, K. Paliwoda. "A Multicast Transport Protocol". *Proceedings of ACM SIGCOMM '88.* pp. 247-256.

[Deering89] S. Deering. "Host Extensions for IP Multicasting". *STD 5. RFC1112.* (August 1989).

[DKM93] D. Dolev, S. Kramer, D. Malki. "Early Delivery Totally Ordered Multicast in Asynchronous Environments." *23rd Annual International Symposium on Fault-Tolerant Computing (FTCS).* (Toulouse, France, June, 1993). pp. 544-553.

[Jacob88] V. Jacobson. "Congestion Avoidance and Control." *Proceedings of ACM SIGCOMM '88 Symp* (Sept. 1988). pp. 314-329.

[JHC94] P. Jain, N. Hutchinson, and S. Chanson. "A Framework for the Non-Monolithic Implementation of Protocols in the x-kernel." *Proceedings of USENIX High Speed Networking (August 1994).* pp. 13-30.

[KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, H. E. Bal. "An Efficient Reliable Broadcast Protocol." *Operating Systems Review.* 23, 4 (Oct. 1989), pp. 5-19.

[Lamp78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM.* 21, 7

[MaCh84] N. F. Maxemchuk and J. M. Chang. "Analysis of the Messages Transmitted in a Broadcast Protocol." *Proceedings of the International Computer Conference* (Amsterdam, May 1984). pp. 1263-1267.

[MeBo76] R. M. Metcalf. and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM.* 19, 7 (July 1976). pp. 395-404.

[MMA90] P. M. Meillar-Smith, L. E. Moser, D. Agrawala. "Broadcast Protocols for Distributed Systems." *IEEE Transactions on Parallel and Distributed Systems.* 1, 1 (Jan. 1990). pp. 17-25.

[Mont94] T. Montgomery. "The Design and Implementation of the Reliable Multicast Protocol". *M.S. Thesis, West Virginia University.* December 1994.

[PEA94] S. Pejhan, A. Eleftheriadis, D. Anastassiou. "Distributed Multicast Address Management in the Global Internet." *Submitted to IEEE Journal on Selected Areas in Communication* on March 1, 1994.

[PBS89] L. L. Peterson, N. C. Buchholz, R.D. Schlichting. "Preserving and Using Context Information in Interprocess Communication". *ACM Transactions on Computer Systems.* 7, 3 (Aug. 1989). pp. 217-246.

[Postel80] J. Postel. “Transmission Control Protocol”. *RFC 761.* (January 1980).

[Purtilo85] J. Purtilo. "Polylith: An Environment to Support Management of Tool Interfaces." *ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, WA, June 25-28, 1985. pp. 12-18.

[TNML93] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. "Implementing Network Protocols at User Level." *IEEE/ACM Transactions on Networking*, 1(5), 1993. pp. 554-565.

[Verissmo92] Verissmo. "xAMp: A Multi-primitive Group Communications Service. *Proceedings of the 11th Symposium on Reliable Distributed Computing.*

[WhKa94] B. Whetten, M. Kalfane. “A Fast Track Architecture for High Performance TCP/IP”. *Work in progress.*

[WhMoKa94] B. Whetten, T. Montgomery, and S. Kaplan. “A High Performance Totally Ordered Multicast Protocol”. To be published in the *Springer-Verlag Proceedings of the 1994 Dagstuhl Workshop “Unifying Theory and Practice in Distributed Systems”*. Dagstuhl, Germany (September, 1994).

[WhStFe94] B. Whetten, S. Steinberg, and D. Ferrari. "The Packet Starvation Effect in CSMA/CD LANs and a Solution". *Proceedings of IEEE Local Computer Networks Conference* (Minneapolis, Minnesota, Oct. 1994).

APPENDIX A: MBUSII USERS MANUAL

TODO: Add this in