UNIVERSITY OF ILLINOIS AT CHAMPAIGN-URBANA

THE MESSAGE BUS II AND THE RELIABLE MULTICAST PROTOCOL

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF COMPUTER SCIENCE
IN CANDIDACY FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ENGINEERING

BY

BRIAN WHETTEN

CHAMPAIGN-URBANA, ILLINOIS

FEBRUARY, 1995

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS v

LIST OF TABLES vi

ACKNOWLEDGMENTS vii

1. INTRODUCTION 1

2. BACKGROUND 7

IP Multicasting 7

Ordering Guarantees 8

Virtual Synchrony 10

Reliability and Guarantees 11

Fault Tolerance 12

3 MOTIVATION 15

Performance Metrics 17

History and Advantages of the MBusI 19

Problems With the MBusI 21

Additional Requirements For the MBusII 22

Design Decisions 24

4. PREVIOUS WORK 26

Classification of Approaches 26

Summary of Previous Approaches 27

MBusI 27

ISIS 28

Transis 31

Totem 33

ToolTalk 34

Multicast Transport Protocol (MTP) 35

CCITT T.122 -- Multipoint Communication Service for Audiographics and Audiovisual Conferencing 39

The IP Multicasting White Board 40

Other Protocols 40

5. DEFINITION OF TERMS AND ASSUMPTIONS 42

System Model 42

6. SYSTEM USAGE 44

Token Rings 47

Sample MBus Syntax 48

7. SYSTEM ARCHITECTURE 51

8. RELIABLE MULTICAST PROTOCOL DESCRIPTION 54

Token Ring Protocol Description 54

Extensions To The Token Ring Protocol 61

Multiple Token Rings and Groups 62

Distributed Name Server 64

Membership Change Protocol 69

Flow Control and Congestion Control 73

9. PROTOCOL OPTIMIZATIONS 79
Analysis Metrics, Factors and Goals 80
Performance Metrics 80
Performance Factors 81
Profiling and Optimization 82
Methodology 82
Results 84
10. PERFORMANCE ANALYSIS 87
Methodology 87
Analysis of Data 90
Conclusions of Performance Measurements 94
Efficiency of IP Multicasting 95
11. CONCLUSIONS AND FUTURE WORK 101
BIBLIOGRAPHY 104
APPENDIX A: MBUSII USERS MANUAL 108

LIST OF ILLUSTRATIONS

Figure Page

Figure 1. Classification of Reliable Multicast Approaches 27

Figure 2. Scalability and Efficiency of Approaches 28

Figure 3. Interconnection of Clients and Servers 46

Figure 4. Use of Token Rings 48

Figure 5. MBusII Architecture 52

Figure 6. Operation of RMP Normal Phase 59

Figure 7. Locking and Reformation Algorithms for Global NSs 67

Figure 8. Locking and Reformation Algorithms for Global NSs 69

Figure 9. Algorithm for Adding a Site to a Token Ring 71

Figure 10. Algorithm for Removing a Site From a Ring 72

Figure 11. Packet Size Algorithm 77

Figure 12. Aggregate Throughput (KB/sec) 89

Figure 13. Single Sender Throughput (Logarithmic Scale, KB/sec) 91

Figure 14. Single Sender Latency (ms) 93

Figure 15. Network Utilization (percent) 94

Figure 16. Sample Network Topologies 97

LIST OF TABLES

Table Page

Table 1. Classification of NS Information..... 66

Table 2. Bottleneck Optimizations 84

Table 3. Profiling Statistics (% sender / % receiver)..... 84

Table 4. Intra-Site Cost Calculations..... 98

Table 5. Inter-Site Cost Calculations..... 99

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 long 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 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 at ARMY CERL in Illinois for providing financial support for version 2.

Dedicated to Version 2 of the Reliable Multicast Protocol

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

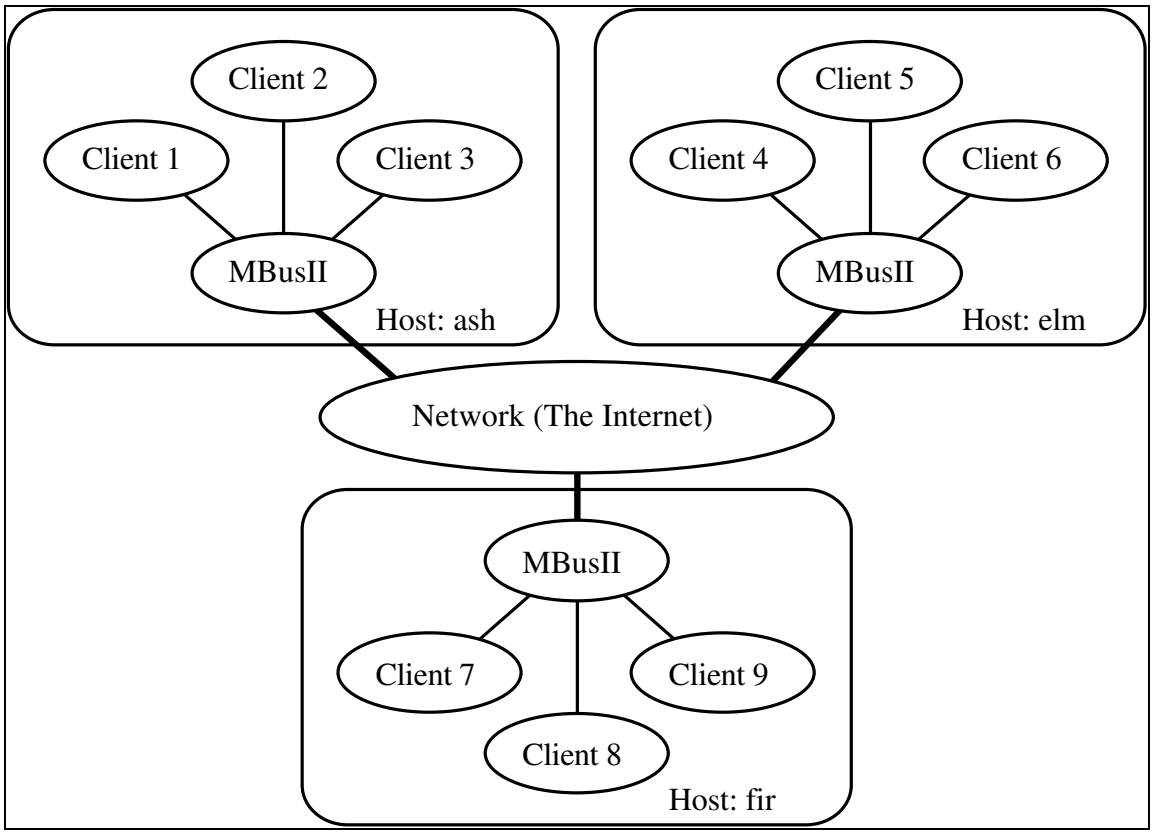


Figure 3. Interconnection of Clients and Servers

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

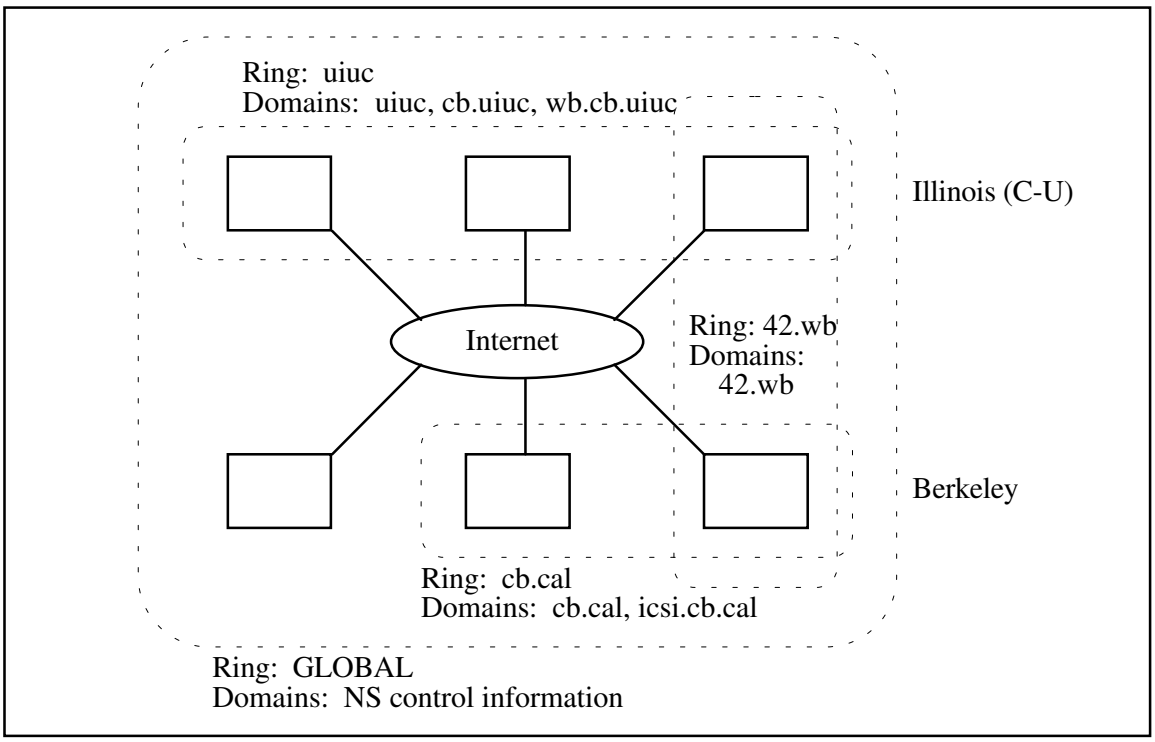


Figure 4. Use of Token Rings

maintained across multiple domains, and this can only be done if the domains are all part of the same token ring.

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.

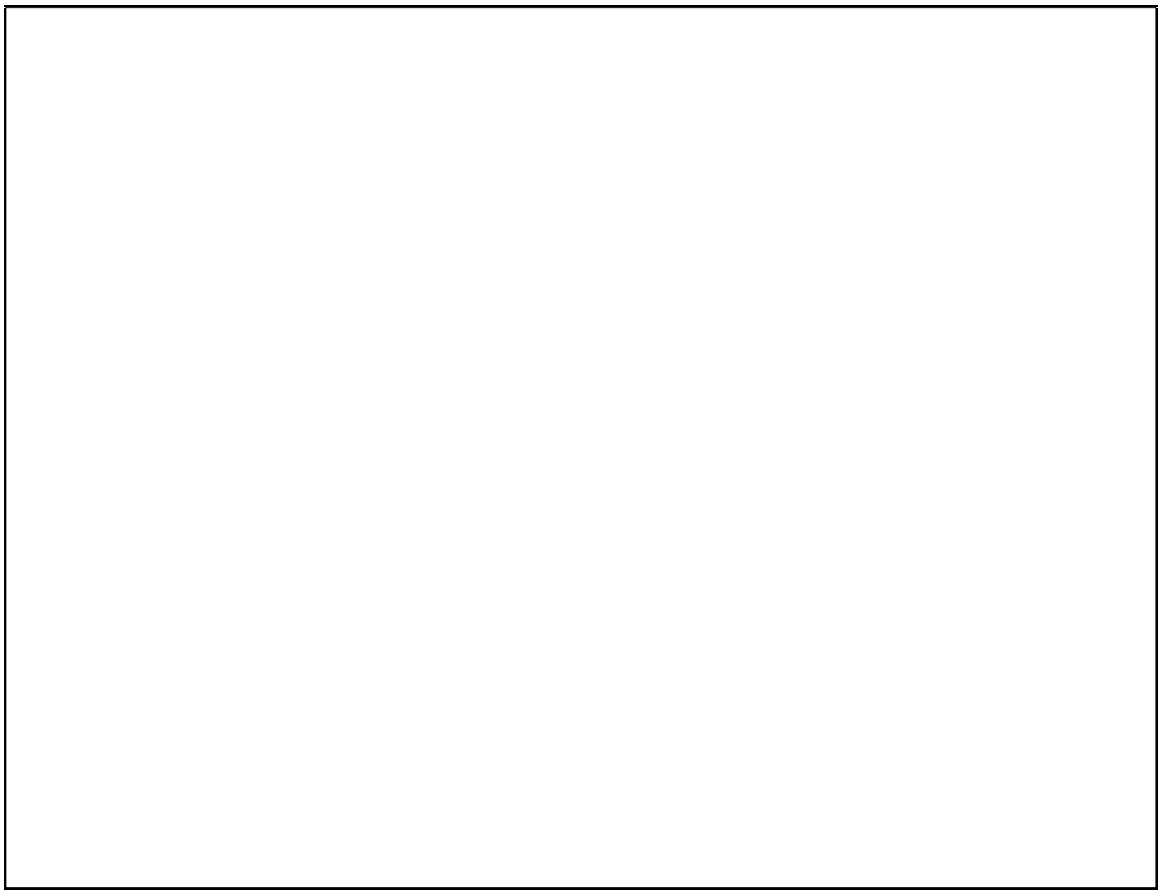
The OSI transport layer consists of RMP library and the NS library. Together, they provide a reliable, distributed, atomic, fault-tolerant, multicasting messaging facility over multiple domains.

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



Offending Command = pp_clip
Error = nametype : undefined
this name is not defined in a dictionary
Stack =