NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

Efficient Kernel-level Reliable Multicast Communication in Distributed Systems

by

Garnik Bobloian Haftevani

B.Sc., Simon Fraser University, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

December 1988

ISBN 0-315-48745-3

Approval

Name : Garnik Bobloian Haftevani

Degree : Master of Science

Title of Thesis: Efficient Kernel-level Reliable Multicast Communication in Distributed Systems

Dr. Jia-Wei Han
Assistant Professor
Chairman

Dr. Stella M. Atkins
Assistant Professor
Senior Supervisor

Dr. Woshun Luk
Associate Professor

Dr. Gerald Neufeld
Assistant Professor
External Examiner
University of British Columbia

Dec 8 88

Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Efficient Kernel-level Reliable Multicast Communication in Distributed Systems.

Author:

(signature)

Garnik B. Haftevani

(name)

December 8, 1988

(date)

Abstract

Multicast communication allows a Sender to send a message to a group of Receivers and, depending upon the intent of the message either none, some, or all of the receivers will respond. Multicast communication in a distributed system connected by a Local Area Network can reduce the amount of network traffic compared to that required for a sequence of one-to-one transactions; moreover, it can reduce the number of context switches in the Sender required by an equivalent sequence of one-to-one message exchanges, and it can also provide a greater functionality than one-to-one communication.

Instead of requiring the Sender to know each receiver to which it wishes to send a message, the Sender directs a message to a named group of Receivers. The membership of the named group is maintained by the system in a decentralized distributed data structure, maintained by each host's kernel in the network. This allows receivers to be specified by function without requiring the Sender to know the specific members of the group. Contrary to computer folk-lore, we found that the overhead of providing reliable multicast over a single Local Area Network was very small, mainly due to the fact that our reliable protocol operates at the kernel level rather than the user level.

This thesis describes several forms of reliable multicast communication expressed as simple message-passing communication primitives, and illustrates the effectiveness of our protocol through an example of a distributed application. Performance analyses and actual performance data of the protocol are presented.

To my parents

Acknowledgments

I am thankful to Stella Atkins for her critique of my work and for teaching me EVERYTHING about distributed systems. Discussions with Woshun Luk were also very helpful.

I am grateful to the host of professors at the School of Computing Science for making available their computing hardware to me. Steve Cummings' technical support and work on the micro Vax II is greatly appreciated. Also, Tony Speakman and Tamara Badkerhanian's close reading of the thesis and their thoughtful questions and comments improved this thesis.

This research was supported in part by Mobile Data International.

Table of Contents

Approval ii

Abstract iii

Acknowledgments v

Table of Contents vi

List of Figures vii

List of Tables viii

1. Introduction 1

2. Related Work 5

3. Design Issues 8

3.1 Reliability Semantics 8

3.2 Reliable Multicast Communication Primitives 10

4. Detailed Design and Implementation 16

4.1. Reliable Local Group Communication 16

4.2 . I_Am_Here Server 19

4.3. Reliable Global Group Communication 22

5. Performance 31

5.1. Experiment Environment 31

5.2. Test Models 32

5.3. Timing Results 34

5.4. Performance of ALL_REPLY and K_REPLY Communication Primitives 37

5.5. Performance of ALL_DELIVER and K_DELIVER Communication Primitives 39

6. Effect of Reliable Communication Primitives on Distributed Algorithms 42
6.1. Static Environment 44
6.2. Dynamic Environment 55
7. Conclusion and Future Work 61
7.1 Possible Improvements and Future Work 62
Appendix A. Improved User Interface for Reliable Communication Primitives 63
References 65

List of Figures

Figure 1. A Request-Response Message Transaction 1

Figure 2. V User Level Message Format 12

Figure 3. Logical inter-process interactions for local groups 17

Figure 4. Reliable Intra-kernel protocol of K_DELIVER and ALL_DELIVER 17

Figure 5. Reliable Intra-kernel Protocol of K_REPLY and ALL_REPLY 18

Figure 6. I_Am_Here Server Data Structures 20

Figure 7. Logical Inter-Process Interactions For Global groups 23

Figure 8. Reliable Inter-kernel protocol of K_DELIVER and ALL_DELIVER 26

Figure 9. Reliable Inter-kernel Protocol of K_REPLY and ALL_REPLY 28

Figure 10. Hardware environment 31

Figure 11. Performance of ALL_REPLY and K_REPLY Sends 37

Figure 12. Performance of ALL_DELIVER and K_DELIVER Sends 40

Figure 13. Unisend Algorithm for Static Environment 45

Figure 14. Unreliable Multicast Algorithm for Static Environment 47

Figure 15. Unreliable Multicast Algorithm 48

Figure 16. Reliable Kernel Algorithm 51

Figure 17. Network Message Complexity Analysis Summary 53

Figure 18. Time for each transaction using three algorithms 54

Figure 19. Process Interactions for Unisend Algorithm in Dynamic Environment 56

Figure 20. Unisend Algorithm for Dynamic Environment 57

Figure 21. Unisend Algorithm for Dynamic Environment(cont.) 58

Figure 22. Unreliable Multicast Algorithm for Dynamic Environment 59

List of Tables

Table 1 - Comparison of standard V-kernel and enhanced V-kernel 34

Table 2 - Performance of ALL_REPLY and K_REPLY Sends 37

Table 3 - Performance of ALL_DELIVER and K_DELIVER Sends 40

Table 4 - Time for each transaction using three algorithms 54

1. Introduction

Distributed computer systems consisting of several workstations connected over a Local Area Network (LAN) provide many advantages over centralised systems, including cost/performance benefits and enhanced reliability through replication of resources. In such distributed systems, communication mechanisms become a major design issue. These communication issues range from low level services such as packet exchanges, to high level ones that provide atomicity and ordered delivery of messages. In general, because of data buffering and asynchronous communication between protocol layers, these higher level mechanisms are considerably more costly than the basic low-level mechanisms used by the kernels themselves[Clark85].

The standard method of communication between processes in a message based operating system is one-to-one communication in which a Sender sends a message to a specific Receiver which usually responds with a reply message. One such request-response message transaction between a Sender and Receiver process is illustrated below:

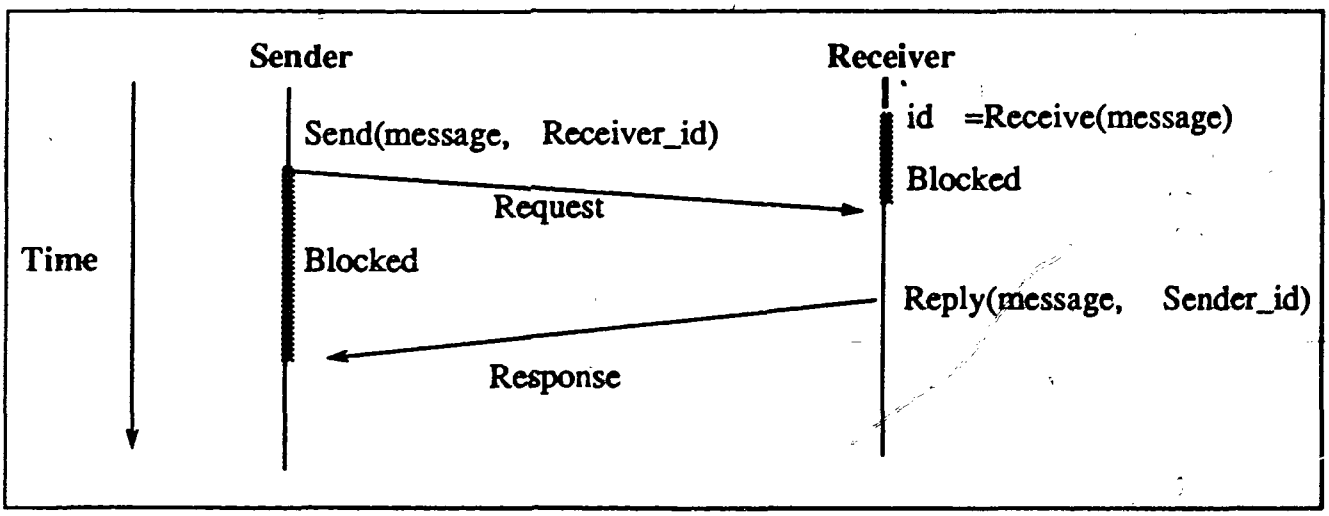

Figure 1. A Request-Response Message Transaction

An extension, called *multicast communication* allows a Sender to send a message to a group of Receivers and, depending upon the intent of the message either none, some, or all of the Receivers will respond. Multicast communication is an elegant and useful com-

munication mechanism for distributed systems; it is claimed that multicast is a fundamental operation of distributed applications [Cheriton87]. Our experience confirms this view. We are particularly interested in providing an efficient multicast implementation which offers reliable delivery and reply where the sender is assured that the message has been delivered to all or k of the receivers or replies from all or k of the receivers have arrived.

There are many applications that can use reliable multicast communication. One such application occurs in the implementation of replicated files or databases where reliable multicast is needed to modify all the copies of data [Schlichting85]. For example, consider the update of a small distributed database, copies of which are maintained on each host in order to ensure robustness in case of host failure. In order to perform an update, a process first queries the database managers (DBMs) on each host to obtain a lock on the item to be updated. Each DBM will reply indicating whether or not they agree to the lock. Clearly this query must be performed reliably in order to ensure that all the database managers consent to the lock. Once it is confirmed that each DBM does consent then a notification containing the update is sent to each DBM. This notification must also be reliable to make certain that each DBM receives the update.

Many linguistic constructs for concurrent programming such as the co..oc in Synchronizing Resources [Andrews82], [Andrews88] could be implemented using a reliable multicast mechanism. Furthermore, many parallel computation algorithms which require synchronization or exchange of partial results, such as distributed chess, and a myriad of distributed algorithms like selection of the k-th smallest element of a file distributed among sites of a communication network[Santoro83] require reliable multicast.

Cheriton and Zwaenepoel [Cheriton85] have presented a model for multicast communication in the V-kernel [Cheriton84]. They propose that processes may join groups to which members may multicast a single message. Their model, however, only provides a kernel-level unreliable "best efforts" delivery for multicast messages. Hence, reliability

must be provided at a higher level to enable distributed applications such as those described above to exploit reliable multicast communication. Thus, if a Broadcasting process wishes to receive every reply from a group of processes, the Sender must explicitly maintain a list of group members, or have access to such a list from a group coordinator, an inherently expensive mechanism.

The weak level of reliability provided in the V-kernel can be used by itself in some applications. For example, if a process wants to locate a particular server, e.g. print server, it may multicast a message to the group of servers and the particular type of server(s) will respond to the query giving their name, current load, etc. The Sender then may either choose the first replier or wait and choose the lightest loaded server. However, the latter case is difficult to implement at the user level as the Sender does not know the number of members in the group. There are also applications that only use the multicast for notification like name publishing or distributed computation applications. In the case of name publishing applications, each server registers itself with the local name server which in turn multicasts that information to the other name servers; this mechanism is used in the V system. This multicast is not essential but it provides additional information to help the name servers in locating servers.

Many applications requiring efficient reliable multicast have to contend with the complications and inefficiencies of implementing reliability at the application level, and thus have not been implemented on distributed computer systems except for research purposes. The designers of the V system argue that such higher degrees of reliability would be too costly to implement and would heavily tax the performance of the applications that do not use such features.

In this thesis, the design, implementation and analysis of the performance of an extremely efficient kernel level reliable multicast in the V distributed system is presented. This reliable multicast is based on the design in [Atkins86] with major extensions.

This protocol does not add significant overhead to the performance of normal one-to-one (1:1) communication, and which does not add to the cost of so called *1-reliable multicast*, where at least one Receiver is guaranteed to reply. To achieve this, our protocol for reliable multicast is implemented efficiently *inside* the distributed kernel. The protocol assumes that the kernels are connected over a single LAN, so that each machine on the LAN running that kernel receives every broadcast packet for which reliable delivery is specified. Our protocol also assumes that a group of Receiver processes can be accessed by a single *group_id*, and that the knowledge of group membership is distributed in that each local host on the LAN "knows" only the group members on its local host. In absence of global group membership information by the Sender, we require that every host on the network acknowledge every reliable multicast message. This actually adds very little overhead in terms of delay because the acknowledgments from kernels with no members arrive before the genuine replies to the multicast messages, and the kernels with members piggyback the acknowledgments on replies. Also, these acknowledgments are dealt with very quickly by the Sender's kernel. Kernel level reliability avoids extra messages between user and kernel level which are required for implementing reliability at the user level. Performance data assure us that the processor and network costs are insignificant compared with the alternative of providing reliability at a higher level [Navaratnam88].

Furthermore, we have developed an integrated approach to the protocol design so that the application dictates the level of reliability required. Through tailoring the reliability to the needs of the application, those applications which can tolerate unreliable datagram broadcasts can still exploit this efficient feature with scarcely any overhead.

This thesis is organized as follows. Chapter 2 relates our kernel protocol to other multicast protocols. Chapter 3 describes high level design issues and the reliable group communication primitives which allow the Sender to specify the degree of reliability required for multicast messages. Chapter 4 presents the detailed design and the implementation of reliable multicast communication. Chapter 5 presents the performance results. Chapter

6 describes a distributed application which utilises the reliable kernel-level multicast, and compares it with the alternatives of user-level reliability, and the concluding remarks are given in Chapter 7.

2. Related Work

Usefulness of reliable group communication has prompted substantial amount of research. Before we look at some of proposed designs and implementations of reliable group communication, we need to define some common terms:

- Atomicity - An operation is atomic if either it is performed in its entirety or is not performed at all.
- Network partitions - A network partition occurs when groups of processors can not communicate with each other because of communication link or site failures.
- Ordered delivery - Deliveries are ordered when they arrive in the same order in which they were sent or when they are delivered in the same order at all receivers.

The approach presented in this thesis is quite different from the method in [Chang84] where the responsibility of reliability is with the Receivers; Receivers take turns in acting as a broadcast co-ordinator to provide resiliency in case of site failures. The co-ordinator is responsible for sending the acknowledgments to the Sender and it also has to service lost message requests from the other Receivers. Moreover, extensive actions are taken in case of site failure when a reformation phase is initiated to elect a new co-ordinator. Our protocol needs no such recovery procedure.

Other systems such as LOCUS [Walker83] provide full reliability for updates of replicated files to the extent that the operating system contains a great deal of code dealing with and attempting to recover from node and network failures.

Communication reliability plays an important role in distributed applications such as distributed data bases. Two such systems, A System for Distributed Databases (SDD-1) [Hammer80][Rothnie77] and A Distributed Database Manager (DDM) [Walter82], depend on a reliable network service providing guaranteed delivery, atomicity, and a network clock. Also, a site monitoring mechanism, similar in function to the I_Am_Here server, is used to maintain the global site status. However, their system does not use

multicast.

ISIS [Birman85] provides reliable communication mechanisms with atomicity and ordered delivery features at the kernel level. These extra features tax the system performance [Birman87]; our reliable protocol is much more efficient.

There are several multicast communication mechanisms in UNIX-like environments. [Ahamad85] presents one such implementation which is integrated with the UNIX sockets and provides group membership management in case of network partitions and site failures. However, his system is based on an unreliable datagram service, and offers no reliable delivery or reply mechanism. COCANET [Rowe82] is another of the enhanced UNIX systems which uses multicast. However, it uses virtual circuits to obtain reliability.

INCAS (INCremental Architecture for distributed Systems) multicomputer project [Nhemer87] provides process groups, like V, and multicast communication. Similar to our design, in INCAS the Sender process defines the success conditions for multicast operations. Their definition of success is the number of Acknowledgments (ACKs) expected by the Sender which is equivalent to the K_REPLY multicast operation. The functionality of the ALL_REPLY, ALL_DELIVER, and K_DELIVER[1] is not provided in INCAS. Also, the Sender has to know how many Receivers there are in the system.

[Ravindron87] discusses the usefulness of variants of ALL_REPLY and K_REPLY group communication mechanisms in distributed systems. Also, a taxonomy of reliable broadcast protocols is given in [Narayavan85] including a quasi reliable broadcast mechanism. This protocol uses a periodic mechanism to distribute group membership information.

Garcia-Molina has initiated several reliable multicast schemes at Princeton Universi-

1 - K_REPLY, K_DELIVER, ALL_REPLY, ALL_DELIVER will be described in the Reliability Semantics section.

ty. In [Garcia88a] a reliable broadcast protocol is described which is built upon an unreliable multicast mechanism. However, the algorithm is designed to execute on a wide-area network, and it contains fault-tolerant features concerning network partitioning which are not necessary in our LAN implementation. No performance data are given, but the algorithm's performance will certainly not be competitive with ours as our performance data show only a few percent overhead compared with the basic unreliable multicast of the V system. In [Garcia88b] an ordered reliable broadcast protocol is described. The actual performance data are not presented, but clearly there will be an extensive overhead for managing the message propagations.

In [Navaratnam88] a reliable group communication mechanism is described, which is based on the V system, and which is designed for failure atomicity and for ordered reliability. It is implemented entirely above the V-kernel, at the user level, and so the performance data are an order of magnitude more costly than ours, as user-level inter-process communication takes around 2 msecs, which their scheme requires. The data given in that paper refers to the unenhanced V-kernel with pre-delay (See Performance chapter), so although the percentage penalty for their reliable group broadcast appears small (around 20%), on our no-delay kernel, the percentage overhead is several hundred.

3. Design Issues

3.1. Reliability Semantics

From experience gained in programming applications which use multicast inter-process communication, we found that it is the Sender process which has the knowledge of how reliably the multicast message must be received, and the knowledge of how to deal with reported failures. For example, in a database application the Sender of a message may not require replies from all members; replies from a majority of the members may suffice. We therefore take the approach that the Sender process must specify the reliability of the multicast message. For efficiency, if the Sender wishes to employ a reliable multicast, the reliability should be provided by the kernel. Furthermore, Senders which do not require high degree of reliability should not be penalized at run-time with the extra overhead of the reliable communication mechanisms. To meet these requirements, our design specifies that the reliability of the multicast message must be specified in the Send message by the Sender.

The word reliability has been used quite often in the computing literature, often with different definitions. The definition of reliability used in this thesis is that a communication mechanism is k-reliable, where 0 <= k <= number of group members, when the initiator of the communication primitive is assured that the desired action (e.g. message delivery) has been done by at least k members. There are distributed operating systems like LOCUS[Walker83] which provide full reliability, where k is all the members of the group. LOCUS uses this reliability to keep its replicated files up-to-date. On the other hand, the V-kernel supports a 1-reliable multicast feature[Cheriton85]. This means that the system guarantees that at least one process receives and replies to a message. V system does this by retransmitting the Send packet when no reply is received after a timeout. If there is no reply after several retransmission then Send returns with failure. This provides a simple and efficient implementation for multicast communication.

The designers of V state that higher degrees of reliability would either require the Receiver to keep track of messages and request missed messages (which does not work well in the case of lost request messages), or require the Sender to have access to the list of members of the group which is against one of the V design philosophy of minimization of global information[Cheriton85].

However, we have designed an efficient mechanism to achieve higher degrees of reliability without depending on the two above mentioned methods. Our multicast provides Sender-initiated, non-atomic, non-ordered reliable multicast. This is done by providing ALL_REPLY, K_REPLY, ALL_DELIVER, and K_DELIVER options for the Sender.

For K_REPLY multicast, the Sender is blocked until K replies arrive from the group members, where as for ALL_REPLY the Sender is blocked for the replies to arrive from all the group members. All the group members is the set of processes that are *alive* on the reachable operating hosts at the time of transmission of the message. K_DELIVER and ALL_DELIVER are counterparts of the above mentioned primitives with the success condition changed to the delivery of the message to the group members. For example, an ALL_DELIVER Send succeeds when the message is delivered to all the group members.

In this scheme, if a Sender requests an ALL_REPLY or ALL_DELIVER reliability and the multicast message cannot reach a node on the network, the operating system will report FAILURE to the Sender, but will make no *automatic* attempt to recover from the site failure. We assume that the processors are *fail-stop* processors in that a fail-stop processor never performs an erroneous state transformation due to a hardware or operating system failure; instead, it simply halts and all its volatile information is lost[2]. We therefore assume that when a host is not reachable, it is not up; this is a reasonable

2 - If the failed processor is rebooted with a server process which joins a group as a Receiver in the middle of a reliable transaction, although the processor is now reachable, reliable Send to that group will still fail.

assumption for reliable local area networks. Also, the problem of network partitions is not addressed here since it is not a major issue in a LAN environment using broadcast mediums where no groups are split on opposite sides of a gateway. The issue of group multicast for LANs with gateway is an active research area[Deering88].

The design goals for our reliable multicast communication are consistent with those of the V-kernel namely efficiency, simplicity, and minimization of global information. In the V-kernel the one to one and one to group communication mechanisms are handled by the same code; our added features did not alter this design to provide simplicity, and avoid duplication. In general, efficiency of the implementation was favored over space saving, although minor efficiency was sacrificed in favor of simplicity. Adhering to the V philosophy, the amount of global information is kept to a minimum; it is limited to the list of alive processors maintained by the I_Am_Here server which will be discussed later.

3.2. Reliable Multicast Communication Primitives

The V-kernel is a distributed message-based operating system with the hosts communicating over a local area network. Process groups and group management routines have already been implemented in the V system[Cheriton85], so we used that as a basis and focussed our attention on providing an efficient reliable multicast protocol in the kernel. Associated with each process is a globally unique process identifier (PID). Each process may communicate with any other process on any host by directing a message to the Receiver's PID. After the message is sent, the Sender is blocked until the Receiver receives the message and responds with a reply. Multicast communication in the V system is performed by specifying the group identifier of a group or processes to which a message is to be delivered. The operating system then uses its membership information about the group to deliver the message to each member. Membership of a group is dynamic in that processes may join or leave a group at any time. Full details on the V system group management routines are available in [Cheriton85].

We modeled the user level communication primitives as much as possible to those of the V system, so only the Send primitive is affected by the addition of reliability. We now describe the reliable Send primitive. The other relevant group communication primitives are given for completeness.

Send(message, id)

The fixed length message is sent to a single process if the id is a process id or to a group of processes if the id specifies a group, in which case multicast is used. The multicast Send blocks for zero or one replies depending on the *syscode* and *code* fields defined in the message. The new features are specified in the same way. The user specifies the type of Send by setting appropriate fields in the message. The **syscode** field may be set to K_REPLY, K_DELIVER, ALL_REPLY, or ALL_DELIVER. For K_REPLY or K_DELIVER the **code** field must contain an integer specifying the value of K, where K >= 1[3]. For the new reliable multicast Send, the Send blocks until either conditions specified by the syscode field are satisfied or conditions are found unsatisfiable, in which case failure is returned.

The format of a user level message is shown in Figure 2.

SYSCODE
FILL
CODE
UNSPECIFIED
DELIVERY
SEGMENT PTR
SEGMENT SIZE

Figure 2. V User Level Message Format

3 - For K=0, reliable communication is inappropriate and the standard V-kernel datagram can be used.

The syscode field types for the new features of group communication are defined below:

K_REPLY

Send blocks until K replies are received from the group members specified by id, in which case SUCCESS is returned, and the Sender's message is overwritten by the contents of the first reply. Otherwise FAILURE is returned because K replies are not possible, i.e. either the maximum number of timeouts and retransmissions have been done or the number of members is less than K. In either case, Send sets the *code* field in the message to the number of replies received.

K_DELIVER

Send blocks until the message is delivered to K group members specified by the id, after which a SUCCESS is returned. Otherwise FAILURE is returned because delivery to K members is not possible, i.e. either the maximum number of timeouts and retransmissions have been done or the number of members is less than K. In either case, Send sets the *code* field in the message to the number of members to which the message was delivered. Note that the Sender is unblocked after the status is returned from the kernel. Thus unlike standard V group communication the Send does not block for a reply message.

ALL_REPLY

Send blocks until replies from all the alive group members in the group specified by id are received, after which SUCCESS is returned, and the Sender's message is overwritten by the first reply. A FAILURE is returned if reception of all the replies is not possible, i.e. the maximum number of timeouts and retransmissions have been done or message delivery to a member failed. In either case, Send sets the *code* field in the message to number of replies returned.

ALL_DELIVER

Send blocks until the message is delivered to all the alive group members specified by id, in which case a SUCCESS is returned. Otherwise FAILURE is returned because delivery to all the members is not possible, i.e. the maximum number of timeouts and retransmissions have been done or message delivery failed for a member. In either case, Send sets the *code* field in the message to the number of members to which the message was delivered.

GetReply(message, time_limit)

GetReply is used by the Sender to retrieve subsequent replies in the reply queue of the Sender. If there is a reply, it is copied to the message and the PID of the replier is returned. Otherwise, it blocks until a reply comes in or the timeout specified in *time_limit* expires. In case of a timeout, FAILURE is returned.

Receive(message)

ReceiveSpecific(message, id)

Receive or **ReceiveSpecific** block until a message is received from any Sender or a message is received from the specific id which may specify a process or a group of processes. When a message is received, the PID of the Sender is returned and the received message is copied to the message. If no messages arrive, Receive blocks forever, whereas ReceiveSpecific may return with a FAILURE after timing out. The timeout occurs only if the process with *id* process id does not exist in the system.

Reply(message, id)

Reply sends a reply message to the process specified by *id*.

The complete list of communication primitives in the V system and their descriptions may be found in [Cheriton86].

One of the major contributions to the design in [Atkins86] was in investigating ALL_RECEIVE multicast and choosing ALL_DELIVER multicast over ALL_RECEIVE. ALL_RECEIVE multicast returns success if all the Receivers have received the message and the message is available to the user level. Choosing of ALL_DELIVER was due to the way that messages are handled inside the V-kernel. When a Receiver invokes the Receive system call, it is blocked and the control is passed to the kernel. When an awaited for message arrives, it is copied to the Receiver's message buffer inside its process control block (PCB). If another valid message addressed to this Receiver arrives before the Receiver is readied for execution (as can easily occur with multiple Senders), the previous message in the Receiver's process block is overwritten by the new one. Currently when this happens on a specific 1:1 message, the first Sender will eventually timeout and retransmit. The present group implementation on the published V system allows message overruns and the message is lost.

The ALL_RECEIVE success implies that the sent message is available to the Receiver. This semantic is very difficult to implement in the current V-kernel implementation where messages can be overwritten in the PCBs. One solution is to use a flag in the PCB of the Receiver. However, this would introduce non modular/structured code since the flag will have to be set after a message delivery in the kernel and reset in the user library level where the message is copied from the PCB to the user message data structure. In lieu of the above, the ALL_DELIVER semantic was chosen, which succeeds if the message is delivered to the Receiver (implying that the message may be overwritten before the Receiver has read it).

Some of the other major additions to the original design were in the area of providing variable degrees of reliability. These are provided by the K_REPLY and K_DELIVER multicasts [Cheriton85][Ravindron87]. These plus the extended user level functionality of I_Am_Here server which will be discussed in the following sections, provide powerful tools in implementing higher level communication services.

4. Detailed Design and Implementation

V system provides two types of groups, local or global. Only local processes may join a local group whereas both local and remote processes, i.e. processes running on other kernels, may join a global group. The original V system only provides 0 or 1 reliable broadcast, i.e. best efforts only[4]. This chapter describes our detailed design of kernel level reliable multicast for both local and global group communication. It also presents the design of the I_Am_Here server which has both kernel and user level applications.

The reliable multicast and the I_Am_Here server was implemented inside the Version 6 V system. The V-kernel has three major logical components: the kernel server, device server, and the inter-process communication. The global reliable multicast was implemented in the inter-process communication section which is entirely written in the C programming language. Our reliable multicast uses some of the V kernel's IPC facilities such as duplicate elimination and message retransmission. V system's inter-process communication mechanism eliminates duplicates and out of sequence messages by using message and transaction sequence numbers, and it also retransmits messages after each timeout. It has four timeouts of 2 seconds each. The I_Am_Here server implementation affected V system's timer interrupt, ethernet device module, and the kernel server, which is a pseudo process. The reliable local group multicast, though completely designed, has not been implemented due to the recent trend in V system evolution toward global groups only.

4.1. Reliable Local Group Communication

In implementing the reliable multicast, the intra-kernel protocol executed at the Sender and Receiver's kernel was modified[5]. These changes, however, did not affect the

---

4 - 0 reliable is a user level datagram communication mechanism that does not require the Receiver to reply.

5 - In case of local group communication, the Sender's and Receiver's kernel are the same.

logical inter-process interactions.

The logical inter-process interactions between 1 Sender and 4 Receivers who are members of a local group for standard group communication are shown below:

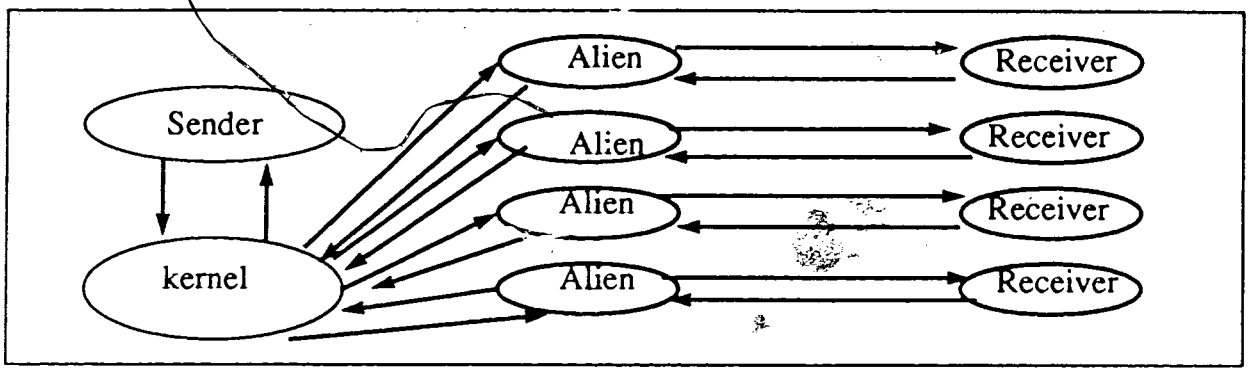

Figure 3. Logical inter-process interactions for local groups

The Sender does a Send, thus invoking a kernel request. The kernel recognizes the request as a Send to a local group. It creates an Alien process for each Receiver in the group to act as a go between and as a message buffer. Then the message is delivered to each Receiver and the Sender blocks until a reply is send to it. As each Receiver replies the Alien is either destroyed if it is the first replier. Otherwise, it is reused to queue the reply in the reply queue of the Sender, and the message is delivered to the Sender. For the sake of simplicity the interactions between Receivers and the kernel are not shown. Note that the Figure 3 represents what logically happens.

4.1.1. DELIVER reliability

The Reliable Intra-kernel protocol for K_DELIVER and ALL_DELIVER of local

group communication is shown below:

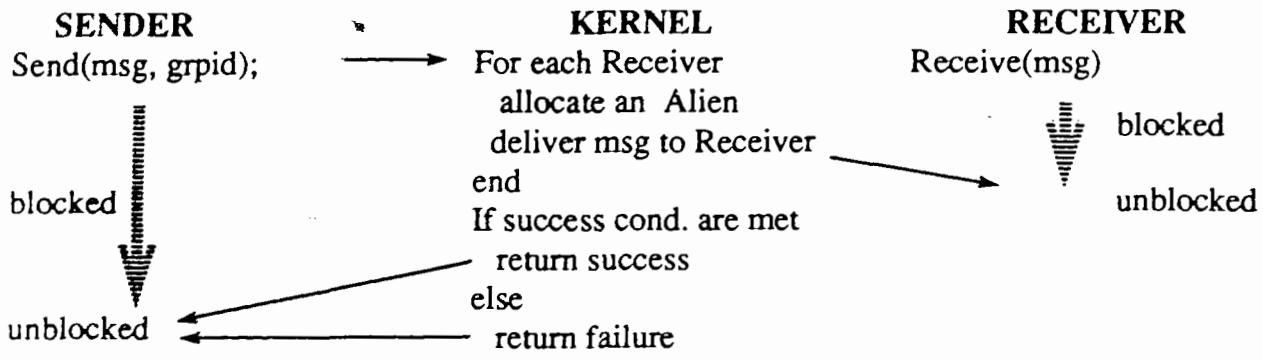

Figure 4. - Reliable Intra-kernel protocol of K_DELIVER and ALL_DELIVER

In the modified group communication mechanisms, the Send succeeds if the message is delivered to K Receivers for K_DELIVER or to all the Receivers for ALL_DELIVER. Note that the Sender is unblocked after the status is returned from the kernel, thus unlike standard V group communication the Send does not block for a reply message. The success condition of the Send is stored in the Sender's process descriptor block.

There are no retransmissions since the Receivers and the Sender are local[6]. The ALL_DELIVER only fails when an Alien can not be allocated; the maximum number of Aliens in the system is predefined, and because of the asynchronous characteristic[7] of K_DELIVER and ALL_DELIVER, the number of free Aliens diminishes quite rapidly.

4.1.2. REPLY reliability

6 - This is also done in the standard V system

7 - Where the Sender does not block for replies and is free to send messages before the Receiver has read previous messages.

The Reliable Intra-kernel Protocol for K_REPLY and ALL_REPLY is shown below:

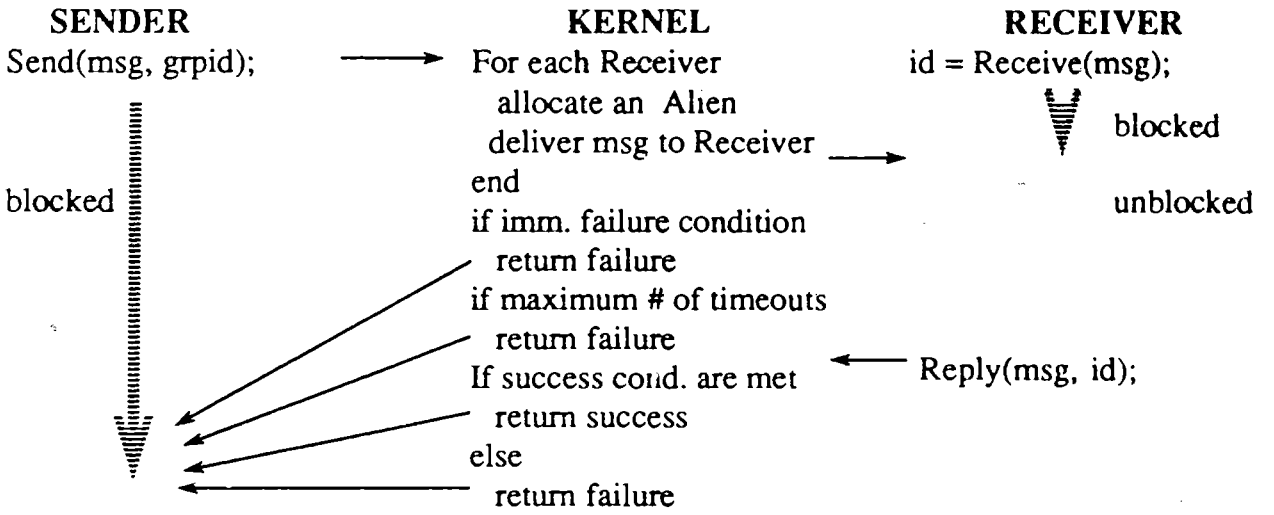

Figure 5. Reliable Intra-kernel Protocol of K_REPLY and ALL_REPLY

The immediate failure condition is satisfied when the Send type is K_REPLY and the message is delivered to less than K Receivers or when the Send type is ALL_REPLY and a delivery fails. If K or all Receivers do not reply, after maximum number of timeouts, failure status is returned to the Sender. As the Receivers reply to the sent message, the kernel checks for having K replies for a Send type of K_REPLY or having received the last reply for ALL_REPLY Send. If a success condition is met, then success is returned to the Sender. Note that timeouts may also happen after replies start to be received by the kernel .

4.2. I_Am_Here Server

Unlike reliable local group communication where all the information needed for determining success or failure of reliable communication is available locally, the knowledge of the Sender's kernel is not complete in the case of reliable global group communication. In order to provide the information needed to determine the success of a Send, some form of global information is needed. One obvious method would require the Sender to acquire the process ids of all the group members. So that in case of ALL_REPLY, it could deter-

mine the success of a Send, or in the case of K_REPLY, it could know that K replies are possible or not. Since V provides dynamic groups, which processes may join or leave at any time, the provision and maintenance of such information will be difficult.

However, the knowledge of alive hosts/kernels is easier to obtain and maintain, and it is sufficient for this reliable multicast implementation[8]. This information is provided by using an I_Am_Here (IAH) server which maintains the list of reachable/alive hosts. Every host on the network has an I_Am_Here server which executes the following algorithm:

When a host boots up, it broadcasts an I_Am_Here message to all the hosts in the V domain and thereafter it broadcasts the message once every second. The I_Am_Here server, upon receiving the first I_Am_Here message from a host, marks the Sender host up in its information base and in turn sends its own I_Am_Here message after a random delay (0-10 milliseconds) to help the Sender host in compiling its Alive Kernel Table (AKT). The data base in each kernel is composed of two parts: an Alive Kernel Table with a row for each alive host on the network, and an Alive Kernel Bit Vector (AKV) with a bit entry, **up_bit**, for each host. These data structures are shown in Figure 6:

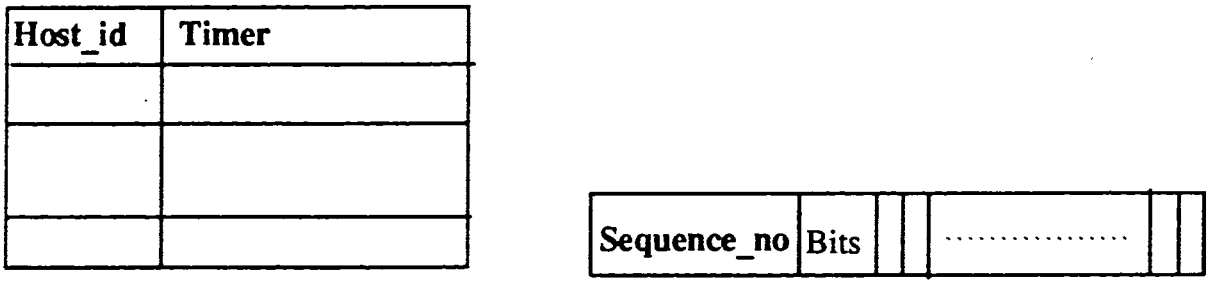

Figure 6. I_Am_Here Server Data Structures

There is a one to one correspondence between the position of a host in the table

8 - Recall that the standard V-kernel "knows" all group members on its local machine.

and the bits in the vector; e.g. the status of the host in the fourth table entry is stored in the fourth bit of the AKV **up_bit** field. The table has two entries for each host: the **host_id** entry and a **timer** entry. In this implementation, the **host_id** contains the Ethernet address of hosts. Every time an I_Am_Here message arrives from a host, its bit position in the AKV **up_bit** is set to TRUE and its **timer** is set to 2 which is decremented once every second. If the **timer** becomes 0, then the host is assumed to be down; its timer is set to -1, its bit position in the AKV **up_bit** is set to FALSE and its space in the AKT is freed up. This timer mechanism facilitates marking a host down only if no I_Am_Here messages arrive from that host in 3 seconds. This 3 second grace period may be increased with ease.

The **sequence_no** variable associated with the Alive Kernel Bit Vector is incremented every time there is status change, i.e. a host comes up or goes down. In the current implementation, the bit vector is 32 bits wide, so it can accommodate up to a 32 host network environment; the number of bits may be increased quite easily.

The global view of the I_Am_Here server quickly stabilizes with the multiple broadcasts of a given host when it comes up, the periodic broadcast, and the very low error rate of Ethernet Local Area Networks. The delayed discovery of false or genuine host failures is tolerated by the communication protocol's timeouts.

It is worthwhile to note that there is some randomness in the broadcasting of I_Am_Here messages due to the following factors:

- The clocks of all the hosts are not synchronized, i.e. interrupts don't all happen at the same time [9].
- In V, the I_Am_Here message is only broadcast immediately if the timer interrupt

9 - The V-kernel timer interrupts once every 10 milliseconds.

did not interrupt the kernel, otherwise it is delayed. So the exact time of the broadcast is affected by the current system activity.

This randomness plus the assumption that the number of hosts on the LAN, not counting gatewayed hosts, is not extremely large avoids collisions and excessive loading of the LAN.

The I_Am_Here server also provides the following user level services:

- getting a copy of the alive kernel vector (AKV) - This is done by sending a message to the local kernel with *syscode* field set to GET_IAH_BIT_VECTOR command constant. The local kernel replies with a message containing the bit vector and the *sequence_no* of the vector.
- getting the host number, given the bit position and sequence number - This is done by sending a message to the local kernel with *syscode* field set to GET_IAH_HOST_NO command constant, and two of the other fields set to the host position number (e.g. 4 for the fourth host) and the *sequence number*, previously obtained by a GET_IAH_BIT_VECTOR command, in the message. If the user's *sequence number* matches the kernel's, the local kernel replies with a message containing the *host_id* (i.e. ethernet address) of the host in the reply message. Otherwise FAILURE is returned.
- resetting the kernel bit vector - This is done by sending a message to the local kernel with *syscode* field set to RESET_IAH_BIT_VECTOR command constant. The local kernel replies to this message after resetting the vector.

These services coupled with the new reliable communication primitives could be used in building higher levels of reliability, atomicity, ordered delivery, etc. with ease.

4.3. Reliable Global Group Communication

Members in a global group may reside at local or remote hosts. The logical inter-pro-

cess interactions between 1 Sender and 2 Receivers who are members of a global group are shown below:

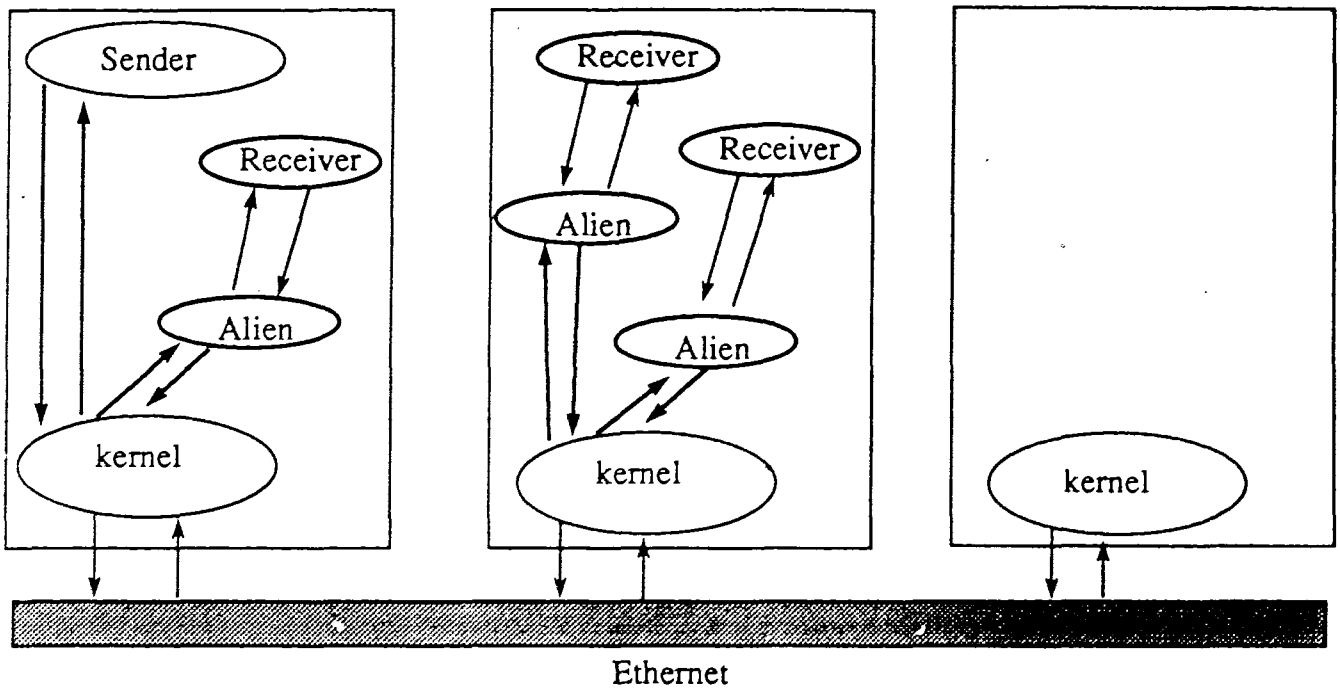

Figure 7. Logical Inter-Process Interactions For Global groups

The Sender blocks, and the message is serviced locally and is also broadcast on the network[10]. The kernels with Receivers create an Alien process for each group member and deliver the message to their local Receiver(s). Replies from remote Receivers are also sent via the network and picked up by the Sender's kernel.

It is necessary for each host, whether it has group members or not, to acknowledge the ALL_REPLY or ALL_DELIVER group Sends. This acknowledgment is also used for early detection of K_REPLY or K_DELIVER's failures, though it is not strictly required. For Reply reliable Sends (i.e. ALL_REPLY and K_REPLY), the host acknowledgments actually add very little overhead, because the acknowledgments from kernels with no

10 - Unless the operation's success conditions are met locally.

members arrive before the genuine replies to the multicast messages and are dealt with quickly, and the kernels with members piggyback the acknowledgments on reply message. For Deliver reliable Send, there are no replies from the Receivers and the kernel acknowledgments are efficiently processed by the Sender's kernel. The performance figures (in the Performance chapter) show that the extra host acknowledgments add insignificant overhead to processor or network costs, as network interrupts are fielded efficiently in the kernel.

In this implementation, one unique multicast address is set aside for all the reliable communication, whereas specific multicast addresses are used for standard V group Sends. One multicast address for reliable communication is used mainly because of ease of implementation of the inter-kernel protocol. Use of more than one multicast address for reliable communication is discussed in the Future work section.

4.3.1. DELIVER reliability

The reliable Inter-Kernel protocol of K_DELIVER and ALL_DELIVER communication primitives is shown in Figure 8.

The Sender blocks. The Sender's kernel first takes a copy of the alive kernel's vector and stores it in the Sender's process control block. The Sender's process control block has an *expected kernel ack vector* and a *num_receivers* entry for the number of Receivers, initially set to zero. The Sender's kernel creates an Alien for each local Receiver (if any), increments *num_receivers*, and delivers the message. It also sets the bit for the local kernel in the expected kernel ack vector.

An immediate failure can occur on the Sender's kernel if the free Alien resource pool is exhausted, and the message is of type ALL_DELIVER. In this case, the kernel aborts the Send and returns failure to the Sender. Otherwise, the message is broadcast over the net. On the Receiver's kernel, an immediate failure happens when the kernel runs out of free Aliens and the message is of type ALL_DELIVER. This

causes a negative acknowledgment (NACK) message to be sent to the Sender's kernel.

If a NACK message arrives at the Sender's kernel and the message is of type ALL_DELIVER, then failure is returned to the Sender.

Remote kernels with no Receivers immediately send an ACK message with 0 as the number of deliveries made. Otherwise, the remote kernel delivers the message to the Receivers and sends an ACK message with number of deliveries made. Both the ACK and the NACK message contain the number of deliveries made.

If immediate success conditions hold, i.e. the local host is the only alive host and delivery succeeded to all the local Receivers and the message is of type ALL_DELIVER or if the number of deliveries done locally >= k, then success is returned to the Sender. Otherwise, as kernel acknowledgments plus number of deliveries messages come in from other hosts, their respective bit in *expected kernel ack vector* is set and number of deliveries is added to Sender's *num_receivers* field; the Send succeeds if expected kernel ack vector is identical to the Sender's copy of alive kernel vector and the message is ALL_DELIVER, or if *num_receivers* >= K for K_DELIVER messages.

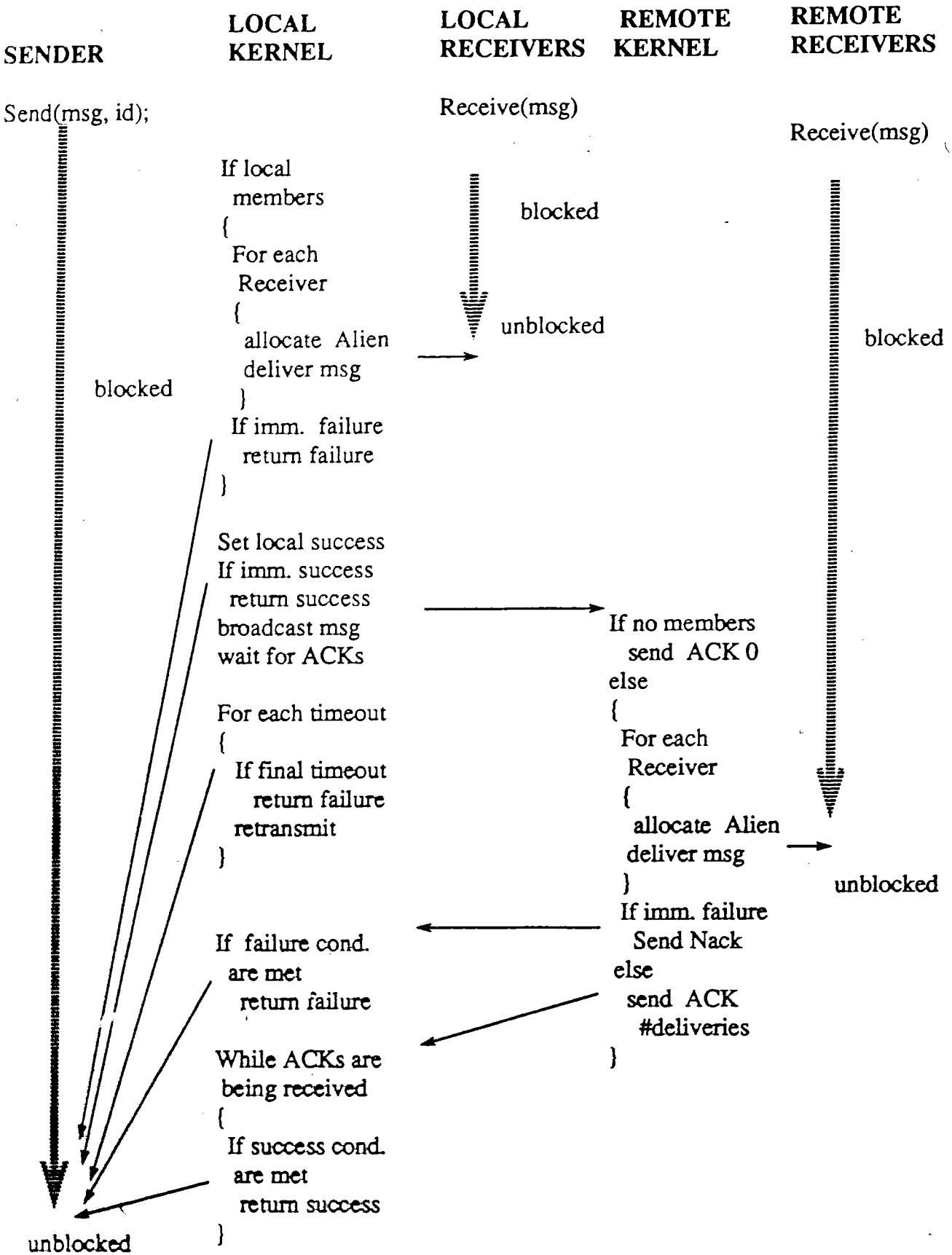

Figure 8. - Reliable Inter-kernel protocol of K_DELIVER and ALL_DELIVER

If immediate success conditions hold, i.e. the local host is the only alive host and delivery succeeded to all the local Receivers and the message is of type ALL_DELIVER or if the number of deliveries done locally >= k, then success is returned to the Sender. Otherwise, as kernel acknowledgments plus number of deliveries messages come in from other hosts, their respective bit in *expected kernel ack vector* is set and number of deliveries is added to Sender's *num_receivers* field. The Send succeeds if expected kernel ack vector is identical to the Sender's copy of alive kernel vector and the message is ALL_DELIVER, or if *num_receivers* >= K for K_DELIVER messages.

If timeout occurs and success conditions have not been met at the Sender's kernel, the message is retransmitted by the kernel. The Receiver's kernel recognizes retransmitted messages and re-sends the kernel ACK messages. After the maximum number of timeouts and retransmissions, the Send is aborted and failure is returned to the Sender. Note that because both the ACK and the NACK messages contain the number of group members on the host, it is possible for a K_REPLY to succeed even if NACK messages are received from some of the remote kernels.

4.3.2. REPLY reliability

The algorithm of K_REPLY and ALL_REPLY for global group Sends are shown in Figure 9:

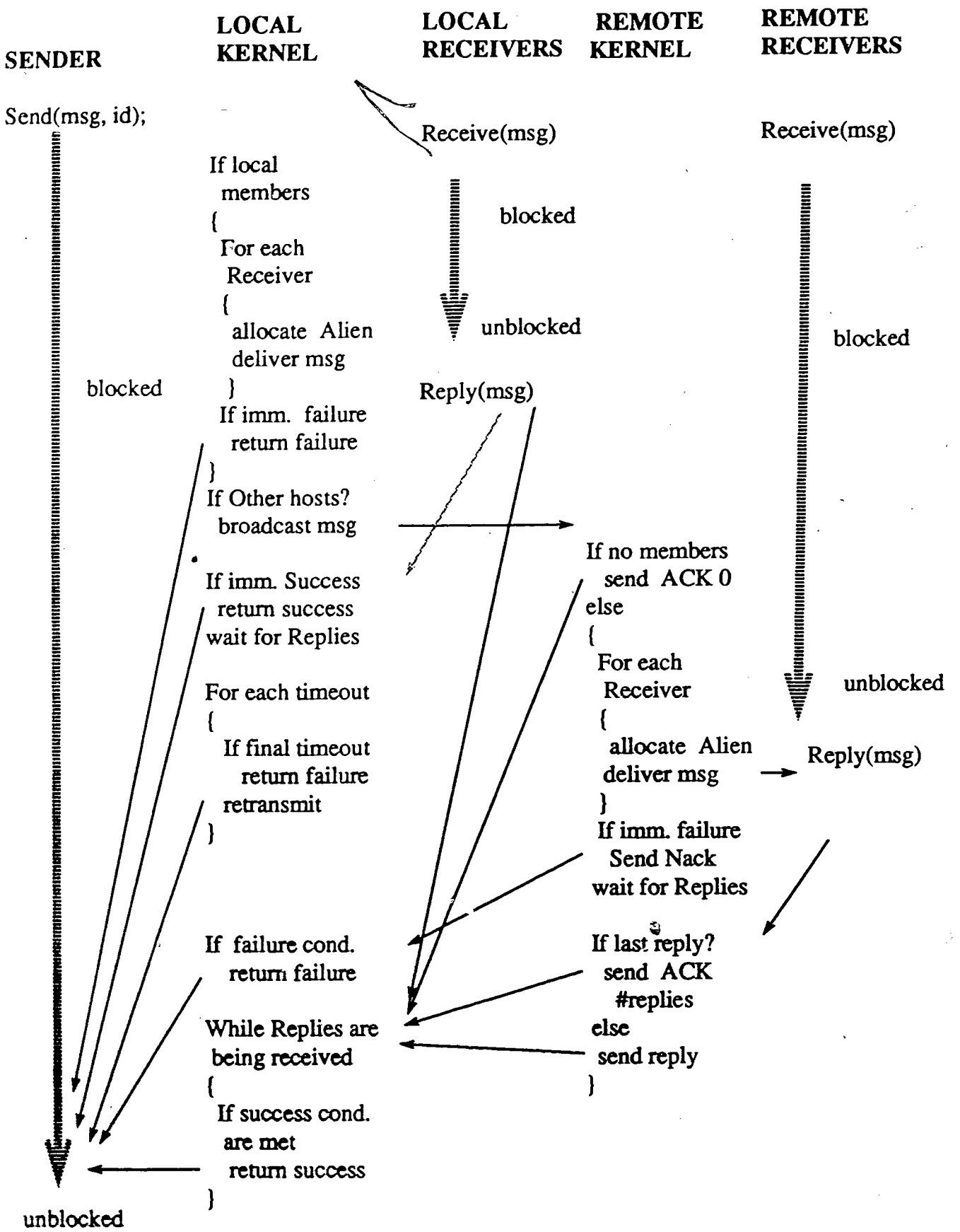

Figure 9. Reliable Inter-kernel Protocol of K_REPLY and ALL_REPLY

The Sender blocks. The Sender's kernel takes a copy of the AKV and stores it in the Sender's PCB. The Sender process's control block's *expected kernel ack vector, num_receivers and num_replies_received* fields are initially set to 0. The Sender's kernel creates an Alien for each local Receiver (if any), updates *num_receivers*, and delivers the message. The immediate failure conditions are similar to the Deliver reliability algorithm and are caused by lack of resources to either deliver the message to the Receiver or reply to the Sender. If immediate failure does not occur, at the Sender's kernel, the message is broadcast over the net.

As replies are received, the Sender's kernel increments *num_replies_received* field in Sender's PCB. If immediate success conditions hold, i.e. the local host is the only alive host and replies arrived from every local Receiver and the message is of type ALL_REPLY (or if the number of replies received locally >= K for K_REPLY), then success is returned to the Sender. Otherwise, the Sender's send operation waits for remote kernel messages. If a NACK message arrives at the Sender's kernel and the message is of type ALL_REPLY, failure is returned to the Sender. A NACK message is sent by remote Receiver's kernel if the sent message can not be delivered due to lack of Alien processes, which causes an immediate failure. Otherwise, the remote kernel sends the replies from the Receivers, piggybacking an ACK and the local number of Receivers on the last reply. As the kernel acknowledgments arrive at the Sender's kernel from other hosts, the respective bit in the *expected kernel ack vector* is set and number of replies is added to *num_replies_received*.

For ALL_REPLY multicast, the Send succeeds if the *expected kernel ack vector* is identical to the Sender's copy of alive kernel vector and *num_receivers* is equal to *num_replies_received*. K_REPLY succeeds if *num_replies_received* >= K. Note that because both the ACK and the NACK messages contain the number of group members on the host, it is possible for a K_REPLY to succeed even if NACK messages are received from some of the remote kernels.

If timeout occurs and success conditions have not been met at the Sender's kernel, the message is retransmitted by the kernel. The Receiver's kernel recognizes retransmitted messages and re-sends the replies previously sent from that kernel; the Sender's kernel discards duplicate replies by using sequence numbers. If there are some Receivers that have received the message but have not replied yet, a special message called Breath-of-life is sent to the Sender's kernel. This message resets the timeout time in the Sender's kernel lengthening the transaction time limit beyond 4 timeouts of 2 seconds each. After the maximum number of timeouts and retransmissions, the Send is aborted and failure is returned to the Sender.

Two outstanding worthwhile issues regarding this reliable multicasts design are its interactions with view of the system (i.e. the alive hosts) and scalability. Change of the view of the system is handled correctly by these reliable multicast protocols in that the Send does not succeed if the requested operation failed. If a site fails in the middle of a send transaction before sending the ACK message to the Sender's kernel, the transaction will fail. Also the ACKs/NACKs that may arrive from sites that recover/come up in the middle of a Send transaction will not contribute to the success or failure of the transaction since the Sender's kernel uses the view before the start of the transaction for determining success. The only other possibility for reporting false success to the Sender is when a site fails after the transaction has started and recovers before the end of the transaction (before the last retransmission). This is very unlikely since the failure and recovery have to happen in three retransmission times(i.e. 6 seconds in the V kernel). Also, this condition may not be viewed as a false success but true success.

Scalability comes into the picture because of the I_Am_Here server's periodic messages and the ACK 0 messages from hosts with no Receivers. These messages are about 100 bytes long and with a 10Mbit Ethernet, their transmission takes about 1 millisecond. The V kernel, like many other kernels, handles Ethernet interrupts

using interrupt service routines. The interrupt service routines receive these messages and invoke appropriate functions which extract the needed information from the messages and then discard the messages. The handling of each of these messages takes about 0.1 msec on Sun 3 with a 50 host network. Because of the equal transmission and kernel processing times for each of the I_Am_Here and ACK 0 messages, the kernel percentage loading for handling ACK 0 and I_Am_Here messages is equivalent to Ethernet percentage loading by ACK 0 and I_Am_Here messages. With a 55% Ethernet loading, where ACK 0 messages from 50 hosts consume 50% and I_Am_Here messages consume 5% of the Ethernet bandwidth, the Ethernet will handle 100 multicast messages per second. This will also result in 55% loading of the kernel, thus indicating the upper limits of scalibility of our protocol.

5. Performance

The timing measurements of communication models for global group communication are given in this section. First the experimental environment, both hardware and software, is presented, followed by the specification of test models used in timing measurements. Then, the performance of the test models is discussed.

5.1. Experiment Environment

Sun 3/50 workstations were used for the performance analysis of reliable communication primitives. These workstations use a 32 bit Motorola 68020 processor running at 16 MHz. Each workstation contains 4 megabytes of main memory with no secondary storage. In this diskless environment, a micro Vax II was used as a file server performing all the disk I/O.

The machines were connected via a 10 Megabit/second Ethernet[Shoch80]. The experiments were done using up to 8 Sun 3/50 workstations. The hardware environment is shown in Figure 10.

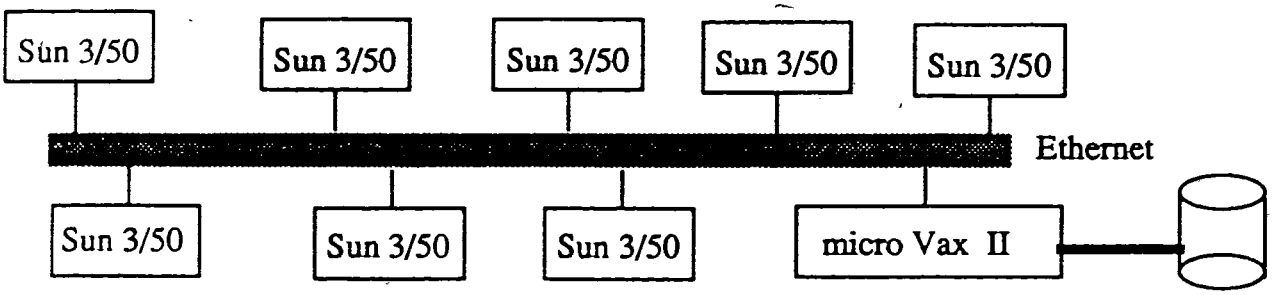

Figure 10. Hardware environment

The Sun 3/50 workstations ran the Version 6 V system, and the micro Vax II ran 4.2 BSD UNIX. The software role of the micro Vax II will not be discussed further, as the micro Vax II was only used as file server and hence did not effect the timing results.

5.2. Test Models

The test model used in obtaining the performance figures consists of a Sender and one or more Receiver processes. The following three types of communication mechanisms are used in the test models:

- **Standard One To One** (1:1) - The Sender uses specific Send to send messages to the Receiver(s). If there are multiple Receivers, the Sender sends the message to each one in a sequential manner.
- **Standard One To Global Group** (Standard 1:many) - The Sender and Receivers belong to the same global group and the Sender uses V system's 1-reliable multicast to send messages to the Receivers.
- **Reliable One To Global Group** (Reliable 1:many) - Similar to Standard 1:many except that one of the ALL_REPLY, K_REPLY, ALL_DELIVER, and K_DELIVER reliable primitives is used.

The Sender and Receiver(s) first establish contact with each other, and then execute a tight loop which is bracketed with timing measurement mechanisms. In the loop, the Sender sends messages to the Receiver(s) and waits for appropriate number of replies or deliveries in a loop of 10,000 iterations. In all the Reply reliable and standard V 1:many models, the Sender awaits and retrieves ALL or K replies, using the GetReply primitive. Note that the Sender and the Receiver(s) each execute on a separate host.

In the V system, each message is 32 bytes long, but it may have a 1 kilobyte data segment associated with it [Cheriton85]. For simplification of test models, our messages do not have data segments associated with them.

The following terminology is used when discussing performance:

- Early Standard V-kernel - Version V.6 kernel before any enhancements.

- Standard V-kernel - Version V.6 kernel with no delay[11] on replies to group Sends.
- Early Enhanced V-kernel - Version V.6 kernel with our reliable group communication enhancements. This corresponds to an enhanced version of the Early Standard V-kernel.
- Enhanced V-Kernel - Version V.6 kernel with our reliable group communication enhancements, and with no delay on replies to Group Sends. This corresponds to an enhanced version of Standard V-kernel.

The timing results obtained in the rest of this chapter were obtained while the network was in normal usage. In order to get valid results, the tests were done at different times and the minimum was chosen. Moreover, they were executed at the user level with no attempt to optimize their performance by adjusting priorities, etc.

11 - The meaning of the term no delay is explained later in this chapter.

5.3. Timing Results

First, we determine the overhead of the enhanced communication mechanisms in the V-kernel by comparing the performance of 1:1 and standard 1:many Sends in standard V-kernel and in the enhanced V-kernel. The results are tabulated in Table 1. All the times given are msecs per message sent.

Table 1 - Comparison of standard V-kernel and enhanced V-kernel (msecs/message)

Type of comm.	Comments	Early Std. (1-reliable)	Std. V-kernel (1-reliable)	Early Enh. V-kernel	Enhanced V-kernel
1:1	same host (timeipc)	0.684	0.684	0.696	0.696
1:1	diff. hosts (timeipc)	1.919	1.919	2.331	2.331
1:3	4 hosts	20.0	3.86	20.5	3.84
1:8	9 hosts		6.945		6.993

We used the standard V system program *timeipc* for measurement of the performance of 1:1 specific Send for local communication on the same host (row 1 of the table) and remote communication between two hosts (row 2 of the table). *Timeipc* is a special program which executes in a tight loop sending or receiving messages. These differences

between standard V and our enhanced V are insignificant. They are caused by the extra code in the kernel for handling the reliable multicast. The gap between 1.913 and 2.331 msecs per 1:1 message may be greatly reduced by streamlining the kernel code in handling the standard V communication primitives.

The results for all the group communication described hereafter were obtained by executing Sender and Receiver programs, each on a different host. The Sender sends a message to the Receiver(s) and waits for appropriate number of replies or deliveries in a loop of 10000 iterations. In case of waiting for replies, the Sender retrieves all the replies using the **GetReply** primitive. Each Receiver blocks on a message reception and replies to all the messages. Unlike *timeipc* program, these programs were executed at the user level with no attempt to optimize their performance by adjusting priorities, etc.

The Sender is given the number of Receivers in the standard kernel test, and it executes **GetReply** to obtain all the replies in both cases. Note however, that it is conceivable that as the standard kernel only makes "best-efforts" delivery attempts, the standard kernel could fail to receive replies from its group members, incurring a heavy timeout cost on **GetReply**. This did not occur in our test runs, thus indicating that the LAN is very reliable.

Row 3 of Table 1 shows that the performance of the early versions of the group communication are not optimized, since the replies to a global group Send are delayed by a random amount of time. This delay, which can be as long as 30 milliseconds per reply, reduces the number of system buffer overruns at the Sender's host [Cheriton85]. This overrun problem exists mainly in Sun-2 hosts which have minimal system buffering capabilities in the Ethernet interface. Sun 3/50's are much faster and the number of allocated system buffers in the Ethernet interface is large. Therefore, the 30 milliseconds delay is not needed, so we removed the delay from both the early standard kernel and the early enhanced kernel. This shows vast performance improvement which makes multicast cost

effective when the number of Receivers is more than 1, which is usually the case in group communication. The cost overhead remained insignificant, so we only continued further measurements on the *no-delay* kernels.

5.4. Performance of ALL_REPLY and K_REPLY Communication Mechanisms

The performance data for 1-reliable, ALL_REPLY and K_REPLY reliable 1:many Send communication mechanism is presented in Table 2. The data are plotted in Figure 11. All the times given are in msecs per message sent.

Table 2 - Performance of ALL_REPLY and K_REPLY Sends (msecs/message)

Number of Receivers	Comments	Enhanced V-kernel		
		(1-reliable)	(ALL_REPLY)	(K_REPLY)
1	2 hosts	2.418	2.707	2.980
2	3 hosts	3.18	3.926	3.954
3	4 hosts	3.84	4.634	4.739
8	9 hosts	6.993	8.878	8.991

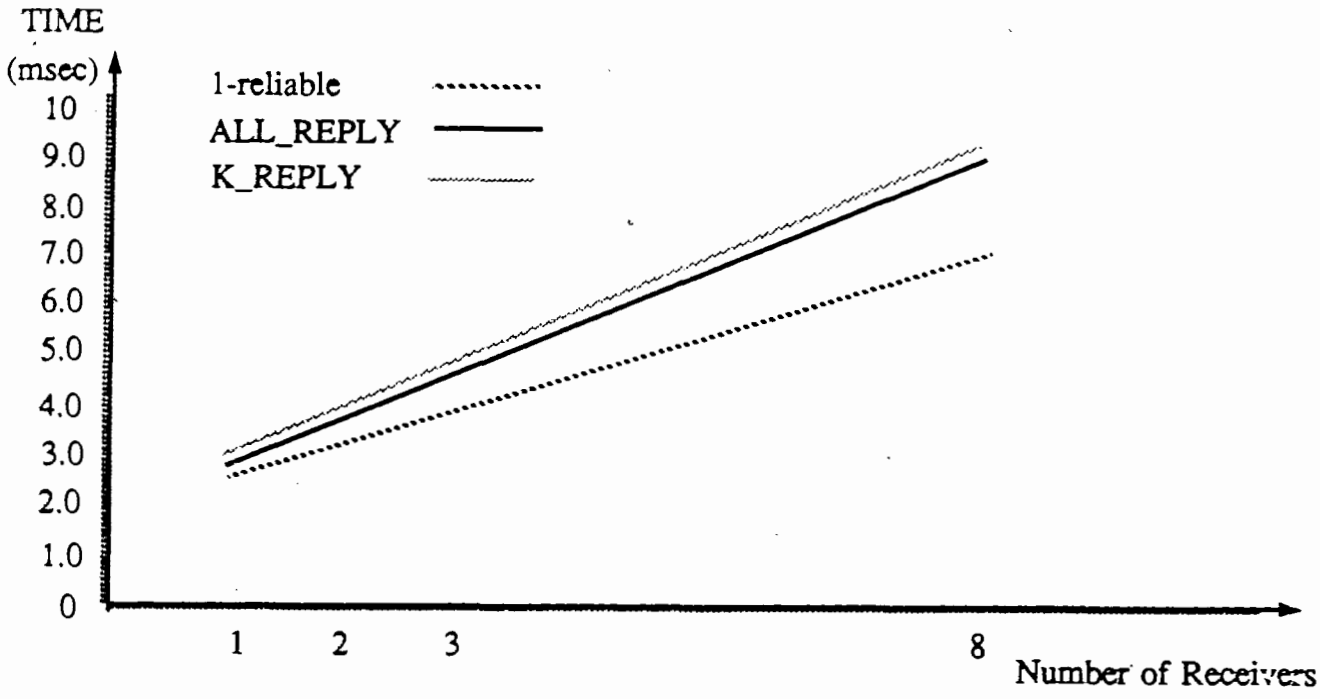

Figure 11. Performance of ALL_REPLY and K_REPLY Sends

The performance data in Table 2 and Figure 11 was obtained by executing each of the Sender and Receiver processes on a different host. Tests done with a mixture of hosts with and without Receivers resulted in the same performance times. Therefore the overhead of ACKs from hosts with no Receivers is negligible.

The performance of the enhanced ALL_REPLY group Send is about 20% slower than the 1:reliable group Send, mainly because the Sender is blocked until all the replies have arrived; only then can the Sender read the replies with **GetReply**. In the 1-reliable case, the Sender is unblocked as soon as one reply arrives, thus allowing concurrency in reading replies and accepting more replies. Another reason for the slower speed of the ALL_REPLY reliable Send is the extra code in the kernel to deal with this case. This still makes ALL_REPLY attractive since the ALL_REPLY group communication mechanism does not need to know the number of Receivers in the group and it has a suitable semantic interface for applications that require reliability.

The percentage timing difference between the 1-reliable 1:many and ALL_REPLY 1:many Sends remains constant as the number of Receivers increase, therefore ALL_REPLY should perform quite well even with large number of Receivers; it scales up well.

The ALL_REPLY mechanism was also tested when not all the alive hosts had Receivers on them; no additional overhead was observed. This is due to the concurrency of the model; the hosts with no Receivers reply before the Receivers on other hosts reply to the message and the Sender's kernel handles these early replies very efficiently.

The standard 1:1 mechanism takes about 2 msecs for a message exchange between two machines. Therefore if all the replies are needed reliably, it pays to use the ALL_REPLY group communication over the sequence of 1:1 Sends, if there is more than one Receiver in the group.

K_REPLY 1:many communication mechanism's performance is similar to

ALL_REPLY's performance since K was set to the total number of Receivers in the system. The difference is insignificant; it is mainly caused by placement of tests for ALL_REPLY success condition tests before the tests for the success conditions of K_REPLY in the kernel. Therefore, the above discussion about the performance of ALL_REPLY mechanism also applies to K_REPLY. Of course, K_REPLY would perform better than ALL_REPLY in applications that require K replies where k < total number of Receivers; for example, an application that wants to find the K available servers would be faster using the K_REPLY Send than using ALL_REPLY Send.

5.5. Performance of ALL_DELIVER and K_DELIVER Communication Mechanisms.

The performance of ALL_DELIVER and K_DELIVER reliable 1:many Send communication mechanism is presented in Table 3 and plotted in Figure 12. Note, that in DELIVER reliable 1:many models, the Sender does not block for replies, instead it blocks for a status to be returned from the Sender's kernel. Also, the Receivers do not reply to messages. Most asynchronous operations have a tendency to exhaust system resources if done in a tight loop; the ALL_DELIVER and K_DELIVER are not exempt from this rule. A tight loop of 10000 iterations would quickly use up the free pool of Aliens and system buffers on the Receiver's kernels, and overrun problems similar to the ones described in the Design Issues chapter would occur. Therefore, the number of iterations was lowered to 100, and the experiment was repeated several times. The alternative to this would involve the introduction of a fixed delay (e.g. half a second), before each Send in the Sender's 10000 iteration loop. The overhead of 10000 half a second delay operations would later be deducted to obtain the time required for the tight loop of 10000 DELIVER reliable Sends. This method would increase the length of timing test (10000 half a seconds = 1.5 hours), and increase the effect of bursty network traffic on the results.

Unlike the ALL_REPLY case, there is no real counter part to ALL_DELIVER communication mechanism in the standard V-kernel; V, however, supports a user level datagram communication mechanism that does not require the Receiver to reply. But, this datagram mechanism often fails to deliver the message if the Receiver is not waiting for a message, and it does not return any delivery status.

The performance data of the reliable DELIVER 1:many and the 1:many datagram communication is shown in Figure 12. All the times given are in msecs per message.

Table 3 - Performance of ALL_DELIVER and K_DELIVER Sends (msecs/message)

Number of Receivers	Comments	Enhanced V-kernel		
		(datagram)	(ALL_DELIVER)	(K_DELIVER)
1	2 hosts	0.6	2.0	2.1
2	3 hosts	0.65	2.1	2.2
3	4 hosts	0.7	2.2	2.3

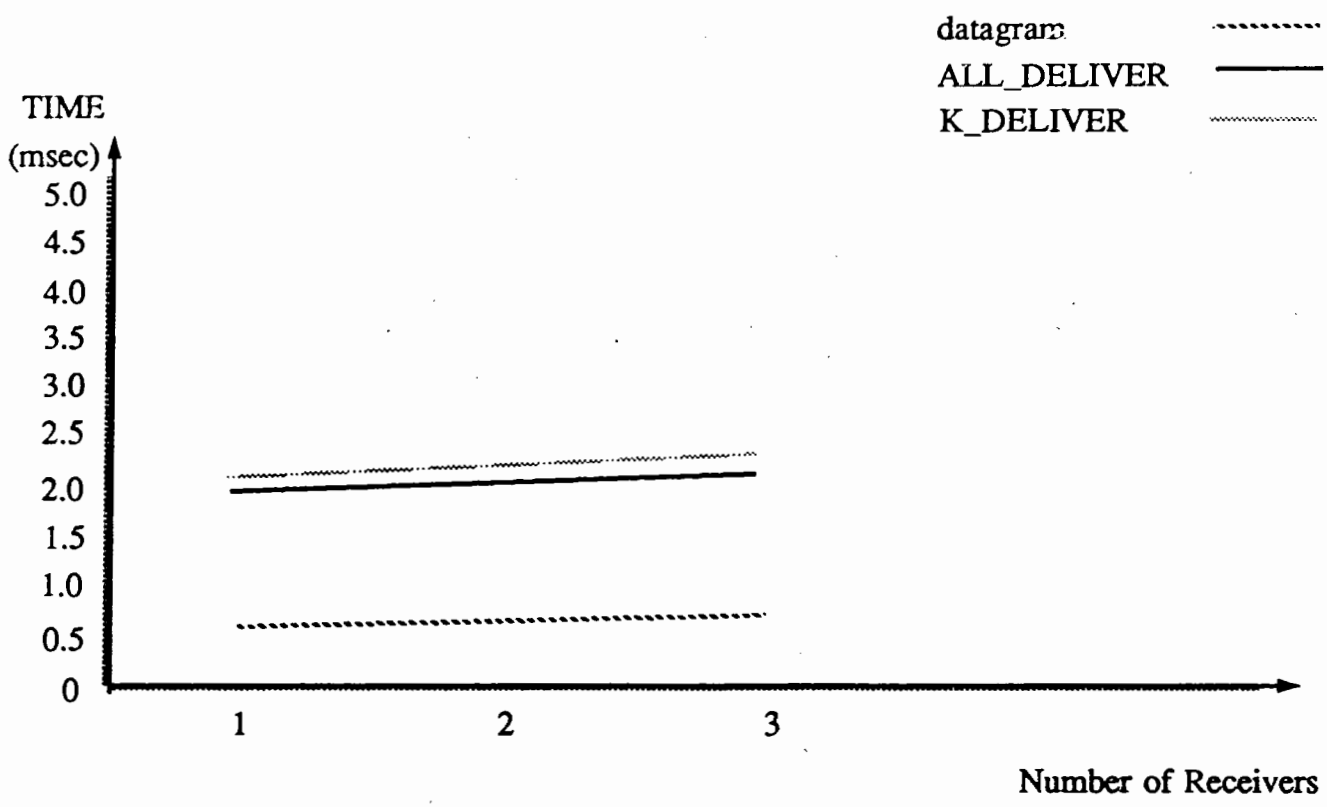

Figure 12. Performance of ALL_DELIVER and K_DELIVER Sends

Although, the datagram ran faster than the DELIVER reliable communication mecha-

nisms, the DELIVER reliable communication mechanism is superior. In the timing tests, the datagram Receivers lost about 50% of the Sender messages, due to kernel level overwrites whereas the DELIVER reliable only lost about 8% of the messages (The discussion of why messages are lost was given earlier in Reliability Semantics section).

The ALL_DELIVER communication mechanism is quite fast when compared with the standard and ALL_REPLY communication mechanisms. This becomes apparent when the performance slope's of DELIVER reliable Send is compared with the standard and REPLY reliable Sends; there is a very small increase in performance time as the number of Receivers increase. Although, the ALL_DELIVER and K_DELIVER communication mechanisms are faster than their REPLY reliable counterparts, they do not provide the same functionality and are intended for different tasks like for notification or publishing purposes which are common in distributed applications.

6. Effect of Reliable Communication Primitives on Distributed Algorithms

To investigate the effect of reliable multicast communication primitives on algorithm design, we describe a typical distributed task, and compare three algorithms for modeling the task. The distributed task involves N transactions between a Sender process and R identical Receiver processes. The objective of the task is for the Sender to send N messages reliably to all the alive Receivers. The description of the task is intentionally kept general to note the occurrence of it in many distributed algorithms.

The following assumptions are made in the design and the analysis of the algorithms:

- the Sender and the Receivers run as processes of an operating system on separate processors connected via a broadcast medium such as Ethernet.
- processes may be addressed as a group in a group **Send**.
- the operating system retransmits the sent message T times or until operation's success conditions are met.
- the following communication primitives are available:

Unisend(*msg*, *Receiver_id*) - The *msg* is sent to the process with the *Receiver_id* process id. The issuer is blocked until a reply message is returned in the buffer *msg* from the Receiver process. The replier's process id (PID) is returned as status.

OneReply_multisend(*msg*, *group_id*) - The *msg* is sent to processes which belong to the group named group_id. The issuer is blocked until one reply message is returned in the buffer *msg* from a Receiver process. The replier's PID is returned as status. The other replies may be obtained by using the **GetReply** primitive (see below).

AllAliveReply_multisend(*msg, group_id*) - Similar to **OneReply_multisend** except that the issuer is blocked until all the replies are received from the alive Receivers. The first reply is returned in the buffer *msg* along with the status of the operation. The replies are handled similarly to **OneReply_multisend**.

GetReply(*msg, timeout*) - A reply to a previous Send is returned in the buffer *msg* . GetReply will wait up to the *timeout* value for the reply. The replier's PID is returned as status.

Receive(*msg*) - blocks the issuer until a message is received. The message is returned in the buffer *msg* along with the Sender's PID which is returned as status.

The three algorithms we compare are based on the following communications protocols:

- Using a series of reliable 1:1 communication primitives (**Unisend**) - the **Unisend Algorithm.**
- Using reliability implemented at the user level, based on unreliable multicast (**OneReply_multisend**) - the **Unreliable Multicast Algorithm.**
- Using our new kernel-level reliable multicast (**AllAliveReply_multisend**) - the **Reliable Kernel Algorithm.**

The distributed algorithms are analysed in 2 different environments namely static and dynamic. In the static environment the number of Receivers remains constant whereas in the dynamic the number of Receivers is in flux. For both environments, the algorithms using Unisend, OneReply_multisend, and AllAliveReply_multicast will be presented along with their analysis. Furthermore, the actual performance of the three algorithms under V system will be discussed.

6.1. Static Environment

In this environment, the number of Receivers remains constant during the transmission of the N messages.

6.1.1. Unisend Algorithm

The algorithm, in pseudo-code form, for the Sender and Receiver is shown in Figure 13.

One Sender process is required to serially send the same message to each of the R Receiver processes. Each Receiver acts on the message and replies (sends application level acknowledgments) to the Sender. First, the Sender has to know the PIDs of the Receivers. Therefore the Receivers must register their PID with the Sender[12]. The Sender then executes a 1:1 **Send** message to each Receiver in turn.

One of the main disadvantages of this algorithm is that the Sender needs to know the number of Receivers. This is a limiting requirement in distributed systems. The other negative point is the difficulty of synchronizing the Sender with the Receivers for the initialization phase plus the danger of Sender's lockups if one of Receiver's messages does not reach the Sender process. The lockup problem may be solved by using Receive service call with timeouts in combination with a more fault tolerant (and consequently more complicated) initialization phase. Note that lockup will not occur in the main transaction loop since Send service call returns with timeout status if no reply is received after T retransmissions by the operating system.

12 - To avoid Send-Send cycles, each Receiver should use an intermediary worker process to Send its *PID* to the Sender. For brevity this additional complexity has been omitted from the algorithm.

Receiver's algorithm

```
/* Initialization phase: register PID with the Sender */
Unisend(msg, Sender_pid);
/* end of initialization phase */

/* receive messages and perform the requested operation */
For j=1 to N   /* N = the number of messages */
  Sender_pid = Receive(msg);
  Reply(msg, Sender_pid);  /* allows Sender to continue */
  < act on message >
End /* j loop */
```

Sender's algorithm

```
/* Initialization phase: collect the PIDs of the Receiver processes */
For i=1 to R  /* R = the number of Receivers */
  Receiver_pid[i] = Receive(msg);
  Reply(msg, Receiver_pid[i]);
End /* i loop */
/* end of initialization phase */

/* send data to all the Receivers */
For j=1 to N ( the number of messages)
  For i=1 to R
    replier_pid = Unisend(msg, Receiver_pid[i]);
    if  < Unisend failed >
      < perform recovery/cleanup actions >
      < error exit >
  End /* i loop */
End /* j loop */
```

Figure 13. Unisend Algorithm for Static Environment

In the best case, when there are no lost messages, the Unisend algorithm's network message complexity is 2*R + 2*N*R where R is the number of Receivers and N is the number of messages the Sender sends to each Receiver. In the initialization phase, for each message the Receiver sends there is a reply from the Sender hence 2*R messages are sent in the system. In the main loop, there is a reply message for each message that the Sender sends which adds up to 2*N*R.

Network message complexity increases considerably, when messages are lost - a common occurrence in most distributed systems. The worst cast for Unisend occurs when each message has to be retransmitted T times by the operating system, where T is the maximum number of retransmissions, before it is received by either side. This increases the total number of network messages to 2*R*T + 2*N*R*T.

Another type of message complexity is the number of service calls which initiate user level messages (e.g. Unisend, Reply, etc). It is interesting to note that the best and the worst case for the Unisend algorithm is 2*R + 2*N*R. This is due to the absence of user level retries in this algorithm.

6.1.2. Unreliable Multicast Algorithm

These performance measurements improve when we use a multicast Send since the number of network message decreases and the parallelism between the Sender and the Receiver increases. The algorithm for the Receiver and Sender processes using OneReply_multisend is shown in Figure 14 and 15:

Receiver's algorithm

```
/* Initialization phase: register PID with the Sender */
Unisend(msg, Sender_pid);

/* join the process group */
JoinGroup(my_pid, group_id);
/* end of initialization phase */

/* receive messages and perform the requested operation */
For j=1 to N  /* N - the number of messages */
   Sender_pid = Receive(msg);  /* allows Sender to continue immediately */
   Reply(msg, Sender_pid);
   < act on message >
End /*j loop */
```

Figure 14. Unreliable Multicast Algorithm for Static Environment

Sender's algorithm

```
/* Initialization phase: collect the PIDs of the Receivers processes */
For i=1 to R  /* R= the number of Receivers */
  Receiver_pid[i] = Receive(msg);
  Reply(msg, Receiver_pid[i]);
End /* i loop */
/* end of initialization phase */

/* send data to all the Receivers */
For j=1 to N  /* N =  the number of messages */
    success = FALSE;
    For i=1 to L /* L = maximum number of user level retransmissions */
      replier_pid = OneReply_multisend(msg, group_id);
      If  < OneReply_multisend failed >
          < perform recovery actions >
          < error exit >
      < Mark reply received from the replier_pid >
      While there are outstanding replies( R-1 of them ) and no timeout
          Replier_pid = GetReply(msg, timeout);
          < Mark reply received from the replier_pid >
      End /* while loop */
      If < all Receivers replied >
        success = TRUE;
    End /* i loop */
    If < OneReply_multisend failed >
      perform recovery actions
End /* j loop */
```

Figure 15. Unreliable Multicast Algorithm

This algorithm's initialization phase also suffers from the need for the Sender to know the number of Receivers, and from Sender and Receiver synchronization problems like the

Unisend algorithm. Both Unisend and OneReply_multisend's Sender accumulate the list of Receivers' PIDs. Unisend algorithm needs the PIDs for directing its Sends to each of the Receivers and Unreliable multicast algorithm needs the list for checking off replies and determining the success of the Send operation. Note that the number of Receivers is sufficient for the Unreliable multicast algorithm in the static environment. The core of Unreliable_multicast algorithm on the Sender's part is more complicated than Unisend because of the two extra loops, one for retransmitting the message L times or until success conditions are met and the other for collecting and checking off the replies. Since the operating system stops retrying as soon as first reply arrives for OneReply_multisend, the retransmission loop is needed to ensure each Receiver having L (usually L == T) chances to receive the message.

This increased complexity at the user level, however, doesn't increase the number of needed network messages. The best case performance of this algorithm is when there is no operating system or user level retries: 2*R + N(1 + R)

The initialization phase, contributes the 2*R number of messages. N(1+R) comes from N transactions; each transaction has one Send and R replies. However, the situation deteriorates when there are retries. The worst case occurs when each message is retransmitted T times by the operating system before a reply comes back. Each reply is in turn sent T times by the Receiver's operating system before it arrives at the Sender's system. These operating system retransmissions increase (1+R) to T(1+R). The situation is worsened when user level OneReply_multisend has to be repeated L times. The worst case network message complexity of OneReply_multisend algorithm becomes:

$$2*R*T + N*T*L(1+R)$$

This is worse then the worst case performance of the Unisend Algorithm by a factor of L. The worst case complexity is exacerbated with the need for the user level retransmission loop of L iterations. This loop guarantees that the Receivers have at least K chances

of receiving the sent message. However, in this algorithm's worst case performance, the receivers may receive the message T*L times which puts this particular algorithm in a disadvantageous position. This disparity is mainly due to the semantics of the **OneReply_multisend** communication primitive.

6.1.3. Reliable Kernel Algorithm

In contrast to the Unisend and Unreliable_multicast algorithms, The Reliable Kernel algorithm is very simple to code at the user level because the user has specified that all the updates/messages must be reliably received and replied to by the Receivers for the Send operation to succeed. The algorithm for the Sender and Receiver using AllAliveReply_multisend is given below:

Receiver's algorithm

```
/* Initialization phase: join the process group */
JoinGroup(my_pid, group_id);
/* end of initialization phase */

/* receive messages and perform the requested operation */
For j=1 to N  /* N = the number of messages */
  Sender_pid = Receive(msg);
  Reply(msg, Sender_pid);
  < act on the message >
End /* j loop */
```

Sender's algorithm

```
/* send data to all the Receivers */
For j=1 to N  /* N = the number of messages */
  replier_pid = AllAliveReply_multisend(msg, group_id);
  if  < AllAliveReply_multisend failed >
     < perform recovery actions >
     < error exit >
End /* j loop */
```

Figure 16. Reliable Kernel Algorithm

The algorithm is clearly simpler than the two previous ones both for the Receiver and the Sender. The Receiver's initialization phase has been reduced to just a JoinGroup operation which registers the Receiver's PID as a member of the group. The Sender's initialization phase has totally disappeared since the Reliable kernel Algorithm's Sender does not need the Receivers' list to determine success of the Send. It also doesn't need to go through the Reply messages/acknowledgments to check off the repliers and check for success conditions. Since AllAliveReply_multisend returns success only after the Receivers have replied, the user level retransmission is not needed; the operating system will retry the Send T times until all the replies are in, thus giving all the repliers the same (T) opportunity to respond. The elimination of this loop significantly decreases the number of generated network messages. All above advantages contribute to the best case performance of this model: N(1+R+H) where H is the number of hosts without receivers that generate extra ACK messages.

Note that the initialization phase cost is 0. The worst case occurs when the operating system has to retransmit the message the maximum number of times and the repliers' systems have to retransmit the reply the same number of times to get them in to the Sender's system which accounts for the worst case network message complexity of: N*T(1+R).

It is noteworthy that the number of messages generated by the Sender in the worst case is not dependent on the number of Receivers. It is always N*T, unlike the Unreliable_multicast's N*T*R. Because of this characteristic, the maximum number of replies each Receiver may generate for each message is T hence N*T*R replies for N messages. This is drastically lower than N*T*L*R messages that may be generated by the Unreliable_multicast's Receivers.

Reliable kernel Algorithm also outperforms the Unisend model in that the Sender may only send T messages per transaction instead of T*R messages, thus cutting the

number of network messages generated by the main loop by (N-1)T*R.

The network message complexity of the three models in static environment is summarized below:

Network Msg Complexity / Algorithm	Best Case	Worst Case
Unisend	2*R + 2*N*R	2*R*T + 2*N*R*T
Unreliable Multicast	2*R + N(1+R)	2*R*T + N*L*T(1+R)
Reliable Kernel	N(1+R+H)	N*T(1+R+H)

Figure 17. Network Message Complexity Analysis Summary

This performance analysis doesn't even begin to measure all the advantages of Reliable kernel Algorithm such as the effects of parallelism achieved by using this reliable multicast communication primitive.

6.1.4. Performance Results

These algorithms were implemented in the Enhanced V system and the actual elapsed time (in seconds) for the transaction loop of 10,000 messages was measured. The graph and table below demonstrate the results for the best case performance of 1 Sender and up to 8 Receivers.

Table 4 - Time(in msecs) for each transaction using three algorithms

Algorithm \ Number of Receivers	1	2	3	8
Unisend	2.2	4.5	7.0	17.8
Unreliable Multicast	2.4	3.3	4.0	7.2
Reliable Kernel ——	2.7	3.5	4.2	7.5

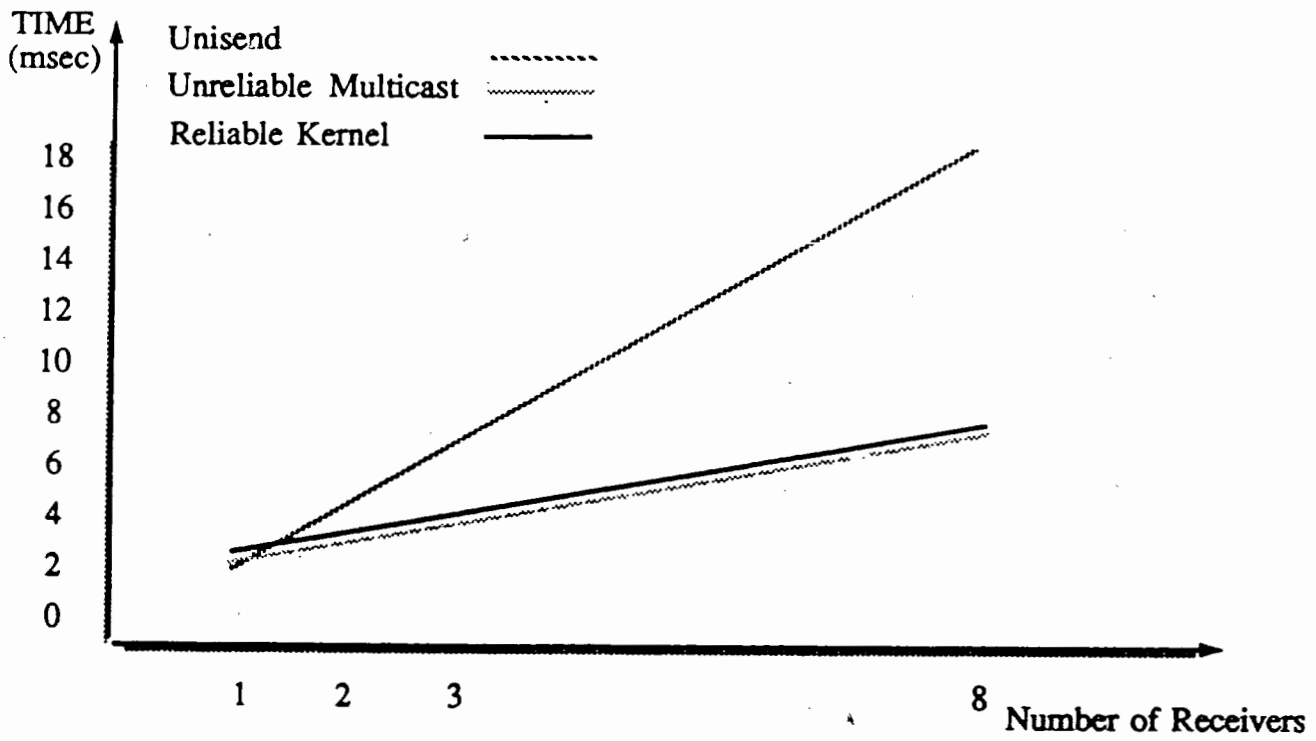

Figure 18. Time(in msecs) for each transaction using three algorithms

The group communication primitives outperform the Unisend model for 2 or more Receivers because of the lower number of messages required and the increased parallelism. In fact, the time required for Reliable kernel algorithm to send 10,000 messages to 3 Receivers is only 60% of the time for Unisend algorithm.

Moreover, the gap between the best case performance of the Reliable kernel and Unreliable_multicast algorithm is only about 5%. This performance data agree with the preceding analysis for the best case network message complexity.

Given the actual performance of the best case for the three algorithms, it is clear to see that with message loss the measurements would separate the performance of AllAliveReply_multisend communication primitive from the other primitives; hence, given the ease of programming and increased functionality of the Reliable Kernel algorithm, this is to be preferred at all times.

6.2. Dynamic Environment

Static environments are not always the norm in real distributed systems where the number of Receivers may be in flux during transactions. Static environment analysis is nevertheless useful in providing a form of lower boundary for the complexity of algorithms. In dynamic environments, the algorithms are generally more complicated to deal with varying number of Receivers. This increased intricacy arises from the need to recognize new Receivers in the group which should get the message and to detect expired Receivers which the Sender should not expect replies from. These tasks may be done in various ways. We will show one way of doing them for the distributed task discussed in the static environment section to highlight the added complexity.

6.2.1. Unisend Algorithm

The Sender's initialization phase, where the process ids of Receivers are collected, needs to be done continually even during the main transaction loop to accommodate joining or leaving Receivers. This may be done by a separate process which we will call

Member Manager. The Member Manager will loop awaiting messages from Receivers. Depending on the type of the message, the Receiver's process id will be added or removed from the Member list. Assuming that the operating system allows shared memory between processes, the Member List will be used by the Sender process which will now contain only the main transaction loop. Of course, some mechanism is needed to coordinate operations on this common data structure (e.g. semaphore, etc). The process interactions are shown in Figure 19.

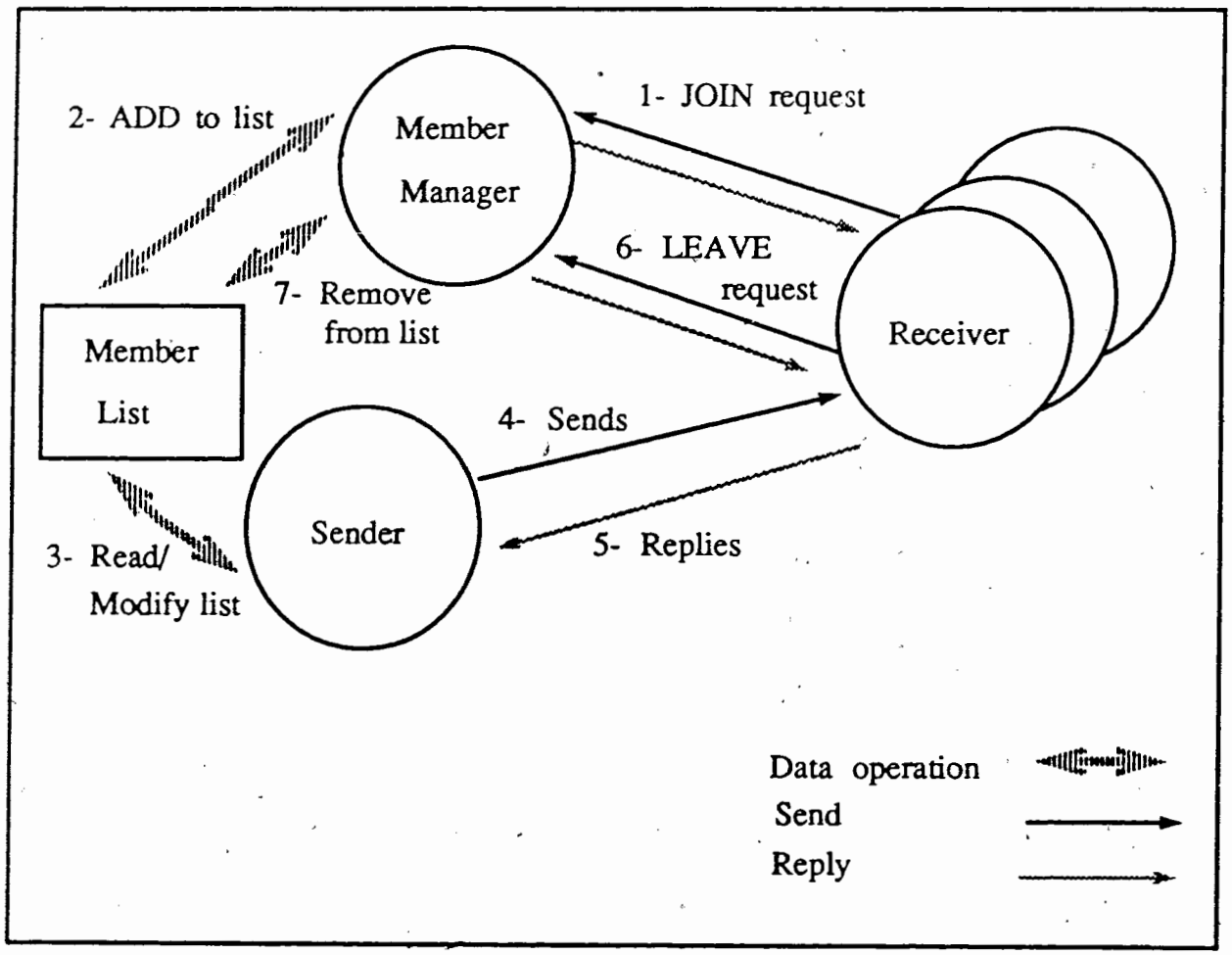

Figure 19. Process Interactions for Unisend Algorithm in Dynamic Environment

The algorithm for the Member Manager, Receiver, and Sender is given in Figure 20.

Member Manager's Algorithm

```
/* service requests from Receivers */
Forever      /* or some termination criteria */
   member_pid = Receive(msg);
   If < valid request >  /* JOIN and LEAVE requests */
        < Wait for permission to operate on the member list >
        If JOIN request
            < Add new member to member list >
        Else
            < Remove member from the list >
   Else
        < take appropriate action( log an error message ) >
   Reply(msg, member_pid)
END /* FOREVER loop */
```

Receiver's algorithm

```
/* Initialization phase: register  PID with the Sender */
Unisend(msg, member_manager_pid);
/* End of initialization phase */

/* receive messages and perform the requested operation */
For j=1 to N  /* N =  the number of messages */
    Sender_pid = Receive(msg);
    Reply(msg, Sender_pid); /* allows Sender to continue immediately */
    < act on message >
End /* j loop */

/* Termination phase: register  PID with the Sender */
Unisend(msg, member_manager_pid);
/* End of Termination phase */
```

Figure 20. Unisend Algorithm for Dynamic Environment

Sender's algorithm

```
/* send data to all the Receivers */
For j=1 to N  /* N = the number of messages */
  < Allow Member Manager to operate on member list if needed >
  For i=1 to R
     replier_pid = Unisend(msg, Receiver_pid[i]);
     if  < Unisend failed >
        < Remove Receiver  PID from the member list[12] >
  End /* i loop */
End /* j loop */
```

Figure 21. Unisend Algorithm for Dynamic Environment(cont.)

The algorithm's increased intricacy is apparent. This increase is due to additional LEAVE messages from the Receivers, the added Member Manager process for servicing requests, and the coordination of the interactions between the Sender and Member Manager processes.

6.2.2. Unreliable Multicast Algorithm

Similar to the Unisend model, a Member Manager mechanism is needed for constructing the list of Receivers from which replies should be collected. This is again done by a Member Manager process. The overall process interactions are shown in Figure 19. The Unreliable Multicast algorithms for the Member Manager and Receivers are the same as the Unisend Algorithm with one difference; the Receiver joins the global group using the

13 - Note, that the Sender also has to deal with Receivers who leave ungracefully, because of a fault, without sending LEAVE requests to the Member Manager process.

standard V **JoinGroup** service call in its initialization phase after registering its PID with the Member Manager process. The Sender's algorithm, however, is different and is given below:

Sender's algorithm

```
/* send data to all the Receivers */
For j=1 to N  /* N =  the number of messages */
   < Allow Member Manager to operate on member list if needed >
   success = FALSE;
   For i=1 to L /* L = maximum number of user level retransmissions*/
      replier_pid = OneReply_multisend(msg, group_id);
      If  < OneReply_multisend failed >
          < perform recovery actions >
          < error exit >
      < Mark reply received from the replier_pid >
      While < there are outstanding replies( R-1 of them ) >
          Replier_pid = GetReply(msg, timeout);
          < Mark reply received from the replier_pid >
      End < while loop >
      If < all Receivers replied >
        success = TRUE;
      Else
          < Remove non replying Receivers from the member list >
          < Perform the needed recovery actions >
   End /* i loop */
End /* j loop */
```

Figure 22. Unreliable Multicast Algorithm for Dynamic Environment

The Unreliable multicast algorithm performs the same additional functions as the

Unisend Algorithm in the Dynamic environment.

6.2.3. Reliable Kernel Algorithm

Because of AllAliveReply_multisend's semantic and characteristic, the Reliable Kernel algorithm for the dynamic environment is the same as the static environment's algorithm. This demonstrates another strength of kernel level reliable multicast communication primitives in specific AllAliveReply_multisend.

Since AllAliveReply_multisend at the kernel level uses knowledge of the alive hosts coupled with the inter-kernel protocol in providing reliable multicast, the user level algorithm need not handle the complicated task of maintaining the list of alive Receivers in the system. This task is done by the Sender kernel's knowledge of the alive host and each kernel's knowledge of local membership lists. Therefore the worst/best case network message complexity remains constant in both static and dynamic environments for the Reliable Kernel algorithm.

The preceding algorithms and analysis show the superiority of using kernel level reliable multicast communication primitives for distributed tasks, requiring reliability, over the user level implemented reliable multicasts. The key features of kernel level reliable multicast communication (e.g. AllAliveReply_multisend) are the efficiency in number of generated messages both at user and system level, better performance over user level implemented reliability, and ease of use at the user level.

7. Conclusion and Future Work

Reliable multicast communication has the advantages of ease-of-programming over unreliable broadcast for applications where reliability is important. It also has greater efficiency resulting from only having to transmit the Send message once. However, there is a small overhead which does not occur in one-to-one communication or unreliable multicast. When no errors occur this overhead occurs from requiring those hosts with no Receivers to acknowledge the Send packet. When an error does occur, and a Send message, or the response to a Send message, is lost then the packet is rebroadcast. There is an increased likelihood of an error occurring since the message must be picked up and acknowledged by more hosts. Despite these facts, for two or more Receivers, reliable multicast communication is faster than a series of 1:1 communications.

Analysis of the number of messages exchanged also shows the effectiveness of multicast. For example, consider a network with M hosts, and a group with N members, excluding the Sender. On a sequence of one-to-one reliable Sends, 2*N messages are exchanged. On our reliable multicast communication scheme 1 + (M-1) messages are exchanged. Therefore, if $N > M/2$ our scheme for reliable multicast is more efficient in terms of the number of messages exchanged. This is the case for many applications such as highly parallel computations in a PROLOG or a DATA-FLOW environment; and for communication between the group of kernel processes (which has M members, one for each host).

There are many possible applications for an interprocess communication primitive which can send a message and receive replies from more than one process. This thesis has extended the work of Cheriton and Zwaenepoel to provide reliable multicast communication in the V-kernel. The reliable protocol has been described, along with a detailed discussion of its implementation and performance.

We found that our reliable multicast out-performed a series of reliable 1:1 Sends for

two or more Receivers and added no overhead to the performance of standard V inter-process communication primitives.

The I_Am_Here server also provides user-level services such as getting a copy of the alive kernel vector and obtaining the host id given the bit position. These services, plus the new ALL_REPLY and K_REPLY communication primitives, provide powerful tools to implement higher level communication services.

The reliable multicast scheme has been implemented in the V-kernel, as part of a project exploring the use of multicast in distributed programming.

7.1. Possible Improvements and Future Work

The V system's Send user interface was modified to accommodate reliable multicast communication. In the current interface, SUCCESS or FAILURE is returned to the Sender of the Reply reliable Send primitive, hence the first replier's PID is not accessible to the Sender. This interface should be modified so that the Reply reliable Send returns the PID of the first replier and puts the return status in the reply message itself (see Appendix A for the detailed description of the new interface).

Another possible improvement is in the area of overhead reduction. The kernel code could be streamlined such that the standard V multicast users incur a lower performance overhead. Such optimizations should also be done for data structure accesses, such as alive host table searches, that are done in providing reliable multicast communication.

The need for getting ACKS from hosts with no Receivers should be investigated and eliminated. This would increase scalability of these protocols. One possible way of accomplishing this is the use of multiple multicast addresses, one per group. This may be done by modifying the I_Am_Here server to send the active group ids in the local kernel in each I_Am_Here message. The knowledge of group ids and hosts may be used in determining the list of hosts which should send ACK messages.

An interesting research project could involve the performance measurement of the

reliable primitives for messages that have 1K or more of data. The insights gained from these measurements could increase the appeal of the reliable multicasts for distributed applications that need to reliably exchange large amounts of data such as distributed file systems. Lastly, the effect of these reliable multicast primitives should be studied in different distributed applications. This will be valuable in refining the reliable multicast primitives' interface and protocol.

Appendix A - Improved User Interface for Reliable Communication Primitives

The Reply reliable multicast Send's interface should be improved to provide the PID of the first replier to the Sender. The following describes this new interface for the reliable Send primitive:

Send(message, id)

The fixed length message is sent to a single process if the id is a process id or to a group of processes if the id specifies a group, in which case multicast is used. The multicast Send blocks for zero or one replies depending on the *syscode* and *code* fields defined in the message. The user specifies the type of Send by setting appropriate fields in the message. The **syscode** field may be set to K_REPLY, K_DELIVER, ALL_REPLY, or ALL_DELIVER. For K_REPLY or K_DELIVER the **code** field must contain an integer specifying the value of K, where K >=1. For the new reliable multicast Send, the Send blocks until either conditions specified by the syscode field are satisfied or conditions are found unsatisfiable.

The syscode field types for the new features of group communication are defined below:

K_REPLY

Send blocks until K replies are received from the group members specified by id, in which case the process id (PID) of the first replier is returned, and the Sender's message is overwritten by the contents of the first reply. After the maximum number of timeouts and retransmissions have been done, if no replies were received then FAILURE is returned. Otherwise, the PID of the first replier is returned, and the Sender's message is overwritten by the contents of the first reply. In either case, Send sets the *code* field in the message to the number of replies received. Note that the issuer should check the code field to determine success of the K_REPLY send when a PID is returned.

ALL_REPLY

Send blocks until replies from all the alive group members in the group specified by id are received, after which the PID of the first replier is returned, the *syscode* field is set to SUCCESS, and the Sender's message is overwritten by the first reply. After the maximum number of timeouts and retransmissions have been done, if no replies were received then FAILURE is returned. Otherwise, (replies did not come in from all the alive group members) the PID of the first replier is returned, the *syscode* filed is set to FAILURE, and the Sender's message is overwritten by the contents of the first reply. In either case, Send sets the *code* field in the message to the number of replies received. Note that the issuer should check the *syscode* field to determine success of the ALL_REPLY send when a PID is returned.

References

[1] Ahamad, Mustaque and Bernstein, A.J., "Multicast Communication in UNIX", *Proceedings 5th International Conference on Distributed Computing Systems*, pp. 80-87, May 1985.

[2] Andrews, G.R., "The Distributed Programming Language SR--Mechanisms, design and implementation", *Software P&E*, vol. 12(8), pp. 719-754, August 1982.

[3] Andrews, G.R. , and et. al., *ACM Transactions on Programming Languages and Systems*, vol. 10(1), Jan 1988.

[4] Atkins, M.S. and Carter, A.W., "Reliable Multicast Interprocess Communication", *CIPS Conference*, Vancouver, April 1986.

[5] Birman, K., "Replication and Availability in the ISIS system", *Proc. of 10th ACM Symposium on Operating Systems Principles*, pp. 79-86, 1985.

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

[7] Chang, Jo-Mei and Maxemchuk, N.F., "Reliable Broadcast Protocols", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp 251-273, August 1984.

[8] Cheriton, D.R., "The V kernel: a Software Base for Distributed Systems", *IEEE Software*, vol. 1(2), April 1984.

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

10] Cheriton, D.R., "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design", *Proceedings 6th International Conference on Distributed Systems*, May 1986.

[11] Cheriton, D.R., "Report on the Second European SIGOPS Workshop -- Making Distributed Systems Work", *Operating Systems Review*, pp. 72-73, Jan. 1987.

[12] Clark, D.D., "The structuring of Systems Using Upcalls", *Proceedings of 10th ASCM Symposium on Operating Systems Principles*, pp. 171-180, Dec. 1985.

[13] Deering, S.E., "Multicast Routing in Internetworks and Extended LANs", *ACM SIGCOMM Symposium*, Aug 88.

[14] Garcia-Molina, H. and Kogan, B., "An Implementation of Reliable Broadcast using an Unreliable Multicast Facility", *Symposium on reliable Distributed Systems*, Oct. 1988.

[15] Garcia-Molina, H. and Spauster, A., "Message Ordering in a Multicast Environment",

Technical Report CS-TR-161-88, Princeton University.

[16] Hammer, M. and Shipman, D., "Reliability Mechanisms for SDD-1", *ACM Transactions on Database Systems*, vol5(4), 1980.

[17] Narayanan, Parthasarathy, "Study of Reliable Broadcast Protocols in Fault Tolerant Distributed Computing Systems", *PhD. dissertation*, 6th National Polytechnic Institute, Toulouse, July 1984

[18] Navaratnam, S., Chanson, S., and Neufeld, G., "Reliable Group Communication in Distributed Systems", *8th International Conference on Distributed Computer Systems*, June 88.

[19] Nehmer, J., Haban, D., and et. al., "Key Concepts of INCAS Multicomputer Project", *IEEE Transactions on Software Engineering*, vol. SE-13(8), pp. 913-923, August 1987.

[20] Ravindron, K., Chanson, S., and Ramakrishnan, K.K., "Application-driven Failure Semantics of Interprocess Communication in Distributed Programs", *TR 87-3*, Dept. of Computer Science, University of British Columbia, Jan. 1987

[21] Rothnie, J.B. and Goodman, N., "An overview of the Preliminary Design of SDD-1: A System for Distributed Databases", *Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, 1977.

[22] Rowe, L.A. and Birman, K., "A Local Network based on the UNIX Operating System", *IEEE Transactions on Software Engineering*, vol. SE-8(2), pp. 137-146, March 1982.

[23] Santoro, N. and Sidney, J.B., "A Reduction Technique for Selection in Distributed Files", *TR SCSTR-23*, School of Computing Science, Carleton University, Ottawa, Canada, April 1983.

[24] Schlichting, R.D., Andrews, G.R., and Purdin, T.D.M., "Mechanisms to Enhance File Availability in Distributed Systems", *TR 85-24*, Computer Science Dept., University of Arizona, Oct. 1985.

[25] Shoch, J. and Hupp, J., "Measured Performance of an Ethernet Local Network", *Communications of ACM*, vol 23(12), Dec. 1980.

[26] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., "The LOCUS Distributed Operating System", *Proceedings of the 9th Symposium on Operating Systems Principles*, ACM, Oct. 1983.

[27] Walter, B., "A Robust and Efficient Protocol for Checking the Availability of Remote Sites", *Proc. 6th Int. Workshop on Distributed Data Management and Computer Networks*, 1982.