Approved for public release; distribution unlimited

A SCALABLE DECENTRALIZED GROUP MEMBERSHIP SERVICE FOR AN ASYNCHRONOUS ENVIRONMENT

by

David S. Neely
Lieutenant , United States Navy
B.S.C.S., University of Washington, 1986

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June, 1994

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A SCALABLE DECENTRALIZED GROUP MEMBERSHIP SERVICE FOR AN ASYNCHRONOUS ENVIRONMENT			5. FUNDING NUMBERS
6. AUTHOR(S) David S. Neely			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis presents a globally scalable, decentralized group membership service to manage client process groups operating in a distributed, asynchronous environment. This group membership service is totally scalable, handling process groups spanning a single LAN to groups spanning the entire global Internet equally well. It provides for nested and overlapping groups, as well as multiple groups residing on a single LAN. It also provides various Quality of Service selections which permit individual groups to be configured for an optimal balance between high quality with strong consistency semantics for group membership, and weaker consistency semantics with reduced complexity and latency.

This thesis describes the complete design of the protocol used to implement the group membership service. It presents the design requirements and goals, and underlying assumptions about the network. The various Quality of Service selections provided by the group membership service are described in detail, as well as the interface between the process groups, the membership service, and the underlying network. The use of a hierarchical architecture to obtain the desired scalability, flexibility, and robustness is explained. A proof of correctness for the protocol is presented, and a partial implementation of the group membership service is described.

14. SUBJECT TERMS group membership, process groups, scalability, multicast, reliable distributed computing			15. NUMBER OF PAGES 194
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

ABSTRACT

This thesis presents a globally scalable, decentralized group membership service to manage client process groups operating in a distributed, asynchronous environment. This group membership service is totally scalable, handling process groups spanning a single LAN to groups spanning the entire global Internet equally well. It provides for nested and overlapping groups, as well as multiple groups residing on a single LAN. It also provides various Quality of Service selections which permit individual groups to be configured for an optimal balance between high quality with strong consistency semantics for group membership, and weaker consistency semantics with reduced complexity and latency.

This thesis describes the complete design of the protocol used to implement the group membership service. It presents the design requirements and goals, and underlying assumptions about the network. The various Quality of Service selections provided by the group membership service are described in detail, as well as the interface between the process groups, the membership service, and the underlying network. The use of a hierarchical architecture to obtain the desired scalability, flexibility, and robustness is explained. A proof of correctness for the protocol is presented, and a partial implementation of the group membership service is described.

C.1

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to express my sincere gratitude and appreciation to Professor Shridhar Shukla for the help he provided in the completion of this thesis. He has been not only an advisor, but a mentor, collaborator, partner, and friend. Without his extensive contributions, the accomplishments of this thesis would not have been possible.

I would also like to thank my family for their love and support while I was working on this thesis. My wife Lynda, sons Daniel and Dayan, and daughter Brionne, endured my ordeal very well, and kept me going through the difficult times. It is to them that I dedicate this thesis.

I. INTRODUCTION

A. BACKGROUND

Distributed networks of computers are being used increasingly to provide computational power and services beyond the capabilities of a single computer system. Distributed application programs specifically designed to utilize the distributed networks of computers are gaining wide recognition as a powerful, flexible, and efficient method of performing computation. Often, these distributed applications can be logically grouped to allow more efficient and capable interaction. The process group paradigm has been shown to be particularly well suited to organizing these distributed applications into a single entity working toward a common goal. Examples of distributed applications that can benefit from the use of process groups include multimedia teleconferencing, distributed system management, remote monitoring and control systems, distributed reliable databases, banking and brokerage services, distributed interactive simulation (DIS), as well as a multitude of other applications. These process groups can be arranged in many possible configurations to suit the needs of the particular application. Examples of various process group configurations are shown in Figure 1.

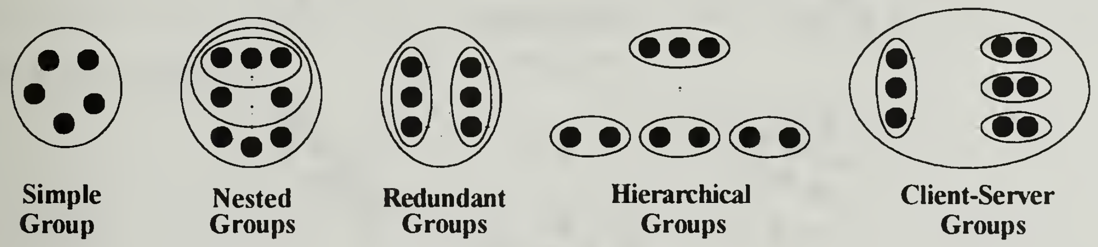


Figure 1: Process Group Configurations

Process group oriented computation based on reliable communication primitives has been shown to be particularly effective in a wide variety of environments [1, 2, 3, 4]. In this paradigm, a group may correspond to a set of processes that must behave consistently to provide a service or make a decision. Changes in the membership of the group may occur due to the voluntary arrival and departure of members, or failures and recoveries caused by the dynamic nature of the members. Therefore, a Membership Service (MS) to manage the group membership is a fundamental building block for distributed applications using the process group model.

To construct usable process groups, an MS must first overcome the group membership problem (GMP); that is, providing consistent agreement on the membership of the group at all members in spite of dynamic changes to the group [5]. This problem is compounded by the asynchronous nature of the networks upon which the process groups operate. Additionally, an MS must be scalable to support groups of any size and distribution. The MS must be efficient, robust, and flexible to continue to provide services to the client process groups under any circumstances. The MS must provide a uniform interface to all applications, hiding the details of the process group management from the users of the MS. An illustration of the logical representation of the MS is shown in Figure 2.

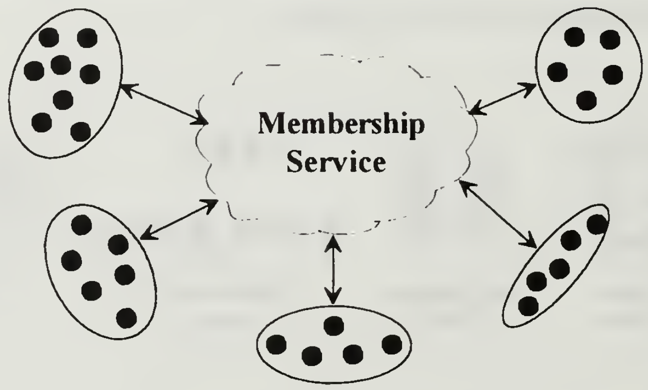


Figure 2: Membership Service and Application Process Groups

B. SCOPE AND ORGANIZATION

This thesis presents the design of a globally scalable, decentralized group membership service to manage application process groups operating in a distributed, asynchronous environment. The scope of this thesis covers an investigation into current group membership protocols and membership services; the identification of the design requirements for an MS; the design of a hierarchical, scalable MS that meets all of the design goals; the detailed specification of the protocols which form the MS; and a partial implementation of an MS running on a campus network.

The organization of this thesis is as follows. The first chapter provides an introduction to the needs and requirements of distributed application process groups and the services provided by a membership service. The second chapter describes the necessary and useful attributes of a full-featured MS, followed by a survey of current group membership protocols and membership services. The third chapter provides a detailed description of the hierarchical architecture and components of the MS. The fourth chapter provides a detailed description of the five protocols required to implement the MS, including algorithmic psuedo-code specifications of each. The fifth chapter provides a proof of correctness for the MS protocols, ensuring that the MS meets the stated design requirements. The sixth chapter includes an implementation of a set of software utilities used by the MS and a partial implementation of the MS protocols. The final chapter provides conclusions about the design of the MS and a discussion of future work to be completed.

II. ATTRIBUTES OF A MEMBERSHIP SERVICE

In this chapter the desirable and necessary attributes that a general purpose MS must possess are described. Design goals for an MS which has all of the required attributes are outlined. The network and user interfaces for an MS are defined. Finally, a survey of current group membership protocols and services is provided, showing the need for a full-featured MS.

A. DEFINITIONS AND ASSUMPTIONS

Before describing the attributes, requirements, and features of an MS, the operating environment must first be defined. Certain assumptions about the functioning of the underlying network and the processes which comprise the MS and application groups must be made. These assumptions are outlined below.

1. The Network

Few assumptions about the service provided by the underlying networks and internetworks are made. These networks are assumed to be asynchronous and unreliable, with only connectionless, "best effort" datagram delivery provided, with unbounded delivery time. Messages may be lost, delayed, duplicated, garbled, or arrive out of order. Furthermore, the networks may suffer partitions, leading to the interruption in communications between end stations for variable periods of time. It is assumed that a network multicast capability is provided, such as IP multicast [6, 7, 8]. This multicast capability is assumed to provide rudimentary group management for the set of hosts which share a common multicast address, including the creation and maintenance of a multicast routing tree, and the detection and removal of processes which are not responding.

2. The Processes

Computer processes executing on distributed host computers throughout the network are the entities which form the MS as well as the application process groups which use the MS. It is assumed that the host computers and processes running on them are unreliable and may fail at any time. The failure of the host computer or the process running on the computer are indistinguishable from the perspective of the MS. It is assumed that these failures will be fail-stop, or crashes [5, 9, 10, 11]. The computers or processes will simply cease to function, with no malicious behavior.

The exchange of messages is the only way that distributed processes can learn of each other's status. Due to the unreliable nature of the network described above, these messages may never reach their destination, even though both sender and receiver are functioning normally. For this reason, it is impossible for distributed processes to distinguish between network partitions and the actual failure of other processes [5, 9, 10, 11]. Therefore, the failure of another process can only be perceived, never known for sure. Perceived failures are detected by the lack of response within a timeout period. Although these perceived failures may be caused by a partition of the network or the actual failure of the process, they will be treated as if the process had actually failed.

B. DESIRABLE ATTRIBUTES OF A MEMBERSHIP SERVICE

A membership service must provide a suite of services to manage group-oriented applications. Some of these services are explicitly invoked, such as calls to create new process groups, to have processes join or depart the process group, or to split or merge the process group. Other services are implicitly and automatically provided by the MS, such as detecting and processing member failures within the group, detecting and processing partitions of the network, ensuring unique group names within a given scope and providing consistency of ordering of group membership changes at all members. Still other services provide information to applications upon request, such as group name, size, membership, view number, and automatic notification of group membership changes. A membership service also has certain inherent attributes, such as scalability, fault-tolerance, efficiency and flexibility. Table 1 lists several desirable attributes that a general purpose MS should posses to fully support application process groups.

TABLE 1: DESIRABLE ATTRIBUTES OF A MEMBERSHIP SERVICE

Attribute	Interpretation	Significance
A: Adaptive status monitor	Adjust timeouts based on local conditions	Minimize wrongly perceived failures
H: Hierarchical protocol	Multilevel membership maintenance	Exploit hierarchy in WANs, support very large groups
L: sca**L**ability to large groups	Absence of centralized actions in the protocol	Support of large, extensively overlapped groups
M: Multiple network support	Distribution over heterogeneous networks	Novel applications
N: Non blocking reconfiguration	Processing of continuous status changes	Enhanced performance for highly dynamic groups
O: topology-based **O**ptimization	Use of physical topology and LAN features	Support of widely distributed groups
P: network **P**artitioning	Merging after recovery with required consistency	Increased applicability of Membership Service
R: Real-time service	Guaranteed detection and processing latency for changes	Support real-time applications
S: multiple **S**imul-taneous changes	Quick update with weaker consistency	Multiple classes of service with overhead proportional to quality
X: fle**X**ible member-ship semantics	Availability of a range of consistency semantics	

It should be noted that some of these desirable attributes listed in Table 1 conflict with each other. For example, adjustment of timeouts based on local conditions will violate the real-time aspect of the MS. Non-blocking reconfiguration and merging after partitions conflict with providing strongly ordered membership change semantics. Thus, a fully-featured MS must permit the membership service user (MSU) to choose which of these conflicting desirable attributes will have priority. The MSU is given the option of choosing various Quality of Service (QoS) selections to configure the MS to the exact needs of the application.

C. MEMBERSHIP SERVICE DESIGN GOALS

In this section the design goals of a full-featured MS are described.

1. Scalability

The MS must be completely scalable. Application process groups spanning a single local area network (LAN) or the worldwide Internet will see the same level of service. To accomplish this goal, the membership information for all groups must be maintained hierarchically. Information about process groups will be distributed throughout the hierarchy, so that each node need only store and process information for the application groups that it supports directly below it. In this manner, the MS nodes that have no member processes for a particular application are in no way impacted by the processing of membership changes for this application. Additionally, the MS will use a decentralized, hierarchical decision making scheme, since a centralized scheme is not scalable. The decisions about membership changes to application groups will be made by a set of distributed nodes located in the hierarchy, which will then propagate the decision to all process group members. By using the hierarchical nature of the MS, the number of nodes involved in each membership change decision will be small. Additionally, the level of the set of nodes in the hierarchy will be different for most process groups, since the span of most groups will be different. Thus, different parts of the hierarchy can function concurrently, processing membership change decisions for different groups at different levels without affecting the operation of the other parts of the hierarchy.

2. Efficiency

The MS must be efficient in the use of computational and network resources in order to be scalable. Using the decentralized hierarchical structure, each node in the hierarchy need only process and store a small part of the information needed to support all application process groups.

Since hosts computers attach to the internetwork through a LAN, access to the MS must be present at the LAN level at all times, even if there are no groups present on a particular LAN. This will drastically reduce the latency for creating new groups and permits the use of special LAN-level features such as hardware multicast. To provide this continual access, a daemon process should be running on each MS capable host computer, and an MS node should be running on a dedicated server for the LAN. The daemon process provides an interface between the MSU and the MS.

Multicast messages must be used to process all changes, since multicasting is an extremely efficient method for multiple processes to communicate. Additionally, the use of a hierarchy provides a natural funneling effect for multiple messages propagating to higher levels in the hierarchy. This is a form of concast [12], reducing the load on the network at each level in the hierarchy.

3. Resilience to Failures and Partitions

The MS must provide membership semantics that handle failures of members as well as the underlying network. Failures of either members or the network must be automatically detected and processed, reforming the group without any direct intervention by the application processes or the MSU. The MS must use a decentralized protocol to eliminate any single point of failure. Multiple simultaneous failures of member processes must be detected and processed without blocking, usually by "batching" the failures into a single change to the membership.

4. Levels of Consistency

As identified in other group membership protocols [11, 13, 14], there are various possible levels of consistency in the ordering of changes to the membership view at members of a process group. Strong consistency guarantees that all members see exactly the same changes to the group membership in exactly the same order. Weak consistency guarantees that all group members will eventually reach the same view of the group membership, but may hold disparate views for some period of time. Strong consistency requires added complexity and overhead to ensure that all members have the same ordering of membership changes, while weak consistency relaxes the requirements required by strong consistency, and therefore is less complex. Strong consistency must block all changes to the membership until the current change finishes, while weak consistency may process concurrent changes. Thus, weak consistency generally has a reduced latency over strong consistency. The MS must provide flexible membership semantics for the application groups supported, allowing the MSU to select the level of consistency needed for the particular application.

5. Membership and Name Scope Control

The MS must provide a means to limit the extent of individual application groups. Without such a limit, all application groups could potentially use the whole MS hierarchy, even if only a small part of the hierarchy was actually needed, creating a bottleneck at the highest level in the hierarchy. The use of "scope control" parameters limits the maximum span of an application group in the MS hierarchy to the referenced level. Membership scope control limits the extent of group name searches whenever an application group is referenced, such as a request to create a new group or join an existing group. Name scope control limits the maximum span in the MS hierarchy which an application group can cover. This parameter can be used when the application group is created, and specifies the highest level in the MS hierarchy at which the group name should be registered. References to an application group outside of the name scope will not find the application group, and will propagate to the highest level in the MS hierarchy unless limited by the membership scope control parameter.

6. Selectable Quality of Service

The MS must permit user selection of the conflicting desirable attributes identified in Table 1. Some of the QoS selections which must be supported include: the level of consistency in ordering of membership changes, methods of resolving partitions in application groups, adaptive status monitor conditions to adjust the MS for local conditions, designation of a limited scope for the application group, and user configuration of the MS hierarchy for special purpose applications. An MSU must be able to select the desired level of service by specifying certain parameters related to the QoS. These parameters specify how application group partitions are handled, how the scope of a group name is controlled, how the membership change information is ordered, the setting of the failure detection timeouts, and the aggregation of multiple simultaneous changes.

D. MEMBERSHIP SERVICE INTERFACES

In this section the relation of the MS protocols is defined with respect to the Internet Protocol (IP) protocol stack, which is the de-facto standard for internetworking communications. Additionally, the application user's interface and the MS system configuration interface are described.

1. Network Protocol Layers

Figure 3 illustrates the relation of the MS protocols to the Transmission Control Protocol/ Internet Protocol (TCP/IP) suite of internetworking protocols in the layer below, and the application programs and upper-layer protocol modules in the layer above, using the common layering model of depicting the hierarchical dependencies of network protocols.

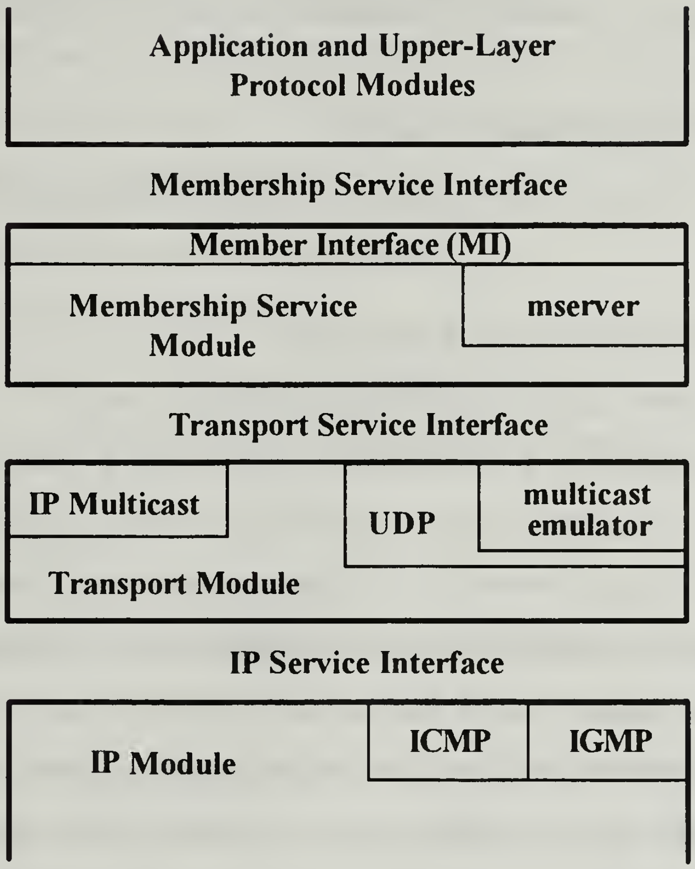

Application and Upper-Layer Protocol Modules
Membership Service Interface
Member Interface (MI)
Membership Service Module
mserver
Transport Service Interface
IP Multicast
UDP
multicast emulator
Transport Module
IP Service Interface
IP Module
ICMP
IGMP

Figure 3: Protocol Layers

2. User Interface

The application user's interface to the MS is provided through explicit system calls to alter the membership or provide information about application process groups. The MS is implicitly called to change the membership of application groups any time a process failure or network partition occur. The following lists explain these system calls and events in more detail.

a. *Informational*

1. **Group View** ("group")

Provide the current group view number maintained by the MS. Used by application processes to guarantee all members have the most recent view of the group membership.

2. **Group Statistics** ("group")

Provide current group view number and membership list maintained by the MS.

b. *Explicit Membership System Calls*

1. **Join** ("group", membership_scope, name_scope)

Request by a new member process to join a group which may or may not already exist. If the group does not presently exist, a new group is formed with only this member. If the group does exist within the requested scope, the MS processes the change and informs the application group of the addition. The membership_scope field is used to specify the highest level in the MS hierarchy which should be searched for the application group name during a join, thus limiting the time required to determine if the group exists, and the impact on other groups. The name_scope field is used during the creation of the process group to specify the maximum span the application group will ever cover in the MS hierarchy. This field limits the extent of the search required whenever a group is referenced.

2. **Leave** ("group", gid, membership_scope)

Request by member with group identity number "gid" to leave a group. The departing member is able to leave immediately, without waiting for a response from the MS.

3. **Merge** ("group1", "group2")

Request by member of group1 to merge group1 and group2. Upon successful completion, the union of the two groups will be formed, using group name "group1", with a new group view. This request is the general form of the join request.

4. **Split** ("group1", "group2", g2MemberList)

Request by a member of group1 to remove one or members of group1, listed in the parameter g2MemberList, and form a new group2 with these members. This request is the general form of the leave request

c. Implicit Membership Altering Events

1. Failures and Partitions

The MS will automatically handle perceived failures of group members, up to and including all members. Automatic notification of member failures is provided to the application group.

3. System Configuration Interface

The system calls used to configure the MS hierarchy are virtually the same as those used by application groups, with the exception of calls to make certain nodes parent nodes of others, thus creating the hierarchy. The configuration of the MS is performed by a system administrator, using individual command line system calls or an MS configuration program called *MS_mgr*.

E. CURRENT PROTOCOLS

A summary of existing membership protocols is provided in Table 2. The category headings are the same desirable attributes of a membership listed in Table 1. Finally, a listing of the design goals and desirable attributes contained in the MS presented in this thesis is shown in Table 3 for comparison.

Unlike any known group membership protocol, the group membership service described in this thesis is totally scalable, handling process groups spanning a single LAN to groups spanning the entire global Internet equally well. It provides for nested and

TABLE 2: A SUMMARY OF EXISTING MEMBERSHIP PROTOCOLS

Index to Columns: see Table 1.

Index to Entries: ✓: Supported, **X** : Not supported,
E : Support possible with extensions, -- : unknown

Protocol	**Required Network Properties**	**Principle Feature**	**A**	**H**	**L**	**M**	**N**	**O**	**P**	**R**	**S**	**X**
Asynchronous Environment:												
Chang *et al.* [15]	unreliable message	token site	X	X	X	X	X	X	X	X	✓	X
Bruso [16]	message diffusion	version numbers, stable storage	E	X	X	X	X	X	X	X	E	X
El Abbadi *et al.* [17]	unreliable message	virtual partitions	E	--	✓	✓	X	E	✓	X	✓	X
Verissimo *et al.* [18]	broadcast LAN	two-phase accept	X	X	X	X	X	X	X	✓	E	X
Moser *et al.* [19]	ordered, reliable	ordinal numbers	X	X	X	X	X	E	X	X	X	X
Riccardi *et al.* [9]	unreliable message	reconfiguration manager	E	E	E	E	✓	E	X	X	X	X
Mishra *et al.* [20]	ordered, reliable	Psync & conversations	X	X	E	E	X	E	--	X	✓	E
Auerbach *et al.* [21]	multicast hardware	multicast sequences	X	E	X	X	--	✓	✓	X	✓	X
Jahanian *et al.* [13]	unreliable message	crown prince	E	E	E	E	--	X	E	X	✓	✓
Golding *et al.* [22]	unreliable message	time-stamped anti-entropy	E	✓	✓	✓	✓	X	✓	X	✓	X
Synchronous Environment:												
Cristian [5]	bounded delay	attendance lists	X	X	X	X	✓	X	X	✓	E	X
Ezhilchelvan *et al.* [23]	bounded delay	time-domain multiplexing	X	X	X	X	✓	X	X	✓	✓	X
Kim *et al.* [24]	TDMA bus	reception history	X	X	X	X	✓	X	X	✓	✓	X
Rodrigues *et al.* [25]	exposed LAN interface	transmit-with-response	X	X	X	X	✓	X	X	✓	✓	X

overlapping groups, as well as multiple groups residing on a single LAN. It also provides various Quality of Service selections which permit individual groups to be configured for an optimal balance between high quality with strong consistency semantics for group membership, with the associated complexity and latency, and weaker consistency semantics with reduced complexity and latency.

TABLE 3: ATTRIBUTES OF THE MEMBERSHIP SERVICE

Required Network Properties	**Principle Feature**	A	H	L	M	N	O	P	R	S	X
Unreliable messages	Decentralized protocol based on ordered membership	✓	✓	✓	✓	✓	✓	✓	X	✓	✓
Bounded delay message delivery		**X**	✓	✓	✓	✓	✓	**X**	✓	✓	✓

III. MEMBERSHIP SERVICE ARCHITECTURE

At the foundation of the scalable and efficient Membership Service lies the architectural structure. The key to a scalable Membership Service is a decentralized, hierarchical architecture. The Membership Service uses a hierarchical architecture designed to follow the pre-existing physical topology of the subnetworks, networks, and internetworks upon which the distributed application process groups that the Membership Service supports will be running. This chapter describes the structure and composition of the physical hierarchy of the MS and how this architecture supports application process groups.

A. PHYSICAL HIERARCHY

The relevance of the architecture of the MS to the scalability of the MS is obvious when the global scale is considered. There are presently over 120 million computers and 1 million LANs world-wide, connected by bridges and routers to form global internetworks. A centralized MS would require the central node to interact directly with all of these computers distributed throughout the world, clearly an impossibility. By forming a logical hierarchy, the interaction required by each node in the hierarchical tree is limited to those nodes directly above and below, providing a uniform load for any node in the hierarchy. The significance of the hierarchical structure is illustrated in Figure 4, where an *n-ary* hierarchical tree is formed with eight levels of ten nodes each, providing support for virtually all of the world's computers at the leaf level. With this hierarchy it is possible for any leaf computer to communicate with the root level of the tree with only six intermediate relays by nodes in the tree. If these intermediate nodes are logically connected in a manner which closely follows their physical connectivity, the connection from leaf to root level could require as few as six physical communication links.

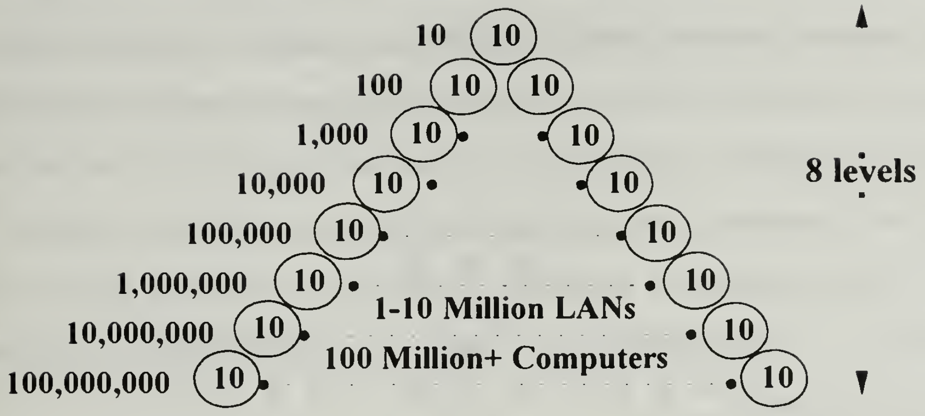


Figure 4: Global Hierarchy

The other significant aspect of the hierarchical structure is that a node at each level of the tree only need interface with the parent node above and the children nodes below. In Figure 4, each node communicates directly with only ten children nodes and one parent node. This is in comparison to the interaction in a centralized MS, where a single manager node must communicate directly with all other nodes in the MS - potentially millions of nodes managed by a single manager.

1. Mservers and Member Interfaces

The MS is comprised of two primary entities: Membership Servers (mservers) and Member Interfaces (MI). The mservers are the heart of the MS, forming the nodes of the hierarchy. The mservers are processes running on routers or host computers distributed throughout the internetwork. The mservers provide connectivity, routing, and record-keeping functions in a distributed, decentralized manner for the MS. The mservers are primarily responsible for processing changes and providing information to the members of both the physical hierarchy as well as the application process groups using the MS. Typically, one mserver process runs on each router or name server in the network, and one mserver runs on a dedicated host or the designated router for each connected LAN. Application group processes interface with the MS through an MI process running on each host computer. Each MI accepts requests for changes to or information about application groups from the individual application member processes running on the particular host computer. The MI then reliably relays these requests to the LAN mserver for submission to the MS. The MI receives responses from the LAN mserver and reliably propagates these responses to the application member processes that it supports. Each MI is able to support numerous application groups and numerous individual member processes from each application group, limited only by the available resources of the individual host computer.

2. Organization and Configuration

a. Logical Hierarchy

The physical hierarchy of the MS is formed with mserver nodes logically connected together to form an *n-ary* tree. The MIs are located at the leaf level of the physical tree, at the host computer level, providing an immediate interface for the application group processes running on the host computer. Figure 5 illustrates an example logical hierarchy of mservers, MIs, and application group processes. The architecture shown is a representative configuration for a small area encompassing a single institution, such as a campus or business. In this case, the architecture shown is the configuration of the Naval Postgraduate School (NPS), where the MS is under development. In Figure 5, the set of mservers labeled *NPS* are servers at the root level, attached to the campus backbone, representing the whole campus. At the next lower level are sets of mservers representing individual buildings at the campus, labeled *Spanagel, Root,* and *Ingersoll.* Each of the mservers in these sets are servers on LANs located in the buildings. At the next lower level are the MIs running on individual host computers on each LAN. The LANs are labeled as *ECE1*, *ECE2*, *SP1*, and so on. Below the MIs are the application group processes running on each host computer. In this example, there are four application groups shown. Some MIs are shown supporting more than one group, each with one or more members per host, while other MIs have no application groups to support. The MI process remains resident on the host computer even if no applications are running to provide quick access to the MS.

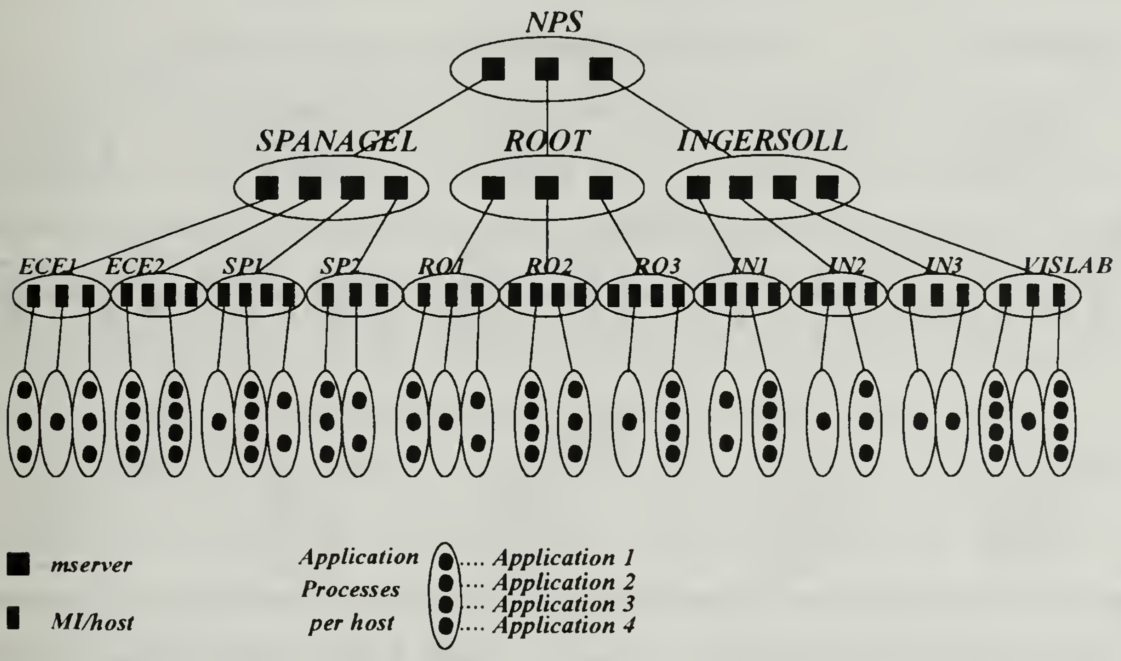


Figure 5: Logical MS Hierarchy

b. Physical Topology

The logical hierarchy shown in Figure 5 corresponds to the physical topology of networks and computers shown in Figure 6. In this illustration, each successively larger grouping of computers and networks is indicated by dotted lines and the associated name, corresponding to the sets of MI or mservers shown at each level in Figure 5.

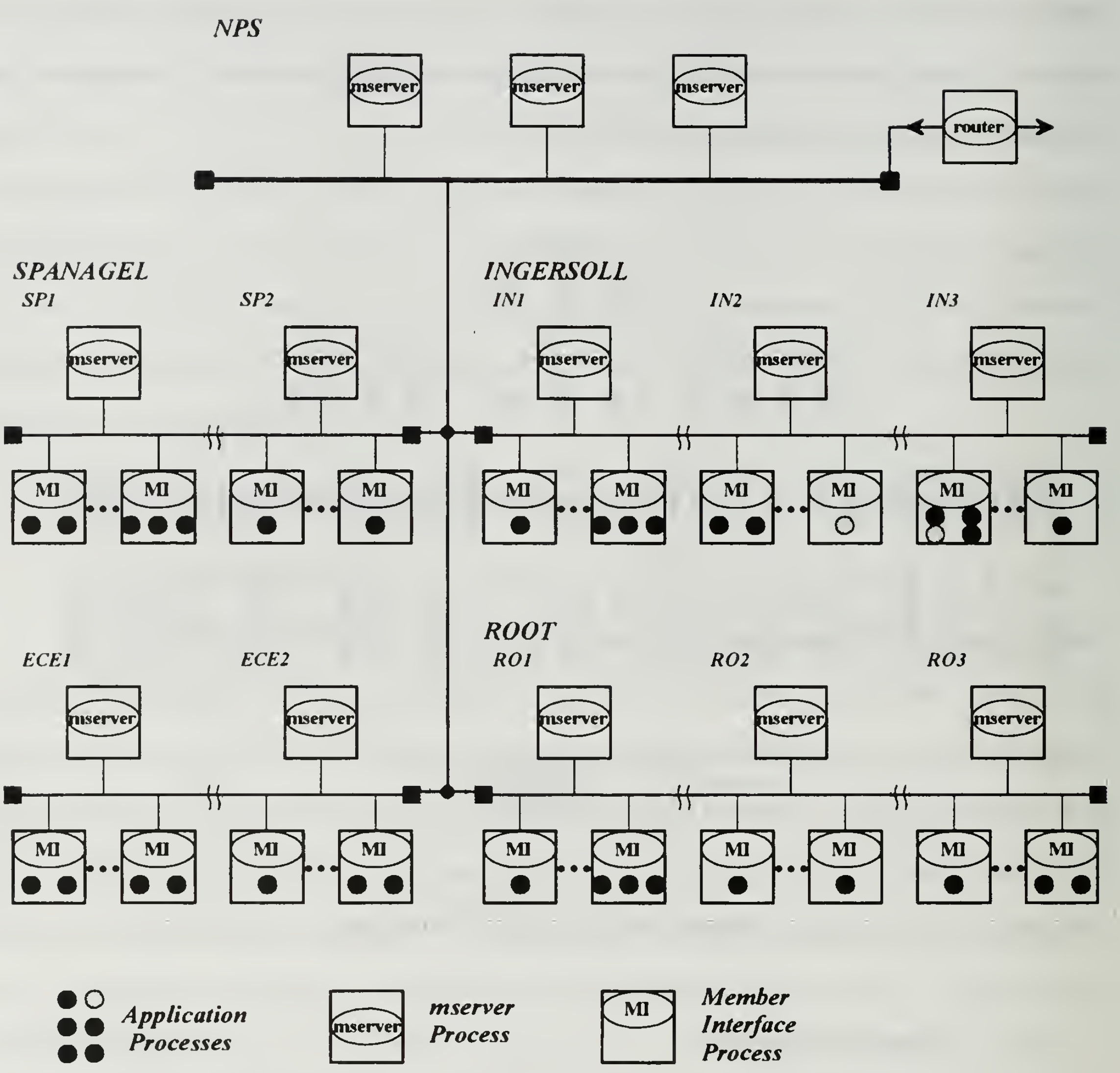


Figure 6: Physical MS Hierarchy

c. *Semi-static Configuration*

The mservers and MIs of the MS are manually configured into the desired physical hierarchy by a local system administrator or cognizant authority. This configuration is expected to be semi-static, normally changing only when additions and deletions to the networks maintained by the administrator are made. The system administrator will assign appropriate names for each set of mservers at each level, corresponding to the multicast group which connects the set of mservers. The assignment of a set name and multicast address are accomplished when the set of mservers are created and joined together, using software calls to the MS.

d. Failures, Partitions, and Dynamic Reformation

Although the mserver and MI configuration is not expected to change very often, there is still a possibility of the failure of the mserver or MI processes, the host computers or servers upon which they are running, or partitions in the network. These failures and partitions lead to a dynamic reconfiguration of the physical structure of the MS, with the surviving mservers and MIs automatically reforming into partitioned sets. Since any failure or perceived failure of an mserver is actually a virtual partition of the network, all failures and partitions will lead to the creation of one or more partitioned subsets of the original set of mservers. Each partitioned subset of mservers will correspond to that subtree of the physical hierarchy on one "side" of the partition; that is, all of the mservers which are still able to communicate over the non-partitioned network. Each reformed physical hierarchy of the MS will continue to function, providing service to all application process groups with members still existing in the partition. The application process groups which span the partitioned network will also experience a partition in their membership. This condition will continue until the physical network partition is repaired, at which time the physical hierarchy of mservers will either manually or automatically be reformed to the original configuration. Once the physical hierarchy is restored, the surviving application groups will also be reformed, if this is the QoS related to partition resolution chosen by the application user at start up time.

In addition to the overall hierarchical structure of the MS, each set of mservers in the physical hierarchy is also organized into a monitoring-set and change-processing core-set. The LAN mservers also are responsible for monitoring the status of all MIs on the LAN. These organizations of mservers will be explained in the next sections.

3. Monitoring Set

a. Definition and Purpose

The first criterion for an MS to be dynamically reconfigurable is to be able to detect failures of the component entities. To accomplish this, each set of mservers in the physical hierarchy is organized into a monitoring-set. The purpose of this monitoring-set is to detect and announce the failure of any failed or perceived failed mserver in the set. The detection method used is pairwise, peer-to-peer monitoring of the mservers in the monitoring-set. Each mserver is responsible for monitoring one other mserver in the set, and in turn is monitored by one other mserver. The monitoring is accomplished by the monitor sending periodic *Query* messages to the monitored mserver, which then responds with a *Reply* message, indicating normal status.

b. Structure

An illustration of a monitoring-set is shown in Figure 7. The pairs of monitoring and monitored mservers are determined by the order in which the mservers join the monitoring-set. Each newly joining mserver is connected into the pair-wise

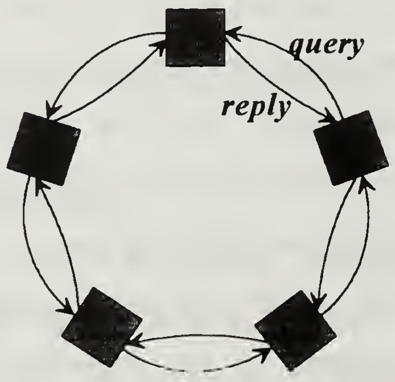


Figure 7: Monitoring-set of Mservers

monitoring sequence as the mserver monitored by the highest rank (oldest) mserver in the set, and will begin monitoring the previously lowest rank (youngest) mserver.

c. *Failure Detection*

As with nearly every message sent within the MS, the monitor will set a timer upon sending the *Query* message. If a *Reply* message is not received before the timer expires, the monitor will suspect the monitored mserver of failure. One or more retries will be conducted, and if the monitored mserver does not respond in this time, it will be declared failed by the monitor, which will then announce the failure to all other mservers in the set. The mserver detected failed may have actually failed, or may be unable to communicate with the monitor; in either case, it will be considered failed by all mservers which receive the monitor's announcement.

4. Change-processing Core-set

a. *Definition*

A second organization applied to the set of mservers at each level in the hierarchy is that of a change-processing core-set. This set of mservers is responsible for processing all membership change requests submitted by the application process groups that it supports, as well as enacting changes in the physical hierarchy. The change processing involves reaching a consistent agreement amongst all core-set mservers about the change being submitted, then to reliably propagate this change back to the application process members, who are then guaranteed to have a consistent view of the changed application group membership.

b. *Purpose*

This organization is termed a core-set because it is the small set of mservers at that "top" level for the group in the physical hierarchy which connects the particular application process group supported. For example, the set of mservers labeled *NPS* in Figure 5 serve as the core-set for all four application groups, since each application group has members distributed on all LANs. The sets of mservers at lower levels in the hierarchy will not process these application membership changes, but will submit them to the core-set, then relay the results back to the MIs. In this manner, the hierarchical structure of the MS is used to reduce the number of mservers that cooperate to process a membership change for an application group to those in the core-set for that group. This organization leads to very efficient and fast processing of membership changes for groups of any size and distribution, since only the core-set of mservers will be processing the change. It also provides the necessary scalability for the MS, since application process groups of any size or distribution will have a small core-set of mservers processing the membership changes, and thus will experience nearly the same small processing time. The primary difference in membership change processing times for different application groups will be caused by the level of the core-set in the physical hierarchy. A core-set at a higher level will have more intermediate relaying mservers between it and the application member processes, thus creating a longer transmission path.

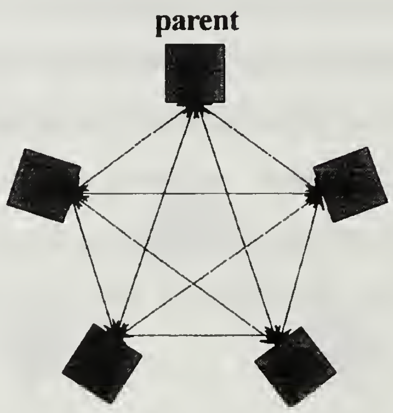


Figure 8: Change-processing Core-set of Mservers

c. *Structure*

An illustration of a core-set of mservers is shown in Figure 8. The mservers in the core-set are connected in a multicast tree, using a common multicast group to multicast a change message from one mserver to all others at once. For each membership change request submitted to the core-set, a coordinator is chosen. The criteria for selecting the coordinator depends on the particular type of change and how it was submitted to or detected by the core-set. The fact that the coordinator is not a fixed member of the core-set, but instead varies from change to change, is a powerful feature of the MS. Since the coordinator does not exist as such unless a change is actively being processed, there is no need to ensure an operational coordinator exists when no change is being processed, thus greatly reducing the core-set overhead.

Each set of mservers in the physical hierarchy is configured as a core-set. This serves the dual purpose of having a core-set readily available for use by application groups at any level in the hierarchy, and allowing each set of mservers to process membership changes among the mservers of the core-set locally. Thus, each level of the MS hierarchy is responsible for managing the mservers at that level only. Changes to the membership of the core-set are processed in exactly the same manner as membership changes submitted by application groups, with the exception that these changes directly affect the core-set membership and are not propagated outside of the core-set. Membership changes to the core-set are generated by failure detections from within the core-set or by change requests sent to the core-set when manual configuration of the MS physical hierarchy is conducted by the system administrator.

d. *Change-processing Sequence*

The basic change-processing sequence uses a modified form of the three-way handshake often seen in unreliable networks for reliable message delivery. The coordinator initiates the change processing with a multicast to all core-set mservers, collects acknowledgment (ACK) messages from all, then multicasts a final message for all to commit the change. Timeouts and retries are used by mservers waiting to receive *ACKs* or *Commit* messages from other mservers to ensure that continual progress is made toward completion of the change. As with the monitoring scheme, if the correct reply is not received from an mserver after the timeout period and all successive retries, then that mserver is declared failed and the failure is announced to all other mservers in the core-set.

e. Multicasts and Failure Detection

The use of timeouts and retries on change-processing messages creates a secondary but essential method of detecting mserver failures. Since mserver monitoring uses unicast messages and change-processing uses multicasts, it is possible that a network partition could occur that affected only multicast message delivery between one or more mservers. The inability of mservers to communicate all necessary data creates a virtual partition between the mservers. Without the use of this secondary detection method, it is possible that one or more mservers could be functioning perfectly well, sending the required monitoring messages, but unable to respond to change-processing messages, thus creating a deadlock situation. The timeout and retries on change-processing messages ensures that an mserver unable to communicate will be detected failed, and the remaining mservers will be able to complete the change in a timely manner. In the event of a coordinator failure during the change processing, a distributed election is conducted and a new coordinator is elected to continue the original change.

5. LAN Mserver Monitoring

Due to the high bandwidth, low latency, hardware multicast capability, and limited number of MIs to monitor, the mserver representing each LAN uses a simple polling scheme to conduct status monitoring of the MIs and host computers on the LAN. Each MI on the LAN is successively polled with a *Query* message by the LAN mserver. The MI responds with a *Reply* message indicating normal status. Timeouts and retries are used to detect a non-responding MI, declare that MI failed, and announce the failure. A depiction of the LAN mserver monitoring scheme is shown in Figure 9.

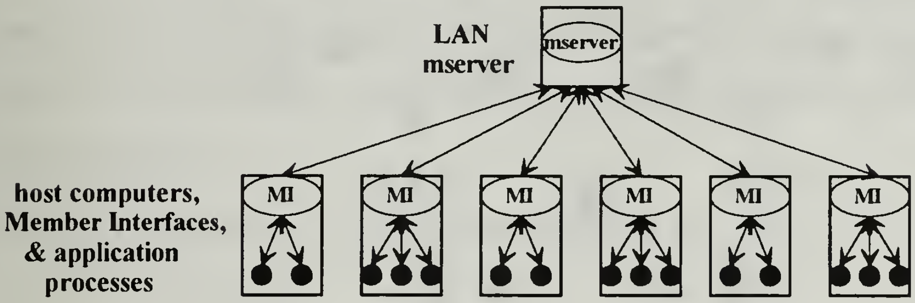


Figure 9: LAN Mserver Monitoring of MIs

6. Hierarchical Structure

a. Collapsing the Tree

The final organization of mservers and I s involves forming the monitoring-sets and core-sets of mservers into the physical hierarchical structure used by the IS, with the I s at the leaf level. All core-sets are also monitoring-sets, thus providing the failure detection needed by a core-set to manage the mserver membership locally. As shown in Figure 5, each mserver in the hierarchy has either a set of children mservers or I s. All mservers and I s also have a parent mserver, except the mservers at the highest level of the hierarchy. To create this physical structure, the logical hierarchy of Figure 5 is "collapsed", so that each parent mserver becomes a member of the core-set of children mservers below it, as well as a member of the core-set of peer mservers. Thus, each mserver above the lowest level in the hierarchy has a dual membership in the "child-set" as well as the original core-set of mservers. Figure 10 illustrates this structure.

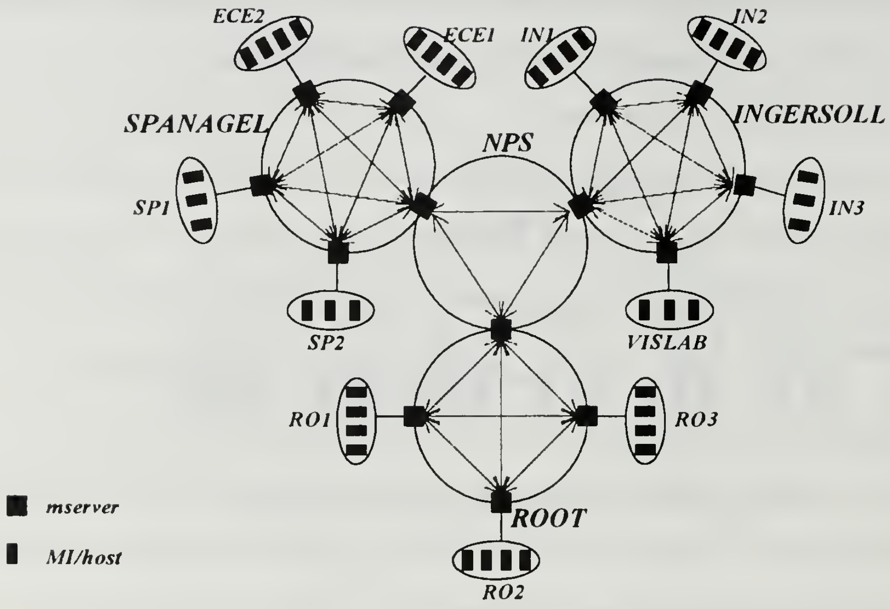


Figure 10: "Collapsed" MS Architecture

b. Parent Mservers

A comparison of the logical MS hierarchy shown in Figure 5 with the physical MS hierarchy shown in Figure 10 shows the same sets of mservers and MIs. However, the sets can now be identified as change-processing core-sets, linked to the level above by the dual membership of the parent mserver. Having the parent mserver as a member of the child-set has two primary advantages. First, the parent mserver is part of the child monitoring-set; thus, the child-set will immediately learn of the failure of the parent mserver by monitoring. Second, the parent mserver takes part in all change processing conducted by the child-set; therefore, it will learn of any changes in the membership of the child-set directly. Together, these two points ensure that "vertical monitoring" is conducted in the hierarchy. This provides the means to ensure that a failure or partition between levels in the MS hierarchy will be detected, allowing the MS to reform as necessary.

B. APPLICATION GROUPS

Support for application process groups is the primary reason for the MS. The MS is responsible for managing the membership of the application process groups and providing services to the application process groups. The following sections describe how the MS accomplishes this.

1. Application Groups and the Physical Hierarchy

a. Scalability

The application groups consist of processes running on host computers distributed throughout the networks supported by the MS. As shown in Figure 2, the MS provides the necessary services to make an application consisting of numerous distributed processes to appear as a unified application running at a single site. Because of the scalability of the underlying MS architecture, the application process groups are completely scalable in number and distribution of processes, with the end result being complete transparency of the distributed nature of the MS to the service users.

b. Consistency

The primary service that the MS provides application groups is a consistent view of the group membership at all members, as well as a consistent ordering of changes to the membership of the group at all members. These consistency guarantees ensure that a process group member either acquires the same consistent view as all other members of the group eventually, or is excluded from the membership of the group. The term "eventually" refers to the asynchronous nature of the environment, leading to delays at some sites. The MS allows for reasonable delays, thus ensuring that all surviving processes will receive the revised group view. Using this guarantee of consistent membership at all processes, the application can safely make certain assumptions about the member processes. The application can expect that processes with the same group view number have seen the same sequence of membership changes, and currently have the same view of the membership of the group. Using this knowledge, the application can decide to accept or reject messages from other application processes depending on the included group view number. The guarantee of consistent membership can be used as the foundation upon which to build many distributed applications.

The MS provides consistent ordering of membership changes to application groups by ensuring that only one change is ever processed at a time in the core-set of that application group, and that all active member processes eventually receive this change. The selected change is committed by all core-set mservers, then reliably propagated to the MIs, and finally, to the distributed application member processes. The MS provides the guarantee that an application member process either receives each revised group view or is detected as failed, and excluded from the group. In this manner, all surviving application member processes are guaranteed to have exactly the same ordering of membership changes.

c. Naming

The MS manages the names of all application groups using the MS. Application group names are guaranteed unique within a predetermined scope. When an application group is created, the software call from the application to the MS includes as a parameter a level in the MS physical hierarchy, under which the application group name will be guaranteed unique. This name-scope parameter is either the actual name of the core-set or a level number above the MI level in the physical hierarchy. For example, to guarantee an application group name of "application1" as unique under the scope of the *NPS* core-set from Figure 5, the name *NPS* or the level number 2 would be used as the name-scope parameter. The name-scope level must be at or above the core-set level for the application.

With the creation of each new application group, the name-scope parameter is checked at each level in the mserver hierarchy up to and including the name-scope level.

If the name already exists, the creation of the new group is refused, and an error code is returned to the calling application. If the name is not found, then it is registered at the name-scope level of mservers and at each level in the hierarchical tree of the application, and a successful group creation is reported to the calling application. When new application member processes at distributed locations wish to join an existing application group, a join request is submitted via the resident MI, then propagated up the hierarchy until either an mserver is located with the application name stored or the highest level in the physical hierarchy is reached and the application name is not located. If the desired application group name is located, the new member is joined into the application group through the normal change-processing sequence, and a successful join is reported back to the requesting process. If the name is not located, an unsuccessful join attempt is reported back. Through judicious use of the name-scope parameter, application names may be used freely with little concern about duplicate name usage.

d. Membership Scope Control

An additional feature provided by the MS is the ability for an application to decide at what level in the MS physical hierarchy to limit the scope of the application group. By providing a membership-scope parameter with the creation call for a new application group, the application guarantees that the span of the application's membership will not exceed the given core-set level in the physical hierarchy. In return, the MS is able to provide more efficient service by limiting the scope of application group name searches to the membership-scope level and below. Instead of propagating every unsuccessful application group name search to the highest level of the MS hierarchy, the name search will cease at the membership-scope level. Without use of the membership-scope, it might be possible for a bottleneck to form at the "top" of the MS hierarchy.

2. Member Interfaces

a. Purpose

As previously described, the MIs provide the interface between application group member processes and the MS physical hierarchy. They accept application membership change and information requests from application processes and submit these changes to the mserver hierarchy for processing. When the change or information data is returned, the MI passes the data to the requesting member processes.

As shown in Figure 9, each MI is running on an individual host computer. Each MI is capable of interfacing multiple application groups, each with multiple members, with the LAN mserver and the MS. Each MI maintains a list of all application groups it is managing as well as all member processes from these groups running on the host computer. Thus, the membership information for each application group is maintained in a decentralized, scalable manner. When an application member process needs to communicate with another application member process on a different host, it submits a request for addressing information to the MI. The MI relays this information request to the MS, which obtains the desired information from the MI managing the desired member process, and relays the information back to the requesting MI and application member process.

b. Application Member Process Monitoring

The MIs monitor the application member processes in exactly the same manner that the LAN mserver monitors the MIs on the LAN: using polling. In the same manner, non-responding application processes are detected failed, the failure is announced, then submitted to the MS for an application group membership change.

3. Application Group Change Processing

As previously discussed, application group change processing begins with the submission of a change request to the host MI. This request is relayed to the core-set of the application, which conducts the mserver change-processing procedure, resulting in all core-set mservers committing the change. Each core-set mserver then reliably relays the change directive down the hierarchy to the MI, and then to the requesting application process. When the change is submitted by the MI, a timer is set to ensure a timely response to the change. The MI waits for the returning *Direct* message from the LAN mserver. If the timer expires before receiving the *Direct* message, a query message is sent to the LAN mserver requesting the status of the change submitted. The LAN mserver will respond with a *Wait* message if the change is still being processed, causing the MI to wait for a period before querying the mserver again. If the MI completes all timeouts and retries and still has not received a reply from the LAN mserver, it detects the LAN mserver failed and announces the failure. In the same manner, each intermediate mserver also sets a timer for a response from the next higher level mserver. A non-response leads to a partition in the physical hierarchy. To ensure reliable transmission from the core-set to the application process, each intermediate mserver and MI send an *ACK* message back to the mserver above upon receipt of the *Direct* message. Timeouts and retries are again used to detect failures and partitions. At the end of the application change processing sequence, every application member process is guaranteed to have received the change message or to have been detected as failed.

IV. MEMBERSHIP SERVICE PROTOCOLS

The previous chapter described the component entities of the MS: the mservers and MIs. The organization of the mservers and MI into the MS physical hierarchy was described in detail, as well as their basic functionality. This chapter describes in detail the protocols used by the mservers and MI to implement the MS, and the general format of messages used to exchange membership information between mservers and MI.

1. General Message Types

The general message types used by the MS and descriptions of each are listed in Table 4. There are three general classifications of messages: Monitoring, Initiate, and Change Processing. Many of these messages are used for more than one purpose, such as processing changes to the physical hierarchy of mservers and MI as well as changes to application process groups. The Monitoring messages are used by mserver in the monitoring-set to conduct pairwise peer-to-peer monitoring, by the LAN mserver to monitor the MIs on the LAN, and by the MI to monitor application process members. The type of monitoring being conducted is determined by the members involved and the context of the message used. The Initiate category of messages are used to initiate a change for either the physical hierarchy or an application process. The *Join* message is used to join a new mserver to an existing core-set of mservers, create a new core-set with one mserver, to join a new application member process to an existing group, or to create a new application group. The *Leave* message allows a voluntary departure by an mserver from a core-set or an application process from the group. The *Split* and *Merge* message types are the general form of *Join* and *Leave*, allowing multiple mservers or application processes to join or leave a core-set or application group, respectively. The *Add_parent* and *Del_parent* message types are used by an existing core-set to adopt or remove a

TABLE 4: MS GENERAL MESSAGE TYPES

Message Type	Description
Monitoring	Used by mservers and MI to determine status of others
Query	Query status of another mserver or MI
Reply	Reply to Query
Initiate	Initiate a physical or application group change
Join	Mserver join a core-set or application process join a group
Leave	Mserver leave a core-set or application process leave a group
Split	Split from core-set or group to form a new set or group
Merge	Merge separate core-sets or application groups into one
Add_parent	Core-set add an mserver as parent
Del_parent	Core-set remove the existing parent mserver
Fail	Mserver, MI, or application member process detected failed
Coord_Fail	Mserver coordinator of current change detected failed
Submit	MI submit change to core-set (same types as Initiate)
Direct	Core-set change directive to MIs (same types as Initiate)
Process Change	Used to process a physical or application group change
ACK	Acknowledge Initiate or Direct messages
Wait	Wait to begin processing change or for next message in change
Commit	Commit the current change
Msg_Query	Query mserver for status of next message expected in change
Init	Initial parameters message from coordinator to joining mserver

parent mserver. This action is the primary function used to create the hierarchy of core-sets. The *Fail* message type is used to announce the failure of an mserver or application member process and initiate the change to remove the failed member from the core-set or application group. The *Coord_Fail* message is used to announce the failure of the coordinator mserver for the current change being processed. This message will prompt the election of a new coordinator, which will complete the original change. The third category of general message types are those actually used to process membership changes to a core-set of mservers or an application group. The *ACK* message is a general acknowledgment message used to indicate successful reception of an *Initiate* or *Direct* message. The *Wait* message is used by the coordinator of the current change or by an

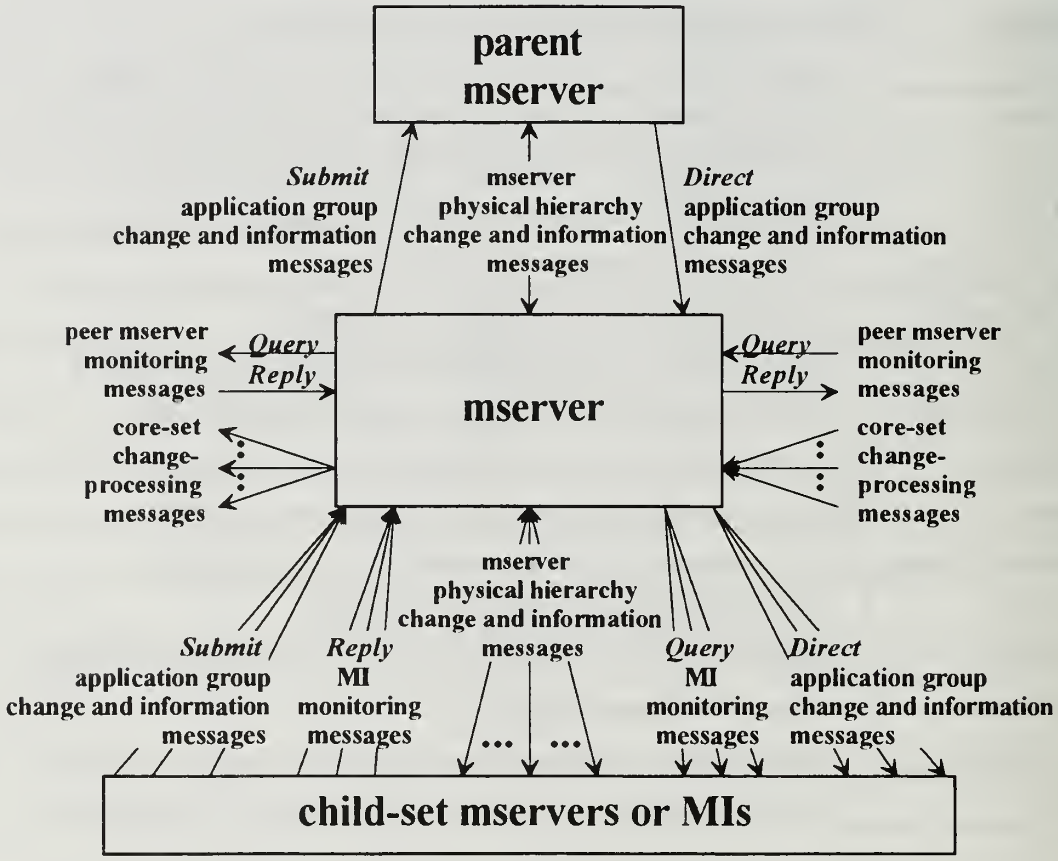

parent mserver
Submit application group change and information messages
mserver physical hierarchy change and information messages
Direct application group change and information messages
peer mserver monitoring messages
Query
Reply
mserver
Query
Reply
peer mserver monitoring messages
core-set change-processing messages
core-set change-processing messages
mserver physical hierarchy change and information messages
Submit application group change and information messages
Reply MI monitoring messages
Query MI monitoring messages
Direct application group change and information messages
child-set mservers or MIs

Figure 11: Mserver Messages

mserver propagating a *Submit* message to inform querying mservers that a there is a delay in completing the current change and that they should wait for a period for the next expected message. It is also used by a core-set mserver to inform another mserver attempting to initiate a new change that the current change is not yet completed, and the new coordinator should wait a period before beginning the next change. The *Commit* message is sent by the coordinator to inform all core-set mservers that it is safe to commit the current change as the new group view and to propagate this view to application processes as needed. The *Msg_Query* message is used by an mserver or MI to query

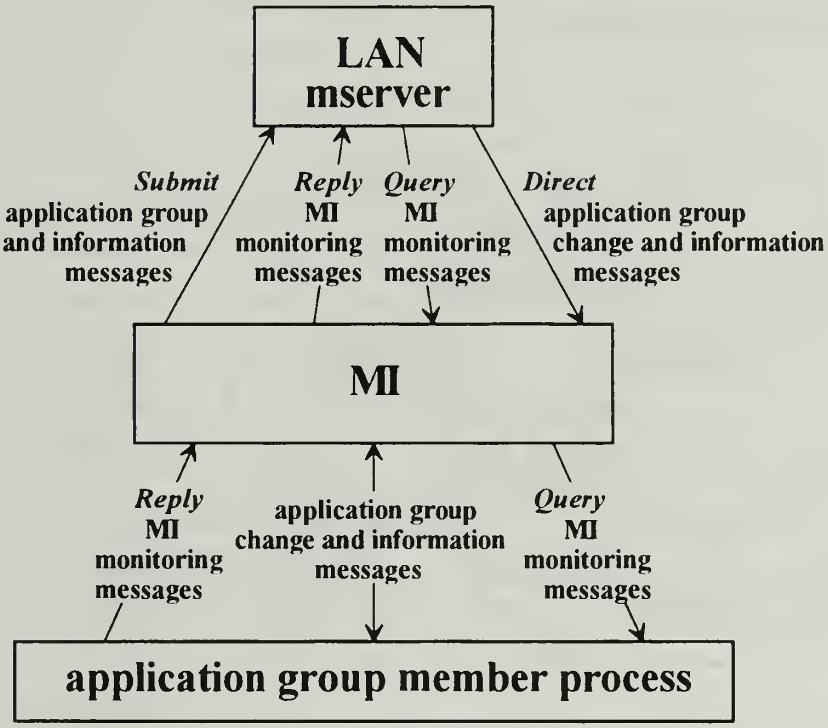

LAN mserver
Submit application group change and information messages
Reply MI monitoring messages
Query MI monitoring messages
Direct application group change and information messages
MI
Reply MI monitoring messages
application group change and information messages
Query MI monitoring messages
application group member process

Figure 12: Member Interface (MI) Messages

another mserver about the status of the message for the current change expected from the queried mserver. The mserver receiving the *Msg_Query* will usually respond with a *Wait* message or the expected message, if it is determined that the message was lost. The *Submit* message is used by an MI submitting an application group change to the LAN mserver, then by each mserver in the hierarchy to propagate the change request to the application core-set. The *Submit* is in effect a remote *Initiate* message, and has the same category of types as an *Initiate* message. The *Direct* message is used by the core-set mserver with application members at their leaf level to propagate the committed application change down the hierarchy to the MIs representing the application group, and also has the same category of message types as an *Initiate* message. Figure 11 illustrates the messages sent to and received by an mserver, while Figure 12 illustrates the messages sent to and received by an MI.

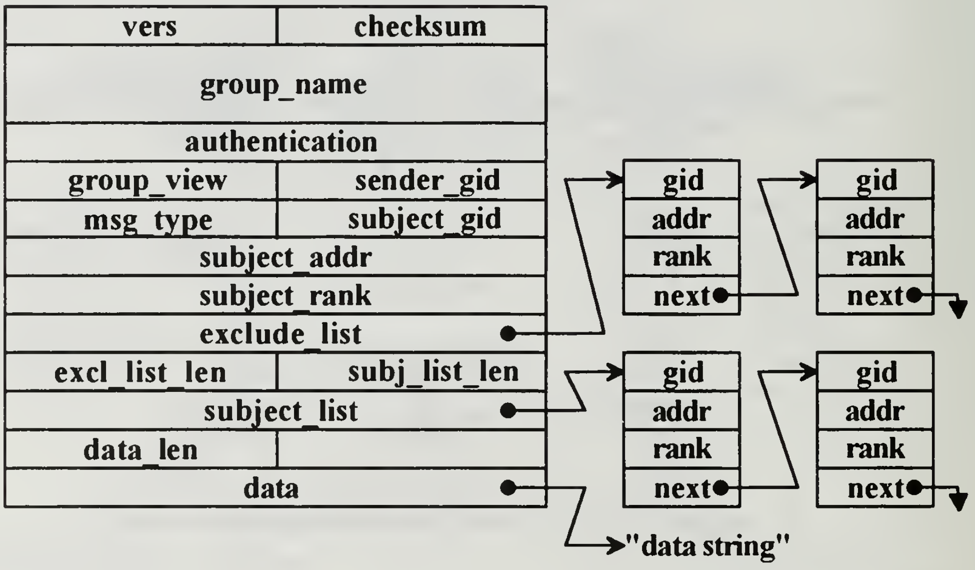


Figure 13: Membership Service General Message Format

2. General Message Format

The general message format used by the MS is shown in Figure 13. An explanation of the meaning of each message field is provided in Table 5. The exclude and subject lists shown in Figure 13 are queues maintained by each mserver, which are included with certain types of messages. Each element of these lists contains the minimal amount of information to uniquely identify an mserver or application process, when

TABLE 5: MS GENERAL MESSAGE FIELDS

Message Field	Description
vers	MS version number
checksum	used for message error detection
group_name	core-set or application group name
authentication	used for group security
group_view	current core-set or application group view number
msg_type	message type
sender_gid	message sender's group identification (gid) number
subject_gid	message subject's group identification (gid) number
subject_rank	seniority based rank of mserver or application member process
exclude_list	list of mservers to be excluded from core-set due to failure
excl_list_len	number of mservers in exclude_list
subject_list	list of subjects for a *Merge* or *Split*, or failed mservers or members
subj_list_len	number of mservers or members in subject_list
data	general purpose data field
data_len	length of data included with message

combined with the information about the core-set or application group contained in the message. The lists are used to communicate information about sets of mservers or application processes. The exclude list serves a dual purpose: to ensure that failed mservers are not included in the communications of the core-set, and to inform other core-set mservers of mservers perceived failed before they are actually removed from the core-set. Because the network multicast capability assumed by the MS is unable to dynamically tailor the receivership of each multicast message, a filter mechanism must be used to ensure that unintended mservers do not receive the current message. An mserver which is detected as failed is added to the exclude list of the current message sent by the detecting mserver. Mservers receiving this message will cease all communications with the excluded mserver. If an mserver is still functioning and receives this message with itself listed in the exclude list, it will immediately cease all communications with all mservers in the core-set, with the possible exception of other mservers in the exclude list, with which it will attempt to reform a new core-set. This method of "piggy-backing" the detected failure of mservers with another message is referred to as "gossip" [9].

3. General State of Mservers and MIs

The MS maintains information about the physical hierarchy and application groups in a decentralized manner. Individual mservers and MIs need only maintain the information necessary to perform their required functions. This decentralized storage of MS information is essential to the scalability of the MS.

a. Mservers

For an mserver, the information stored about the physical hierarchy includes the gids, ranks, and addresses of other mservers in the core-set and child-set of mservers of which it is a part; the monitor, and monitored mservers for each of these sets of mservers; the parent mserver of the core-set; information about the current and previous changes processed; and queues of mservers detected failed , received change requests, and excluded mservers. Each mserver also maintains information about the application groups that it supports. This information includes a list of application groups supported by the mserver; which children mservers are in the application group's hierarchy; and whether the mserver level is the core-set, memberhsip-scope, or name-scope level of the application.

b. MIs

Each MI must store information about the application groups and their members that it is supporting on the host computer, as well as the address of the parent mserver. Any other required information is obtained through the MS hierarchy.

4. Physical and Application Group Protocols

To perform the various functions described, mservers and MI use five primary protocols. These protocols are: the physical monitoring protocol, the application group monitoring protocol, the physical core-set change-processing protocol, the application group change-processing protocol, and the network partition resolution protocol. Each of these protocols are described in detail in the following section, using psuedo-code algorithm listings and event diagram illustrations.

A. PHYSICAL MONITORING PROTOCOL

1. Pairwise Monitoring

As described in the previous chapter, mservers in the physical hierarchy are arranged into monitoring-sets for the purpose of detecting mserver failures. Within these monitoring-sets, the mservers conduct pairwise, peer-to-peer monitoring. The monitoring mserver periodically sends a *Query* message to the monitored mserver, which responds with a *Reply* message indicating normal operation. The algorithm for this physical monitoring protocol is shown in Figure 14, with the description following.

Monitoring mserver

```
/* when monitoring timer has expired */
1.  form_message (Query, current_change, exclude_list)
2.  send_message (Query_message, monitored mserver)
3.  messg = Reliable_receive (Reply_message, Query_message)
4   if (messg != Reply_message)   /* failed mserver */
5.          declare the monitored mserver failed
6.  else
7.          reset monitoring timer
```

Figure 14: Physical Monitoring Protocol

Figure 14 shows the procedure used by the monitoring mserver. In lines 1 and 2 the *Query* message is sent to the monitored mserver. Line 3 uses a function called Reliable_receive, explained in the next section, which a uses timeouts and retries to ensure the *Reply* message is received in a reasonable period of time. Lines 4 and 5 detect the monitored mserver failed if it did not respond to the *Query*. Finally, line 7 resets the monitoring timer to repeat the process after a suitable period.

2. Failure Detection, Timeouts, and Retries

The primary means of detecting an mserver failure through monitoring is the use of timeouts and retransmissions of the *Query* message. If after a preset number of timeouts and retries the monitored mserver still has not responded, it is assumed to have failed. The Reliable_receive function in Line 3 of Figure 14 performs this timeout and retry sequence. The function is termed "reliable" because it ensures a reliable communication over a single link: that is, the monitored mserver either responds in a reasonable period or is determined to have failed. The algorithm for this function is listed in Figure 15.

```
Reliable_receive (expected_messg, query_messg)
/* returns the received message */
1.  set_timer (timeout)
2.  retries = n
3.  while ((timer not expired) and (messg != expected_messg))
4.          receive_message (messg)
5.  if ((messg != expected_messg) and (retries > 0))
6.          retries = retries - 1
7.          send_message (query_messg, destination)
8.          set_timer (timeout)
9.          goto 3.
10. return messg
```

Figure 15: Reliable_receive

Line 1 and 2 initialize the function. Line 3 begins the main reception loop. Line 4 is a timed receive function, which returns the received message immediately upon reception or times out waiting and returns a NULL message. The main loop is executed until the expected message is received or the main timeout period expires. Lines 5 through 9 perform the retry sequence. If the expected message is received the function returns immediately, otherwise, the function times out and returns whatever message, if any, was received. By examining the returned message, the monitor is able to decide if the monitored mserver has failed.

B. APPLICATION GROUP MONITORING

The protocol used by an MI to monitor the status of the application member processes that it is interfacing is exactly the same as that used by the LAN mserver to monitor the MIs running on the host computers of the LAN. The MI periodically queries each application member process, using the Reliable_receive function, and declares any application processes not responding as failed. Figure 9 shows the arrangement of LAN mserver, MIs, and application member processes.

C. PHYSICAL CORE-SET CHANGE PROCESSING

At the heart of the MS is the ability of a small set of mservers to make a consistent, mutually agreed upon decision about membership changes to physical sets of mservers and application groups of any size. This section describes the types of changes processed, the basic change-processing protocol used by a core-set to commit these changes, and additional protocols used in the event of failures of mservers within the core-set.

1. Coordinator

The coordinator is the core-set mserver responsible for coordinating the processing of the current membership change. One of the strengths of the MS is that any mserver can become the coordinator, either initially upon detecting or receiving a change, or following the failure of the current coordinator. Also, the coordinator only exists in that capacity while the current change is being processed; when there are no membership changes to process in a core-set, there is no coordinator. The coordinator for each change is determined by a combination of the type of change, which core-set mservers detect or receive the change, and a priority associated with each change. In addition to determining which mserver will act as coordinator, these criteria also ensure that only one change at a time is committed by a core-set.

2. Types of Changes

There are three primary types of membership changes processed by a core-set of mservers: requests, failures, and dynamic reconfigurations.

a. Requests

Requests are voluntary, planned membership changes, submitted to the core-set for processing by an application process, membership service user, or system administrator. Change requests for the MS physical hierarchy may be of any type listed in Table 4, with the exception of *Fail* and *Coord_Fail*. Application group change requests may be of any type used for physical change requests except *Add_parent* and *Del_parent*. Physical change requests are multicast to a specific core-set in the hierarchy by a system configuration call, usually invoked by a system administrator during manual configuration of the MS hierarchy. The physical change request is received by all mservers in the selected core-set. Each receiving mserver queues the request, to be processed when other higher priority changes have completed processing. Application group change requests are submitted to the resident MI process on the host computer by the application or the MSU. The MI then propagates the request to the core-set mserver above it in the hierarchy. The receiving core-set mserver queues the request to be processed when other higher priority changes have completed processing.

b. Failures

The second primary type of membership changes are detected failures. These detected failures may be the result of the actual failure of an mserver, MI, or application process, or the host machines upon which they are running. Additionally, network partitions will be perceived as failures of the partitioned mservers, and will lead to the processing of failures and reformation of the partitioned subsets of mservers and subgroups of application processes. The partitioning of the MS physical hierarchy leads to a partitioning of the application groups residing on this hierarchy. The MS automatically reforms both the physical hierarchy and the supported application groups in the event of a network partition. Failures detected or received by a core-set mserver are queued and processed according to their priority. Multiple failures queued at a core-set mserver are processed all at once, in a "batched" manner. This greatly reduces the time required to reform physical core-sets or application groups.

c. Dynamic Reconfigurations

The final type of changes are the result of automatic actions taken by core-sets of mservers. As part of the processing of multiple failures caused by a network partition, a core-set is often partitioned into two or more subsets. After the reformation into subsets has occurred, these sub-core-sets attempt to reform into the original core-set by sending messages to the other subsets of mservers. Since the sub-core-sets still share the same multicast address, once the network partition is mended, the other sub-core-sets receive these reformation messages. Upon learning of the existence of a sub-core-set from the original core-set, the partitioned subsets of mservers reform into the original core-set automatically. In addition to reforming the physical core-set, all application groups which were partitioned and are still functioning are also reformed. The reformation process for both physical core-sets and application groups merges the currently existing membership of each, taking the union of all subsets or subgroups, and making the reformed core-set or application group membership the current view. In the event that the network partition is not repaired in a predetermined period of time, the partitioned subsets of mservers will abandon their attempts to reform the original core-set, and will create a new multicast group with only the current core-set mserver included.

Another type of dynamic reconfiguration occurs when new members join an application group, causing the span of the application group to increase beyond that presently covered by the current application core-set. In this event, the application core-set must be moved from the present level in the physical hierarchy to a higher level covering the new span of the application. This new level must be at or below the name-scope and membership-scope levels of the application group, if these levels were designated when the application group was created. The MS automatically moves the application core-set to the new level. In a similar manner, the departure of application member processes may lead to a reduced span of the application. An application core-set must have at least two mservers with application members in their subtrees; otherwise, there is no need to have the application core-set at this level in the hierarchy. If the application core-set is reduced to only one mserver supporting an application, the application core-set will automatically move down to the child-set of this mserver.

The repositioning of an application core-set is initiated by the set of mservers detecting the need to move the application core-set. Messages are exchanged between the old and new core-sets and a change involving the join or departure of the instigating application member is processed along with the change in application core-set level by both core-sets. After committing the changes, the internal state of all mservers in both core-sets is changed to reflect the new application core-set level.

3. Ordering and Priority of Change Processing

A key issue associated with processing membership changes to sets of mservers or application groups is the ordering of changes committed by the core-set. As previously described, to guarantee consistent ordering of membership changes at all mservers in the core-set, only one change may be committed at a time. However, it is possible that more than one membership change may be submitted to or detected by the core-set at one time. Each receiving or detecting mserver in the core-set will attempt to become the core-set coordinator and initiate the change it received or detected. These multiple change initiation attempts are referred to as "virtually simultaneous", since they have all been initiated before the core-set has reached a consistent and uniform decision on the current change to process. If the core-set had already chosen a current change and coordinator, a newly initiated change would be processed after completion of the current change.

To resolve these virtually simultaneous changes and select only one change to be processed, a prioritization scheme is used. This prioritization scheme uses the type of change and the gid and rank of the subject of the change to decide which change will be processed by the core-set. The highest priority is given to any current change being processed by the core-set; that is, a change which has been consistently accepted by all core-set mservers. It is essential that such a change progresses to completion at all core-set mservers; otherwise, the possibility of inconsistent membership views exists if some mservers commit the change while others do not. The next higher priority is that physical changes always have priority over application group changes. This is because it is important to ensure a complete and whole MS before attempting to change the membership of an application group using the MS. Once these decisions have been made, the priority of the change is determined by the rank of the subject of the change. The only exceptions to this rule are for the failure of the coordinator of the current change or a *Join*. The failure of the coordinator has priority over otherwise equal status changes. A newly joining mserver or member will not have an associated rank until after the join is completed. For this reason, the network address of the joining member is used as a rank number to give a priority among *Joins*. The final rule used to determine the priority of virtually simultaneous changes is applicable when changes are submitted to the core-set by different application groups with identical subject rankings in each group. In this case, a tie-breaker is needed, and the rank of the receiving mserver is used to decide which change will be processed.

4. The Basic Change-processing Protocol

As discussed in the description of an mserver core-set, the basic change-processing protocol used by a core-set is a modified version of the three-way handshake used in unreliable networks to ensure reliable message delivery. An event diagram illustrating the sequence of message transmissions and receptions is shown in Figure 16. The algorithm for the coordinator of the basic change is listed in Figure 17. The algorithm for the non-coordinator core-set mservers is listed in Figure 18.

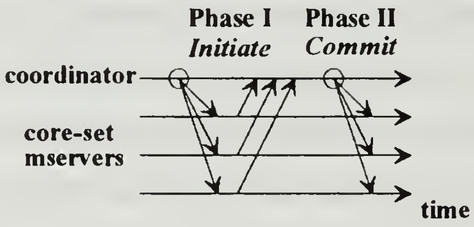


Figure 16: Basic Two-Phase Change-Processing Protocol

The basic change-processing protocol consists of two phases: the Initiate and Commit phases. In the Initiate phase, the coordinator multicasts an *Initiate* message to all mservers in the core-set. The core-set mservers respond with *ACKs*, acknowledging reception of the *Initiate* message. When all *ACKs* have been received by the coordinator, the Commit phase is begun with the coordinator multicasting a *Commit* message to all core-set mservers, indicating that it is safe to commit the change. All change-processing messages use timeouts and retries to ensure continual progress in completing the change. The procedures Reliable_receive and Reliable_multi_receive perform these functions.

Coordinator Basic Change

```
/* coordinator has been identified by reception of a change request or detection of a failed
mserver */
/* Initiate phase */
1.  current_change = change data (received or detected)
2.  update (exclude_list, internal state)
3.  form_message (Initiate, current_change, exclude_list)
4.  multicast (Initiate_message, {core_set - exclude_list - coordinator})
5.  Reliable_multi_receive (ACK_message, Initiate_messg, {core_set - exclude_list -
    coordinator})
/* Commit phase */
6.  update (exclude_list, internal state, current_change)
7.  form_message (Commit, current_change, exclude_list)
8.  multicast (Commit_message, {core_set - exclude_list - coordinator})
9.  update (internal state)
10. previous_change = current_change
11. group_view = group_view + 1
```

Figure 17: Coordinator Basic Change Protocol

Non-coordinator Basic Change

```
/* core-set mserver has received and decoded Initiate message */
/* Initiate phase */
1.  current_change = change data (received)
2.  update (exclude_list, internal state)
3.  form_message (ACK, current_change, exclude_list)
4.  send_message (ACK_message, coordinator)
5.  messg = Reliable_receive (Commit_message, Msg_Query_message)
6   if (messg != Commit_message)   /* failed coordinator */
7.      goto Broadcast Election Protocol
/* Commit phase */
8.  update (internal state)
9.  previous_change = current_change
10. group_view = group_view + 1
```

Figure 18: Non-Coordinator Basic Change Protocol

a. Timeouts and Retries For All Messages

The need for timeouts and retries of messages has been discussed previously. The function Reliable_receive accomplishes this for unicast messages, as described in the monitoring protocol section. The same function is performed by the procedure Reliable_multi_receive when multiple responses must be received, as shown in Figure 19. The algorithm for the Reliable_multi_receive is listed in Figure 20.

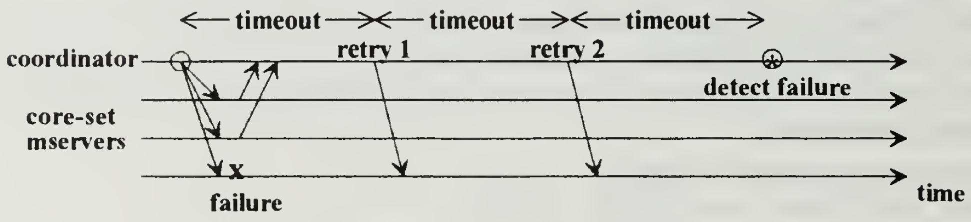


Figure 19: Message Timeout, Retries, and Failure Detection

```
Reliable_multi_receive (expected_messg, last_messg, responders)
/* Last_messg has been sent, now collect expected_messg responses . Modifies the set of
responders to reflect those not responding */
1.  set_timer (timeout)
2.  retries = n
3.  initialize (all responders = not_responded)
4.  num_responders = number of responders
5.  responses = 0
6.  while ((timer not expired) and (responses < num_responders))
7.      receive_message (messg)
8.      if ((messg == expected_messg) and (responder == not_responded))
9.          responder = responded
10.         responses = responses + 1
11. responders = {responders - (all responding responders)}
12. if ((responses < num_responders) and (retries > 0))
13.     retries = retries - 1
14.     multicast (last_messg, responders)
15.     set_timer (timeout)
16.     goto 6.
```

Figure 20: Reliable_multi_receive Algorithm

Lines 1 and 2 of the Reliable_multi_receive algorithm initialize the timer and number of retries. Lines 3 through 5 initialize the set of responders and number of responses. Line 6 begins the main loop to collect responses from the set of expected responding mservers. Line 7 is the timed receive function described in the Reliable_receive function. Lines 8 through 10 determine if the response is valid and not a duplicate, and if so, mark the responding mserver as having responded. Line 11 calculates the new set of expected responding mservers after the loop has completed. If any mservers have not responded and the retries have not been exhausted, lines 12 through 16 initialize for another timed reception loop to attempt to collect the remaining responses. At the end of the procedure, the set of responders has been reduced to only those who failed to respond. These mservers will be declared failed by the calling mserver; in this case, by the coordinator in line 5 of the basic change protocol.

Figure 21 is an event diagram showing the sequence of messages in the event of a lost or delayed *ACK* message from a non-coordinator core-set mserver. After a

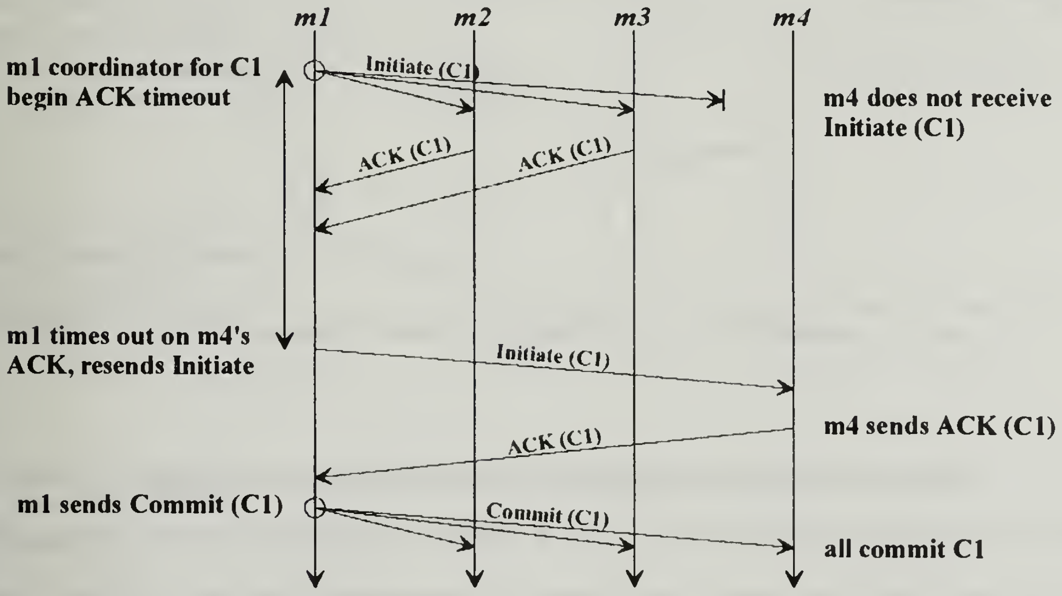


Figure 21: Lost or Delayed *ACK* Message During Initiate Phase

timeout period, the coordinator resends the *Initiate* message, and receives an *ACK* message. The core-set then commits the change. In this event diagram, the multicast of a message is indicated by multiple arrows emanating from a small circle.

Figure 22 shows a similar situation, in which a non-coordinator mserver has not received an expected *Commit* message from the coordinator. This mserver sends a *Msg_Query* message to the coordinator, querying the coordinator about the missing *Commit* message. The coordinator realizes that the querying mserver must not have received the original *Commit* message, so it resends the message. The core-set then completes the change.

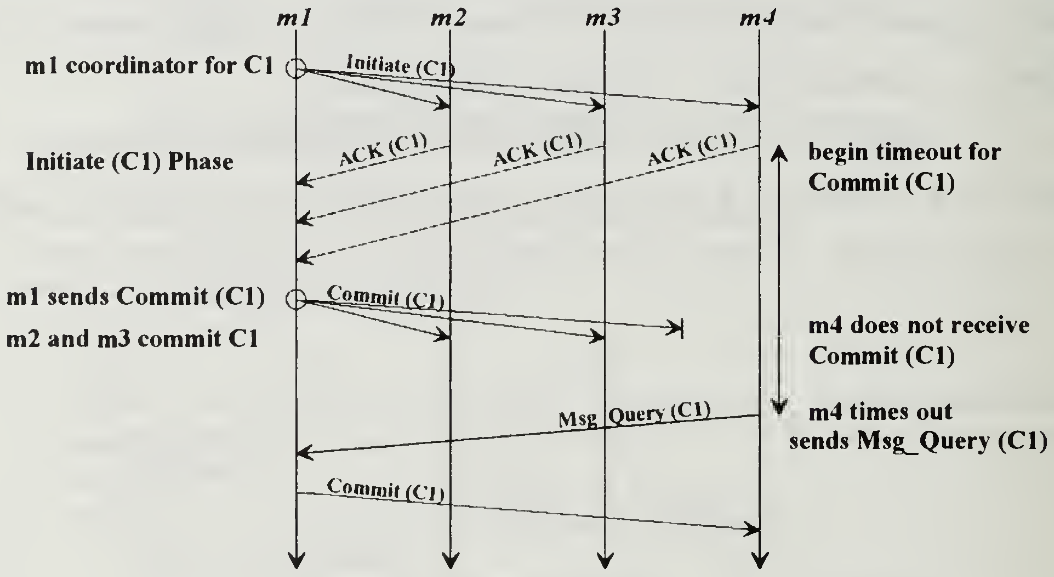


Figure 22: Lost or Delayed *Commit* Message

Figure 23 shows the sequence of events when a non-coordinator mserver is unable to receive from the coordinator. *m2* is unable to receive the *Initiate* message from coordinator *m1*, and after timeouts and retries, the coordinator declares *m2* failed. While the coordinator was waiting to receive an *ACK* message from *m2*, it received a

Msg_Query message from *m3* and *m4*, querying the coordinator about the expected *Commit* message. The coordinator responds with a *Wait* message, telling *m3* and *m4* that the coordinator is still collecting *ACKs*, and will send the *Commit* message when done. The use of the *Msg_Query* and *Wait* message is described in the next section. After the coordinator has detected *m2* failed, it sends the *Commit* message with gossip about *m2*'s failure to *m3* and *m4*, completing the change and informing them that *m2* has failed.

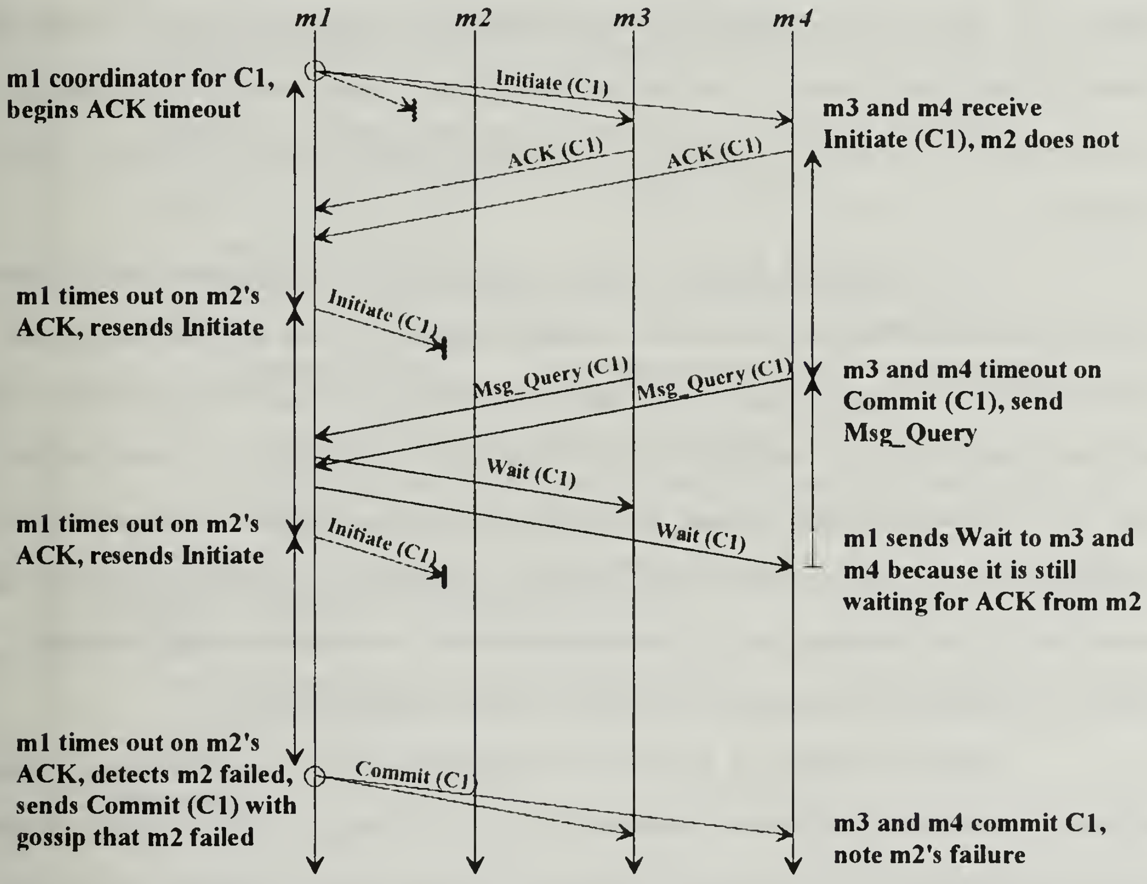

m1
m2
m3
m4
m1 coordinator for C1, begins ACK timeout
Initiate (C1)
m3 and m4 receive Initiate (C1), m2 does not
ACK (C1)
ACK (C1)
m1 times out on m2's ACK, resends Initiate
Initiate (C1)
Msg_Query (C1)
Msg_Query (C1)
m3 and m4 timeout on Commit (C1), send Msg_Query
Wait (C1)
Wait (C1)
m1 times out on m2's ACK, resends Initiate
Initiate (C1)
m1 sends Wait to m3 and m4 because it is still waiting for ACK from m2
m1 times out on m2's ACK, detects m2 failed, sends Commit (C1) with gossip that m2 failed
Commit (C1)
m3 and m4 commit C1, note m2's failure

Figure 23: Lost or Delayed *ACK* Message During Initiate Phase

b. Virtually Simultaneous Changes

The basic change protocol , listed in Figures 17 and 18 for the coordinator and non-coordinator, respectively, is unable to resolve the virtually simultaneous changes discussed previously. The Reliable_receive and Reliable_multi_receive functions used by the basic change protocol are only capable of receiving the expected message or declaring the non-responding mserver or mservers as failed. They are not able to handle unexpected messages, including an *Initiate* message from another mserver attempting to initiate a change. To allow for the occurrence of simultaneous changes and other unexpected messages, the Reliable_receive and Reliable_multi_receive functions were augmented to cover all possible unexpected messages. These augmented versions are listed in Figures 24 and 25.

The augmented Reliable_receive function has the same name and is called with the same parameters as the simpler version in Figure 15. The new version is used in place of the simpler version in line 5 of the non-coordinator basic change protocol. The modified or added lines to the algorithm are underlined in Figure 24. The new version is used by a non-coordinator core-set mserver waiting for a *Commit* message from the coordinator, by an MI or non-core-set mserver submitting an application change and waiting for a *Direct* message, and by the monitor mserver waiting for a *Reply* message. A detailed description of the augmented Reliable_receive function follows.

Lines 5 through 8 in the new algorithm detect an overlapping change initiated by an mserver that already completed the current change. A *Wait* message is sent to the attempting mserver to postpone initiation of the new change until the old change is completed. An example of this situation is shown in Figure 26. Lines 9 and 10 detect the failure of the coordinator and call the election protocol, which will be described in the next section. Lines 11 and 12 detect a virtually simultaneous change initiation of higher priority. The mserver drops the current change and begins processing the new change. An example of this situation is shown in Figure 27. Lines 13 through 15 detect the

Reliable_receive (expected_messg, query_messg)
/* Augmented **Reliable_receive** used by mserver to reliably receive *Commit* from coordinator or *Direct* from parent-mserver. Handles relevant unexpected messages. Returns the received message */

```
1.  set_timer (timeout)
2.  retries = n
3.  while ((timer not expired) and (messg != expected_messg))
4.      receive_message (messg)
5.      if (messg == Initiate_message or Coord_Fail_message)
6.          if (messg is from next change)    /* overlap from next view */
7.              form_message (Wait, current_change, exclude_list)
8.              send_message (Wait_message, responder)
9.      if (messg == Coord_Fail_message of higher priority or current coordinator)
10.         goto Broadcast Election Protocol
11.     if (messg == Initiate_message of higher priority or current coordinator)
12.         goto Non-Coordinator Basic Change Protocol
13.     if (messg == Wait_message)        /* response to Msg_Query */
14.         wait (wait_timeout)
15.         goto 1.    /* restart Reliable_receive for expected_messg */
16.     if (messg == Msg_Query_message)     /* other mserver querying status */
17.         if (current change)        /* other mserver must wait for next message */
18.             form_message (Wait, current_change, exclude_list)
19.             send_message (Wait_message, responder)
20.         if ((previous unfinished change) and (mygid == previous coordinator))
21.             form_message (Commit, current_change, exclude_list)
22.             send_message (Commit_message, responder)
23. if ((messg != expected_messg) and (retries > 0))
24.     retries = retries - 1
25.     send_message (query_messg, destination)
26.     set_timer (timeout)
27.     goto 3.
28. return messg
```

Figure 24: Augmented Reliable_receive Algorithm

reception of a *Wait* message in response to a *Msg_Query* message sent previously. The mserver waits for a period, then restarts the Reliable _receive. An example of this action is shown in Figure 26. Lines 16 through 22 detect a received *Msg_Query* message and perform the necessary actions. If the *Msg_Query* is about the current change, it is from an MI or child mserver waiting for a *Direct* message. The querying MI or mserver is sent a *Wait* message to stall their reception of the expected message. If the *Msg_Query* is about the previous change and the receiving mserver was the coordinator of the previous change, then the querying mserver did not receive the *Commit* message. The receiving mserver sends a *Commit* message to complete the last change. An example of this situation is shown in Figure 26. The remaining lines of the function are the same as the original, and perform the primary function of receiving the expected message within the timeout period.

The augmented Reliable_multi_receive function has the same name and is called with the same parameters as the simpler version. The augmented Reliable_multi_receive function is used in place of the simpler version in line 5 of the coordinator basic change protocol shown in Figure 17. The modified or added lines to the algorithm are underlined in Figure 25. The new version detects unexpected messages received while collecting *ACK* messages, and responds to them appropriately. A detailed description of the Reliable_multi_receive function follows.

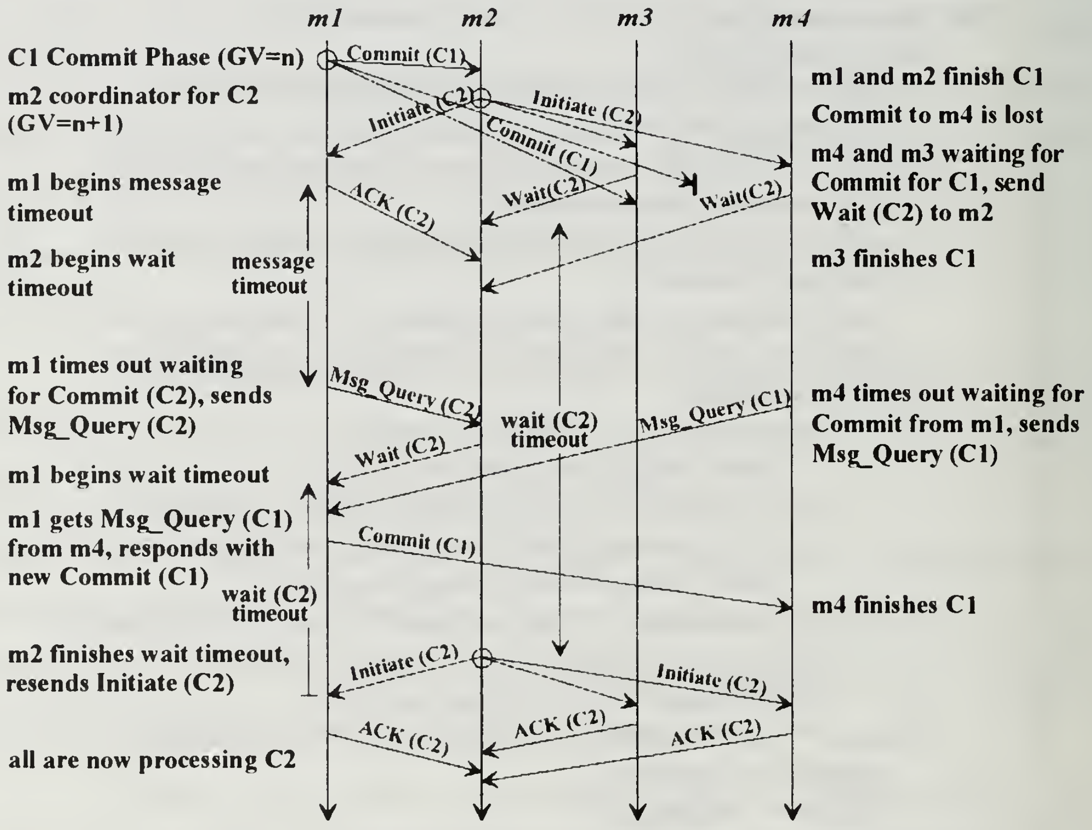
Lines 8 and 9 detect a simultaneous change of higher priority. The coordinator stores, then drops the current change, ceases to be a coordinator, and begins processing the new change. An example of this action is shown in Figure 27. Lines 10 and 11 detect the failure of the current coordinator or a new coordinator for a virtually simultaneous change of higher priority than the current change, and call the election protocol, which will be described in the next section. A received *Coord_Fail* message for a change of lower priority is quietly discarded. Lines 12 and 13 detect the reception of a *Wait* message sent by an mserver still processing the previous change. The coordinator

Reliable_multi_receive (expected_messg, last_messg, responders)
/* Augmented **Reliable_multi_receive,** used after last_messg is multicast to reliably collect all responses from other mservers or MIs. Handles relevant unexpected messages. Modifies the set of responders to reflect those not responding*/

```
1.  set_timer (timeout)
2.  retries = n
3.  initialize (all responders = not_responded)
4.  num_responders = number of responders
5.  responses = 0
6.  while ((timer not expired) and (responses < num_responders))
7.      receive_message (messg)
8.      if (messg == Initiate_message of higher priority than current change)
9.          goto Non-Coordinator Basic Change Protocol
10.     if (messg == Coord_Fail_message of higher priority than current change)
11.         goto Broadcast Election Protocol
12.     if (messg == Wait_message from previous change)   /* group_view - 1 */
13.         wait (wait_timeout)
14.         goto Coordinator Basic Change Protocol   /* restart change */
15.     if (messg == Msg_Query_message)   /* other mserver querying status */
16.         if (current change)
17.             if (responder did not receive last_messg)
18.                 send_message (last_messg, responder)
19.             else   /* responder already received last_messg, so must wait */
20.                 form_message (Wait, current_change, exclude_list)
21.                 send_message (Wait_message, responder)
22.         if ((previous unfinished change) and (mygid == previous  coordinator))
23.             form_message (Commit, current_change, exclude_list)
24.             send_message (Commit_message, responder)
25.     if ((messg == expected_message) and (responder  == not_responded))
26.         responder == responded
27.         responses = responses + 1
28. responders = {responders - (all responding responders)}
29. if  ((responses < num_responders) and (retries > 0))   /* more responses to collect */
30.     retries = retries - 1
31.     set_timer (timeout)
32.     multicast (last_messg, senders)   /* resend original message */
33.     goto 6
```

Figure 25: Augmented Reliable_multi_receive Algorithm

will perform a wait timeout, the resume the current change. An example of this is shown at the top of Figure 26. Lines 15 through 21 detect a received *Msg_Query* message and perform the necessary actions. If the *Msg_Query* is about the current change and the querying mserver did not receive the *Initiate* or *Direct* message sent prior to the



m1
m2
m3
m4
C1 Commit Phase (GV=n)
m2 coordinator for C2 (GV=n+1)
m1 begins message timeout
m2 begins wait timeout
message timeout
m1 times out waiting for Commit (C2), sends Msg_Query (C2)
m1 begins wait timeout
m1 gets Msg_Query (C1) from m4, responds with new Commit (C1)
wait (C2) timeout
m2 finishes wait timeout, resends Initiate (C2)
all are now processing C2
Commit (C1)
Initiate (C2)
Initiate (C2)
Commit (C1)
ACK (C2)
Wait(C2)
Wait(C2)
Msg_Query (C2)
wait (C2) timeout
Wait (C2)
Msg_Query (C1)
Commit (C1)
Initiate (C2)
Initiate (C2)
ACK (C2)
ACK (C2)
ACK (C2)
m1 and m2 finish C1
Commit to m4 is lost
m4 and m3 waiting for Commit for C1, send Wait (C2) to m2
m3 finishes C1
m4 times out waiting for Commit from m1, sends Msg_Query (C1)
m4 finishes C1

Figure 26: Resolution of Overlapping Changes

Reliable_multi_receive call, the mserver resends the appropriate message. An example of this situation is shown in Figure 26. If the querying mserver did receive the prior message, then the *Msg_Query* is about a new change which has been started before the old change completed. The coordinator sends a *Wait* message to stall the processing of the next change until the last change is completed. An example of this is shown in Figure 23. If the *Msg_Query* is about the previous change and the receiving mserver was the coordinator of the previous change, then the querying mserver did not receive the *Commit* message. The receiving mserver sends a *Commit* message to complete the last change. An example of this situation is shown in Figure 26. The remaining lines of the function are the same as the original, and perform the primary function of collecting the expected response message within the timeout period.

The event diagrams in Figures 26, 27, and 28 show various occurrences of concurrent changes attempting to be processed in a core-set at one time. Since the MS guarantees that only one change at a time will ever progress to completion, a method of resolving the various concurrent change attempts must be used. The algorithms in Reliable_receive and Reliable_multi_receive provide the necessary capability to resolve

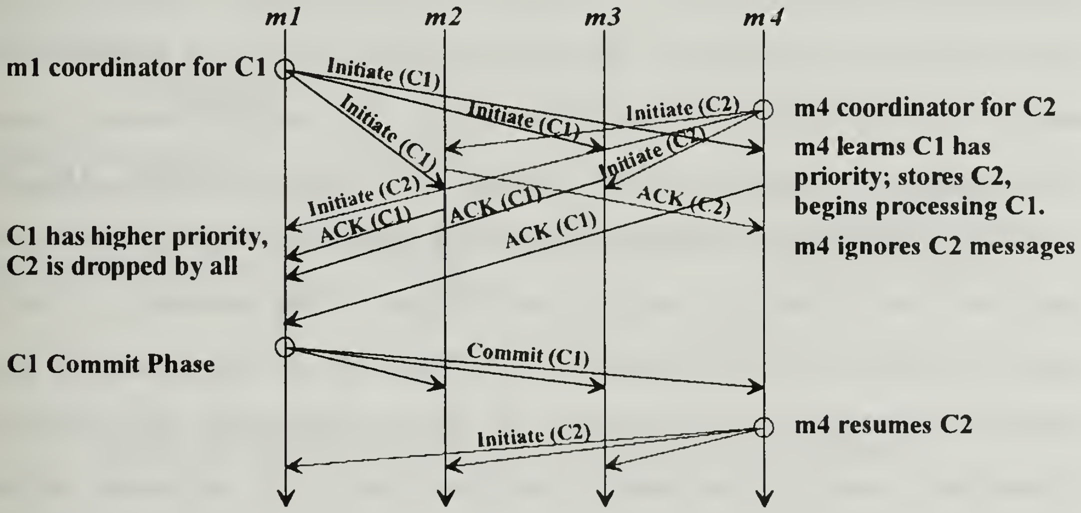


Figure 27: Resolution of Virtually Simultaneous Changes

concurrent changes to a single change. Figure 26 shows one case where an mserver *m2* has completed the previous change C1 and immediately initiates the next change C2. This is a very likely occurrence, since each mserver will queue failures and changes, waiting for the next opportunity to initiate them. Mserver *m4* did not receive the *Commit* message for C1, so when it receives the *Initiate* message for C2 it sends a *Wait* message to the new coordinator *m2* to postpone the new change C2 until the old change C1 is completed. *m4* times out waiting for the *Commit* message for C1 and sends a *Msg_Query* to *m1*, which is now processing C2. *m1* receives *m4*'s *Msg_Query* while using the Reliable_receive function and sends *m4* another *Commit* message. *m4* is now able to complete C1. After *m2* finishes the wait timeout, it resumes C2.

Figure 27 shows two core-set mservers attempting to initiate virtually simultaneous changes. All core-set mservers receive both *Initiate* messages, although perhaps in different order due to the asynchronous environment. All core-set mservers have the same core-set state information, and therefore make exactly the same decision about which change has priority. In this case, C1 has priority, and is therefore recognized as the change to be processed by all core-set mservers, while C2 is dropped by all. The candidate coordinators must collect *ACK*s from all other core-set mservers, including the other candidate coordinator, before sending the *Commit* message. The candidate coordinators make the same decision about which change has priority; therefore, only one coordinator will be selected and only one change will be processed. If all core-set mservers receive all *Initiate* messages, it is impossible for more than one change to progress to completion in the core-set. It will be shown later, that under non-ideal circumstances, some core-set mservers may have failed or do not receive all messages, leading to a situation where more than one change is being processed in the same core-set. However, it will also be shown that if this occurs, the core-set will always partition in such a manner that all core-set mservers in the partition will be processing the same change, and will arrive at the same consistent view.

Figure 28 is a combination of the overlapping change shown in Figure 26 and the virtually simultaneous changes shown in Figure 27. This event diagram shows that even under the circumstances when changes overlap at the beginning and end of a change, only one change at a time progresses to completion.

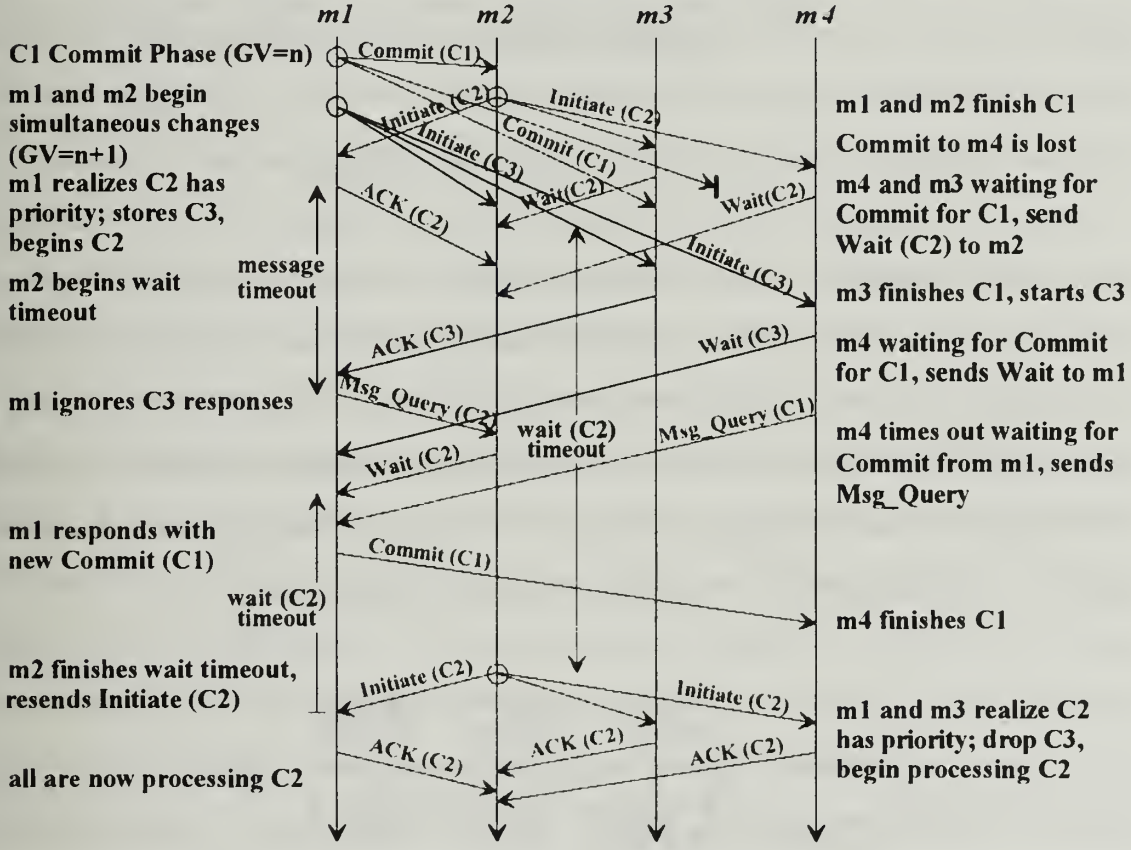

m1
m2
m3
m4
C1 Commit Phase (GV=n)
m1 and m2 begin simultaneous changes (GV=n+1)
m1 realizes C2 has priority; stores C3, begins C2
m2 begins wait timeout
message timeout
m1 ignores C3 responses
m1 responds with new Commit (C1)
wait (C2) timeout
m2 finishes wait timeout, resends Initiate (C2)
all are now processing C2
Commit (C1)
Initiate (C2)
Initiate (C3)
Commit (C1)
Wait(C2)
ACK (C2)
Wait(C2)
Initiate (C3)
ACK (C3)
Wait (C3)
Msg_Query (C2)
wait (C2) timeout
Msg_Query (C1)
Wait (C2)
Commit (C1)
Initiate (C2)
Initiate (C2)
ACK (C2)
ACK (C2)
ACK (C2)
m1 and m2 finish C1
Commit to m4 is lost
m4 and m3 waiting for Commit for C1, send Wait (C2) to m2
m3 finishes C1, starts C3
m4 waiting for Commit for C1, sends Wait to m1
m4 times out waiting for Commit from m1, sends Msg_Query
m4 finishes C1
m1 and m3 realize C2 has priority; drop C3, begin processing C2

Figure 28:Virtually Simultaneous and Overlapping Changes

5. Coordinator failure

The basic change-processing protocol assumes that the coordinator of the change will continue to function throughout the processing of the change. The protocol definitions and examples to this point handle various situations, including the failure of non-coordinator core-set mservers. However, it is entirely possible that the coordinator of a change may fail or be unable to communicate with others during the processing of a change. In the event of the coordinator failure, a new coordinator must be elected. Birman and Riccardi [9] have proven that when the coordinator of a change can fail, a three-phase change protocol is required. To this end, another phase must be added to the two-phase basic change protocol. This phase is a broadcast election phase, as described in [26], which is conducted to elect a new coordinator to resume the original change being processed. After the distributed election is accomplished, the new coordinator will restart the original change with a new *Initiate* message to all surviving core-set mservers, or, under special circumstances, will simply send a *Commit* message to complete the change. An illustration of the three-phase election and change processing protocol is shown in Figure 29, and the algorithm or the broadcast election is listed in Figure 30.

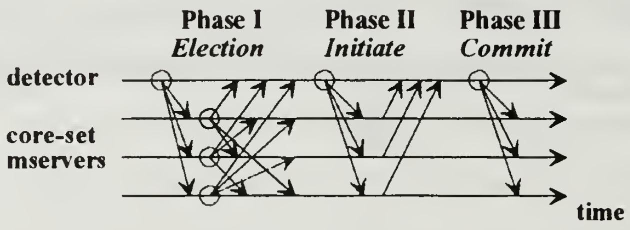


Figure 29:Election and Change-processing Protocol

```
Broadcast Election
/* Coordinator failure has been detected.  Elect new coordinator */
1.  update (exclude_list, internal state)
2.  form_message (Coord_Fail, current_change, exclude_list)
3.  multicast (Coord_Fail_message, {core_set - exclude_list - coordinator})
4.  Reliable_multi_receive (Coord_Fail_message, Coord_Fail_message, {core_set -
    exclude_list - coordinator})
5.  update (exclude_list, internal state)
    /* determine new coordinator from responding mservers */
6.  coordinator = highest rank mserver with current change active
7.      Resume_change (current_change, coordinator)
```

Figure 30:Broadcast Election Protocol

The broadcast election protocol is commenced upon detection by one or more mservers of the failure of the current coordinator. This detection could occur by monitoring or by the timeout of an expected message while using the Reliable_receive function. The detecting mserver will multicast a *Coord_Fail* message to all other core-set mservers, which will include the status of the mserver in processing the current change. This mserver will then collect responses from all other mservers with the Reliable_multi_receive function. The other mservers receiving the *Coord_Fail* message will also multicast their status and collect responses from all others. In this way, all core-set mservers learn of the status of all other core-set mservers with respect to the interrupted change. These steps are covered by lines 1 through 5 in Figure 30. In order for an mserver to become the new coordinator, it must have received the original *Initiate* message, but not yet have committed the change. Only an mserver still processing the change will have sufficient information to restart the change. There is guaranteed to be at least one such mserver, since only an mserver still processing the change could determine that the coordinator had failed. Since all mservers have learned the status of all other mservers, a uniform distributed decision can be made by all as to the identity of the new coordinator. To select the new coordinator from those mservers still processing the current change, a priority scheme is used. The mserver with the highest rank in the core-set, still processing the original change, will become the new coordinator. All core-set mservers know the rank of all other core-set mservers, so they all make exactly the same distributed decision. Thus, a single new coordinator is chosen.

Once the new coordinator is chosen, it will resume the original change using the two-phase basic change-processing protocol, as shown in Figure 29. However, if at least one core-set mserver has committed the change, then it is safe for the coordinator to immediately multicast a *Commit* message to have all core-set mservers commit the change. This is possible due to the fact that in order for any mserver to have received a *Commit* message from the failed coordinator, that coordinator must have received *ACK*s from all surviving core-set mservers. This means that all mservers in the core-set have knowledge of the change, and can therefore commit the change. Any mserver that did not have knowledge of the change would have been detected failed by the old coordinator, using the Reliable_multi_receive procedure. The old coordinator would include all detected failures in the exclude_list added to each multicast message, and thus any mserver receiving the *Commit* message would learn of the detected failure of all mservers which had not received the original *Initiate* message. The mserver, learning by gossip of the failure of other mservers, would cease to communicate with them. These excluded mservers will be removed from the core-set, so that only mservers which had received the original change remain.

Figure 31 shows the event diagram for the compressed election and change-processing protocol described above. Figure 32 is the listing for the Resume_change function used in Line 7 of the broadcast election protocol. This function makes the decision for the new coordinator whether to restart the original change with an *Initiate* message or use the compressed protocol and simply multicast a *Commit* message to complete the original change.

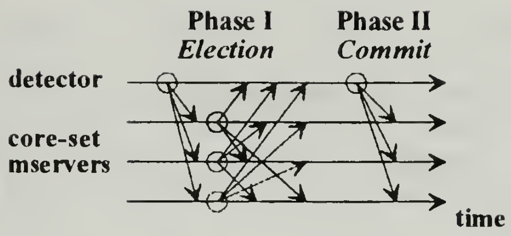


Figure 31: Compressed Election and Change-processing Protocol

```
Resume_change
/* Following broadcast election of new coordinator. */
1.  if (any core_set mserver has committed the change)
    /* then use compressed change protocol - send/receive Commit only */
2.      if (coordinator)
3.          form_message (Commit, current_change, exclude_list)
4.          multicast (Commit_message, {core_set - exclude_list - coordinator})
5.      else  /*non_coordinator */
6.          messg = Reliable_receive (Commit_message, Msg_Query_message)
7.  else  /* no mservers have committed the change - must restart */
8.      Reinitiate (current_change)
```

Figure 32: Resume_Change Algorithm

Examples of various scenarios involving the failure of the current coordinator are shown in five event diagrams on the following pages. These examples illustrate some of the more likely scenarios which might be encountered when a coordinator is detected failed, and the sequence of events leading to the election of a new coordinator and completion of the original change.

Figure 33 shows the sequence of events when the coordinator fails in the Initiate phase, immediately after multicasting the *Initiate* message. All other core-set mservers time out waiting for the *Commit* message, detect the coordinator failed, and conduct an election for a new coordinator. The new coordinator completes the original change.

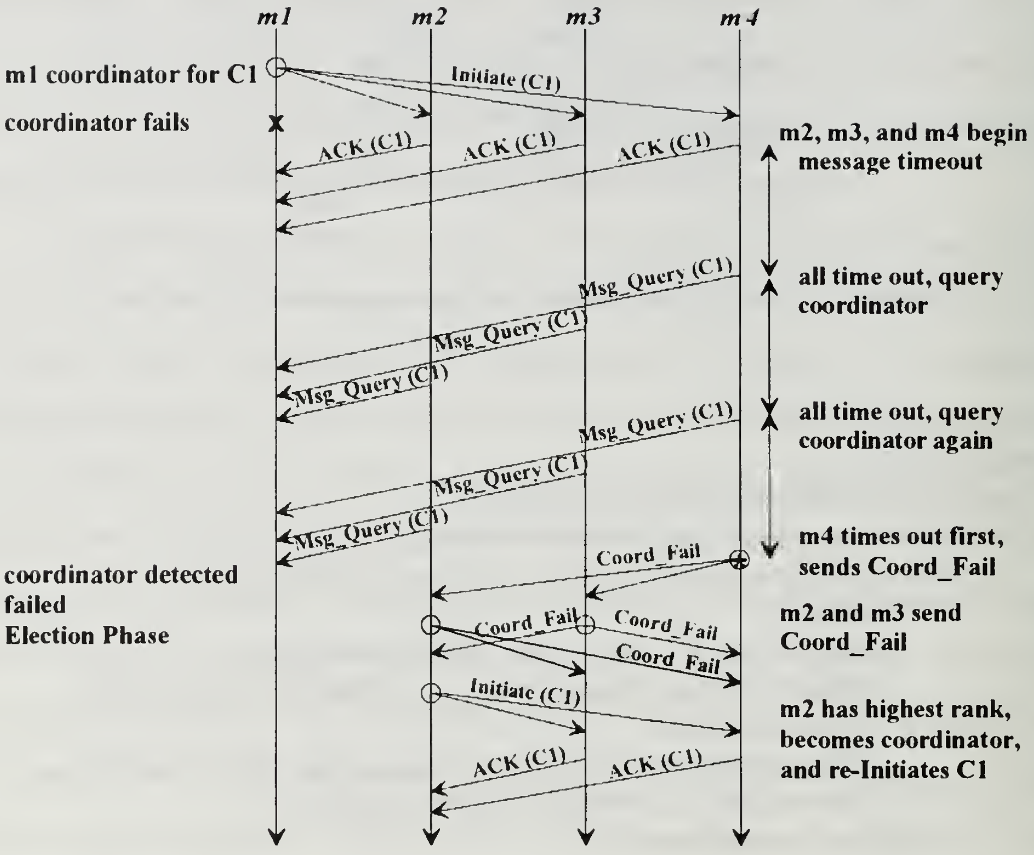

m1
m2
m3
m4
m1 coordinator for C1
coordinator fails
Initiate (C1)
ACK (C1)
ACK (C1)
ACK (C1)
m2, m3, and m4 begin message timeout
Msg_Query (C1)
Msg_Query (C1)
Msg_Query (C1)
all time out, query coordinator
Msg_Query (C1)
Msg_Query (C1)
Msg_Query (C1)
all time out, query coordinator again
Coord_Fail
m4 times out first, sends Coord_Fail
coordinator detected failed
Election Phase
Coord_Fail
Coord_Fail
Coord_Fail
m2 and m3 send Coord_Fail
Initiate (C1)
m2 has highest rank, becomes coordinator, and re-Initiates C1
ACK (C1)
ACK (C1)

Figure 33: Coordinator Failure During Initiate Phase

Figure 34 illustrates the case where the monitor of the coordinator is unable to receive from the coordinator. The monitor *m2* detects the coordinator *m1* failed by monitoring and initiates a change C2 for the failure of *m1* by multicasting an *Initiate* message to all core-set mservers. The other core-set mservers are already processing the change C1. The change C2 is recognized by *m3* and *m4* as a failure of the current coordinator; however, it is also a virtually simultaneous change, since no mserver has committed C1. For this reason, C2 is treated as a virtually simultaneous change of higher priority than C1, avoiding the need to elect a new coordinator. Since the failed coordinator initiated the original change and then failed, there is no need to resume processing of this change. If an

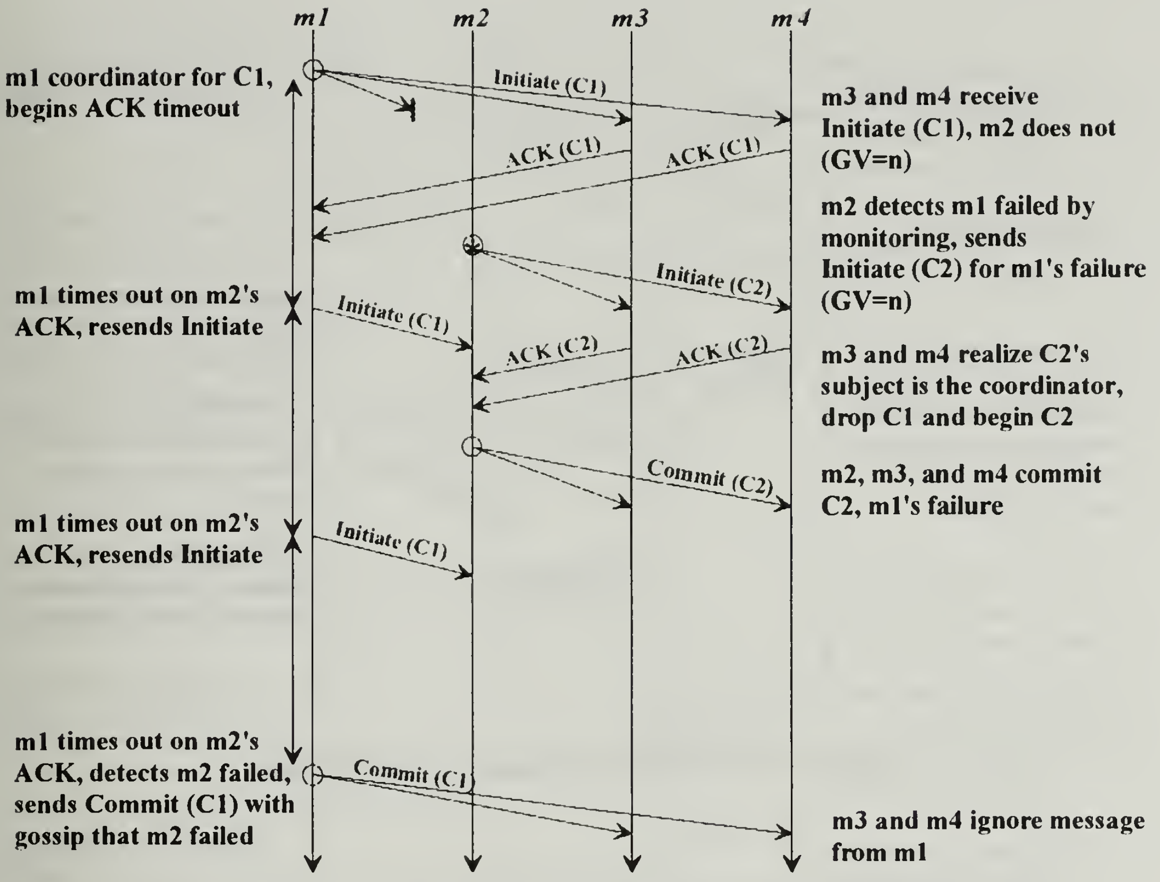

m1
m2
m3
m4
m1 coordinator for C1, begins ACK timeout
Initiate (C1)
m3 and m4 receive Initiate (C1), m2 does not (GV=n)
ACK (C1)
ACK (C1)
m2 detects m1 failed by monitoring, sends Initiate (C2) for m1's failure (GV=n)
Initiate (C2)
m1 times out on m2's ACK, resends Initiate
Initiate (C1)
ACK (C2)
ACK (C2)
m3 and m4 realize C2's subject is the coordinator, drop C1 and begin C2
Commit (C2)
m2, m3, and m4 commit C2, m1's failure
m1 times out on m2's ACK, resends Initiate
Initiate (C1)
m1 times out on m2's ACK, detects m2 failed, sends Commit (C1) with gossip that m2 failed
Commit (C1)
m3 and m4 ignore message from m1

Figure 34: Coordinator Failure With Lost *Initiate* Message

application group submitted the change to *m1*, the group will be partitioned at *m1* anyway, so there is no need to process the submitted change on the other side of the partition. If *m1*'s change was a physical change about a core-set mserver, it will either be redetected and processed, or perhaps will remedy itself.

Figure 35 illustrates the case where the coordinator is detected failed in the Commit phase, after one or more mservers have received the *Commit* message. The core-set mservers conduct a broadcast election in which *m2* becomes the new coordinator. Since *m4* committed the original change, the compressed change protocol is used, and *m2* multicasts a *Commit* message to finish the change.

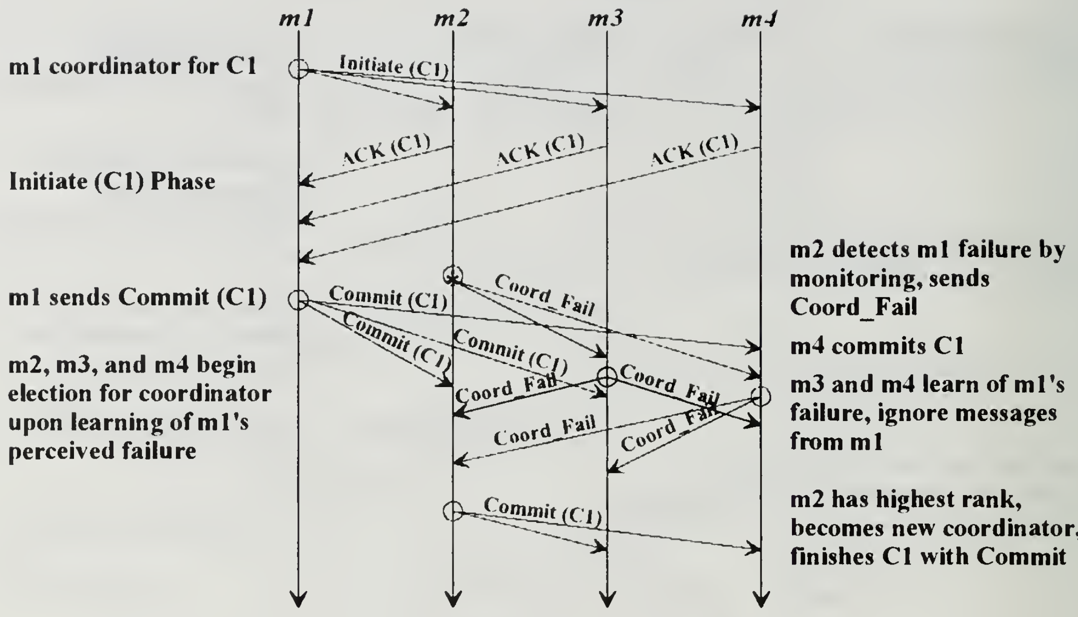
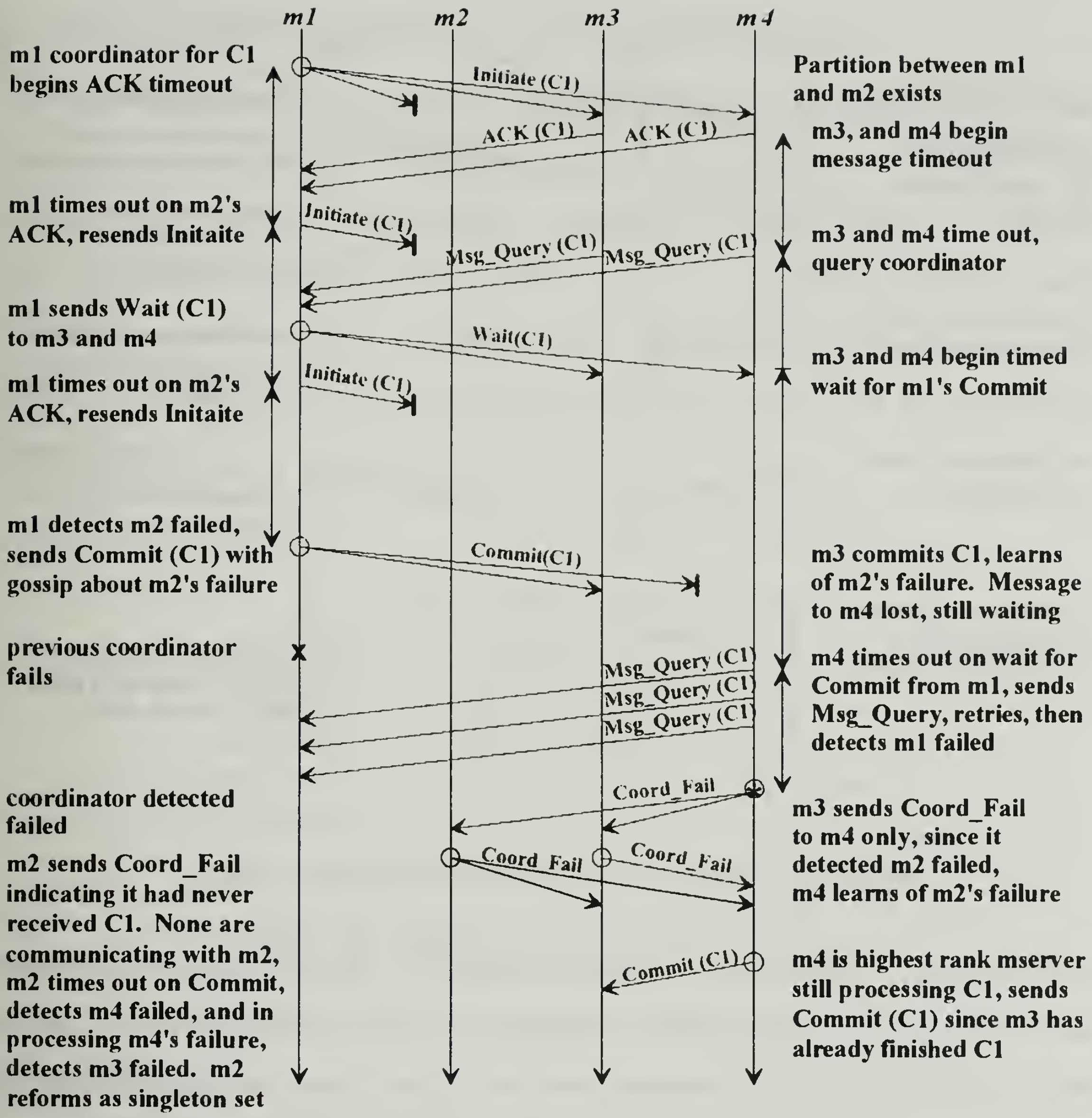

m1
m2
m3
m4
m1 coordinator for C1
Initiate (C1)
ACK (C1)
ACK (C1)
ACK (C1)
Initiate (C1) Phase
m1 sends Commit (C1)
Commit (C1)
Commit (C1)
Commit (C1)
Coord_Fail
Coord_Fail
Coord_Fail
Coord_Fail
Coord_Fail
m2, m3, and m4 begin election for coordinator upon learning of m1's perceived failure
m2 detects m1 failure by monitoring, sends Coord_Fail
m4 commits C1
m3 and m4 learn of m1's failure, ignore messages from m1
Commit (C1)
m2 has highest rank, becomes new coordinator, finishes C1 with Commit

Figure 35: Coordinator Failure In Commit Phase

Figure 36 illustrates an unusual case where the failure of the coordinator and lost messages lead to one core-set mserver committing the change C1 (*m3*), one mserver still processing the change (*m4*), and one mserver never having received the change (*m2*). As a result of this situation, *m4* will become the new coordinator, since it is the only mserver still processing C1. Mserver *m3* learned of *m2*'s detected failure with the *Commit* message received from the original coordinator *m1*. The end result is that *m3* and *m4* commit C1 and reform into a new core-set, while *m2* never learns of C1, and is excluded from the original core-set.

The final event diagram shown in Figure 37 shows a situation in which the coordinator has failed after multicasting an *Initiate* message which was received by only one core-set mserver. Another core-set mserver, *m4*, also initiated a virtually simultaneous change of lower priority. *m1* receives the *Initiate* message for both changes



m1 coordinator for C1 begins ACK timeout

m1 times out on m2's ACK, resends Initaite

m1 sends Wait (C1) to m3 and m4

m1 times out on m2's ACK, resends Initaite

m1 detects m2 failed, sends Commit (C1) with gossip about m2's failure

previous coordinator fails

coordinator detected failed

m2 sends Coord_Fail indicating it had never received C1. None are communicating with m2, m2 times out on Commit, detects m4 failed, and in processing m4's failure, detects m3 failed. m2 reforms as singleton set

Partition between m1 and m2 exists

m3, and m4 begin message timeout

m3 and m4 time out, query coordinator

m3 and m4 begin timed wait for m1's Commit

m3 commits C1, learns of m2's failure. Message to m4 lost, still waiting

m4 times out on wait for Commit from m1, sends Msg_Query, retries, then detects m1 failed

m3 sends Coord_Fail to m4 only, since it detected m2 failed, m4 learns of m2's failure

m4 is highest rank mserver still processing C1, sends Commit (C1) since m3 has already finished C1

Figure 36: Coordinator Failure With Lost Messages

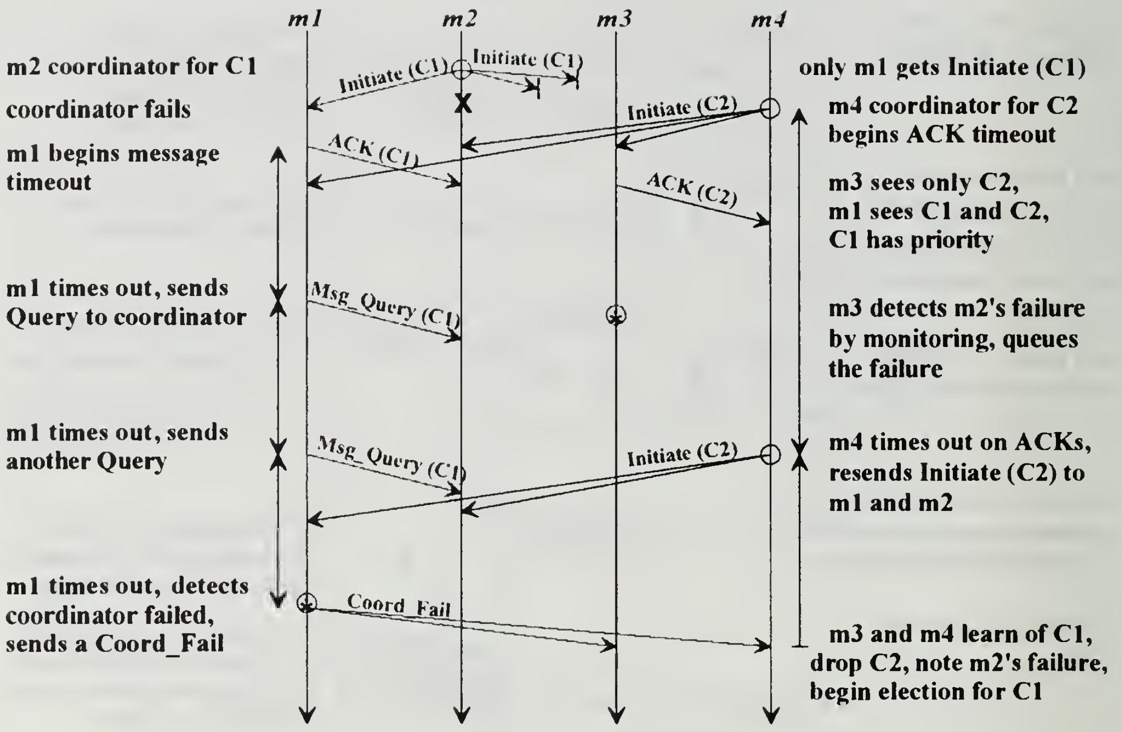

m1
m2
m3
m4
m2 coordinator for C1
coordinator fails
m1 begins message timeout
m1 times out, sends Query to coordinator
m1 times out, sends another Query
m1 times out, detects coordinator failed, sends a Coord_Fail
Initiate (C1)
Initiate (C1)
Initiate (C2)
ACK (C1)
ACK (C2)
Msg_Query (C1)
Msg_Query (C1)
Initiate (C2)
Coord_Fail
only m1 gets Initiate (C1)
m4 coordinator for C2 begins ACK timeout
m3 sees only C2, m1 sees C1 and C2, C1 has priority
m3 detects m2's failure by monitoring, queues the failure
m4 times out on ACKs, resends Initiate (C2) to m1 and m2
m3 and m4 learn of C1, drop C2, note m2's failure, begin election for C1

Figure 37: Coordinator Failure With Simultaneous Change

and decides that C1 has priority, and therefore drops C2, assuming that all other core-set mservers will make the same decision. However, *m3* and *m4* did not receive C1, so they continue to process C2. *m1* eventually times out on the *Commit* message expected from the *m2*, detects the coordinator failed, and multicasts a *Coord_Fail* message to all core-set mservers. *m3* and *m4* now learn of the higher priority change C1. Since no mservers had committed C2, the change is dropped and *m3* and *m4* begin processing C1 with an election for a new coordinator.

D. APPLICATION GROUP CHANGE PROCESSING

The general description of application group change processing has already been presented in previous sections. In this section, the protocols necessary to submit a change to the application core-set and then reliably propagate the core-set change directive back to the application are presented. These protocols are divided into the algorithms used by MIs submitting or receiving a change directive, and those used by mservers in the hierarchy or in the core-set of the application. Figure 38 shows the basic application change protocol.

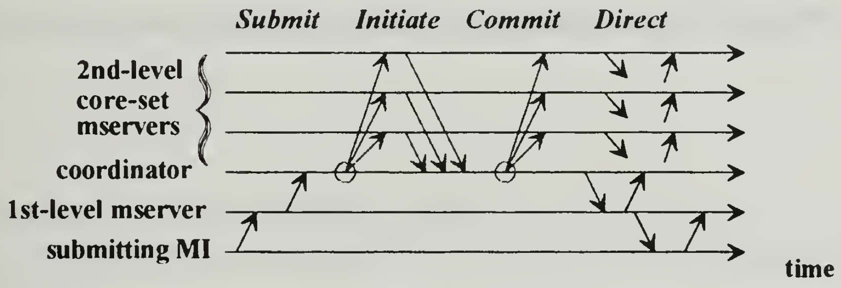


Figure 38: Application Group Change Protocol

1. MIs

The MI accepts change requests from the application groups that it supports and relays these requests to the LAN mserver for submission to the application core-set. The MI may also detect application process members failed and submit these changes as well. The algorithms used by an MI are listed in Figures 39 and 40, for the submitting MI and a non-submitting MI, respectively.

Submitting MI Basic Application Change

```
/* MI received change request or detected change in an application group */
/* Submit phase */
1.  current_change = application change data (received or detected)
2.  update (internal state)
3.  form_message (Submit, current_change)
4.  send_message (Submit_message, parent_mserver)
5.  messg = Reliable_receive (Direct_message, Msg_Query_message)
/* Direct phase */
6.  form_message (ACK, current_change)
7.  send_message (ACK_message, parent_mserver)
8.  update (internal state)
9.  application_group_view = application_group_view + 1
10. reliably inform application of change
```

Figure 39: Submitting MI Application Group Change Protocol

Non-submitting MI Basic Application Change

```
/* MI received change Direct message from parent_mserver */
/* Direct phase */
1.  form_message (ACK, current_change)
2.  send_message (ACK_message, parent_mserver)
3.  update (internal state)
4.  application_group_view = application_group_view + 1
5.  reliably inform application of change
```

Figure 40: Non-submitting MI Application Group Change Protocol

2. Mservers

The LAN mserver accepts application change requests and failures submitted by the MIs running on the host computers of the LAN. These changes are then submitted up the MS hierarchy of mservers to the application core-set, where the change is processed. Once the core-set commits the change, all core-set mservers with application members below them multicast the change directive to their children mservers with application members below them. At each level an *ACK* is sent to the parent mserver to ensure reliable delivery of the change directive to all application member processes. The change directive is propagated to each MI with members of this application, which then inform the application members of the completed change. The algorithms used by mservers are listed in Figures 41 and 42, for the non-core-set mserver and application core-set mserver, respectively.

Non-core-set Mserver Basic Application Change

```
/* mserver received Submit message relayed from submitting MI; will reliably relay to
parent_mserver */
/* Submit phase */
1.  send_message (Submit_message, parent_mserver)
2.  messg = Reliable_receive (Direct_message, Msg_Query_message)
3.  if (messg != Direct_message)   /* failed parent_mserver */
4.          goto Broadcast Election Protocol
5.  update (internal state)
/* Direct phase */
6.  form_message (ACK, current_change)
7.  send_message (ACK_message, parent_mserver)
8.  form_message (Direct, current_change, exclude_list)
9.  multicast (Direct_message, {children with application members - excluded})
10. Reliable_multi_receive (ACK_message, Direct_message, {children with application
    members - excluded})
11. update (internal state)
```

Figure 41: Non-core-set Mserver Application Group Change Protocol

```
Core-set Mserver Basic Application Change
/* core-set mserver learned of application change by Submit message relayed from
submitting MI or Initiate message from application change coordinator */
1.  execute Basic Change Protocol
2.  form_message (Direct, current_change, exclude_list)
3.  multicast (Direct_message, {children with application members - excluded})
4.  Reliable_multi_receive (ACK_message, Direct_message, {children with application
    members - excluded})
5.  update (exclude_list, internal state)
```

Figure 42: Core-set Mserver Application Group Change Protocol

Figure 43 is an event diagram showing the actions when a *Submit* message is lost. In this case, the *Submit* message is lost between the LAN mserver and the core-set mserver. The MI times out waiting for the *Direct* message from the LAN mserver and sends a *Msg_Query*. The LAN mserver resends the *Submit* message to the core-set and also sends a *Wait* message to the querying MI, indicating that the mserver is still pursuing the application change and the MI should wait for a while longer before detecting a failure. The core-set now receives the *Submit* message, completes the processing of the application change, and propagates a *Direct* message to the LAN mservers and then to the MIs. The LAN mservers send an *ACK* to the core-set, and the MIs send an *ACK* to the LAN mserver, indicating successful propagation of the *Direct* message.

Figure 44 is very similar to Figure 43 except the *Direct* message is lost instead of the *Submit* message. The MI times out waiting for the *Direct* message from the LAN mserver and sends a *Msg_Query*. Instead of sending a *Wait* message to the querying MI, the LAN mserver sends a *Msg_Query* to the core-set. The mserver in the core-set receiving the query resends the lost *Direct* message, which is propagated to the MI with *ACK*s returned at every level, and then to the application.

Figure 45 shows the failure of a core-set mserver after processing the application change, but before multicasting the change directive to the children mservers. Using message timeouts and retries, the LAN mserver detects the parent mserver in the core-set failed. The MS hierarchy is partitioned at the failed core-set mserver, causing a partition in the application group as well.

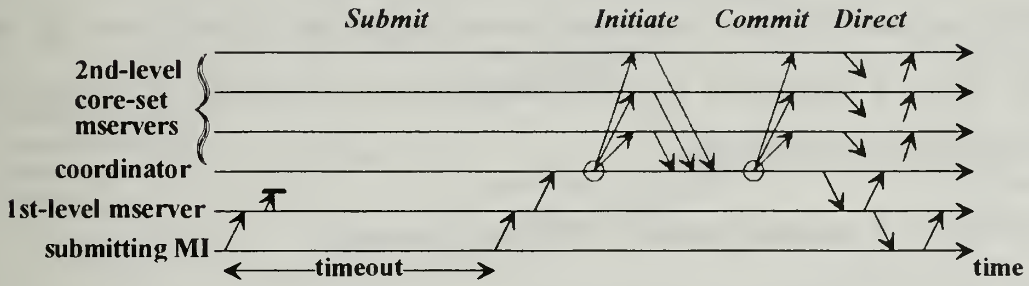


Figure 43: Application Group Change With Lost *Submit* Message

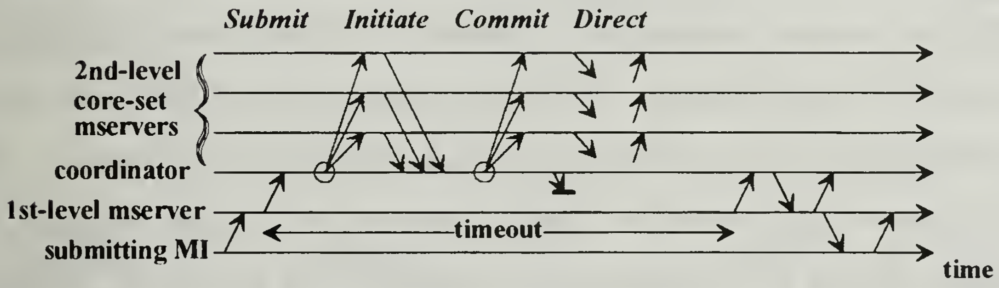


Figure 44: Application Group Change With Lost *Direct* Message

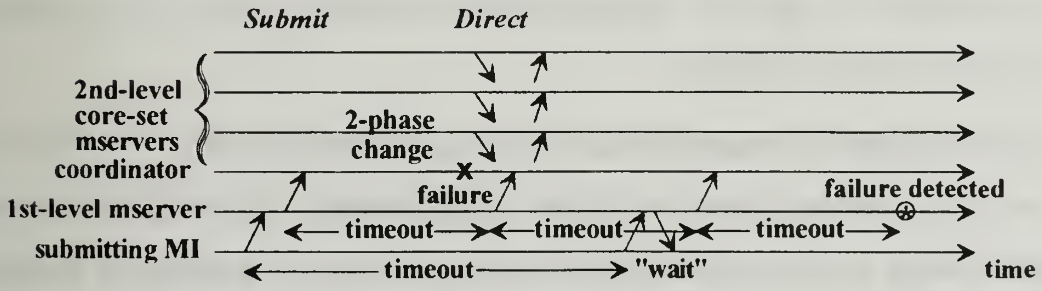


Figure 45: Application Group Change With Failed Coordinator

E. PARTITION RESOLUTION

The final protocol provides the means for the MS hierarchy and application groups to dynamically reconfigure in the event of network partitions. The reconfiguration method of the physical hierarchy is fixed, whereas the reconfiguration method used by each application group is determined by QoS selections made by each MSU when the application group is initially created.

1. Dynamic Reconfiguration of Physical Core-set

a. Perceived Failures and Partitions

As discussed previously, the actual failure of one or more mservers in a core-set is indistinguishable from the perceived failure of these mservers caused by an interruption in the network communication capability. For this reason, all perceived failures are treated as actual failures. The failed mservers are excluded from further core-set communications, and the core-set is reformed without the failed mservers. One partition of the core-set will contain the original parent mserver, the other partitions will not. This means that the physical hierarchy of mservers is also partitioned. However, there exists a possibility that the mservers perceived as failed are still functioning. It would be desirable to have these mservers automatically rejoin the original core-set when the network partition is repaired. This protocol provides the means for this automatic reformation of the physical hierarchy.

b. Automatic Reformation Using the Shared Multicast Group

The monitoring protocol, basic change protocol, and broadcast election protocol provide the means to detect failures or perceived failures of mservers. The basic change protocol provides the means to process the failure of core-set mservers and reform the core-set. An example of such a reformation due to a network partition is displayed in Figure 46, with the reformed subsets of mservers shown in Figure 47. Although the

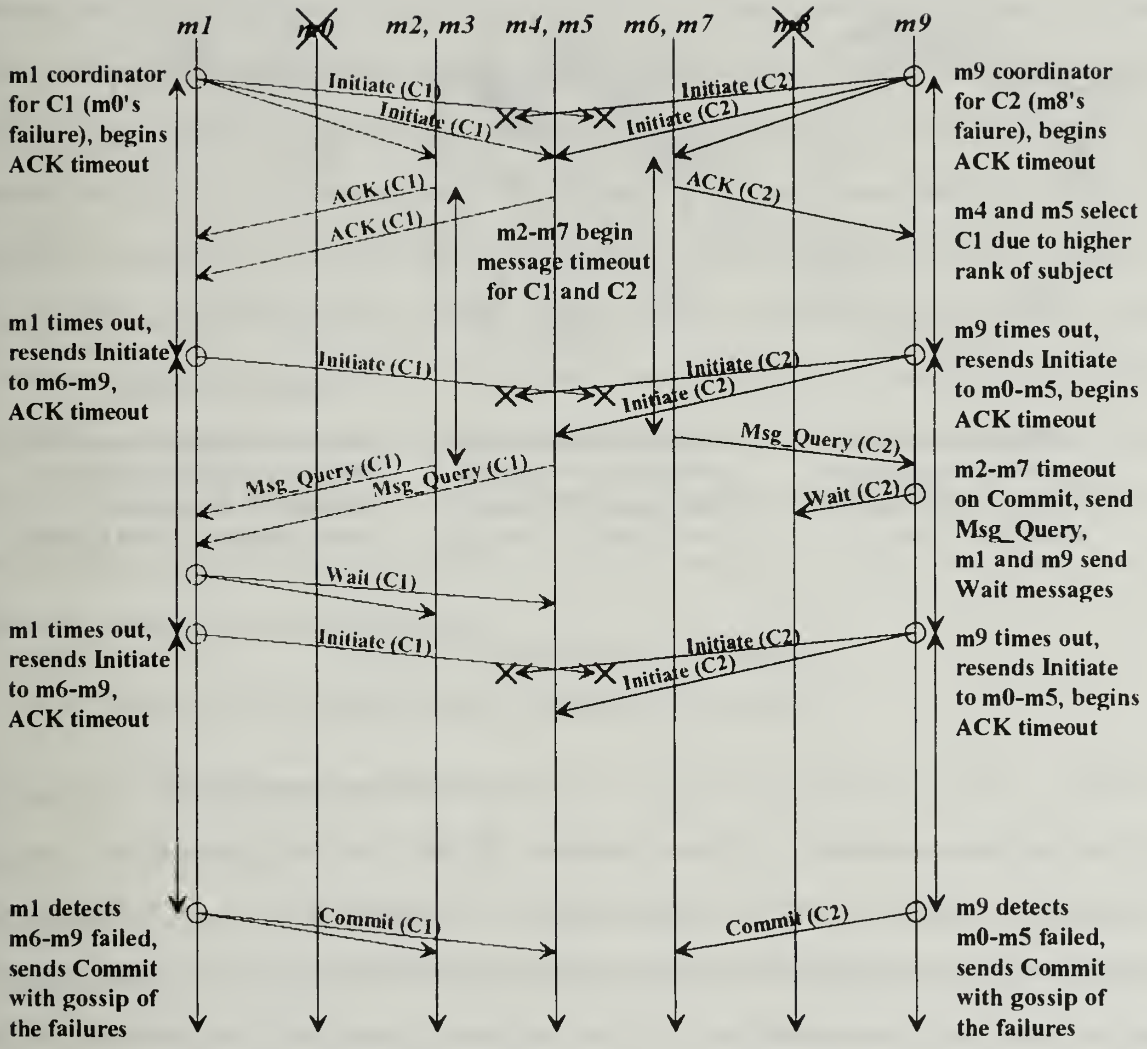
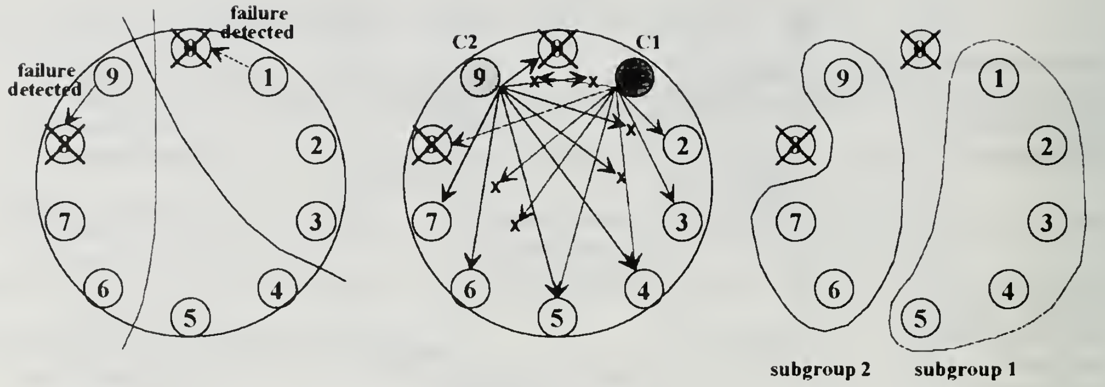

m1
m0
m2, m3
m4, m5
m6, m7
m8
m9
m1 coordinator for C1 (m0's failure), begins ACK timeout
m9 coordinator for C2 (m8's faiure), begins ACK timeout
Initiate (C1)
Initiate (C1)
Initiate (C2)
Initiate (C2)
ACK (C1)
ACK (C1)
ACK (C2)
m2-m7 begin message timeout for C1 and C2
m4 and m5 select C1 due to higher rank of subject
m1 times out, resends Initiate to m6-m9, ACK timeout
m9 times out, resends Initiate to m0-m5, begins ACK timeout
Msg_Query (C2)
Msg_Query (C1)
Msg_Query (C1)
Wait (C2)
m2-m7 timeout on Commit, send Msg_Query, m1 and m9 send Wait messages
Wait (C1)
m1 detects m6-m9 failed, sends Commit with gossip of the failures
m9 detects m0-m5 failed, sends Commit with gossip of the failures
Commit (C1)
Commit (C2)

Figure 46: Partitioning of a Core-set

perceived failed mservers have been removed from the core-set membership, they have not been removed from the membership of the multicast group which the core-set uses to multicast change-related messages. This provides the means by which an automatic reformation of the original core-set may be accomplished.

Once the core-set is reformed without the perceived failed mservers, attempts are periodically made to reestablish communications with the other partition of



Simultaneous failures detected by m1 and m9. Two partitions also exist in the set as shown.

Coordinator m1 unable to send to m6, m7, and m9. Coordinator m9 unable to send to m1, m2, and m3.

After processing C1 and C2, the set has reformed into two subgroups. Subgroup 1 is larger because C1 had priority.

Figure 47: Partitioning of a Core-set

mservers. These attempts are made by multicasting query messages to the original core-set multicast address. Current members of the core-set ignore these queries; however, an mserver in the other partition will respond to a query, if able. If communication is reestablished within a predetermined timeout period, a simple merge of the two partitions is conducted, restoring the original core-set, with the exception of any new additions or deletions to either partition. The group view of the reformed core-set is set to one more than the higher view number of two formerly partitioned subsets. This is the same action that would be performed with an ordinary merge of two separate core-sets of mservers. The original parent mserver is now a member of both of the reformed subsets, so the physical hierarchy is also automatically restored.

c. *Unique Names and Addresses of Partitioned Core-sets*

In the event that the partitions of mservers are unable to restore communciations, the reformed subsets are converted to completely independent core-sets. Since all core-sets of mservers must have a unique name and multicast address, some method must be used to automatically obtain these unique values. To obtain a unique name, each sub-core-set appends a unique suffix to the original core-set group name. This suffix value must be automatically derived by each partitioned subset of mserves independently, and with a guaranteed unique value for all partitioned subsets. The most readily available attribute that all subsets can use to obtain a guaranteed unique name is the original group identity (gid) of a significant mserver remaining in each partition. The lowest mserver gid of the mservers remaining in each partition is appended to the original core-set group name. In this manner, all partioned core-sets are guaranteed a unique core-set name. However, all partioned core-sets are still easily identifiable as subsets of the original core-set, which simplifies the task of manually reconfiguring the physical hierarchy when the network is repaired.

2. Dynamic Reconfiguration of Application Groups

a. Reconfiguration Rules

Any partition of the MS physical hierarchy results in a partition of all application groups which spanned the original hierarchy. The method of resolving the partitions of each of the application groups depends on the QoS selection made by the MSU at the time the application group was created. The MSU uses the size, membership and name characteristics associated with an application group as the parameters to specify how partitions will be resolved. These parameters are used by two rules which explicitly determine how partitioned subgroups will be handled. These rules are:

1. Keep alive any partitioned subgroups that meet a certain condition specified by the user. Any subgroups which do not meet the condition will be terminated.
2. Partitioned subgroups will attempt to find and merge with other partitioned subgroups that have a certain user-specified property.

The first rule utilizes a user-specified condition related to group size and/or membership to determine which subgroups will continue to function. Using group size as the condition for deciding which subgroups survive, the MSU may specify that all subgroups terminate upon a partition by selecting a size equal to the original group size. All partitioned subgroups would be smaller than the original group, and would therefore terminate, also terminating the application. Similarly, the MSU may specify that all subgroups survive a partition by selecting a limiting size equal to zero. All partitioned subgroups would be larger than the selected size and would continue to function. Any size between zero and the original group size may be selected, permitting subgroups larger (or smaller) than the specified size to continue to function, and terminating all subgroups smaller (or larger). The membership of the group may also be used to determine which subgroups survive. The MSU may specify the condition that a subgroup must have a particular member or type of member to continue to function. Any subgroups not containing such a member will terminate.

The second rule utilizes an MSU specified property related to the original group name or the identity and location of significant members of the group to determine which partitioned subgroups will attempt to merge. The simplest case is that all subgroups attempt to merge with all other subgroups of the original group. The property used is the same base group name common to all subgroups from the original group. Another simple case is that no subgroups attempt to merge. Use of the null property ensures that no subgroups attempt to merge with other subgroups. The identity of certain key members of the original group may be used as the property, also. Partitioned subgroups attempt to merge with the subgroup containing these key members.

By combining these two rules, a wide variety of partition resolution methods can be produced. The first rule determines which partitions survive, and the second rule determines which partitions attempt to merge. Each rule can also combine multiple parameters to provide very specific and flexible methods of handling partitions. For example, all subgroups larger than a size of three which contain a particular member type will survive and attempt to merge with subgroups with the same base group name containing another particular member type.

V. CORRECTNESS ARGUMENTS

In the previous chapters the architecture and protocol descriptions for a global, decentralized membership service were presented. In this chapter arguments and proofs are presented to show that the MS protocol performs correctly under all circumstances. The correct performance of the MS protocol leads to achievement of the desired attributes of the MS, as discussed in Chapter II. The arguments presented here focus on the functioning of a single core-set of mservers, treating the set as a group in itself, with the individual mservers in the core-set as members of the group. The proofs show that changes to the membership of this group are made in a manner which always maintains strong consistency of the membership information at all members of the core-set. The arguments about the correct operation of a single core-set of mservers can then be extended to the physical hierarchy of core-sets of mservers, and then to the application groups which utilize the MS, showing that consistent membership information is always obtained at all application process group members.

The assumptions and definition of terms used in the proofs are listed first, with their specific implications with respect to the correctness arguments described. These are followed by a description of the criteria for correctness and a summary of the actions that the protocol takes to maintain the membership knowledge accordingly. Finally, key statements about different aspects of the protocol are proven, thus proving the correctness of the MS protocol.

A. ASSUMPTIONS

As described previously, an asynchronous communication environment is assumed to exist, providing an unreliable message delivery capability with an unbounded delay, as in the present *best-effort* Internet. Thus, network failures that include dropped messages and network partitions are permitted. All member failures are assumed to be crash or fail-stop [5, 9, 10, 11]. In such conditions, failures can only be perceived, and both actual member failures and network partitions lead to perceived failures of the members. For this reason, every perceived failure is processed as an event that partitions the group. Partitions of the membership of a group are assumed to be acceptable to the user of the membership service, who may make QoS selections to determine how partitioned groups will continue to function, as described in earlier chapters. Unlike many other membership protocols, majority-based decisions are not used by the MS protocol to ensure that only a single partition survives; instead, complete agreement is required among all surviving members, leading to the possibility of separate, functioning partitions of any size. Continuous changes to the membership are allowed; however, the changes are committed one at a time, and with a specific order in each partition.

B. TERMS AND DEFINITIONS

The specific terms and implications of their use in the correctness arguments described later are listed below.

1. Change Events

The events that cause a change in the membership are: explicit join and leave requests by members, perception of failure by the monitoring of members by other members, and suspicion of failures resulting from member or network failures which lead to a lack of response during change processing.

2. Change Event Priority

Every change event has an associated priority to enable ordering of virtually simultaneous changes. Failures have a higher priority than voluntary joins or departures. Priority of a failure or departure event is the rank, or seniority, of the failed member in the group. The most senior member always has a rank of 0. When two or more members initiate a change simultaneously, the coordinator initiating the higher priority change, as determined by the rank of the subject of the change, prevails. In virtually simultaneous joins, the subjects do not yet have a group rank, so the network address of each subject is used in place of the rank. The subject with the lower network address will be interpreted as having a higher temporary rank, and therefore will have a higher priority, joining the group first.

3. Isolation

A member that perceives another member as faulty ceases all communication with that faulty member. This leads to the member perceived as faulty also determining that the other member is faulty, since no communications are received.

4. Gossip

A member that isolates another member gossips about the isolation in the subsequent communication it has with every other group member. Thus, in the absence of any other failures, a multicast following an isolation leads to the whole group isolating the member that was perceived faulty by the sender of the multicast.

5. Group View

This term denotes the ordered membership list maintained by each member m_i, and is denoted as View$_x$(m_i), where x denotes the view number.

a. Definition

The group view at a member is the set of members that are believed to be part of the group. It is ordered with respect to the seniority of members in the group and has an integer, called a view number, associated with it.

b. Remarks

Every membership change alters the number of members in the view at a member and leads to the installation of a new view identified by the next higher view number. The number of members in the group may change by more than one in a single view change. The rank of a member denotes its seniority in the group, with the most senior member having rank 0. Identical views imply identical membership as well as ranks.

6. Group Partition

Let G denote the set of all possible potential and current members of a group. A *partition* P of G is defined below.

a. *Definition*

P is a subset of the all members' set G, such that $\forall\ m_i, m_j \in P$, if $\text{View}_x(m_i)$ and $\text{View}_x(m_j)$ are defined, then $\forall m_k \in \text{View}_x(m_i)$: $m_k \in \text{View}_x(m_i) \Leftrightarrow m_k \in \text{View}_x(m_j)$, and all members have the same rank in the two views.

b. *Remarks*

The view associated with partition P is denoted View_P, and the partition containing m_i is denoted $\text{P}(m_i)$. Thus, all members in a partition must have identical views. However, it is possible that there exists an m_k outside a partition, but still in every member's view for a particular partition. Such partitions are called *unstable partitions*. The MS protocol treats such a partition as legal, and eventually removes m_k from the views of all members of the partition. When no such m_k exists for a partition, the partition is called *stable*. Network and member failures lead to the creation of group partitions in asynchronous environments.

7. Group Membership Protocol

Using the definitions of the terms above, a protocol is defined to solve the Group Membership Problem (GMP) as below.

a. *Definition*

A protocol solves the GMP correctly if <u>every</u> change event results in <u>group partitions</u> <u>eventually</u>.

b. Remarks

The above definition of a correct solution of the GMP requires it to satisfy distinct properties corresponding to the underlined conditions in the definition above.

- **E1** This property, arising from the condition of <u>every</u>, requires that a change event observed by a member is processed despite other virtually simultaneous change events and failures during protocol execution, including that of the coordinator. The only situation in which a change event is not processed is in case of catastrophic occurrences in which all the members with knowledge of the change event suffer real failures.

- **E2** This property, arising from the condition of <u>eventually</u>, permits the processing of a change event to be suspended temporarily; however, it requires that the resulting view is always installed at all members of the partition before the change event occurred after only a finite number of changes are allowed to take place.

- **GP** This property, arising from the condition of <u>group partitions</u>, implies identical views at all members of each partition. As per the protocol described, all partitions resulting from change processing always become stable.

Requirements imposed by the **E1** and **E2** properties satisfy the condition commonly known as *liveness* in distributed systems and those imposed by the **GP** property satisfy *safety* [5, 27, 28]. Thus, the uniqueness of views and identical ordering of changes at all operational members is guaranteed by **GP**.

C. REMARKS ON THE PROTOCOL STRUCTURE

The previous chapter described, in detail, how the protocol handles various change events. The functions of the different components of the protocol are summarized in the following paragraph. Unless specified otherwise, the term failure is assumed to imply a perceived member failure that may have been caused by either a network failure or a member's failure.

Any of the members may initiate a change when it perceives a change according to the change events described earlier. The change initiator is called the coordinator for that change and carries out the basic membership change protocol listed in Figure 17. The normal two-phase change processing is illustrated in Figure 16. The first phase consists of a multicast of the *Initiate* message to all the members followed by collection of *ACKs* from all members. As specified in Figure 20, the coordinator collects *ACKs* from all members it believes to be in the group while, at the same time, trying repeatedly to send the *Initiate* message to those that it believes to be present but from whom a response is not forthcoming due to a failure. The second phase consists of multicasting the *Commit* message. The members that do not send an *ACK* are isolated and gossiped about during the commit phase.

The non-coordinator's actions of Figure 18 consist of sending the *ACK* message and committing the change. Once the *Initiate* message is received, the receiving mserver prompts the coordinator repeatedly if a *Commit* message is not received, as specified in Figure 15. If a *Commit* message is not received due to a failure, the mserver expecting the message starts a broadcast election. As specified in Figure 30, all of the members that have received but not yet committed the incomplete change elect the highest rank member as the coordinator. The elected coordinator then resumes processing of this change as specified in Figure 32. If the coordinator failure was initiated before any member could commit the change, it is resumed with an *Initiate* multicast by the elected coordinator. If at least one member that participates in the election had committed the change, then the newly elected coordinator resumes the change by sending a *Commit* message.

Due to the possibility of other changes occurring during a change processing, both the coordinator and non-coordinators must take additional actions as specified in Figures 25 and 24, respectively. In Figure 25, the specification of Figure 20 is augmented to permit the coordinator to handle messages in addition to the *ACKs* for the initiated change. Depending upon the message received by the coordinator as it collects the *ACKs*, it switches to a higher priority change, enters an election, or delays the change it is coordinating due to a previous change that may not yet have completed.

Similarly, Figure 24 is the augmented version of Figure 15 to handle situations in which the non-coordinator does not get the expected *Commit* or *Direct* message. The additional actions permit the non-coordinator to either switch to a higher priority change, start an election if the coordinator has failed, or delay another coordinator that attempts to install the next view change.

D. CORRECTNESS ARGUMENTS

Based on the protocol summary above and the detailed description given in the previous chapter, a proof is presented that shows that the MS protocol has all the properties as identified above for a correct solution to the GMP. Also shown is that a more refined solution to the GMP defined earlier by Ricciardi and Birman [9] is possible.

1. Claim 1

Change event processing always completes at both the coordinator and the non-coordinator except when all members, including the coordinator, with knowledge of the change fail.

2. Proof

Consider a change event *change(subject, coordinator)* initiated in *P*.

a. At the coordinator

Although the coordinator makes multiple attempts to deliver the *Initiate* message to all perceived members of *P*, it does not require a predetermined number of them to respond before it sends a *Commit* message (line 5, Figure 17). If the coordinator switches to a higher priority change before it sends a *Commit* message, the information about the old change is saved. The old change is reinitiated after all higher priority changes complete.

b. At the non-coordinator

If the coordinator fails, at least one member times out on the *Commit* message and starts an election (line 7, Figure 18). The highest rank member with the change active is elected to resume the change (line 6, Figure 30). The fact that the election is conducted among those with knowledge of the change ensures that the change completes even if the coordinator and the only members to have committed the change fail. This takes care of the invisible commits described by Ricciardi and Birman [9].

3. Claim 2

In any partition, either only one change event proceeds to the commit phase, or members reaching the commit phase for different change events form separate partitions.

4. Proof

Initially, all members have identical views of the membership (definition of a partition). In the set of all potential change events, there exists a unique priority order due to the uniqueness of ranks, which order failures and departures, and network-level addresses, which order joins. This permits every member receiving multiple *Initiate* messages before receiving any *Commit* message to switch to the highest priority change that will install the next view. Overlapping of *Initiate* messages to install successive views with different view numbers is not possible (line 6, Figure 25).

Suppose a member receives a *Commit* message for the current change that will change the view number from x to $x+1$. Suppose this mserver then receives a higher priority change that also corresponds to a view number change from x to $x+1$. It is guaranteed that the sender of the higher priority change appears in the gossip accompanying the received *Commit* message. This happens because the coordinator of the lower priority change will have timed out on the coordinator for the higher priority change and isolated it before generating a *Commit* message (line 6, Figure 17). This ensures further partitioning if more than one change events proceed to the commit phase.

5. Claim 3

If the coordinator fails after sending the commit message, the two-phase protocol consisting of an election followed by a commit can solve the group membership problem correctly.

6. Proof

Begin by proving the contrapositive statement:

The two-phase protocol consisting of an election followed by a commit cannot solve the GMP correctly if the coordinator fails before sending the commit.

If the coordinator fails before sending the *Commit* message, it is possible that one of the members has not yet received the *Initiate* message for the change. This member would respond in the election with a *Coord-Fail* message that announces that it is not aware of the change for which the election has been started. This member must receive an *Initiate* message before it can commit the change for which the coordinator failed. If the *Coord-Fail* message is used to start the change in place of a separate *Initiate* message, and only a *Commit* message is sent to complete the change, then the **GP** property can be violated, as shown in the example below.

Consider a partition consisting of members m_i, m_j, m_k, C_a, and C_b. Let C_a initiate change "*a*" by multicasting *Initiate*$_a$, which is received only by m_i due to network failures. C_a fails immediately after sending *Initiate*$_a$, and this failure is perceived by m_i, which then starts an election by multicasting *Coord_Fail*$_a$. m_j and m_k participate in the election, but C_b does not because it has failed. However, before failing, C_b starts another higher priority change by multicasting *Initiate*$_b$, which reaches only m_j due to network failures. Since change "*b*" is a higher priority change, m_j drops change "*a*" as the current change. At this point, m_j perceive C_b failed and starts an election by multicasting *Coord_Fail*$_b$.

Throughout this time, m_i waits to hear C_b's response to the election for change "*a*", which will not arrive due to C_b's failure before *Coord_Fail*$_a$ reaches it. Eventually, m_i times out in the election, determines that it must be the winner, and assumes the responsibility for completing change "*a*". m_i commits change "*a*" and multicasts *Commit*$_a$ to the group with gossip about C_b's isolation. If the *Commit*$_a$ reaches m_j and m_k after they have switched to change "*b*" due to the *Coord_Fail*$_b$ message, they will quietly discard the *Commit*$_a$ message due to its lower priority. Thus, m_i will have committed change "*a*", whereas m_j and m_k will never commit it. This inconsistency violates the **GP** property and makes the two-phase protocol incorrect. Thus, the contrapositive statement is proved.

The contrapositive statement proves Claim 3 above. It should be noted that the failure of the coordinator after sending the *Commit* message with simultaneous failures of all members that receive the *Commit* message is equivalent to the coordinator failing before sending the *Commit* message. It is not possible to differentiate between these two situations, thus the change must be completed in three phases. In the protocol described in this thesis, the three phases are the broadcast election, initiate, and commit phases. Thus, the Resume_change procedure of Figure 32 requires the elected coordinator to complete the change with a *Commit* message if some member that had committed the change participates in the election, permitting a two-phase processing of the coordinator failure. Otherwise, the elected coordinator simply reinitiates the change, providing a three-phase processing of the original change.

7. Theorem

The proposed group membership protocol is safe and live.

8. Proof

The *liveness* properties follow directly from Claim 1. The s*afety* property follows from Claim 2 and 3.

VI. MEMBERSHIP SERVICE IMPLEMENTATION

This chapter describes a partial implementation of the MS specified in previous chapters on a campus-wide set of LANs with UNIX-based workstations. The use and limitations of the IP multicast capability are described, as well as the needs of the MS not met by the IP multicast capability. To meet some of these unfulfilled multicasting needs, a multicast emulation program, called *mcaster*, was developed. The design and implementation of this program are described. A complete set of utility functions for use by the *mcaster* and MS programs were developed, and are described in detail. High-level descriptions of the algorithms used to implement mservers and MIs are presented. A working implementation of the shell of the *mserver* program is also presented. The software code for the *mcaster* program, the utility functions, and the *mserver* shell program are listed in the Appendix to this thesis.

A. MULTICASTING

The use of multicast message delivery is essential to the efficient and scalable operation of an MS. In this section the general concept of multicast message delivery is explained. Two implementations of multicast facilities are described: the IP multicast and a specially written multicast emulation program, called *mcaster*.

1. IP Multicast

A recent addition to the IP suite of services is the IP multicast capability. A multicast is the multipoint delivery of a single datagram, originated by a single sender and delivered to multiple destinations which are part of a predesignated multicast group. This is in contrast to a broadcast, which is a multipoint delivery of a single datagram to all connected machines, without any capability to limit the scope of the delivery, and a unicast, which is a point-to-point datagram delivery. In effect, a multicast is the generalized form of message delivery, providing broadcasts at one extreme and unicasts at the other [29]. Previously, the capability to multicast efficiently was limited to single LANs, using the LAN hardware protocol. IP multicast provides a similar capability for machines connected over the Internet, allowing the efficient multicast of a single datagram to multiple receiving machines which are included in the multicast group, as shown in Figure 48.

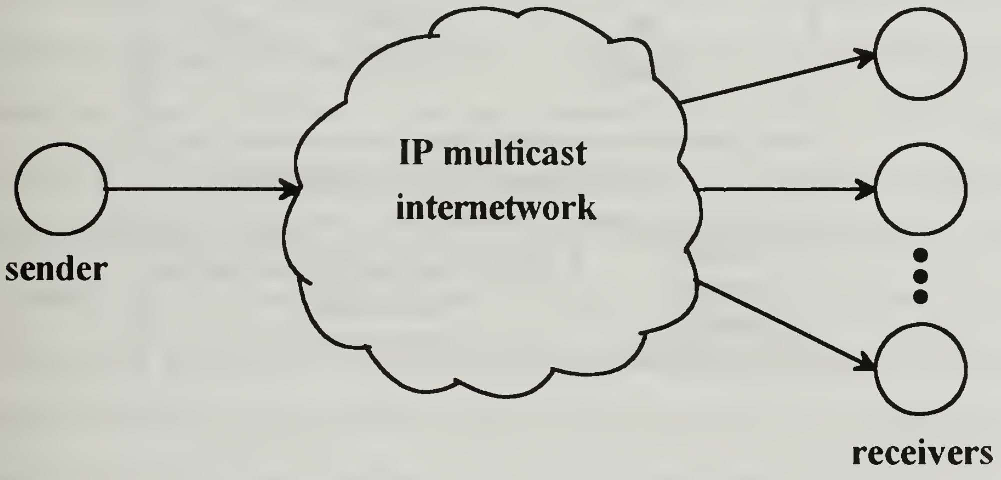


Figure 48: IP Multicast

a. IP Multicast Extensions

Full utilization of the new IP multicasting feature requires an extension to the currently installed IP implementation on each host machine. The document which describes how this extension is accomplished [6] defines three levels of conformance to the specification: Level 0, with no support provided for IP multicast (the current configuration for most machines), Level 1 which provides limited support for sending multicasts but not for receiving multicasts, and Level 2, which provides full IP multicast support. Level 2 requires the implementation of the Internet Group Management Protocol (IGMP), which manages the dynamic multicast groups which a host must join to receive multicast datagrams. A depiction of the layered model for IP multicast is shown in Figure 49, provided by reference [6].

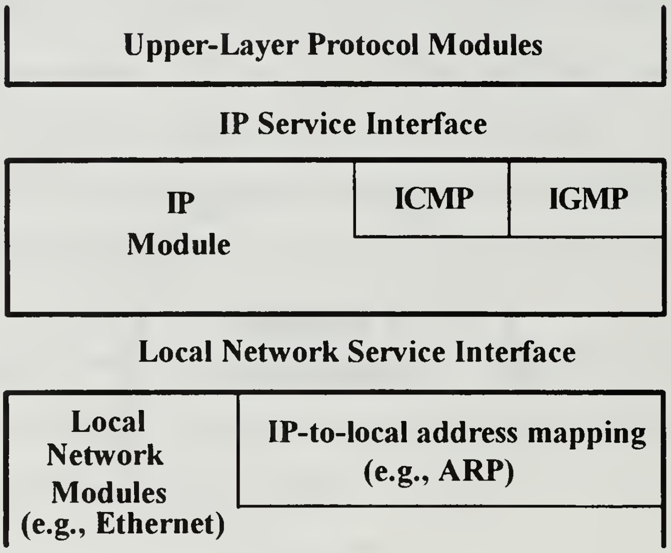

Upper-Layer Protocol Modules
IP Service Interface
IP Module
ICMP
IGMP
Local Network Service Interface
Local Network Modules (e.g., Ethernet)
IP-to-local address mapping (e.g., ARP)

Figure 49: IP Multicast Layered Model

Full use of IP multicast of datagrams requires that hosts join a dynamic multicast group. This group is identified exclusively and uniquely by the 32-bit IP address used to transmit a datagram to the group. A set of IP addresses has been reserved specifically for IP multicast. These are referred to as class D IP addresses, with the first four bits of the address set to '1110' [6]. The range of these class D addresses is from 224.0.0.0 to 239.255.255.255, using the common dotted decimal notation to specify IP addresses. Addresses between 224.0.0.0 and 224.0.0.255 inclusive are reserved for multicast routing and maintenance protocols [7], but all others class D addresses are available for use, providing a total multicast address space of over 268 million addresses. A few of these addresses are permanently assigned, but most are available for transient multicast groups. Additionally, the IP multicast specification provides a time-to-live (*ttl*) variable associated with each multicast, controlling the transmission scope of any multicast datagram. With judicious use of the *ttl* variable, it is possible to use virtually any class D address for a given host group without worrying about prior assignment of that class D address.

As described earlier, full level 2 conformance requires implementation of the IGMP to manage these multicast groups. As shown in Figure 49, IGMP is an integral part of the IP protocol layer when implemented at a host or gateway. IGMP controls the relationship between a multicast router and a set of host machines participating in a multicast group. Multicast routers and host machines use IP datagrams to communicate status back and forth, similar to the Internet Control Message Protocol (ICMP), which is used to report errors and provide information about unexpected circumstances between gateways and host machines [29]. IGMP provides a mechanism for hosts to dynamically join and leave multicast groups, and for local multicast gateways to monitor the group membership as well as provide correct routing of multicast datagrams. Hosts and local gateways use IP multicast datagrams for all IGMP communications, using the "all hosts" reserved multicast address of 224.0.0.1, to conduct very efficient communication [6]. The local gateway maintains status tables to record local group membership of hosts. It also periodically polls all connected hosts to determine if they are still part of the specified groups. In this manner, a very efficient management of IP multicast groups is performed.

b. IP Multicast Implementation

The most common implementation of multicast applications involves the use of the Berkeley sockets abstraction provided in most UNIX environments for network I/O. Sockets are a generalization of a UNIX file object, and provide an endpoint for communications [29]. There are normally three types of communication used for various applications: reliable stream delivery, using SOCK_STREAM type of socket, connectionless datagram delivery, using a SOCK_DGRAM type socket, and a raw type of communication, using the SOCK_RAW type socket. IP multicast supports only the SOCK_DGRAM and SOCK_RAW types of sockets, and provides no support for connected sockets. Additionally, there are several types of system calls for sending and receiving datagrams, most of which are similar to the system calls for UNIX file I/O. IP multicast supports only the *sendto*, *sendmsg*, *recvfrom*, and *recvmsg* system calls for datagram transmission and reception [7]. The *sendto* and *sendmsg* datagram transmission calls require the destination (multicast or unicast) address as an input parameter. The *recvfrom* and *recvmsg* system calls extract the sender's address from the header of the incoming datagram. Together, these calls provide a very efficient means of combined unicast and multicast network communications, since the only difference between communicating with a single host or a multicast group is the address used, and this address is readily extracted in exactly the proper format to send a reply to the sender for either a multicast or unicast transmission. The format of the IP address is contained in a C programming language structure, called sockaddr_in, as shown in Figure 50, containing the address family, port number and IP address for the particular host.

Address Family	**Protocol Port**
IP address	
Unused (0)	
Unused (0)	

Figure 50: IP Socket Address Structure (Sockaddr_in)

2. *Mcaster* program

IP multicasting is a relatively new innovation, and is not widely available at this time. Due to the very limited implementation of level 2 conformance to the IP multicast specification on most current computer networks, it was decided to develop a program that would emulate the IP multicast capability for the currently available unicast environment. The goal was to develop a program that would emulate the services provided by IP multicast as transparently as possible; hopefully to the extent that a user or application program would not need to be concerned with which environment was actually being used. This involved simulating all of the functionality provided by IGMP at the host and gateway level.

a. Mcaster Design Decisions

The overall scheme chosen for the IP multicast emulator, called *mcaster*, was to have a "daemon" process running at a well-known site, which would act as an intermediary between the members of a multicast group, providing essentially the same services as those provided by IGMP, such as controlling members joining and leaving groups, and the routing of multicast datagrams to all members of a particular group. The primary difference between an IP multicast gateway using IGMP and the *mcaster* program is that *mcaster* enjoys none of the hardware support that a router would include - especially the ability to send a datagram over multiple interfaces at once. The *mcaster* program would be running on a standard host computer, probably using a single interface to the internetwork. This limitation is the most significant difference between an IP multicast router and an *mcaster* host computer; whereas a router can send the same datagram to multiple recipients simultaneously over multiple network interfaces, the *mcaster* must iteratively send the datagram over one interface, causing a significant performance degradation over IP multicast. However, the primary goal of the *mcaster* emulator was to provide the capability of multicasting, not to match the performance possible through hardware supported multicasting.

The primary reason for developing the *mcaster* program was to provide a multicast capability for use by the membership service under development in environments which did not support IP multicast. For this reason, the message format used by the *mcaster* program was chosen to correspond as closely as possible to the expected needs of the membership service that it would support. The basic message format for the *mcaster* program was designed to also be the basic message format for the MS. This message format was previously described in Figure 13 of Chapter IV. Special message types are reserved for *mcaster* control messages. Although the *mcaster* program was developed to support the MS, it also provides a general multicast capability for any program or user. The only requirement for the use of the *mcaster* program is that messages sent by the application program using *mcaster* must include a header structure in the format described above. The *mcaster* program will then be able to deliver messages of any type to a designated multicast group.

To make the *mcaster* daemon as capable as possible, it was decided to permit each *mcaster* daemon to support any number of separate groups, each with an unlimited number of members. The primary data structure chosen to store state information for all groups supported by an *mcaster* was a list of groups, each with a list of members, as shown in Figure 51. Groups and their members are dynamically added to and removed from the lists as needed.

A host computer desiring to join an *mcaster* multicast group simply formats a message with the JOIN_GROUP message type and sends it to the well-known IP address of the *mcaster*. The *mcaster* processes the join request and responds with a similar message indicating success or failure of the join request. Leaving an *mcaster* multicast group is done in exactly the same manner, with the message type set to LEAVE_GROUP. Any message received by the *mcaster* which is not a join or leave request is considered to be a message to multicast to the group, and is iteratively sent to each member of the indicated group using the *sendto* socket system call. Whereas IP multicast groups are exclusively and uniquely identified by their class D IP address, *mcaster* multicast groups are identified by the combination of a group name and an *mcaster* IP address.

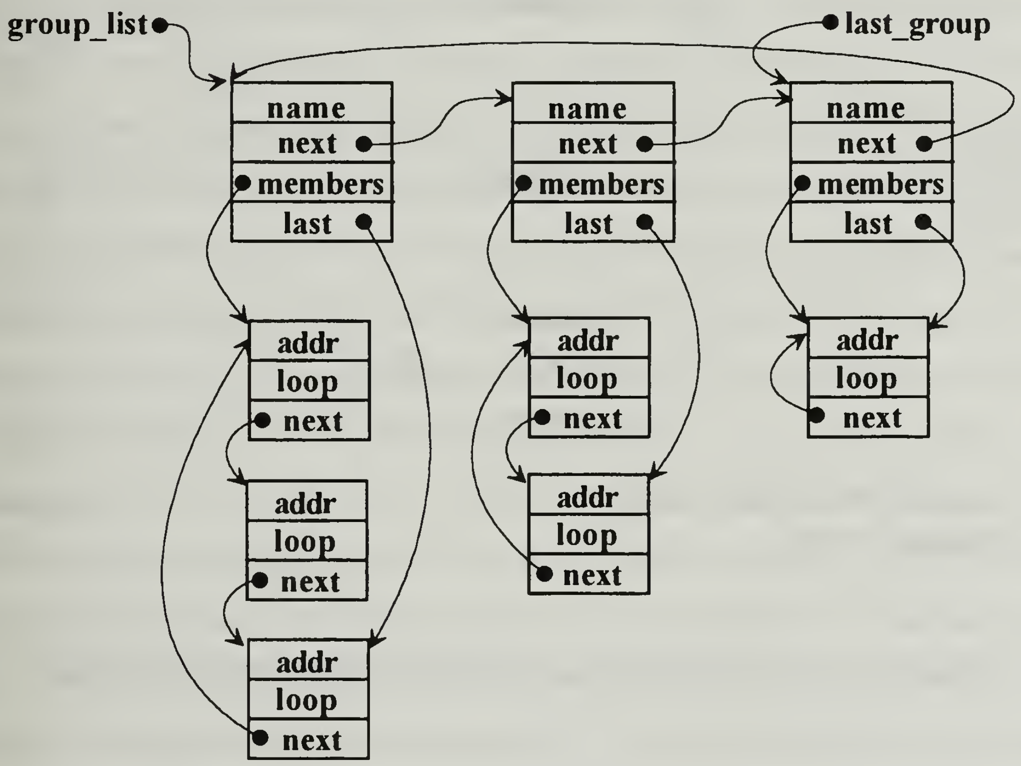


Figure 51: *Mcaster* Data Structures

b. Differences from IP Multicast

Originally, it was hoped that the use of the *mcaster* multicast emulator would be completely transparent to a user or application program; that is, exactly the same system calls would be made with nearly identical arguments for either multicast environment, with identical results, in a manner similar to that shown in Figure 52. It was soon realized that there were several deviations from the desired transparency that would be necessary to make the *mcaster* program as capable as desired.

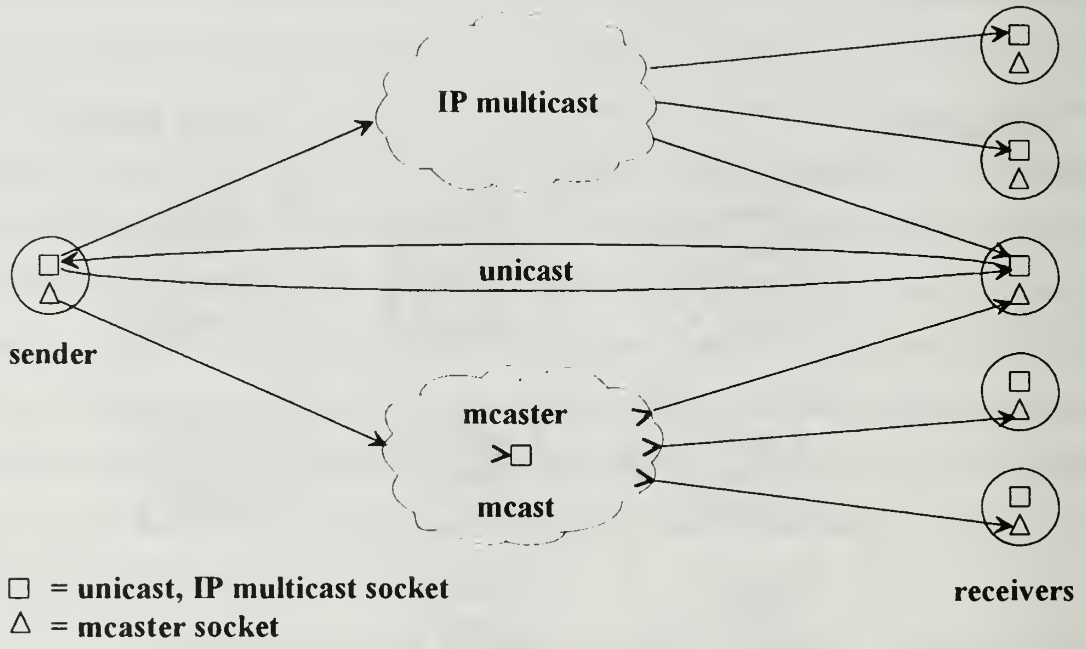


Figure 52: IP Unicast, Multicast, and *Mcaster* Using Separate Sockets

The first deviation had to do with the ability of an *mcaster* to manage more than one group. Whereas IP multicast groups are exclusively and uniquely identified by their class D IP address, *mcaster* multicast groups are identified by the combination of a group name and an *mcaster* IP address. Since an *mcaster* is a daemon process running on a specific host machine with a unique IP address, all of the groups managed by that *mcaster* must share the same group IP address of the host machine. This is in contrast to IP multicast groups, which may share a common local IP multicast router, but each still have unique IP addresses. The only implication of this deviation is that the group name had to be included in the message itself, so that the *mcaster* could extract the group name and reference the desired group. With IP multicasts, the group name would not be required, since the identity of the group is implicit in the unique group address.

The second deviation from the desired transparency between IP multicast and *mcaster* multicast had to do with the procedure for joining and leaving groups. This deviation was inherently necessary due to the fact that *mcaster* emulates the functionality of IGMP, so a mechanism had to be created to perform the same functions. IP multicast uses the *setsockopt* system call to make a socket multicast capable. The sockaddr_in address structure bound to the socket is first loaded with the class D address of the group. The *setsockopt* call is then made with the IP_ADD_MEMBERSHIP option set [7]. If the address used is a legitimate class D address, then IGMP adds the calling host to the specified multicast group. Hosts leaving a multicast group perform the same routine, with the *setsockopt* option set to IP_DROP_MEMBERSHIP. As described earlier, a host desiring to join or leave an *mcaster* group would simply format a message with the appropriate message type and send it to the host running the *mcaster*. The functionality required to join either an IP multicast or *mcaster* group can easily be encapsulated within a single procedure, perhaps in a library file, giving the desired transparency between the two methods of multicasting at the procedure call level. The same holds true for the procedure to leave either type of group.

A third deviation between the two types of multicasting did not directly affect the transparency, but could have adverse effects on the performance of the *mcaster* program. Unlike IGMP, once a host joined an *mcaster* group, no monitoring of group members is performed. The purpose of this monitoring in IGMP is to detect members no longer participating in the group and drop them from the membership. It was decided that this was unnecessary for the *mcaster*; the lack of a monitoring capability did not directly affect the ability to multicast nor the desired transparency between the two types of multicast, since the user would normally not be aware the monitoring was taking place at all. Instead, it was left to the application program to correctly leave an *mcaster* group. Failure to properly leave an *mcaster* group would burden the *mcaster* daemon with sending extra messages to hosts no longer participating in the group, increasing the time required to multicast to other legitimate hosts in the group, as well as the overhead required to store the state of members no longer participating in the group. The functionality required to monitor the status of group members, to detect non-participation, and to remove non-participating members could be added to the *mcaster* program at some future time if desired, but would likely affect the transparency of the *mcaster* program to application programs.

The final deviation in the transparency between the two types of multicasting was the most significant. Due to the sender's IP address being included in the datagram header, the receiving host can easily extract the sender's address using the *recvfrom* system call. Normally this is a very desirable trait, useful for quick and easy replies to the sender of a datagram. The problem encountered was that the *mcaster* program acts as an intermediary for all multicasts between group members, extracting the group name from the message to reference the proper group, then sends the original message to all members. In so doing, the sender's address in the datagram header is changed to the *mcaster* host instead of the original sender. It was therefore no longer possible for a receiving host to extract the original sender's address from the datagram header; instead only the *mcaster's* address could be recovered. To remedy the inability of a receiving host to determine the original sender of an *mcaster* multicast, it was required to prepend the original sender's address to a normal message structure before it was encapsulated in an IP datagram and sent to all members. An illustration of the extended message format is shown in Figure 53. There were two choices as to how to handle the extended format message at the receiving hosts. The first choice was to check every message and decide whether it was a normal or extended format message, and process it accordingly. The

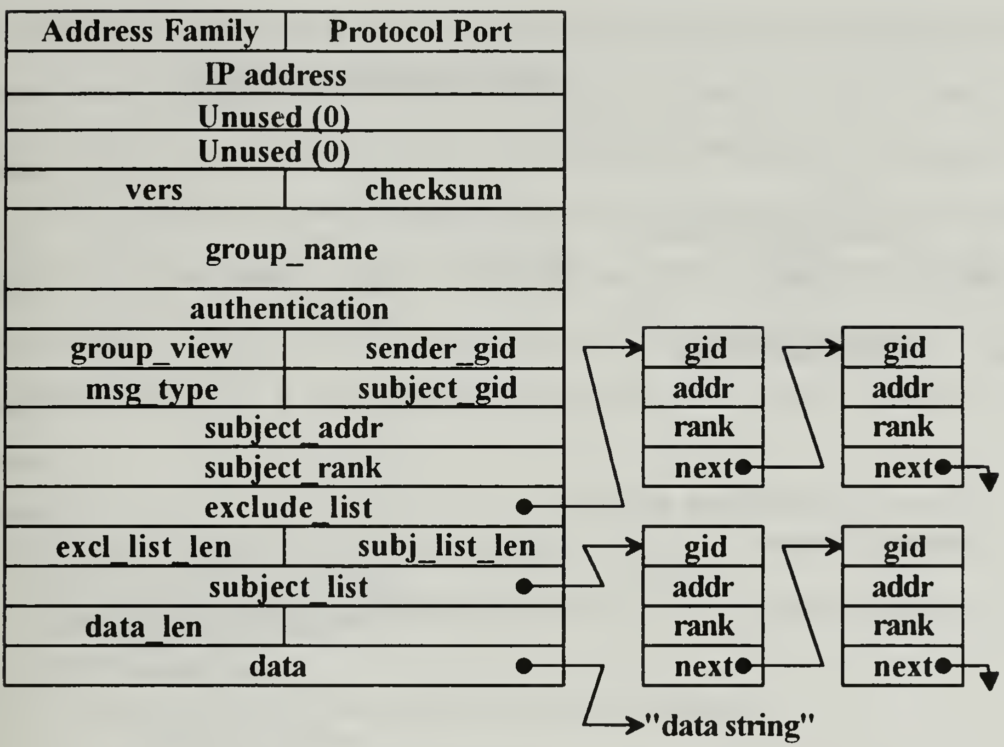


Figure 53: Extended Format *Mcaster* Message Structure

second choice was to have two separate receiving sockets: one for normal messages and one for the extended format messages from an *mcaster* multicast. The latter method was chosen for the simplicity and clean separation it provided between the two multicasting methods, as shown in Figure 54. The drawback to the chosen method was that an application program using an *mcaster* multicast would have to manage an extra socket at all levels of the program, virtually eliminating the desired transparency. However, the amount of overhead required to manage the extra socket is insignificant, and the use of the extra receiving socket could easily be hidden in a separate receive routine in a library file, similar to the join and leave procedures used to hide the access to the two multicasting methods.

The deviations noted above prevent the user from being totally unaware of which method of multicasting is being used: an IP multicast or an *mcaster* multicast.

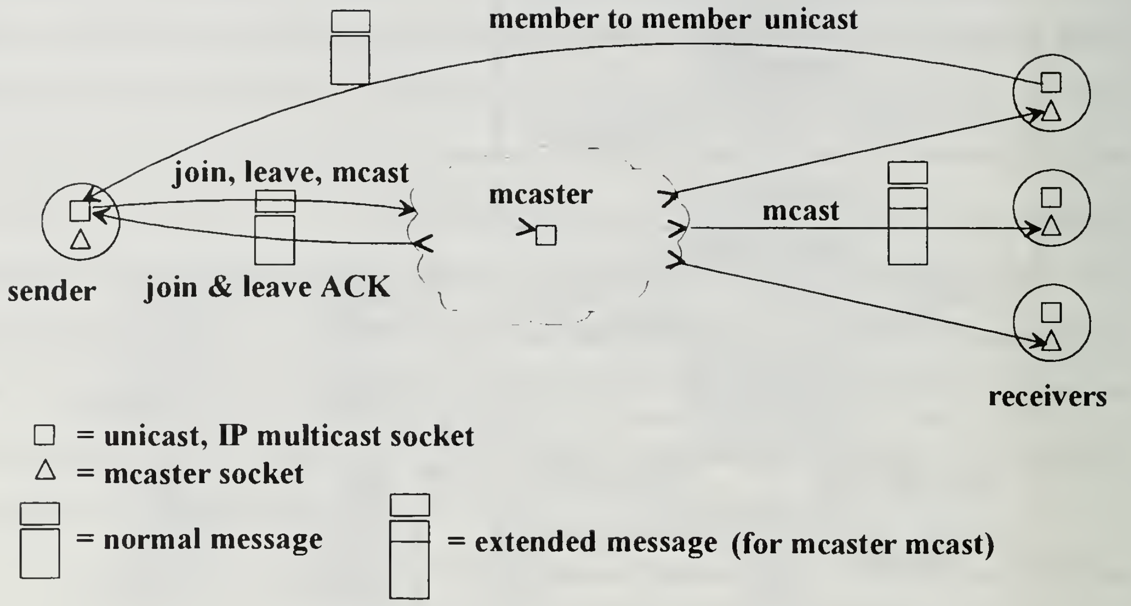


Figure 54: Multicasting Using Extended Format *Mcaster* Messages

However, it would be impossible to completely remove the awareness of the multicasting method used, since an IP multicast only works within a limited range of IP addresses, and the user would have to select the proper IP address to use if intent on using IP multicasting. The deviations from IP multicasting listed above required by the *mcaster* program would not be evident in the normal multicasting of IP datagrams; the user could confidently select an IP address and name for the multicast group and then use the library calls described to join the group, send and receive multicast and unicast messages, and leave the group, without ever being aware of which method of multicasting was being used. Thus, the desired level of transparency in multicasting methods was achieved.

c. *Mcaster Algorithm*

The algorithm for the *mcaster* program is listed in Figure 55 and described as follows. Line 1 is the initialization of the single socket used by the *mcaster* program for all I/O. Line 2 begins the main loop of the program, an infinite loop of waiting to receive a message, then processing the received message and sending a reply or multicast as necessary. Lines 3 and 4 describe the process of blocking to receive an incoming message. Lines 4 and 5 check the received message type for a join request. Lines 5 through 12 perform the join_group sequence. In line 6, the group list is searched to determine if the group already exists or not. If the group is not located in the group list, lines 7 and 8 add the new group to the list. If the group does exist, then lines 9 and 10 determine if the

Mcaster

```
/* Emulates IP Multicast using iterative unicasts */
1.  initialize socket (group address)
2.  wait for incoming messages
3.  when message received
4.      if (messagetype == JOIN_GROUP or LEAVE_GROUP)
5.          if (JOIN_GROUP)
6.              search group list for group (group name)
7.              if (group not located)
8.                  add group to group list
9.              else  /* group located */
10.                 search member list for member (member address)
11.             if (not already a member)
12.                 add member to member list
13.         else  /* LEAVE_GROUP */
14.             locate group (group name) or indicate error
15.             locate member (member address) or indicate error
16.             if (located)
17.                 remove member from member list
18.             if (member list is empty)
19.                 remove group from group list
20.         form ACK message
21.         send ACK message to requesting member
22.     else  /* multicast to all members */
23.         for (all members in specified group)
24.             if (not sender or loopback)
25.                 send message to member
26. goto line 2
```

Figure 55: *Mcaster* Algorithm

member is already in the member list of that group. If it is a new group or if the member is not already in the member list, then the member is added to the member list of the specified group in lines 11 and 12. Lines 13 through 19 perform the similar procedure for leaving a group. Line 14 and 15 locate the specified member. Line 16 and 17 remove the member from the member list. If the member list for the specified group is now empty, lines 18 and 19 remove the group from the group list. Lines 20 and 21 complete the join or leave sequence by forming and sending an acknowledgment message to the requesting member. Lines 22 through 25 perform the multicast of any message other than a join or leave request. Line 24 ensures that the sender does not receive the multicast message if the no loopback option is selected. Line 26 completes the main loop and returns to line 2 to begin again.

The actual code for the *mcaster* program is included in the Appendix in the program file *mcaster.c*. The utility functions used by the *mcaster* program are included in the library files *mcaster.h*, *msutil.h*, and *msutil.c*, also included in the Appendix.

B. MSERVER

The functioning of an mserver process has already been explained from a procedural point of view. The monitoring and change-processing protocols defined in Chapter IV each explain the sequence of actions performed by an mserver with respect to one aspect of the overall operation of the MS and an mserver. The protocols are described in a procedural form, implying that an mserver performs the complete set of actions that make up each protocol sequentially before beginning a new set of actions. In reality, each mserver must continually process incoming messages and changes to the internal state of the mserver concurrently. It is true that for strong membership consistency guarantees, only one change will be committed by a core-set of mservers at a time; however, during the processing of that change, many other events must be registered and processed. These other events include the reception of messages of all types: some that affect the current change; others that do not, but must be stored nonetheless; and some that require an immediate response, such as a monitoring query. Other events include the expiration of timers or a change in the internal state caused by processing the current change.

Simply put, an mserver process performs three primary actions: 1) it receives and stores incoming messages, 2) it changes the internal state in response to internal events or incoming messages, and 3) it sends outgoing messages. The incoming and outgoing messages may be unicast or multicast, depending on the circumstances. In this section, the operation of an mserver process is described in detail from an implementation viewpoint.

1. Internal State and Data Structures

Each mserver process has a dual personality: it is a member of a core-set of peer mservers, as well as the parent of a child-set of mservers. For this reason, the set of data structures and variables used to maintain the internal state of an mserver must be replicated to support both identities. Additionally, each mserver must maintain information about all application groups that it supports. Figure 56 illustrates the data structures and variables used to maintain the internal state of an mserver. Each of these data structures will be described in detail in the following paragraphs. Table 6 lists the variables used to maintain the mserver's internal state and their meaning.

As shown in Figure 56, an mserver maintains two core-tables; one for the peer core-set, and one for the child-set. The core-tables are used to maintain the membership information for each the mserver's core-sets. The index into the table is the gid of each mserver in the core-set. The IP address of each mserver is stored to allow unicast message addressing. The rank of each mserver is maintained, with a rank of 0 corresponding to the highest rank and most senior mserver in the core-set. The *cw* and *ccw* variables are integer pointers representing the clockwise and counterclockwise neighbors of each mserver. These links represent the pairwise monitoring-set; the clockwise neighbor is the monitor and the counterclockwise neighbor is the monitored mserver. It is important that all mservers know the exact monitoring relationship of all

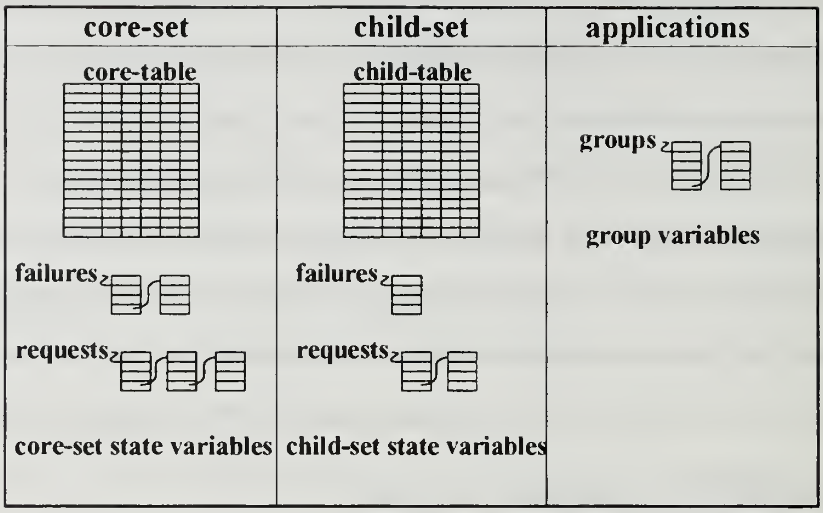


Figure 56: Mserver Data Structures and Internal State

other core-set mservers, so that the correct monitoring arrangements can be made by all each time the core-set membership changes. Figure 57 illustrates the structure of these core-tables, and Figure 58 illustrates a core-set of mservers corresponding to the core-set listed in the core-table of Figure 57.

Each mserver maintains four lists for each side of internal state: an mserver failures list, a physical change requests list, an application group change requests list, and a list of all application groups supported by that core-set. The failures list is a list of all mservers that have been detected failed by this mserver, but not yet processed out of the core-set. The format of the list is the same as the exclude_list and subject_list shown in Figures 13 and 53. The physical change request list is shown in Figure 59. This list stores all of the relevant change data for each physical change request received at an mserver. The data is copied from the received message and a new entry is added to the list of pending changes. The application group change requests list functions in the same manner as the physical change requests list, but is maintained separately to simplify the prioritization of pending physical and application change requests. The list of application

TABLE 6: MSERVER INTERNAL STATE VARIABLES
Note: Separate copies of all state variables are maintained by each mserver for the core-set and child-set of which it is a member.

Variable	Description
group_name	name of core-set
group_address	address of core-set
group_size	size of core-set
group_view	current group view of core-set
authentication	used for core-set security
mygid	group identity number of this mserver
cw	clockwise neighbor (monitor)
ccw	counterclockwise neighbor (monitored)
core_table	pointer to core-table
exclude_list	list of mservers to be excluded from core-set due to failure
subject_list	list of subjects for a *Merge* or *Split*, or failed mservers
current_change	pointer to structure for data about current change
previous_change	pointer to structure for data about previous change
failures	list of failed core-set mservers waiting to be processed
requests	list of pending physical change requests received by core-set
app_changes	list of pending application change requests submitted to mserver
timeouts	timeout vector (recv, query, reply, messg, ACK)
retries	retries vector (reply, messg, ACK)
expected_type	message type expected for current processing
responses	count of number of responses (*ACK*s and *Coord_Fail*s)
app_group_list	list of application groups supported and relevant data

groups is illustrated in Figure 60. The fields in each entry in the application group list represent all of the data that the mserver must maintain for each application group supported. By keeping the data stored minimal, scalability is maintained. The group_name is the string name for the application group. The core_set and name_set fields are boolean variables to indicate whether this mserver is in the core-set or name-set

gid	address	rank	cw	ccw	flag1	flag2
0	131.120.50.103	6	1	5		
1	131.120.50.110	0	3	0		
2						
3	131.120.50.105	1	8	1		
4						
5	131.120.50.108	5	0	7		
6	131.120.50.106	3	7	8		
7	131.120.50.112	4	5	6		
8	131.120.50.115	2	6	3		
9						
. . .						
MAXTBLSIZE						

Figure 57: Mserver Core Table

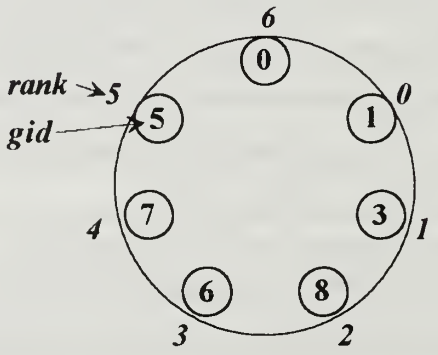


Figure 58: Mserver Core-set Corresponding to Core Table in Figure 57

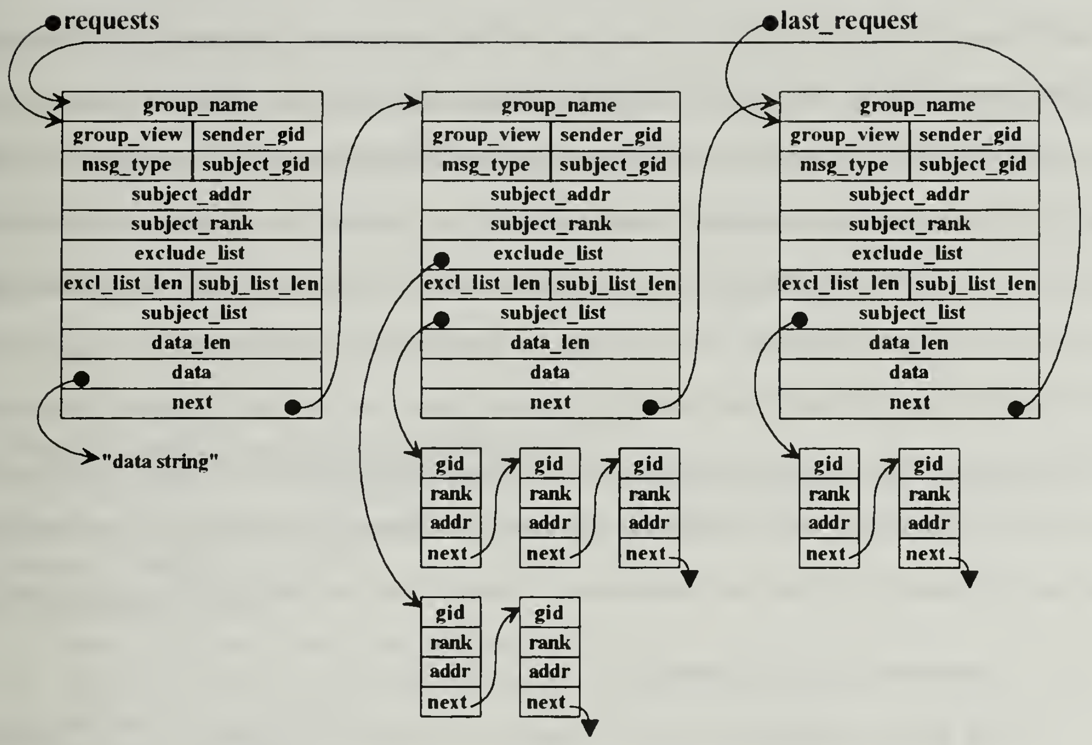


Figure 59: Mserver Requests List

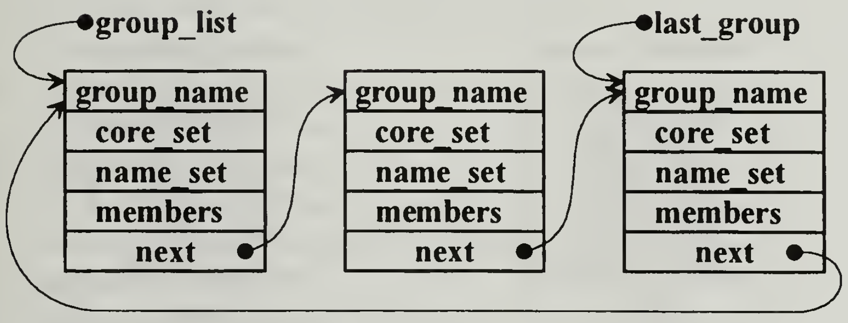


Figure 60: Mserver Application Groups List

of the application. The members field is a list of child mservers with application members in their hierarchy. Only these child mserver will be included in the message exchange and change processing for the application group which they support. Other mservers will not be impacted by the changes to application groups which they do not support, with the exception of processing core-set changes if they happen to be in the core-set for the application. Figure 61 shows the data structure used to store information related to the current and previous changes. The current_change structure maintains a separate exclude list from that included with the mserver's internal state, so that any changes made to the exclude list while processing a change that is subsequently dropped do not affect the main exclude list of the mserver. When the change is committed, the main exclude list for the core-set is updated with the new information contained in the current_change exclude list.

2. Algorithm and Explanation

The general algorithm for an mserver is listed in Figure 62. As described previously, the algorithm for the mserver allows continual processing of incoming messages and internal events, even while a current change is being processed. Outgoing

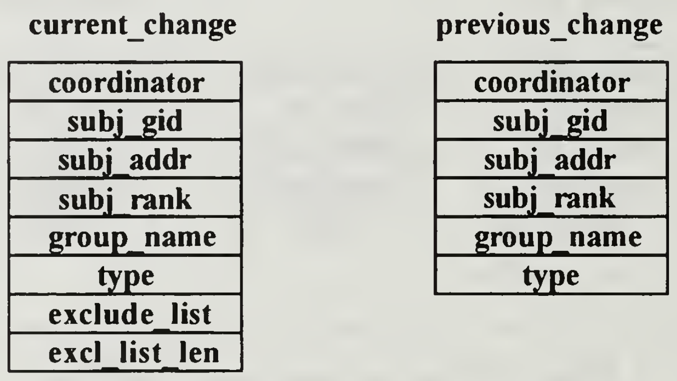


Figure 61: Mserver Current and Previous Change Storage

messages can be sent at the same time as well. The line-by-line description of the algorithm follows.

The mserver is started with a command line call, as shown in the header of Figure 62. The group_address is only included if this mserver is the first to join a new core-set, thus creating the core-set. Lines 1 and 2 make system calls to obtain information about the host computer and core-set multicast group. These calls are made with the name as an

```
Mserver (group_name, [group_address])
/* group_name is the string name of the core-set, group_address is an optional IP
address of the multicast group for the core-set, included only when a new
core-set is being created */
/* Initialize mserver */
1.  obtain_info (host_name, host_address)
2.  obtain_info (group_name, group_address)
3.  initialize sockets (ms, mc)
4.  initialize (internal state)
5.  set timers (recv, query, reply, messg, ACK)
/* Join core-set */
6.  send_message (Join_message, group_address)
7.  messg = Reliable_receive (Init_message, Msg_Query_message)
8   if (messg == Init_message)     /* successful join */
9.      join_mcast_group (group_name, group_address)
10.     update (internal state)
11. else   /* unsuccessful join */
12.     exit and report error
/* Begin main processing loop */
13. for (; ;)  /* infinite loop */
14.     receive_message (messg, recv_timeout)
15.     update (internal state)
16.     process_message (messg, internal state)
17.     process_lists (internal state)
18.     process_timeouts (internal state)
```

Figure 62: Mserver Algorithm

input argument, and return the respective address. The core-set multicast address is obtained from a locally maintained, well-known file, mapping group names to multicast addresses in the local environment, if the group already exists. If the group does not exist, the procedure registers the group_name and corresponding group_address in the file. Lines 3 initializes the two sockets used by the mserver: *ms* for incoming unicast messages and all outgoing messages, and *mc* for outgoing multicast messages (to utilize the *mcaster* utility, if needed). Line 4 initializes the internal state of the mserver, represented by the data structures and variables for each part, as shown in Figure 56. Line 5 initializes all timeout variables used as timers for the reception of messages.

Now that the mserver has been created and initialized, lines 6 through 12 control the mserver's attempt to join the desired core-set. If a new core-set is being created, there is no need to send and receive messages to join a core-set. The mserver simply updates the internal state to reflect that it is the only mserver in the new core-set. If the core-set already exists, a join request message is sent to the core-set multicast address in line 6, followed by a Reliable_receive of the Initial Parameters message from the core-set in line 7 The Initial Parameters message contains the complete state of the sending mserver, which was the coordinator for the join request of this new mserver to the core-set. Since the state of the coordinator is also the state of all other mservers in the core-set, the joining mserver receives the complete state of the core-set in this message. An illustration of the Initial Parameters message is shown in Figure 63. If the Initial Parameters message is returned, the mserver joins the multicast group for the core-set. This is a separate action from joining the core-set; the core-set must have been joined first before allowing the new mserver to become a member of the core-set multicast group. If for some reason the joining mserver is unable to join the multicast group, it will exit and return an error code to the caller. The core-set will soon detect the new mserver failed and remove it from the group automatically.

After successfully joining the core-set, the mserver begins the main loop of operation, shown in lines 13 through 18. The mserver continually repeats a cycle of receiving any incoming messages, processing the received message, then processing any pending failures or requests, and finally checking the internal timers to determine if any

vers	**checksum**
group_name	
authentication	
group_view	**coordinator_gid**
INIT_STATE	**subject_gid**
subject_addr	
subject_rank	
exclude_list	
excl_list_len	**0**
NULL	
data_len	
No. of messages in request list	
Coordinator's core-table	
Coordinator's request list (if any)	

Figure 63: Initial Parameters Message Format

messages have exceeded their timeouts. In line 14 a timed receive function is used; the process is idle waiting for any incoming message or the timeout period to elapse. This is similar to a combination of the *select* and *recvfrom* UNIX socket calls. The timeout for the receive function is a relatively short period, and in the absence of any incoming messages, acts as the clock for the mserver. Each iteration of the main loop represents one "tick" of the logical event clock for the mserver. All other timeouts used are multiples of this basic receive timeout, so that messages are timed in terms of a real clock as well as the logical event clock. When the receive function returns, either a message has been received, or the timeout period expired. If a message was received, it is processed in line 16. The message is decoded, and the appropriate action taken depending on the message type.

Next, the failures list, requests list, and application group requests list are checked for pending items. The lists are checked this order, so that the failures list has priority. Any mserver failures queued are "batched" and processed as one change to the core-set membership, with the rank of the highest ranking subject used for the change priority. Upon completion of processing the failures list, the requests list and application group requests lists are checked, in that order. Only one pending change is processed each main cycle; the request at the head of the selected queue is copied into the current_change structure and processed as the current change. The request is not removed from the head of the queue until the change is committed, so that if the change is dropped, the request will remain at the head of the queue. Once the change is committed, the change data is copied from the current_change structure to the previous_change structure.

Finally, all timers are checked to see if any expected message has exceeded the associated timeout period. If any timers expired, the internal state is updated, messages are sent as needed, and the timers are reset. This completes the main loop processing, which is then repeated continually. The code for a partial implementation of an mserver process is included in the Appendix in file *mserver.c*.

C. MEMBER INTERFACE

As discussed previously, the primary purpose for the MI is to act as an interface between the application and the MS hierarchy. The MI accepts membership change and information requests from the application processes and relays the requests to the LAN mservers for processing. When the change has been processed, the MI accepts and relays the change data to the application processes. The MI must also respond to periodic monitoring queries from the LAN mserver.

1. Internal State and Data Structures

The MI maintains a limited amount of information about the MS hierarchy and the application process group members that it supports, as shown in Figure 64. The only MS hierarchy information stored is the name and IP address of the LAN mserver. The MI maintains a list of the application groups that it supports. This list is very similar to the application groups list maintained by mservers, except the MI is not part of any core-set or name-set. Additionally, the MI must maintain a list of all members running on the host computer for each application group. Information about other member processes is obtained by requests to the MS hierarchy. An optional QoS feature would allow the MI to store limited information about all application member processes for a particular application, thus increasing the storage requirements at the host computer, but greatly reducing the latency to obtain member information.

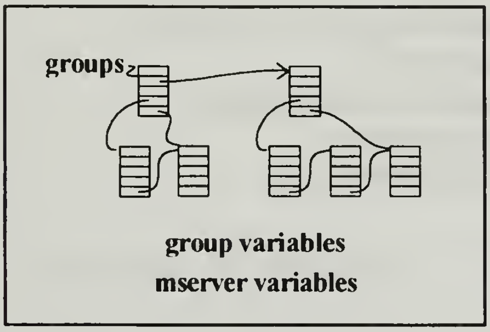


Figure 64: MI Data Structures and Internal State

2. Algorithm and Explanation

Figure 65 lists the algorithm used by the MI. The algorithm is similar to that of an mserver, but much simpler. The same idea of a continual cycle of receiving messages, processing the messages, processing pending requests, and processing expired timeouts is performed. The timed receive function is used again, so that receive cycles act as the internal event clock. The initialization in lines 1 through 11 is very similar to that of an

```
MI (mserver_name)
/* mserver_name is the string name of the LAN mserver */
/* Initialize MI */
1.  obtain_info (host_name, host_address)
2.  obtain_info (mserver_name, mserver_address)
3.  initialize socket (ms)
4.  initialize (internal state)
5.  set timers (recv, messg)
/* Register with LAN mserver */
6.  send_message (Join_message, mserver_address)
7.  messg = Reliable_receive (ACK_message, Msg_Query_message)
8   if (messg == ACK_message)        /* successful */
9.      update (internal state)
10. else  /* unsuccessful */
11.     exit and report error
/* Begin main loop */
12. for (; ;)  /* infinite loop */
13.     receive_message (messg, recv_timeout)
14.     update (internal state)
15.     process_message (messg, internal state)
17.     process_lists (internal state)
18.     process_timeouts (internal state)
```

Figure 65: MI Algorithm

mserver, except the MI does not join a group, but instead registers with the LAN mserver. The main loop of lines 12 through 18 is nearly identical to that of an mserver, except that there are many fewer events to process at each stage. The only messages received by an

MI are application membership change and information requests, *Direct* messages from the LAN mserver, and monitoring *Query* messages from the LAN mserver. The only messages that an MI sends are *Submit* messages for application change requests, *Reply* monitoring messages to the LAN mserver, and configuration messages to the MS. The MI only needs to track two timers: the main receive timer and a message timer for expected responses. Incoming requests are added to a pending requests list if a current request has been submitted to the LAN mserver. As each change request is completed with the reception of a *Direct* message, a new request is taken from the head of the queue and processed.

VII. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

This thesis presented a globally scalable, fully decentralized group membership service which provides the framework for distributed applications of any size and distribution. A complete description of the architectural design and protocol specifications were presented, and an implementation of the membership service was described.

The most significant contribution of the group membership service described herein is the total scalability. The MS provides a consistent suite of services to client applications distributed on any scale, from a single LAN to the worldwide internetwork. No other membership protocol or service provides this capability. Other noteworthy contributions include the decentralized and efficient nature of the MS, and the concept of providing numerous user-selectable Quality of Service options to tailor the MS to the needs of each client application.

B. FUTURE WORK

Although a great deal of work was accomplished in the design and partial implementation of the MS described in this thesis, much more work remains to be done. First, to demonstrate that the MS is truly scalable to global proportions, a working implementation of the complete MS must be developed and installed on progressively larger scales. Second, complete performance analysis of the operation, overhead, network constraints, service latency, and functionality of the MS must be accomplished. Third, a complete formal specification of the protocols used by the MS must be accomplished, with a reachability analysis conducted to demonstrate formally correct operation. It is recommended that an extended communicating finite state machine (CFSM) modeling method be used, such as Systems of Communicating Machines (SCM) [30], for the formal specification. Finally, the MS must be modified to take advantage of the reliable, high-speed networks which are currently being deployed. Advances in network technology, such as ATM (Asynchronous Transfer Mode) and Sonet (Synchronous optical network), provide a different network model than the conventional IP-based model used for the design of this MS. The conceptual design described in this thesis remains valid for any network model; however, by adapting the protocol specifications to take advantage of reliable, high-speed networks, a more efficient and capable version of the MS can be realized.

APPENDIX

```
/*********************************************************************
* MCASTER.H ver 1.0
* Header file for MCASTER.C
* Program to emulate IP multicast in a unicast environment.
*
* Written by Dave Neely, March 1994.
* Modified: 4/23/94
*********************************************************************/

#define MC_PORT           3000
#define JOIN_GROUP        120
#define LEAVE_GROUP       121
#define JOIN_ACK          130
#define DUP_MEMBER        131
#define NEG_JOIN          132
#define LEAVE_ACK         140
#define NO_GROUP          141
#define NO_MEMBER         142
#define NEG_LEAVE         143
#define NO_LOOP           0
#define LOOP              1
```

```
/***********************************************************************
* MSUTIL.H ver 1.0
* Header file for Membership Service (MS) utilities
*
* Written by Dave Neely, March 1994.
* Modified: 4/23/94
***********************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
#include <sys/time.h>

#define VERS            1               /* version number */
#define MS_PORT         2900            /* unicast & IP multicast */
#define MAXGROUPNAME    32              /* chars */
#define MAXMSGLEN       1024            /* 1kB */
#define SEC             1000000         /* 1 million usec */
#define T_RECV          1*SEC           /* recv cycle timeout */
#define T_REPLY         5 * T_RECV      /* incoming reply timeout */
#define T_MESSG         T_REPLY         /* incoming messg timeout */
#define T_ACK           T_REPLY         /* incoming ACK timeout */
#define T_QUERY         60*SEC          /* timeout to send another query */
#define MAXTBLSIZE      250             /* max size of group table */
#define MAXTIME         0x7fffffff      /* to reset timeouts */

#ifdef  IFF_MULTICAST
#ifndef MULTICAST
#define MULTICAST
```

```
#endif
#endif

/* monitoring message types */
#define QUERY              0              /* monitoring query */
#define REPLY              1              /* monitoring reply */

/* mserver INITIATE message types */
#define M_JOIN             10             /* mserver join_group*/
#define M_LEAVE            11             /* mserver leave */
#define M_SPLIT            12             /* mserver split_group */
#define M_MERGE            13             /* mserver merge_group */
#define M_ADD_PARENT       14             /* mserver add parent */
#define M_DEL_PARENT       15             /* mserver delete parent */
#define M_STATS_S          16             /* mserver group stats - short */
#define M_STATS_L          17             /* mserver group stats - long */
#define M_FAIL             18             /* mserver fail */
#define M_MULTI_FAIL       19             /* multiple mservers fail */
#define M_COORD_FAIL       20             /* coordinator fail */

/* change sequence message types */
#define ACK                21
#define COMMIT             22
#define WAIT               23
#define MSG_QUERY          24
#define   INIT             25

/* external physical requests to core-set */
#define M_JOIN_REQ         30             /* mserver join_group request */
#define M_LEAVE_REQ        31             /* mserver leave_request */
#define M_SPLIT_REQ        32             /* mserver split_group request */
#define M_MERGE_REQ        33             /* mserver merge_group request */
#define M_ADD_PAR_REQ      34             /* mserver add parent request */
#define M_DEL_PAR_REQ      35             /* mserver delete parent request */
```

```
#define M_STATS_S _REQ    36                  /* mserver group stats - short */
#define M_STATS_L _REQ    37                  /* mserver group stats - long */

/* application group INTITIATE message types */
#define A_JOIN            70                  /* app join_group */
#define A_LEAVE           71                  /* app leave_group */
#define A_SPLIT           72                  /* app split_group */
#define A_MERGE           73                  /* app merge_group */
#define A_STATS_S         74                  /* app group stats - short */
#define A_STATS_L         75                  /* app group stats - long */
#define SUBMIT            76                  /* app change submission */
#define DIRECT            77                  /* parent's change directive */

/* application group request message types */
#define A_JOIN_REQ        80                  /* app join_group request */
#define A_LEAVE_REQ       81                  /* app leave_group request */
#define A_SPLIT_REQ       82                  /* app split_group request */
#define A_MERGE_REQ       83                  /* app merge_group request */
#define A_STATS_S_REQ     84                  /* app group stats - short */
#define A_STATS_L_REQ     85                  /* app group stats - long */

struct table_entry {              /* member's entry in set table */
    u_long   addr;                /* IP address of member */
    u_short  rank;                /* member's rank */
    u_short  cw;                  /* gid of clockwise member (to "left") */
    u_short  ccw;                 /* gid of counterclockwise member */
    u_char   flag1;               /* status flag for each member */
    u_char   flag2;               /* status flag for each member */
};

struct gid_entry {                /* used for gid lists */
    u_short  gid;                 /* member's group ID */
    u_short  rank;                /* member's rank */
    u_long   addr;                /* IP address of member */
```

```
        struct    gid_entry *next;
};

struct message {                      /* to build and receive messages */
        u_short  vers;
        int      checksum;
        char     group_name[MAXGROUPNAME];
        u_short  group_view;
        long     authentication;
        u_short  sender_gid;
        u_short  msg_type;
        u_short  subject_gid;
        u_long   subject_addr;
        u_short  subject_rank;
        struct   gid_entry *exclude_list;
        u_short  excl_list_len;
        struct   gid_entry *subject_list;
        u_short  subj_list_len;
        char     *data;
        int      data_len;
} messg;

struct group_state{                   /* core and child set internal state */
        char     group_name[MAXGROUPNAME];
        struct   sockaddr_in group_addr;
        u_short  group_size;
        u_short  group_view;
        long     authentication;
        u_short  mygid;
        u_short  cw;
        u_short  ccw;
        struct   table_entry table;
        struct   gid_entry *exclude_list;
        u_short  excl_list_len;
```

```
    struct   gid_entry *subject_list;
    u_short  subj_list_len;
    char     *data;
    int      data_len;
    struct   change_data *current_change;
    struct   change_data *previous_change;
    struct   gid_entry *failures;
    struct   gid_entry *last_failure;
    struct   messg_entry *requests;
    struct   messg_entry *last_request;
    struct   messg_entry *app_requests;
    struct   messg_entry *last_app_request;
    struct   timeval recv_timeout;
    struct   timeval query_timeout;
    struct   timeval reply_timeout;
    struct   timeval messg_timeout;
    struct   timeval ACK_timeout;
    u_short  r_retries;
    u_short  m_retries;
    u_short  a_retries;
    u_short  ACK_count;
    u_short  change_underway;
    u_short  elect_coordinator;
};

struct change_data {                    /* current and previous change info */
    u_short  coordinator;
    u_short  subj_gid;
    u_long   subject_addr;
    u_short  subj_rank;
    u_short  group_name[MAXGROUPNAME];
    u_short  type;
};
```

```
struct messg_entry {                    /* entry in requests lists */
      char      group_name[MAXGROUPNAME];
      u_short   group_view;
      long      authentication;
      u_short   sender_gid;
      u_short   msg_type;
      u_short   subject_gid;
      u_long    subject_addr;
      u_short   subject_rank;
      struct    gid_entry *exclude_list;
      u_short   excl_list_len;
      struct    gid_entry *subject_list;
      u_short   subj_list_len;
      char      *data;
      int       data_len;
      struct    messg_entry *next;
};
```

```
/************************************************************************
* MSUTIL.C ver 1.0
* Utility procedures used by Membership Service programs.
*
* int     init_socket(sin, port)
* int     join_mcast_group()
* void    leave_mcast_group()
* int     addrcmp(addr1, addr2)
* void    form_messg(messg, group, authentication, groupview, sender, type, subject,
*              excl_list, excl_list_len, subj_list, subj_list_len, data, data_len)
* int     send_messg(socket, message, dest)
* int     recv_messg(ms_socket, mc_socket, message, sender, timeout)
* void    set_timeout()
* int     timed_out()
* int     search_gid_list(gid_list, gid)
* int     add_gid_entry(&gid_list, gid)
* int     copy_gid_list(gid_list, &buffer)
* int     extract_gid_list(buffer, &gid_list, list_len)
* void    delete_gid_list(&gid_list)
* void    print_in_addr(in_addr)
* void    print_sock_addr(sin)
* void    print_sock_info(s, sin)
* void    print_hostent(hp)
* void    print_messg(messg)
* void    print_gid_list(gid_list)
*
* Written by Dave Neely, March 1994.
* Modified: 4/26/94
************************************************************************/


#include "msutil.h"
#include "mcaster.h"
```

```
int     init_socket();
int     join_mcast_group();
void    leave_mcast_group();
int     addrcmp();
void    form_messg();
int     send_messg();
int     recv_messg();
void    set_timeout();
int     timed_out();
int     search_gid_list();
int     add_gid_entry();
int     copy_gid_list();
int     extract_gid_list();
void    delete_gid_list();
void    print_in_addr();
void    print_sock_addr();
void    print_sock_info();
void    print_hostent();
void    print_messg();
void    print_gid_list();

/* global variables */
struct  sockaddr_in sin, mcsin;       /* socket addresses */
struct  sockaddr_in group_addr;       /* group mcast address */
int     ms, mc;                       /* IP socket fd's */
#ifdef  MULTICAST
struct  ip_mreq imr;                  /* IGMP control */
#endif
int     debug = 0;                    /* 1 = enable diagnostic prints */

/*********************************************************************
* Initialize socket address structure
*********************************************************************/
int init_socket(sin, port)
```

```
struct    sockaddr_in sin;                /* socket address */
u_short  port;
{
    int  s;                               /* socket fd */
    int  one = 1;
    bzero((char*)&sin, sizeof(sin));/* clear address structure and initialize */
    sin.sin_family = AF_INET;
    sin.sin_port = htons(port);
    sin.sin_addr.s_addr = htonl(INADDR_ANY);

    /* open and bind UDP/IP socket */
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)  {
        perror("can't open socket");
        exit(-1);
    }
    if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one)) < 0)  {
        perror("can't make socket reuseable");
        exit(-1);
  }
    if (bind(s, (struct sockaddr *) &sin, sizeof(sin)) < 0)  {
        perror("can't bind socket");
        close(s);
        exit(-1);
    }
    return s;
}


/*************************************************************************
* Join IP multicast group or mcaster group (if unicast only).
*************************************************************************/
int join_mcast_group(group_name, group_str_addr)
```

```
char    *group_name;                    /* group string name */
char    *group_str_addr;                /* IP dot address */
{
    u_char loop = 0;                    /* turn loopback option off */
    int     len, sent;
    struct  sockaddr_in from;           /* to receive sender's address */
    struct  timeval timeout;            /* time to wait for response */

    timeout.tv_sec = 30*SEC;            /* wait max of 30 seconds */
    timeout.tv_usec = 0;

    /* set up group address structure */
    group_addr.sin_family = AF_INET;
    group_addr.sin_port = htons(MS_PORT);
    group_addr.sin_addr.s_addr = inet_addr(group_str_addr);
    printf("Group Address:\n");
    print_sock_addr(group_addr);

#ifdef MULTICAST                        /* join IGMP multicast group */
    imr.imr_multiaddr.s_addr = inet_addr(group_str_addr);
    printf("group address: %s\n", inet_ntoa(imr.imr_multiaddr.s_addr));
    imr.imr_interface.s_addr = htonl(INADDR_ANY);
    if (setsockopt(ms, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                &imr, sizeof(imr)) < 0)  {
        perror("can't join group");
        return NEG_JOIN;
    }
    if (setsockopt(ms, IPPROTO_IP, IP_MULTICAST_LOOP,
                &loop, sizeof(loop)) < 0)  {
        perror("can't disable multicast loopback");
    }
    printf("group %s joined\n", inet_ntoa(imr.imr_multiaddr.s_addr));
    return JOIN_ACK;
```

```
#else      /* join MCASTER multicast emulator group */
      /* generate and send join request message to MCASTER */
      form_messg(&messg, group_name,0,0,0, JOIN_GROUP,0,0,0,0,0,0,0,0);
      len = sizeof(messg);
      printf("SENDING JOIN MESSAGE:\n");
      printf("message to send: \n");
      print_messg(messg);
      sent = send_messg(ms, messg, group_addr);
      printf("%d bytes sent\n", sent);

      /* wait for ACK message from MCASTER */
      if ((sent = recv_messg(ms, mc, &messg, &from, timeout)) < 0)
            printf("error in message rec'd\n");
      else {
            printf("MESSAGE RECEIVED:\n");
            printf("%d bytes received\n", sent);
            print_messg(messg);
            printf("sender:\n");
            print_sock_addr(from);
      }
#endif
      return messg.msg_type;
}


/************************************************************************
* Leave IP multicast or mcaster group.
************************************************************************/
void leave_mcast_group(group_name)

char * group_name;
{
      int      len, sent;
      short  message_type;
```

```
        struct sockaddr_in from;              /* to receive sender's address */
        struct timeval timeout;

        set_timeout(&timeout, 30*SEC);    /* wait 30 seconds */

        /* leave group */
#ifdef MULTICAST
        if (setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                        &imr, sizeof(struct ip_mreq)) < 0)  {
            perror("can't leave group");
            exit (-1);
        }
        else
            printf("group %s left\n", group_name);
#else
        /* generate and send leave request message to MCASTER */
        form_messg(&messg, group_name,0,0,0, LEAVE_GROUP,0,0,0,0,0,0,0,0);
        len = sizeof(messg);
        printf("SENDING LEAVE MESSAGE:\n");
        printf("message to send: \n");
        print_messg(messg);
        sent = send_messg(ms, messg, group_addr);
        printf("%d bytes sent\n", sent);

        /* wait for ACK message from MCASTER */
        if ((sent = recv_messg(ms, mc, &messg,&from, timeout)) < 0)
            printf("error in message rec'd\n");
        else {
            printf("MESSAGE RECEIVED:\n");
            printf("%d bytes received\n", sent);
            print_messg(messg);
            printf("sender:\n");
            print_sock_addr(from);
            message_type = ntohs(messg.msg_type);
```

```
          printf("message_type = %d\n", message_type);
          if ((strcmp(messg.group_name, group_name)) ||
                (!(message_type == LEAVE_ACK)))
               printf("unable to leave group: error %d\n", message_type);
     }
#endif
}


/*********************************************************************
* Compare two sockaddr_in structs.. return 1 if same, 0 otherwise.
*********************************************************************/
int addrcmp (addr1, addr2)

struct sockaddr_in addr1;
struct sockaddr_in addr2;
{
     return ((addr1.sin_family == addr2.sin_family) &&
                 (addr1.sin_port == addr2.sin_port) &&
                 (addr1.sin_addr.s_addr == addr2.sin_addr.s_addr));
}  /* addrcmp */


/*********************************************************************
* Compose message.  Copy all integer values, point list and data
* pointers to appropriate list or data string.
*********************************************************************/
void form_messg(messg, group, auth, GV, sender, type, subject, excl_list,
                    excl_list_len, subj_list, subj_list_len, data, data_len)

struct   message *messg;
char     *group;                          /* group string name */
long     auth;                            /* authentication number */
u_short GV;                               /* group view number */
u_short sender;                         /* sender gid */
u_short type;                           /* message type */
u_short subject;                        /* subject gid */
struct  gid_entry *excl_list;           /* gid exclude list */
u_short excl_list_len;
struct  gid_entry *subj_list;           /* gid subject list */
u_short subj_list_len;
char    *data;
u_short data_len;
{
/*  bzero((char *)messg, sizeof(*messg));   */
    messg->vers             = VERS;
    messg->checksum         = htons(0xffff);
    strcpy(messg->group_name, group);
    messg->authentication = htons(auth);
    messg->group_view       = htons(GV);
    messg->sender_gid       = htons(sender);
    messg->msg_type         = htons(type);
    messg->subject_gid      = htons(subject);
    messg->exclude_list     = excl_list;
    messg->excl_list_len    = htons(excl_list_len);
    messg->subject_list     = subj_list;
    messg->subj_list_len    = htons(subj_list_len);
    messg->data             = data;
    messg->data_len         = htons(data_len);
}


/*********************************************************************
* Send a variable length message "messg".  The message may contain 2
* lists of gids, and data field.  These are appended to the buffer,
* 'messgbuf', used to store the overall message.  Returns the number
* of bytes sent.
*********************************************************************/
```

```
int send_messg(s, messg, to)

int      s;                                    /* socket fd */
struct message messg;
struct sockaddr_in to;
{
    int   sent;
    int   msglen = sizeof(messg) + (ntohs(messg.excl_list_len) +
                          ntohs(messg.subj_list_len))*2 + ntohs(messg.data_len);
    char      *messgbuf, *mp;                  /* message buffer and pointer */
    char      *datastr;                        /* for diagnostic prints */
    int       i;
    u_short val, *up;                          /* for diagnostic prints */

    if (debug) printf("messglen to send: %d\n", msglen);
    if (!(messgbuf = (char*) malloc (msglen)))  {
        perror("unable to create message buffer");
        return -1;
    }
    /* copy messg into outgoing buffer */
    bzero(messgbuf, msglen);                          /* clear buffer */
    bcopy((char *)&messg, messgbuf, msglen);  /* copy messg into buffer */
    mp = messgbuf + sizeof(messg);                    /* skip over messg */
    /* append excl & subj lists and data */
    copy_gid_list(messg.exclude_list, &mp);

    if (debug) {          /* print excl_list string */
        printf("excl_list to send: ");
        for (i=0; i<ntohs(messg.excl_list_len); i++)  {
            up = (u_short *)(mp + i*2);
            printf(" %d ", *up);
        }
        printf("\n");
    }
    mp += (ntohs(messg.excl_list_len)*2);          /* skip 2* number of gids */
    copy_gid_list(messg.subject_list, &mp);

    if (debug) {          /* print subj_list string */
        printf("subj_list to send: ");
        for (i=0; i<ntohs(messg.subj_list_len); i++)  {
            up = (u_short *)(mp + i*2);
            printf(" %d ", *up);
        }
        printf("\n");
    }
    mp += (ntohs(messg.subj_list_len)*2);          /* skip 2* number of gids */
    bcopy(messg.data, mp, ntohs(messg.data_len));

    if (debug) {   /* create temporary data string to print messg.data */
        if (!(datastr = (char*) malloc (ntohs(messg.data_len)+1)))
            perror("unable to create data string");
        bcopy(mp, datastr, ntohs(messg.data_len));
        datastr[ntohs(messg.data_len)] = NULL;             /* make string */
        printf("data to send:     %s\n", datastr);
        free(datastr);
    }
    if ((sent = sendto(s, messgbuf, msglen, 0, (struct sockaddr *)&to,
                        sizeof(struct sockaddr))) != msglen)
        perror("error in message length sent");
    free(messgbuf);
    return sent;
} /* send_messg */


/*********************************************************************
* Receive a variable length message from either the ms or mc sockets.
* Use select() to receive from ready socket into messgbuf.  If received
* from ms, then messgbuf contains only "messg" and can be transferred.
* If received from mc, then messgbuf has the sender's address at the
* front which is extracted into "from", then extract "messg".
* Note: recv_messg allocates memory for the received gid_lists and
* data.  Messg is returned with pointers pointing to these new lists
* and data.  The lists and data must be deallocated when no longer
* needed, and before a new message is formed.  Otherwise, the links to
* the memory will be lost, and the memory cannot be recocered.
*********************************************************************/
int recv_messg(ms, mc, messg, from, timeout)

int     ms, mc;                         /* socket fd's */
struct message *messg;                  /* to hold incoming message */
struct sockaddr_in *from;               /* extract sender's address */
struct timeval timeout;             /* for variable timeout */
{
    char    messgbuf[MAXMSGLEN], *mp;      /* message buffer and pointer */
    int     len = sizeof(*from);
    int     ready, sent = 0;
    fd_set  fdread;                     /* fd mask for select() */
    char    *datastr, *data;            /* to receive messg.data */

    /* initialize for reception from multiple sockets */
    FD_ZERO(&fdread);
    FD_SET(ms, &fdread);
    FD_SET(mc, &fdread);
    /* wait until either socket is ready to be read */
    if ((ready = select(32, &fdread, 0, 0, &timeout)) < 0)  {
        perror("select error");
        return -1;
    }
    if (ready)  {
        bzero((char *)messg, sizeof(*messg));
        if (FD_ISSET(ms, &fdread))  {      /* received from ms socket */
            printf("received at MS socket\n");
            if ((sent = recvfrom(ms, messgbuf, MAXMSGLEN, 0, from, &len)) < 0) {
                perror("error in message rec'd");
                return -1;
            }
            else  /* extract message from messgbuf */
                mp = messgbuf;                /* set ptr to beginning of message */
        }
        if (FD_ISSET(mc, &fdread))  {       /* received from mc socket */
            printf("received at MC socket\n");
            if ((sent = recvfrom(mc, messgbuf, MAXMSGLEN, 0, from, &len)) < 0) {
                perror("error in message rec'd");
                return -1;
            }
            else  {    /* extract sender address & message from messgbuf */
                bzero((char *)from, len);
                bcopy(messgbuf, (char *)from, len);
                mp = messgbuf + len;  /* set ptr to beginning of message */
            }
        }
        /* extract messg, exclude & subject lists, and any data from messgbuf */
        bcopy(mp, (char *)messg, sizeof(*messg));
        mp += sizeof(*messg);       /* skip to lists */
        if ((len = extract_gid_list(mp, &(messg->exclude_list),
            ntohs(messg->excl_list_len))) != ntohs(messg->excl_list_len))  {
              printf("error in extracting exclude list: len = %d\n", len);
              return -1;
        }
        if (debug) printf("len = %d gids extracted\n", len);
        mp += (ntohs(messg->excl_list_len))*2;       /* skip to end of list */
        if (debug) printf("mp-messgbuf = %d\n", mp-messgbuf);
        if ((len = extract_gid_list(mp, &(messg->subject_list),
            ntohs(messg->subj_list_len))) != ntohs(messg->subj_list_len))  {
              printf("error in extracting subject list: len = %d\n", len);
              return -1;
        }
        if (debug) printf("len = %d gids extracted\n", len);
        mp += (ntohs(messg->subj_list_len))*2;      /* skip to end of list */
        if (debug) printf("mp-messgbuf = %d\n", mp-messgbuf);
        if(!(data = (char *) malloc (ntohs(messg->data_len)))) {
            perror("unable to allocate memory for data");
            return -1;
        }
        /* copy received data into messg.data */
        bcopy(mp, data, ntohs(messg->data_len));
        messg->data = data;
        if (debug) printf("after data: mp-messgbuf = %d\n",mp-messgbuf);

        if (debug) {   /* create temporary data string to print messg.data */
            printf("messg->data_len = %d\n", ntohs(messg->data_len));
            if (!(datastr = (char*) malloc (ntohs(messg->data_len)+1)))
                perror("unable to create data string");
            bcopy(mp, datastr, ntohs(messg->data_len));
            datastr[ntohs(messg->data_len)] = NULL;          /* make string */
            printf("data rec'd:     %s\n", datastr);
            free(datastr);
        }
    } /* ready */
    return sent;
} /* recv_messg */


/**********************************************************************
* Set timer to current time + timeout time t_usec.  Converts t_usec to
* seconds and useconds, and adds to timer.tv_sec & timer.tv_usec,
* respectively.  If useconds exceed 1,000,000, a carry to seconds is
* performed.
*
**********************************************************************/
```

```
void set_timeout(timer, t_usec)

struct timeval *timer;                  /* timer to set */
long   t_usec;                          /* timeout period in usec. */
{
     struct  servent tzp;               /* for timing */

     if (t_usec == MAXTIME)  {          /* set sec & usec to MAXTIME */
          timer->tv_sec = MAXTIME;
          timer->tv_usec = MAXTIME;
     }
     else  {          /* set timer to current time + t_usec */
          if (gettimeofday(timer, &tzp) != NULL)  {
               perror("unable to gettimeofday");
             exit(-1);
          }
          /* add t_usec to timer */
          timer->tv_sec += t_usec / SEC;     /* add seconds */
          timer->tv_usec += t_usec % SEC;    /* add useconds */
          if (timer->tv_usec >= SEC)  {      /* carry 1 sec. */
               timer->tv_usec -= SEC;
               timer->tv_sec += 1;
          }
     }
}  /* set_timeout */


/*******************************************************************
* Check if timer has timed out.  Returns 1 if current time > timer,
* 0 otherwise.
*******************************************************************/
int timed_out(timer)

struct timeval timer;                   /* timeout timeval */
```

```
{
        struct  timeval tp;                          /* for time stamps */
        struct  servent tzp;                         /* for timing */

        if(gettimeofday(&tp, &tzp) != NULL)  {
                perror("unable to gettimeofday");
                exit(-1);
        }
        return ((tp.tv_sec > timer.tv_sec) ||
                        ((tp.tv_sec == timer.tv_sec) && (tp.tv_usec > timer.tv_usec)));
}  /* timed_out */


/*************************************************************************
* Search a list of gid_entries pointed at by gid_list for "gid".  Return
* 1 if the gid is found, 0 otherwise.
*************************************************************************/
int search_gid_list(gid_list, gid)

struct     gid_entry *gid_list;
u_short  gid;
{
        struct gid_entry *gp = gid_list;
        int      found = 0;

        while (gp && !found)  {
                found = (gid == gp->gid);
                gp = gp->next;
        }
        return found;
}  /* search_gid_list */


/*************************************************************************
```

```
* Add a new node to the head of the list of gid_entries pointed at by
* gid_list. Return 1 if successful, 0 if unable to add to list.
*********************************************************************/
int add_gid_entry(gid_list, gid)

struct    gid_entry **gid_list;           /* pointer to gid_list pointer */
u_short gid;
{
    struct gid_entry *gp;

    if (search_gid_list(*gid_list, gid))  /* duplicate gid found in list */
        return 0;
    /* allocate new gid_entry */
    if (!(gp = (struct gid_entry *) malloc (sizeof(struct gid_entry)))) {
        perror("unable to create new gid_entry");
        return 0;
    }
    /* add new entry to head of gid_list */
    gp->gid = gid;
    if(!(*gid_list))  /* if empty gid_list */
        gp->next = NULL;
    else                                  /* nonempty list.. insert at head */
        gp->next = *gid_list;
    *gid_list = gp;
    return 1;
} /* add_gid_entry */


/*********************************************************************
* Copy the gids from a list of gid_entries pointed at by gid_list into
* a buffer of characters.  Since each gid is u_short, it will take 2
* bytes.  Uses pointer math to increment through buffer to place gids.
* Returns the number of gids copied or 0 for an error.
*********************************************************************/
```

```
int copy_gid_list(gid_list, buffer)

struct gid_entry *gid_list;
char   **buffer;                          /* pointer to buffer */
{
      struct gid_entry *gp = gid_list;
      char   *cp = *buffer;

      if (debug) printf("copy_gid_list: cp-(*buffer) = %d\n", cp-(*buffer));
      while (gp)  {        /* copy gids one at a time */
            if (debug) printf("gp->gid = %d\n", gp->gid);
            bcopy((char *)&(gp->gid), cp, 2);
            cp += 2;                      /* u_short == 2 bytes */
            gp = gp->next;
      }
      if (debug) printf("(cp-(*buffer))/2 = %d\n", (cp-(*buffer))/2);
      return (cp - (*buffer)) / 2;  /* number of gids copied */
}  /* copy_gid_list */


/*************************************************************************
* Extract gids from a buffer of characters into a list of gid_entries
* pointed to by gid_list.  Each gid is 2 bytes in the buffer.  Uses
* pointer math to increment through buffer to place gids.
* Returns the number of gids extracted or 0 for an error.
*************************************************************************/
int extract_gid_list(buffer, gid_list, list_len)

char    *buffer;
struct  gid_entry **gid_list;             /* pointer to gid_list pointer */
u_short list_len;
{
      u_short i = 0;                      /* count of gids */
      u_short gid;
```

```
    *gid_list = NULL;
    while (i < list_len)  {  /* extract gids one at a time */
        bcopy((buffer + (i*2)), (char *)&gid, 2);
        if (!(add_gid_entry(gid_list, gid)))
            return 0; /* unsuccessful add */
        i++;
    }
    return i;  /* number of gids extracted */
}  /* extract_gid_list */


/***********************************************************************
* Remove all gids from a list of gid_entries pointed at by gid_list and
* free all memory.  Uses two pointers, ngp and cgp to walk through list
* and free each entry.
***********************************************************************/
void delete_gid_list(gid_list)

struct  gid_entry **gid_list;
{
    struct gid_entry *ngp, *cgp = *gid_list;

    while (cgp)  {                          /* current gid ptr != NULL */
        ngp = cgp->next; /* get next entry */
        free(cgp);                          /* free current entry */
        cgp = ngp;
    }
    *gid_list = NULL;
}  /* delete_gid_list */


/***********************************************************************
* Print message fields.
```

```
*********************************************************************/
void print_messg(messg)

struct message messg;
{
    char *datastr;                          /* to convert data to a string */

    printf("version:        %d\n", ntohs(messg.vers));
    printf("checksum:       %d\n", ntohl(messg.checksum));
    printf("group_name:     %s\n", messg.group_name);
    printf("authentication: %d\n", ntohl(messg.authentication));
    printf("group_view:     %d\n", ntohs(messg.group_view));
    printf("sender_gid:     %d\n", ntohs(messg.sender_gid));
    printf("subject_gid:    %d\n", ntohs(messg.subject_gid));
    printf("subject_addr:   %d\n", ntohl(messg.subject_addr));
    printf("subject_rank:    %d\n", ntohs(messg.subject_rank));
    printf("msg_type: ");
    switch (ntohs(messg.msg_type)) {
        /* monitoring message types */
        case QUERY:             printf("    QUERY\n");              break;
        case REPLY:             printf("    REPLY\n");              break;
        /* mserver INITIATE message types */
        case M_JOIN:            printf("    M_JOIN\n");             break;
        case M_LEAVE:           printf("    M_LEAVE\n");            break;
        case M_SPLIT:           printf("    M_SPLIT\n");            break;
        case M_MERGE:           printf("    M_MERGE\n");            break;
        case M_ADD_PARENT:      printf("    M_ADD_PARENT\n");       break;
        case M_DEL_PARENT:      printf("    M_DEL_PARENT\n");       break;
        case M_STATS_S:         printf("    M_STATS_S\n");          break;
        case M_STATS_L:         printf("    M_STATS_L\n");          break;
        case M_FAIL:            printf("    M_FAIL\n");             break;
        case M_MULTI_FAIL:      printf("    M_MULTI_FAIL\n");       break;
        case M_COORD_FAIL:      printf("    M_COORD_FAIL\n");       break;
```

```
/* change sequence message types */
case ACK:               printf("     ACK\n");               break;
case COMMIT:            printf("     COMMIT\n");            break;
case WAIT:              printf("     WAIT\n");              break;
case MSG_QUERY:         printf("     MSG_QUERY\n");         break;
case INIT:              printf("     INIT\n");              break;
/* external physical requests to core-set */
case M_JOIN_REQ:        printf("     M_JOIN_REQ\n");        break;
case M_LEAVE_REQ:       printf("     M_LEAVE_REQ\n");       break;
case M_SPLIT_REQ:       printf("     M_SPLIT_REQ\n");       break;
case M_MERGE_REQ:       printf("     M_MERGE_REQ\n");       break;
case M_ADD_PAR_REQ:printf("     M_ADD_PAR_REQ\n");     break;
case M_DEL_PAR_REQ: printf("     M_DEL_PAR_REQ\n");     break;
case M_STATS_S_REQ: printf("     M_STATS_S_REQ\n");     break;
case M_STATS_L_REQ: printf("     M_STATS_L_REQ\n");     break;
/* application group INITIATE message types */
case A_JOIN:            printf("     A_JOIN\n");            break;
case A_LEAVE:           printf("     A_LEAVE\n");           break;
case A_SPLITQ:          printf("     A_SPLIT\n");           break;
case A_MERGE:           printf("     A_MERGE\n");           break;
case A_STATS_S:         printf("     A_STATS_S\n");         break;
case A_STATS_L:         printf("     A_STATS_L\n");         break;
case SUBMIT:            printf("     SUBMIT\n");            break;
case DIRECT:            printf("     DIRECT\n");            break;
/* application group request message types */
case A_JOIN_REQ:        printf("     A_JOIN_REQ\n");        break;
case A_LEAVE_REQ:       printf("     A_LEAVE_REQ\n");       break;
case A_SPLIT_REQ:       printf("     A_SPLIT_REQ\n");       break;
case A_MERGE_REQ:       printf("     A_MERGE_REQ\n");       break;
case A_STATS_S_REQ:     printf("     A_STATS_S_REQ\n");     break;
case A_STATS_L_REQ:     printf("     A_STATS_L_REQ\n");     break;
/* mcaster message types */
case JOIN_GROUP:        printf("     JOIN_GROUP\n");        break;
case LEAVE_GROUP:       printf("     LEAVE_GROUP\n");       break;
```

```
        case JOIN_ACK:          printf("    JOIN_ACK\n");            break;
        case DUP_MEMBER:        printf("    DUP_MEMBER\n");          break;
        case NEG_JOIN:          printf("    NEG_JOIN\n");            break;
        case LEAVE_ACK:         printf("    LEAVE_ACK\n");           break;
        case NO_GROUP:          printf("    NO_GROUP\n");            break;
        case NO_MEMBER:         printf("    NO_MEMBER\n");           break;
        case NEG_LEAVE:         printf("    NEG_LEAVE\n");           break;
        default:                printf("    %d\n", ntohs(messg.msg_type));
    }
    printf("exclude_list:  ");
    print_gid_list(messg.exclude_list);
    printf("excl_list_len:  %d\n", ntohs(messg.excl_list_len));
    printf("subject_list:  ");
    print_gid_list(messg.subject_list);
    printf("subj_list_len:   %d\n", ntohs(messg.subj_list_len));
    printf("data_len:        %d\n", ntohs(messg.data_len));

    /* create temporary data string to print messg.data */
    if (!(datastr = (char*) malloc (ntohs(messg.data_len)+1)))
        perror("unable to create data string");
    bcopy(messg.data, datastr, ntohs(messg.data_len));
    datastr[ntohs(messg.data_len)] = NULL;      /* make string */
    printf("data:            %s\n", datastr);
    free(datastr);

} /* print_messg */


/*********************************************************************
* Print in_addr IP addresss.
*********************************************************************/
void print_in_addr(addr)

struct in_addr *addr;
```

```
{
    char *ip_addr = (char*)inet_ntoa(* addr);
    printf("IP address = %s\n", ip_addr);
}  /* print_in_addr */


/************************************************************************
* Print sockaddr_in address structure info.
************************************************************************/
void print_sock_addr(sin)

struct sockaddr_in sin;/* socket address structure */
{
    printf("family: %d \n", ntohs(sin.sin_family));
    printf("port: %d \n", ntohs(sin.sin_port));
    print_in_addr(&sin.sin_addr.s_addr);
}    /* print_sock_addr */


/************************************************************************
* Print socket info.
************************************************************************/
void print_sock_info(s, sin)

int s;                              /* socket fd */
struct sockaddr_in sin;/* socket address structure */
{
    int len = sizeof(sin);
    if (getsockname(s, (struct sockaddr *) &sin, &len) < 0)  {
        perror("can't get socket info");
        exit(1);
    }
    printf("Socket Info: \n");
    printf("socket: %d \n", s);
    print_sock_addr(sin);
} /* print_sock_info */


/**********************************************************************
* Print hostent structure info.
**********************************************************************/
void print_hostent(hp)

struct hostent *hp;
{
    char *af = hp->h_addrtype == 2 ? "AF_INET": "non-AF_INET";

    printf("Hostent Info: \n");
    printf("h_name:          %s\n", hp->h_name);
    printf("h_aliases[0]:    %s\n", hp->h_aliases[0]);
    printf("h_addrtype:      %s\n", af);
    printf("h_length:        %d\n", ntohs(hp->h_length));
    printf("h_addr:          %s\n", inet_ntoa(*(struct in_addr*)(hp->h_addr)));
    printf("h_addr_list[0]: %s\n", inet_ntoa(*(struct in_addr*) (hp->h_addr_list[0])));
} /* print_hostent */

/**********************************************************************
* Print all gids from a list of gid_entries pointed at by gid_list.
**********************************************************************/
void print_gid_list(gid_list)

struct  gid_entry *gid_list;
{
    struct gid_entry *gp = gid_list;

    if (!gp) printf(" empty list");
    else {
        while (gp != NULL) {
            printf(" %d ", gp->gid);
            gp = gp->next;    /* get next entry */
        }
    }
    printf("\n");
}  /* print_gid_list */
```

```
/**********************************************************************
* MCASTER.C ver 1.0     Multicast Emulator
* Program to emulate IP multicast in a unicast environment.
* Uses single socket for send & receive, with the IP address & port
* the same as would be used for an IP multicast (port = MS_PORT).
* Incoming messages are of "message" format, outgoing unicast messages
* are also of "message" format (for join & leave ACKs to members).
* Outgoing multicast messages have the original sender's sockaddr_in
* prepended to the message, since mcaster overwrites the original
* sender's address with its own and the recipients have no other way
* of knowing who was the original sender.
* Note: this version has no error checking or diagnostic print state-
* ments... any erroneous message is simply discarded or delivered as
* is.  For diagnostics, use MCASTERV.C, the same program with
* diagnostic print statements.
*
* Written by Dave Neely, March 1994.
* Modified 4/26/94
**********************************************************************/

#include "msutil.c"

struct member {                          /* element in list of members */
     struct    sockaddr_in addr;
     u_char    loop;
     struct    member *next;
};

struct group {                           /* element in list of groups */
     char      name[MAXGROUPNAME];
     struct  group *next;
     struct  member *members;
     struct  member *last;
};
```

```
struct   sockaddr_in sin;                   /* socket address */
struct   sockaddr_in group_addr;            /* group mcast address */
struct   sockaddr_in from;                  /* received from address */
struct   sockaddr_in member;                /* member address */
struct   hostent *hp;                       /* host entity struct */
struct   group *group_list, *last_group;    /* global group list ptrs */

/* functions */
struct   group *search_group_list();
struct   member *search_member_list();
struct   group *add_group();
int      add_member();
int      join_group();
int      remove_group();
int      remove_member();
int      leave_group();
int      mcast();
void     print_group_list();
void     print_member_list();

main()
{
    int      s;                                 /* IP socket fd */
    u_short  port;
    int      len;
    int      sent;
    char     hostname[MAXGROUPNAME];
    char     hostaddr[17];
    char     msgbuf[MAXMSGLEN];                 /* to recv message */
    char     *msgstr;                           /* to copy message */
    short    message_type, msglen;
    short    cc;
```

```
/* initialize socket */
port = htons(MS_PORT);
s = init_socket(sin, port);            /* mcaster socket */
print_sock_info (s, sin);

/* get info about local host */
gethostname(hostname, MAXGROUPNAME);
if ((hp = gethostbyname(hostname)) == 0) {
    perror("unable to get hostname");
    exit(-1);
}
print_hostent(hp);
strcpy(hostaddr, inet_ntoa(*(struct in_addr*)(hp->h_addr)));

/* initialize group address structure */
bzero((char*)&group_addr, sizeof(group_addr));
group_addr.sin_family = hp->h_addrtype;
group_addr.sin_port = htons(port);
group_addr.sin_addr.s_addr = inet_addr(hostaddr);
printf("Group Address:\n");
print_sock_addr(group_addr);

for (;;) { /*wait for incoming multicast messages */
    len = sizeof(from);
    sent = recvfrom(s, msgbuf, MAXMSGLEN, 0, &from, &len);
    /* extract messg from buffer */
    bzero((char *)&messg, sizeof(messg));
    bcopy(msgbuf, (char *)&messg, sizeof(messg));

    /* check type of received message */
    message_type = ntohs(messg.msg_type);
    if ((message_type==JOIN_GROUP)||(message_type==LEAVE_GROUP)) {
        member = from;
        /* all members receive mcasts on the MC_PORT */
```

```
                member.sin_port = MC_PORT;
                if (message_type == JOIN_GROUP)
                    cc = join_group(messg.group_name, member, NO_LOOP);
                else
                    cc = leave_group(messg.group_name, member);
                /* generate and send ACK for join or leave */
                form_messg(&messg, messg.group_name, 0, messg.group_view,
                              0, cc, messg.sender_gid, 0, 0, 0, 0, 0, 0);
                len = sizeof(messg);
                sendto(s, (char *)&messg, len, 0,
                        (struct sockaddr *)&from, sizeof(struct sockaddr));
            }
            else /* multicast unchanged message to group */
                mcast(s, msgbuf, from);
        }
} /* main */


/**********************************************************************
* Search group list for a group by its string name.  Return a pointer
* to the group before the desired group, for ease of removing the
* group, or NULL if not found.
**********************************************************************/
struct group *search_group_list (groupname)

char *groupname;
{
    struct   group *gp = group_list;
    int       notfound;

    if (group_list)  {                          /* non-empty group list */
    if (!(notfound = strcmp(gp->name, groupname)))        /* found 1st one */
        gp = last_group;                        /* set gp to element before 1st element */
        else /* not the 1st element - search for a match */
```

```
                while ((notfound = strcmp(gp->next->name, groupname))  &&
                            (gp->next != last_group))
                      gp = gp->next;
            if(!notfound) {      /* found! */
                return gp;
            }
      }     /*else group not found or empty group list */
      return NULL;
}  /*search_group_list */


/*************************************************************************
* Search member list of a group pointed to by gp for member "mbr".
* Return a pointer to the member before the desired group, for
* ease of removing the group, or NULL if not found.
*************************************************************************/
struct member *search_member_list (gp, mbr)

struct  group *gp;                    /* points to the desired group */
struct  sockaddr_in mbr;              /* member address to locate */
{
      struct  member *mp = gp->members;
      int       found;

      if (gp->members)  {             /* non-empty member list */
      if (found = addrcmp(mp->addr, mbr))     /* found 1st one */
            mp = gp->last;            /* set mp to element before 1st element */
            else /* not the 1st element - search for a match */
                while ((!(found = addrcmp(mp->next->addr, mbr))  &&
                            (mp->next != gp->last)))
                      mp = mp->next;
            if(found)
                return mp;
      }     /*else member not found or empty list */
```

```
    return NULL;
} /* search_member_list */


/***********************************************************************
* Add new group "groupname" to list of groups.  Return pointer to
* new group.
***********************************************************************/
struct group *add_group (groupname)

char *groupname;
{
    struct group *gp;

    /* create new group element */
    if (!(gp = (struct group *) malloc (sizeof(struct group))))
        return NULL;

    /* connect new group into list */
    if (!group_list)                    /* if group_list is empty */
        group_list = gp;
    else                                /* non-empty group_list */
        last_group->next = gp;
    last_group = gp;

    /* initialize new group element */
    strcpy(gp->name, groupname);
    gp->next = group_list;          /* point new last element to 1st element */
    gp->members = NULL;
    gp->last = NULL;

    return gp;
} /* add_group */
```

```
/***********************************************************************
* Add new member to member list of group pointed to by gp.  Return
* 0 if successful or negative value indicating reason for failure.
* mbr is a sockaddr_in structure with the new member's address.
* loop is used to control loopback of message to sender,
* 0 = no loopback, 1 = loopback.
***********************************************************************/
int add_member (gp, mbr, loop)

struct   group *gp;
struct   sockaddr_in mbr;
u_char loop;
{
      struct member *mp;

      /* create new member */
      if (!(mp = (struct member *) malloc (sizeof(struct member))))
            return -3;

      /*add to list */
      if (gp->members == NULL)          /* if member list is empty */
            gp->members = mp;
      else                              /* non-empty group_list */
            gp->last->next = mp; /* add to end of list */
      gp->last = mp;                    /* new element is last in list */

      /* initialize new member */
      mp->addr = mbr;
      mp->loop = loop;
      mp->next = gp->members;      /* point new last element to 1st element */

      return 0;
}  /* add_member */
```

```
/***********************************************************************
* Join a new member to a group named "groupname". The IP address of
* the new member is in mbr, a sockaddr_in struct. If the group exists,
* then the new member is added to the end of the member list. If the
* group does not exist, then a new group is first added to the group
* list, then the new member is added to the group. Loop is used to
* control loopback of messages to the sender: 0 = no loopback, 1 =
* loopback.
***********************************************************************/
int join_group (groupname, mbr, loop)

char    *groupname;
struct  sockaddr_in mbr;
u_char loop;
{
    struct group *gp;

    /* check if group exists */
    if (!(gp = search_group_list(groupname))) {  /* group doesn't exist */
        if (!(gp = add_group(groupname)))        /* so add a new group */
            return NEG_JOIN;
    }
    else {                                   /* group exists */
        gp = gp->next;                       /* set gp to desired group */
        if (search_member_list(gp,mbr))      /* member found in list */
            return DUP_MEMBER;
    }
    /* add new member to group */
    if (add_member(gp, mbr, loop) < 0)
        return NEG_JOIN;
    return JOIN_ACK;
} /* join_group */
```

```
/************************************************************************
* Remove group pointed to by gp->next from group_list.  The group has
* had all* of its members removed and is now ready to be removed from
* the list.  Return 0 if successful, neg. value if unsuccessful.
************************************************************************/
int remove_group (gp)

struct group *gp;                   /* gp points to group prior to desired group */
{
    struct group *rgp;              /* group to be removed */

    if (group_list == NULL)         /* empty list */
        return -6;

    if (group_list == last_group)   {  /* remove only member */
        free(group_list);
        group_list = last_group = NULL;
    }
    else  {                         /* remove desired group at gp->next */
        rgp = gp->next;             /* group to be removed */
        gp->next = rgp->next;
        if (group_list == rgp)      /* remove first group */
            group_list = rgp->next;
        if (last_group == rgp)      /* remove last group */
            last_group = gp;
        free(rgp);
    }
    return 0;
}  /* remove_group */


/************************************************************************
```

```
* Remove a member pointed to by mp->next in group pointed to by gp.
* mp points to member prior to one to be removed.  Returns 0 on success,
* neg. value on failure, and 1 if list is empty.
*****************************************************************/
int remove_member (gp, mp)

struct group *gp;
struct member *mp;
{
    int cc;
    struct member *rmp;

    if (gp->members == NULL)          /* no members to remove */
        return -7;

    if (gp->members == gp->last)  {   /*last member to remove */
        free(gp->members);
        gp->members = gp->last = NULL;
        cc = 1;
    }
    else  {                           /* remove desired member at mp->next */
        rmp = mp->next;               /* member to be removed */
        mp->next = rmp->next;
        if (gp->members == rmp)       /* remove first member */
            gp->members = rmp->next;
        if (gp->last == rmp)          /* remove last member */
            gp->last = mp;
        free(rmp);
        cc = 0;
    }
    return cc;
}  /* remove_member */
```

```
/***********************************************************************
* Allows a member "mbr" of a group to leave the group "groupname".
* If the member was the last one, the group is also removed from the
* group list.  Trying to remove a member that doesn't exist, or a
* member from a group that doesn't exist, return error codes.
* Successful removal of a member returns LEAVE_ACK code.
***********************************************************************/
int leave_group (groupname, mbr)

char   *groupname;
struct sockaddr_in mbr;
{
    struct group *gp, *dgp;
    struct member *mp;
    int      empty = 0;

    /* check if group exists */
    if (!(gp = search_group_list(groupname)))   /* group doesn't exist */
        return NO_GROUP;

    /* gp points to group prior to desired group */
    dgp = gp->next;                         /* set dgp to desired group */
    if (!(mp = search_member_list(dgp,mbr)))  /* member not found */
        return NO_MEMBER;

    /* mp points to member prior to desired member */
    empty = remove_member(dgp, mp);
    if (empty) remove_group(gp);            /* remove group if empty member list */
    return LEAVE_ACK;
}  /* leave_group */


/***********************************************************************
* Receives "message" and iteratively sends it to all members of
* the group "messg.group_name". Combines "message" with "from" address
* of sender in an extended format message, stored in messgbuf. The
* mcast is sent to the MC_PORT of each member. Loopback of message
* to sender is controlled by a comparison of the sender's address
* (from) with the loop field of each destination member. On success,
* returns a count of the number of destinations sent to, on failure
* returns a neg. value.
***************************************************************************/
int mcast(s, message, from)

int     s;                              /* fd for mcast socket */
char    *message;                       /* message to send */
struct sockaddr_in from;                /* sender of mcast */
{
        char    *messgbuf;
        struct message messg;
        int     len, msglen;
        struct group *gp;
        struct member *mp;
        int     count = 0;

        /* extract messg from buffer */
        bzero((char *)&messg, sizeof(messg));
        bcopy(message, (char *)&messg, sizeof(messg));

    /* form extended message */
        msglen = sizeof(messg) + (ntohs(messg.excl_list_len) +
                        ntohs(messg.subj_list_len))*2 + ntohs(messg.data_len);
        len = msglen + sizeof(from);

        /* allocate space for whole extended message */
        if (!(messgbuf = (char*) malloc (len)))
                return -1;
    /* copy message into outgoing buffer */
    bzero(messgbuf,len);
    bcopy((char *)&from, messgbuf, sizeof(from));
    bcopy(message, (messgbuf + sizeof(from)), msglen);

    /* find group */
    if(!(gp = search_group_list(messg.group_name)))       /*group not found */
        return -1;

    else {    /* group found.. gp points to group prior to desired one */
        gp = gp->next;                    /* get desired group */
        mp = gp->last;                    /* mp = tail of member list */
        /* set from port to MC_PORT for addrcmp search */
        from.sin_port = MC_PORT;
        if (mp != NULL)  {                /* non-empty list */
            do  {                         /* send to all */
                mp = mp->next;
                /* check for loopback to sender, then send to destination */
                if (!((addrcmp(from, mp->addr)) && (mp->loop == NO_LOOP))) {
                    sendto(s, messgbuf, len, 0, (struct sockaddr *)&(mp->addr),
                            sizeof(struct sockaddr));
                    count++;
                }
            }  while(mp != gp->last);
        }
        free(messgbuf);
        return count;
    }
}  /* mcast */


/*************************************************************************
* Print group list.
*************************************************************************/
```

```
void print_group_list()

{
     struct group *gp = last_group;

     printf("Group_List:\n");
     if (gp)                                   /* non-empty group list */
          do {
               gp = gp->next;
               printf("%s\n",gp->name);
          } while (gp != last_group);
     else printf("Empty group_list\n");
} /* print_group_list */


/**********************************************************************
* Print member list of a group pointed to by gp.
**********************************************************************/
void print_member_list (gp)

struct group *gp;                             /* points to the desired group */
{
     struct member *mp = gp->last;

     printf("Member_List for group %s:\n", gp->name);
     if (mp)                                   /* non-empty member list */
          do {
               mp = mp->next;
               print_sock_addr(mp->addr);
               printf("loop = %d\n\n", mp->loop);
          } while (mp != gp->last);
     else printf("Empty member_list\n");
} /* print_member_list */
```

```
/**********************************************************************
* MSERVER.C ver 1.0
* Membership Server program.
* At present, includes:
*          join & leave multicast group
*          message sending & receiving
*          pairwise monitoring
*          working on change processing sequence
* Member failures are logged to file "failures".
*
* Written by Dave Neely, April 1994.
* Modified: 4/25/94
**********************************************************************/

#include "msutil.c"

struct    sockaddr_in to, from;          /* general use address structures */
struct    hostent *hp;                   /* host entity struct */
u_short   mygid, cw, ccw;                /* mserver group IDs */
struct    timeval tp;                    /* for time stamps */
struct    servent tzp;                   /* for timing */
struct    timeval recv_timeout;          /* select() receive timeout */
struct    timeval query_timeout;         /* timeout for monitoring query */
struct    timeval reply_timeout;         /* timeout for monitoring reply */
struct    timeval messg_timeout;         /* timeout for response message */
struct    timeval ACK_timeout;           /* timeout for ACK message */
FILE      *fp;                           /* file to record mserver failures */
int       MCASTER;

main (argc, argv)

int   argc;
char *argv[];
{
```

```
u_short  message_type;
u_short  GV, gsize;                    /* group view no. and group size */
int      len, i, cc;
int      recd, sent;
int      retries = 2;                  /* monitoring retries for no reply */
char     groupname[MAXGROUPNAME];
char     hostname[MAXGROUPNAME];
char     IPaddr[16];
char     groupaddr[16];
struct   table_entry core_table[MAXTBLSIZE];   /* core-set state table */
u_short  coordinator;                  /* for change processing */
long     authentication = 0x7fffffff;
struct   gid_entry *excl_list, *subj_list;     /* lists of mserver gids */
u_short  excl_list_len, subj_list_len;

if (argc != 8)  {
     printf("usage: mserver groupname groupIPaddr");
     printf(" mygid cw_gid cw_addr ccw_gid ccw_addr\n");
     exit(-1);
}
/**** Note: no error checking on arguments ****/
strcpy(groupname, argv[1]);
strcpy(groupaddr, argv[2]);
mygid = atoi(argv[3]);
cw  = atoi(argv[4]);
ccw = atoi(argv[6]);

/*get info about local host */
gethostname(hostname, MAXGROUPNAME);
if ((hp = gethostbyname(hostname)) == 0) {
     perror("unable to get hostname");
     exit(-1);
}
print_hostent(hp);
```

```
        strcpy(IPaddr, inet_ntoa(*(struct in_addr*)(hp->h_addr)));

        /* initialize core_table */
        bzero((char *)core_table, sizeof(core_table));
        core_table[mygid].addr = inet_addr(IPaddr);
        core_table[mygid].cw   = cw;
        core_table[mygid].ccw = ccw;
        core_table[cw].addr     = inet_addr(argv[5]);
        core_table[ccw].addr   = inet_addr(argv[7]);
        core_table[cw].ccw     = mygid;
        core_table[ccw].cw     = mygid;

        /* intialize gid lists */
        excl_list_len = subj_list_len = 0;
        excl_list = subj_list = NULL;

        /* determine if IP multicast or MCASTER will be used */
#ifndef IFF_MULTICAST
        MCASTER = 1;
#else   /* check that group address is in Class D range */
        if ((inet_addr(groupaddr) < inet_addr("224.0.0.255")) ||
                (inet_addr(groupaddr) > inet_addr("239.255.255.255")))
                MCASTER = 1;
#endif

        printf("Mserver\n\n");
        printf("mygid: %d, cw: %d, ccw: %d\n", mygid,
        core_table[mygid].cw, core_table[mygid].ccw);
        printf("my ");
        print_in_addr(&(core_table[mygid].addr));
        printf("cw ");
        print_in_addr(&(core_table[cw].addr));
        printf("ccw ");
        print_in_addr(&(core_table[ccw].addr));
```

```
   /* initialize general purpose "ms" & mcaster "mc" sockets */
   ms = init_socket(sin, htons(MS_PORT));
   print_sock_info (ms, sin);
   mc = init_socket(mcsin, htons(MC_PORT));
   print_sock_info (mc, mcsin);

/* initialize timeouts */
   /* recv_timeout is an absolute period, not referenced to current time */
   recv_timeout.tv_sec  = T_RECV / SEC;          /* set seconds */
   recv_timeout.tv_usec = T_RECV % SEC;          /* set useconds */
   set_timeout(&query_timeout, T_QUERY);         /* set timer for next query */
   set_timeout(&reply_timeout, MAXTIME);         /* reset timer for reply */
   set_timeout(&messg_timeout, MAXTIME);         /* reset messg timer */
   set_timeout(&ACK_timeout, MAXTIME);           /* reset ACK timer */

   cc = join_mcast_group(groupname, groupaddr);
   switch (cc)  {
         case JOIN_ACK :
               printf("Group %s joined.\n", groupname);
               break;
         case DUP_MEMBER :
               printf("Unable to join group %s: duplicate member\n", groupname);
               exit(-1);
               break;
         case NEG_JOIN :
               printf("Unable to join group %s.\n", groupname);
               exit(-1);
               break;
         default :
               printf("Invalid code returned during group join.\n");
               exit(-1);
   }
```

```
for (;;) {      /* begin main loop */
      len = sizeof(from);

      /* check if message ready */
      if ((recd = recv_messg(ms, mc, &messg, &from, recv_timeout)) > 0) {
            printf("MESSAGE RECEIVED:\n");
            printf("%d bytes received:\n", recd);
            print_messg(messg);
            printf("from:\n");
            print_sock_addr(from);
            message_type = ntohs(messg.msg_type);

            /* select appropriate action for received message type */
            switch (message_type)  {
                  /* mserver set message types */
                  case QUERY:
                        /* check if query from cw neighbor in this group */
                        if ((!(strcmp(messg.group_name, groupname))) &&
                               (from.sin_addr.s_addr == core_table[cw].addr))  {
                              /* then send reply */
                              form_messg(&messg, groupname,0,0, mygid, REPLY,
                                                   cw,0,0,0,0,0,0);
                              len = sizeof(messg);
                              if ((sent = send_messg(ms, messg, from)) != len)  {
                                    printf("error in message length sent\n");
                              }
                        }
                        break;
                  case REPLY:
                        /* check if query from ccw neighbor in this group */
                        if ((!(strcmp(messg.group_name, groupname))) &&
                               (from.sin_addr.s_addr == core_table[ccw].addr))  {
                              /* then reset reply and query timers, and # retries */
                              printf("REPLY rec'd from %d, resetting timers\n",
            ntohs(messg.sender_gid));
            set_timeout(&reply_timeout, MAXTIME);
            set_timeout(&query_timeout, T_QUERY);
            retries = 2;
        }
        break;
    /* mserver INITIATE message types */
    case M_JOIN:            printf("    M_JOIN\n");           break;
    case M_LEAVE:           printf("    M_LEAVE\n");          break;
    case M_SPLIT:           printf("    M_SPLIT\n");          break;
    case M_MERGE:           printf("    M_MERGE\n");          break;
    case M_ADD_PARENT:      printf("    M_ADD_PARENT\n");     break;
    case M_DEL_PARENT:      printf("    M_DEL_PARENT\n");     break;
    case M_STATS_S:         printf("    M_STATS_S\n");        break;
    case M_STATS_L:         printf("    M_STATS_L\n");        break;
    case M_FAIL:            printf("    M_FAIL\n");           break;
    case M_MULTI_FAIL:      printf("    M_MULTI_FAIL\n");     break;
    case M_COORD_FAIL:      printf("    M_COORD_FAIL\n");     break;
    /* change sequence message types */
    case ACK:               printf("    ACK\n");              break;
    case COMMIT:            printf("    COMMIT\n");           break;
    case WAIT:              printf("    WAIT\n");             break;
    case MSG_QUERY:         printf("    MSG_QUERY\n");        break;
    case INIT:              printf("    INIT\n");             break;
    /* external physical requests to core-set */
    case M_JOIN_REQ:        printf("    M_JOIN_REQ\n");       break;
    case M_LEAVE_REQ:       printf("    M_LEAVE_REQ\n");      break;
    case M_SPLIT_REQ:       printf("    M_SPLIT_REQ\n");      break;
    case M_MERGE_REQ:       printf("    M_MERGE_REQ\n");      break;
    case M_ADD_PAR_REQ:     printf("    M_ADD_PAR_REQ\n");break;
    case M_DEL_PAR_REQ:     printf("    M_DEL_PAR_REQ\n"); break;
    case M_STATS_S_REQ:     printf("    M_STATS_S_REQ\n");  break;
    case M_STATS_L_REQ:     printf("    M_STATS_L_REQ\n");  break;
    /* application group INITIATE message types */
```

```
            case A_JOIN:            printf("    A_JOIN\n");          break;
            case A_LEAVE:           printf("    A_LEAVE\n");         break;
            case A_SPLITQ:          printf("    A_SPLIT\n");         break;
            case A_MERGE:           printf("    A_MERGE\n");         break;
            case A_STATS_S:         printf("    A_STATS_S\n");       break;
            case A_STATS_L:         printf("    A_STATS_L\n");       break;
            case SUBMIT:            printf("    SUBMIT\n");          break;
            case DIRECT:            printf("    DIRECT\n");          break;
            /* application group request message types */
            case A_JOIN_REQ:        printf("    A_JOIN_REQ\n");      break;
            case A_LEAVE_REQ:       printf("    A_LEAVE_REQ\n");     break;
            case A_SPLIT_REQ:       printf("    A_SPLIT_REQ\n");     break;
            case A_MERGE_REQ:       printf("    A_MERGE_REQ\n");     break;
            case A_STATS_S_REQ:     printf("    A_STATS_S_REQ\n");   break;
            case A_STATS_L_REQ:     printf("    A_STATS_L_REQ\n");   break;
            /* mcaster message types */
            case JOIN_GROUP:        printf("    JOIN_GROUP\n");      break;
            case LEAVE_GROUP:       printf("    LEAVE_GROUP\n");     break;
            case JOIN_ACK:          printf("    JOIN_ACK\n");        break;
            case DUP_MEMBER:        printf("    DUP_MEMBER\n");      break;
            case NEG_JOIN:          printf("    NEG_JOIN\n");        break;
            case LEAVE_ACK:         printf("    LEAVE_ACK\n");       break;
            case NO_GROUP:          printf("    NO_GROUP\n");        break;
            case NO_MEMBER:         printf("    NO_MEMBER\n");       break;
            case NEG_LEAVE:         printf("    NEG_LEAVE\n");       break;
            default:                printf("    %d\n", ntohs(messg.msg_type));
        } /* switch */
    }  /* if (recd > 0) */
    if (recd < 0)
        printf("error in message rec'd\n");

    /* check timeouts */
    if (timed_out(query_timeout))  {  /* time to send a new query */
        /* reset QUERY timer */
        set_timeout(&query_timeout, T_QUERY);
        /* set REPLY timer */
        set_timeout(&reply_timeout, T_REPLY);
        form_messg(&messg, groupname,0 ,0 , mygid, QUERY, ccw, 0,0,0,0,0,0);
        len = sizeof(messg);
        to.sin_family = AF_INET;
        to.sin_port = htons(MS_PORT);
        to.sin_addr.s_addr = core_table[ccw].addr;
        if ((sent = send_messg(ms, messg, to)) != len)  {
            printf("error in message length sent\n");
            exit(-1);
        }
    }  /* query_timeout */

    if (timed_out(reply_timeout))  {        /* retry or note failure */
        /* reset QUERY timer */
        set_timeout(&query_timeout, T_QUERY);
        if ((retries--) < 0)  {             /* then ccw is failed */
            retries = 2;                    /* reset retry counter */
            /* log an entry in failures file */
            gettimeofday(&tp, &tzp);
            if (fp = fopen("failures", "a"))  {
                fprintf (fp, "Member %d is detected failed by %d at %d sec.\n\n",
                ccw, mygid, tp.tv_sec);
                fclose(fp);
            }
            /* At this point, would want to start fail processing */
            set_timeout(&reply_timeout, MAXTIME); /* reset reply timer */
        }
        else  {
            /* set REPLY timer */
            set_timeout(&reply_timeout, T_REPLY);
            form_messg(&messg, groupname,0,0, mygid, QUERY,
                            ccw,0,0,0,0,0,0);
                len = sizeof(messg);
                to.sin_family = AF_INET;
                to.sin_port = htons(MS_PORT);
                to.sin_addr.s_addr = core_table[ccw].addr;
                if ((sent = send_messg(ms, messg, to)) != len)  {
                     printf("error in message length sent\n");
                     exit(-1);
                }
           }
      }  /* reply_timeout */
 }  /* main for loop */
}
```

LIST OF REFERENCES

[1] K. P. Birman, "The process group approach to reliable distributed computing," Technical Report TR91-1216, Cornell University Computer Science Department, Ithaca, NY, July 1991.

[2] F. Cristian, R. Dancey, and J. Dehn, "Fault-tolerance in the advanced automation system," *The 20th International Symposium on Fault-tolerant Computing*, pp. 6-17, June 1990.

[3] L. L. Peterson, N. Buchholz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Transactions on Computer Systems*, vol. 7, no. 3, pp. 217-246, August 1989.

[4] D. Powell, M. Chereque, D. Drackley, "Fault-tolerance in Delta-4," *Operating Systems Review*, vol. 25, no. 2, pp. 122-125, April 1991.

[5] F. Cristian, "Agreeing on who is present and who is absent in a synchronous distributed system," *Proceedings of the 18th International Conference on Fault Tolerant Computing, Tokyo, Japan,* pp. 206-211, 1988.

[6] S. Deering, "Host extension for IP Multicasting," Memo from Network Working Group, Stanford University, August 1989.

[7] S. Deering, "IP Multicasting Extensions for 4.3BSD UNIX and related systems (MULTICAST 1.2 Release)," RFC 1112, Stanford University, August 1989.

[8] R. Braudes and S. Zabele, "Requirements for multicast protocols," Memo from Network Working Group, TASC, May 1993.

[9] A. M. Ricciardi and K. P. Birman, "Using process groups to implement failure detection in asynchronous environments," *ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pp. 341-353, August 1991. Also available as TR91-1188, Dept. of Computer Science, Cornell University.

[10] R. D. Schlichting and F. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 222-238, August 1983.

[11] K. P. Birman and T. A. Joseph, "Reliable communications in the presence of failures," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 47-76, February 1987.

[12] B. Rajagopalan, "A mechanism for scalable concast communication," *Computer Communications*, vol. 16, no. 8, pp. 484-493, August 1993.

[13] F. Jahanian and W. Moran Jr., "Strong, weak and hybrid group membership," *Proceedings of the Second Workshop on the Management of Replicated Data, Monterey, California*, pp. 34-38, November 1992. Also available as Technical Report RC 18040 (79173) 5/28/92, IBM Research Division, T. J. Watson Research Center, 1992.

[14] F. Jahanian, S. Fakhouri, and R. Rajkumar, "Processor group membership protocols: Specification, design and implementation," paper presented at Symposium on Reliable Distributed Systems, October 1993.

[15] J. M. Chang and N. F. Maxemchuk, "Reliable broadcast protocol," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 251-273, August 1984.

[16] S. A. Bruso, "A failure detection and notification protocol for distributed computing systems," *Proceedings of the 5th International Conference on Distributed Computing Systems*, pp. 116-123, May 1985.

[17] A. El Abbadi, D. Skeen, and F. Cristian, "An efficient fault-tolerant protocol for replicated data management," *Proceedings of the 4th ACM Symposium on Principles of Database Systems*, pp. 215-229, 1985.

[18] P. Verissimo and J. A. Marques, "Reliable broadcast for fault-tolerance on local computer networks," *Symposium on Reliable Distributed Systems*, pp. 54-63, October 1990.

[19] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala, "Membership algorithm for asynchronous distributed systems," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 480-488, 1991.

[20] S. Mishra, L. L. Peterson, and R. D. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs," Technical Report TR 91-32, Department of Computer Science, University of Arizona, 1991.

[21] J. Auerbach, M. Gopal, M. Kaplan, and S. Kutten, "Multicast group membership management in high speed wide area networks," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 231-238, 1991.

[22] R. A. Golding and D. D. E. Long, "The performance of weak-consistency replication protocols," Technical Report ucsc-crl-92-30, Department of Computer Science, University of California at Santa Cruz, July 1992.

[23] P. D. Ezhilselvan and R. de Lemos, "A robust group membership algorithm for distributed real-time systems," *Proceedings of the Real-Time Systems Symposium*, pp. 173-179, 1990.

[24] K. H. Kim, H. Kopetz, K. Mori, E. H. Shokri, and G. Gruensteidl, "An efficient decentralized approach to processor-group membership maintenance in real-time LAN systems: The PRHB/ED scheme," *Symposium on Reliable Distributed Systems*, pp. 74-83, 1992.

[25] L. Rodrigues, P. Verissimo, and J. Rufino, "A low-level processor group membership protocol for LANs," Technical Report Oct. 1992, Technical University of Lisbon, Portugal, INESC, 1992.

[26] S. Levi and A. K. Agrawala, *Fault Tolerant System Design*, McGraw-Hill, New York, New York, 1994.

[27] J. Misra and K. M. Chandy, *Parallel Program Design - A Foundation*, Addison-Wesley, New York, New York, 1989.

[28] G. Andrews, *Concurrent Programming - Principles and Practice*, Benjamin/ Cummings, Redwood City, California, 1991.

[29] D. Comer and D. Stevens, *Internetworking with TCP/IP, Vol. I: Principles, Protocols, and Architecture*, 2nd edition, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[30] G. M. Lundy and R. E. Miller, "Specification and analysis of a data transfer protocol using systems of communicating machines," *Distributed Computing*, vol. 5, no. 3, pp. 145-157, December 1991.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4.	Professor Shridhar B. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	3
5.	Professor Gilbert M. Lundy, Code CS/Lu Department of Computer Science Naval Postgraduate School Monterey, California 93943-5118	1
6.	LT David S. Neely P.O. Box 63 Arnold, California 95223-0063	1