DESIGN AND IMPLEMENTATION OF A MEMBERSHIP SERVER AND ITS APPLICATION INTERFACE

by

John Kostrivas

March 10, 1995

Thesis Advisor: Shridhar B. Shukla

Approved for public release; distribution is unlimited.
**REPORT DOCUMENTATION PAGE**

**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.**

<table>
<thead>
<tr>
<th>1. AGENCY USE ONLY (Leave blank)</th>
<th>2. REPORT DATE</th>
<th>3. REPORT TYPE AND DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>March 1995</td>
<td>Master's Thesis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESIGN AND IMPLEMENTATION OF A MEMBERSHIP SERVER AND ITS APPLICATION INTERFACE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5. AUTHOR(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Kostrivas</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naval Postgraduate School</td>
</tr>
<tr>
<td>Monterey CA 93943-5000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. PERFORMING ORGANIZATION REPORT NUMBER</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>10. SPONSORING/MONITORING AGENCY REPORT NUMBER</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>11. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12a. DISTRIBUTION/AVAILABILITY STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approved for public release; distribution unlimited</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12b. DISTRIBUTION CODE</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>13. ABSTRACT (Maximum 200 words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None of the existing membership protocols have all the properties needed to be reliable and fault-tolerant. Therefore, the goal of this work is to implement two major components of a group Membership Service protocol which will provide distributed applications the necessary fault tolerance, reliable communications and consistent group views among all members. These protocols must operate on top of a usually unreliable and best effort network such as the Internet. The first component implements a multicast emulator, to emulate IP multicasting communication over a mixture of multicast-capable and unicast capable local area networks (LANs). The second component implements a membership server that maintains the group memberships using the Membership Service protocol. These components are implemented as programs and then verified to be faithful to the specifications through extensive testing of all possible paths through the program (all combinations of scenarios).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. SUBJECT TERMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group membership, multicast, decentralized, change protocols, network partitions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. PRICE CODE</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>17. SECURITY CLASSIFICATION OF REPORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18. SECURITY CLASSIFICATION OF THIS PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19. SECURITY CLASSIFICATION OF ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20. LIMITATION OF ABSTRACT</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Standard Form 298 (Rev. 2-89)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prescribed by AS/Ed. 239-18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NSN 7540-01-280-5500</th>
</tr>
</thead>
</table>
Approved for public release; distribution is unlimited

DESIGN AND IMPLEMENTATION OF A MEMBERSHIP SERVER
AND ITS APPLICATION INTERFACE

John Kostrivas
Lieutenant JG, Hellenic Navy
Hellenic Naval Academy, 1987

Submitted in partial fulfillment of the requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March, 1995

Author:

John Kostrivas

Approved by:

Shridhar B. Shukla, Thesis Advisor
Gilbert M. Lundy, Second Reader

Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

Ted G. Lewis, Chairman
Department of Computer Science
ABSTRACT

None of the existing membership protocols have all the properties needed to be reliable and fault-tolerant. Therefore, the goal of this work is to implement two major components of a group Membership Service protocol which will provide distributed applications the necessary fault tolerance, reliable communications and consistent group views among all members. These protocols must operate on top of a usually unreliable and best effort network such as the Internet. The first component implements a multicast emulator, to emulate IP multicasting communication over a mixture of multicast-capable and unicast capable local area networks (LANs). The second component implements a membership server that maintains the group memberships using the Membership Service protocol. These components are implemented as programs and then verified to be faithful to the specifications through extensive testing of all possible paths through the program (all combinations of scenarios).
# TABLE OF CONTENTS

I. INTRODUCTION .......................................................................................... 1

II. THE ARCHITECTURE OF THE MEMBERSHIP SERVICE ......................... 3
   A. COMPONENTS OF THE MS ............................................................... 3
      1. Membership Servers And Member Interfaces ............................ 3
      2. Failures, Partitions, And Dynamic Reformation ...................... 4
      3. Change-Processing Core-Set ..................................................... 5
      4. Lan Mserver Monitoring ............................................................ 6
      5. Forming The Hierarchy ............................................................... 6
   B. SUPPORT FOR APPLICATION GROUPS ........................................... 7
      1. Consistency .............................................................................. 7
      2. Naming .................................................................................... 8
      3. Membership Scope Control .................................................... 8
      4. Member Interfaces ................................................................. 9
      5. Application Group Change Processing ................................. 9

III. PROTOCOL DESCRIPTIONS .................................................................. 11
   A. PROTOCOL FUNCTIONS ................................................................ 11
      1. Types Of Changes .................................................................. 12
         a. Requests ........................................................................ 12
         b. Failures ......................................................................... 12
         c. Dynamic Reconfigurations ............................................... 13
      2. Ordering And Priority Of Change Processing ....................... 13
   B. CHANGE PROTOCOL ..................................................................... 15
   C. CHANGE PROTOCOL WHEN COORDINATOR FAILS ..................... 15
   D. PARTITION RESOLUTION PROTOCOL ......................................... 17

IV. IMPLEMENTATION ................................................................................. 21
   A. MULTICAST EMULATOR (MCASTER) ............................................ 21
      1. Algorithm Design ................................................................. 21
      2. Code Description ................................................................. 25
         a. Internal Group Lists ......................................................... 26
         b. Mcaster Specific Messages ............................................. 27
   B. BASIC MESSAGE FUNCTIONS .................................................... 28
      1. Function Receive_Msg .......................................................... 28
      2. Function Recv_Messg ......................................................... 29
      3. Function Form_Messg ......................................................... 30
      4. Function Send_Messg ......................................................... 31
   C. MEMBERSHIP SERVER (MSERVER) .............................................. 32
      1. Algorithm Design ................................................................. 32
ACKNOWLEDGMENT

This thesis is devoted to my father Dionisios Kostrivas for whom, higher education has been his life drive.

It is also devoted to all my professors and especially to my advisor Professor Shridhar Shukla, who helped me to materialize my dream.
I. INTRODUCTION

Distributed applications require co-operation among their groups spread out in a network. These groups sometimes change dynamically and the membership may be based on voluntary, as well as involuntary, actions. As such applications proliferate on a typical organization's wide area network (WAN), access to a Membership Service (MS) to manage and administer the membership information of individual groups becomes necessary.

The type of membership information required depends upon the nature of the cooperation to be achieved by the members of the client groups. Examples of membership-related information are group size, members' identities, their geographical and organizational distribution, and a history of membership changes. This information could be provided by the membership service through a series of functions that in turn could be used to build distributed applications.

A lot of different approaches have been presented [13, 14, 15, 16, 17, 9, 18, 19, 12, 20, 5, 21, 22, 23] for a membership service. None of them includes all the necessary features to provide total scalability over WANs. Typically, they do not provide a range of membership services and, in many cases, assume network properties that are not representative of today's WANs.

The MS described in [28] assumes a "best-effort" network such as the Internet, is scalable with respect to the size/distribution/number of groups, uses network-level multicasting when available, employs a decentralized protocol with provably minimal number of phases for committing changes, and offers different qualities-of-service (QoS).

This thesis presents the algorithms and actual code of the MS implementation started in [28]. Two basic components of the MS, a multicast emulator (mcaster) and a membership server (mserver) were implemented and tested. In this process some of the membership protocols of [28] were refined. In Chapter II, a brief description of the MS architecture described in [28, 29] is given. The protocol is described in Chapter III. In Chapter IV, the implementation of the mcaster and mserver is explained. Chapter V gives
the conclusions and future work. Finally, the Appendix describes how to compile, run
and test the MS code written for this thesis.
II. THE ARCHITECTURE OF THE MEMBERSHIP SERVICE

The key to a scalable MS is a decentralized, hierarchical architecture, designed to exploit the existing physical topology of subnetworks, networks, and internetworks upon which the distributed application process groups that the MS supports will be running. This chapter summarizes the structure and composition of the physical hierarchy of the MS and how this architecture supports application process groups, as it is given in [28, 29].

A. COMPONENTS OF THE MS

1. Membership Servers And Member Interfaces

The MS is comprised of two primary entities: membership servers (mserver) organized in a physical tree hierarchy and member interfaces (MI) that represent the leaves of the tree. The mserver are primarily responsible for processing changes and providing information to the members of the physical hierarchy as well as the application process groups using the MS. 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 to access 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 supports numerous application groups and numerous individual member processes from each application group.

Figure 1 illustrates an example logical hierarchy of mserver, 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. The logical hierarchy shown in Figure 1 corresponds to the physical topology of networks and computers. It shows 11 departmental LANs served by as many mserver. The 11 mserver form 3 groups at the building level. At the next level (backbone) 3 mserver form a group to
serve the entire campus. This hierarchy of *mservers* groups forms the MS infrastructure. Figure 2 shows the messages that are exchanged between LAN *mservers*, Member Interfaces and application group members.

The configuration of the *mservers* and MIs is expected to be semi-static, normally changing only when additions and deletions to the physical topology are made. The system administrator will assign appropriate names for each set of *mservers* at each level. If network-level multicasting is available, the administrator could join each set into a multicast group for efficient communication.

![Logical MS Hierarchy](Image)

Figure 1: Logical MS Hierarchy

### 2. Failures, Partitions, And Dynamic Reformation

*Mserv* failures and network partitions lead to a dynamic reconfiguration of the physical structure of the MS, with the surviving *mservers* and MIs automatically reforming into partitioned sets. Perceived *mserver* failures represent virtual partitioning of the network into one or more subsets of the original set of *mservers*. Each partitioned subset corresponds to the subtree of the physical hierarchy in a single piece of the partition. This subtree corresponds to all of the *mservers* which are still able to communicate over the non-partitioned network. Each partition of the MS reforms and continues to function, providing service to all application process groups which have members still existing in the partition. The application process groups which span the partitioned network will
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 administratively or automatically be reformed to the original configuration. Once the physical hierarchy is restored, the surviving application groups will also be reformed, as per the Quality-of-Service (QoS) related to partition resolution chosen by the MS user at start up time. Partitions can be detected by mservers through monitoring, as described later.

3. Change-Processing Core-Set

The group of mservers at each level in the hierarchy is called a change-processing core-set with respect to a particular application group when it is designated to be responsible for processing all membership change requests submitted by members of that application group. Every such set is also responsible for enacting changes in the physical hierarchy immediately below it. The change processing involves reaching agreement amongst all mservers in the core-set about the change submitted and propagating this change back to the application or physical hierarchy group members, who are then guaranteed to have a new view of the changed group membership.

![Diagram of LAN mserver and MI connections](image)

Figure 2: Messages sent by LAN Members and Member Interfaces
For each membership change request submitted to an mserver group, 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 mserver group. The fact that the coordinator is not a fixed member of the mserver group, but instead varies from change to change, is a powerful feature of the MS.

4. 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 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 and announce the perceived failure. Note that this monitoring emphasizes collection of status of individual MI’s on the LAN. This is to be distinguished from the monitoring done by IGMP which detects if there exists a (any) member on the LAN [6].

5. Forming The Hierarchy

The final organization of mservers and MIs involves forming the hierarchy of the sets of mservers that cooperate for monitoring and change processing, with the MIs at the leaf level. As shown in Figure 1, each mserver in the hierarchy has either a set of children mservers or MIs. All mservers and MIs also have a parent mserver, except the mservers at the highest level of the hierarchy. Each mserver above the lowest level in the hierarchy has a dual membership in the "child-set" as well as the original peer group of mservers.

Having the parent mserver as a member of the child-set has two primary advantages. First, the parent mserver is part of monitoring the child 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. SUPPORT FOR APPLICATION GROUPS

The MS is responsible for managing the membership of the application groups and providing services to the application groups with features as described below.

1. 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 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. Using this guarantee of consistent membership at all members, the application can expect that members with the same group view number have seen the same sequence of membership changes and 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 [11, 27]. 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.
2. 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 backbone core-set from Figure 1, the name backbone 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 a successful group creation is reported to the calling application. When new application members 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.

3. 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 that of 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.

4. Member Interfaces

The MI's accept application membership change and information requests from application processes and submit these changes to the mserv hierarchy for processing. When the change or information data is returned, the MI passes the data to the requesting member processes.

The MI, running on an individual host computer is capable of interfacing multiple application groups, each with multiple members, with the LAN mserv and 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.

5. 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 mserv change-processing procedure, resulting in all core-set mservers committing the change. Each core-set mserv then reliably relays the
change directive down the hierarchy to the MI, and then to the requesting application process. Timeouts and retries are again used to detect failures and partitions.

In this chapter the architecture of the MS was briefly discussed and its components (*mservers* and MIs) along with the MS support for the applications were described. The next chapter describes the MS protocol.
III. PROTOCOL DESCRIPTIONS

This chapter describes in brief the protocol used in MS. More detailed analysis of the MS protocol along with correctness arguments can be found in [28, 29].

A. PROTOCOL FUNCTIONS

As described in [28], the basic change-processing protocol 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 group 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 expected 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 group.

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. This is described later.
1. Types Of Changes

There are three primary types of membership changes processed by a group of mserver: requests, failures, and dynamic reconfigurations.

a. Requests

Requests are voluntary, planned membership changes, submitted to the group for processing by an application process or system administrator. Change requests for the MS physical hierarchy may be to Join to a mserver group, Leave a mserver group, Split a mserver group to form two new ones, Merge two mserver groups to form one, Add_parent to add a parent mserver to the mserver group and Del_parent to remove a parent from the mserver group. Physical change requests are multicast to a specific group in the hierarchy by a system configuration call, usually invoked by a system administrator during manual configuration of the MS hierarchy. Application group change requests are submitted to the resident MI process on the host computer by the application user. The MI then propagates the request to the group mserver above it in the hierarchy. The receiving group 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 group mserver are queued and processed according to their priority. Multiple failures queued at a group mserver are
processed all at once, in a "batched" manner. This greatly reduces the time required to reform physical mserver groups or application groups.

**c. Dynamic Reconfigurations**

The final type of changes are the result of automatic actions taken by mserver groups. This 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.

**2. Ordering And Priority Of Change Processing**

A key issue associated with processing membership changes is the ordering of changes committed by the mserver group. As previously described, to guarantee consistent ordering of membership changes at all mservers in the group, 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 group at one time. Each receiving or detecting
\textit{mserver} in the group will attempt to become the group 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 group has
reached a consistent and uniform decision on the current change to process.

To resolve these virtually simultaneous changes and select only one change to be
processed, a priority scheme is used. This scheme uses the type of change and the unique
group id (rank) of the subject of the change to decide which change will be processed by
the group. The subject of a change refers to the member whose membership status has
changed. The highest priority is given to any current change being processed by the
group; that is, a change which is in progress at an \textit{mserver} (i.e. an \textit{ack} to the \textit{initiate} has
been sent). It is essential that such a change progresses to completion at all group \textit{mser-
vers}; otherwise, the possibility of inconsistent membership views exists if some \textit{mser-
vers} commit the change while others do not. The next lower priority is that physical hierarchy
changes always have priority over application group changes. This is because it is impor-
tant to ensure a complete and whole MS infrastructure 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 or age of the subject of the
change in the group. The only exceptions to this rule are for the failure of the coordina-
tor of the current change or a \textit{Join}. The failure of the coordinator of a change in pro-
gress, has priority over otherwise equal status changes. A newly joining \textit{mserver} or
member will not have an associated rank until after the join is completed. For this rea-
son, the network address of the joining member is used instead of a rank number to de-
cide priority among \textit{Joins}. The final rule used to determine the priority of virtually
simultaneous changes is applicable when changes are submitted to the core-set by differ-
ent application groups with identical subject rankings in each group. In this case, a tie-
breaker is needed, and the ranks of the coordinators in the group are used to decide which
change will be processed. Various scenarios with respect to virtually simultaneous changes are described in [29].

B. CHANGE PROTOCOL

The basic change-processing protocol consists of two phases: the Initiate and Commit phases. A timeline for this protocol is shown in Figure 3. In the Initiate phase, the coordinator multicasts an Initiate message to all mservers in the group. The group mservers respond with ACKs, acknowledging reception of the Initiate message. When the coordinator has received all the acknowledgments, the second phase of the protocol begins with the coordinator sending the Commit message. This message indicates to members of the group that it is safe to commit the change. Phase I is achieved through a reliable schema. The coordinator sends the Initiate message a predetermined number of times if one or more group mservers do not reply. After that, it assumes that the mserv-er(s) that did not reply has (have) failed.

![Figure 3: Basic Two Phase Change Protocol](image)

C. CHANGE PROTOCOL WHEN COORDINATOR FAILS

The two phase protocol is not sufficient in case coordinator of a change fails while processing a change. As shown by Riccardi and Birman [9], a three phase protocol is required. After the coordinator of the current change fails, and its failure is detected by a member of the group, a three phase protocol is initiated, with the election of the new coordinator as the first phase. Figure 4(a) illustrates the three phase election and change processing protocol. In the election phase only those members that have finished phase one of the original change protocol participate.
When the new coordinator is elected, it knows the status of each member with respect to the change, due to a status broadcast during the election phase. If at least one mserver has finished the change (committed) it means that the old (and detected failed) coordinator had already collected all ACKs and had started the Commit phase. So the new coordinator, knowing that phase one was completed, can continue with the final phase of the change, instead of restarting phase one. This compressed three phase protocol is shown in Figure 4(b).

![Diagram](image)

(a) ![Diagram](image)

(b) Figure 4: Three Phase (Election and Change) and Compressed Three Phase Protocol

A simplified algorithm for the two phase and three phase algorithm is shown in Figure 5. Only the important arguments are shown. All sends and receives of messages are done with timeouts. This way the protocol does not block and takes appropriate actions in case of no response. In line 5, the term "reliably" implies that a message is sent and specific answers are expected from all members in a specific time interval. If some or all of them do not reply, a number of retries are attempted. After the retries and the last timeout expire, the sender assumes the non responding mservers to have failed. In line 11, the same function is called recursively to process the failure(s) of the member(s) that did not reply. In line 8, the second phase is executed by sending a commit message. Then the coordinator has to make the change to its internal state. Lines 13 through 25 are devoted to the non coordinator. Since this code is executed everywhere, it contains both cases (coordinator, non-coordinator), and the if statement in line 3 decides which part of the code should execute.

Lines 17 through 25 form the three phase protocol in case the coordinator of the current change fails. Lines 17 through 21 refer to the member(s) that detected the failure
of the current coordinator. Lines 22 through 25 refer to the member(s) that have still not detected the coordinator's failure but wait for a Commit message. Both sides triggered by the coord_fail message, start collecting status of the rest of the members. Then in lines 20 and 24 an election of a new coordinator is done. The lowest rank among the survived and responded mservers gets elected and all members go to process the new (and of higher priority) change. If the current coordinator does not fail, all members commit the change, updating their internal state, in line 27.

```plaintext
/*membership change protocol */
2. process_change (type of change, subject)
3. if coordinator
4. /* start phase */
5. send agree to group reliably
6. receive asks
7. put members that did not reply, on fail list
8. send commit message
9. commit the change
10. if fail list is not empty
11. process_change (fail, first in fail list)
12.
13. else /* non-coordinator */
14. receive agree msg
15. send ack to coordinator
16. wait for commit
17. if commit is not received
18. send coord_fail msg broadcasting status
19. collect status from other members
20. determine new coordinator
21. process_change (fail, old coordinator)
22. else if coord fail msg is received
23. broadcast own status and collect status from other members
24. determine new coordinator
25. process_change (fail, old coordinator)
26. else /* commit received */
27. commit change
```

Figure 5: The Membership Change Protocol (two phase - three phase)

D. PARTITION RESOLUTION PROTOCOL

After a network partition, it is possible that a group is partitioned into two or more subsets. This should happen as some of the group members see at the others as failed and proceed with the processing of these failures. Once the processing of the failures is completed these subgroups will attempt to rejoin once they learn about the existence of other subgroups. Since the subgroups still share the same multicast address, once the network partition is mended, all subgroups receive all the messages from the other subgroups. Upon learning of the existence of a subgroup from the original group, the
partitioned subsets of *mservers* reform into the original group automatically by sending the appropriate reform messages. In addition to reforming the physical group, all application groups which were partitioned and are still functioning are also reformed. The reformation process for both physical subgroups and application groups merges the currently existing membership of each, taking the union of all subsets or subgroups, and making the reformed group 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 group, and will create a new multicast group with only the current group *mserver* included.

If the group partitions, the application groups that span the partition, also experience a virtual partition. These partitions are handled using the following two rules.

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

By combining these two rules, every possible combination of partition handling methods can be produced. The first rule determines who survives, and the second rule determines who will 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 could be permitted to survive and merge with subgroups larger than half of the original group size and containing another particular member type. Note that all partitions of the group necessarily survive network partitions.

In the event that the partitions of *mservers* are unable to restore communications, the reformed subsets are converted to completely independent subgroups. Since all subgroups 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 subgroup appends a unique suffix to the original group name. This suffix value must be automatically derived by each partitioned subset of *mservers* 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 group name. In this manner, all partitioned subgroups are guaranteed a unique group name. However, all partitioned subgroups are still easily identifiable as subsets of the original group, which simplifies the task of manually re-configuring the physical hierarchy when the network is repaired. Once a unique name is obtained, traffic on the same multicast address can be easily filtered by the individual subgroups.

This chapter focused on the MS protocol and its various aspects. The next chapter describes the implementation of two major MS components mcaster and mserver and how some of the features of the MS protocol were embedded into the latter.
IV. IMPLEMENTATION

This chapter describes the actual implementation of some parts of the MS protocol. As mentioned in the two previous chapters, the MS uses multicast to send and receive group messages. Since IP multicasting [30] may not be available at all LANs participating in the MS, there is a need for a multicast emulator that enables running the MS protocol over unicast-capable as well as multicast-capable networks and hosts. So the first step was the implementation of this multicast emulator, referred to hereafter as mcaster. The next step is to implement an mserver capable at least of creating and maintaining an mserver group to be used as a mserver group, handling monitoring and some basic change requests. This is expected to enable the implementer(s) to start working on an application interface and create initial test applications. Mcaster and mserver programs are described next.

The terms "unicast-capable" and "multicast-capable" state the capability of a host or LAN to propagate group messages by sending them point-to-point or multicasting them respectively. The acronyms "uc" and "mc" will be used here.

A. MULTICAST EMULATOR (MCASTER)

1. Algorithm Design

IP multicasting requires that the IP multicast Extensions (1.2 Release) as specified in [30] are installed. Without these extensions, the MS cannot use multicasting to propagate messages to specified groups. Mcaster is an underlying program that enables the MS to virtually use the properties of multicasting in a uc environment with minimum overhead. If all LANs in which the MS runs are mc, the mcaster is not needed.  

Mcaster must be able to listen to all multicast messages in the network and decide according to membership of groups which of them to propagate through the unicast channels. To achieve this, mcaster must be a member of all the mserver groups created. Since some members of an mserver group may have multicast capability while others may not, mcaster must maintain links for both unicast and multicast LANs. The administrator
must foresee if there is any possibility of a mc host to become a member of the mserver group and place the mcaster process on the LAN that supports multicasting. This is an easy decision since one mserver will run per LAN. A typical scenario that includes both types of LANs is presented in Figure 6. If on the other hand, there are no mc LANs involved, mcaster must be able to run, simulating fully the multicast capability by repetitive unicast message transmissions. This is the first restriction that comes with the use of mcaster. The second comes with the use of extended header that messages, propagated through mcaster, have. As described by Neely [29], this is essential so that the final receiver of the message can extract the information about the original sender. Each time an mserver receives a message with a sender’s address the same as the mcaster’s (which is known to all mservers), it tries to extract an extended header from it. This fact prohibits the MS from using the host that mcaster is running on, as the host that will run the mserver routine for the LAN that it belongs to.

Figure 6: Typical mcaster communication diagram

Figure 7 describes the algorithm for m caster, demonstrating its capability of detecting the type of LAN it runs on. In case of multicast LAN, it initializes a second socket to be used for multicasting to mc members. There are two types of messages that arrive at mcaster: those that are going for other members of the group, and those that are for the mcaster specifically. These messages can be either a JOIN_GROUP or a
LEAVE_GROUP and must be transmitted from any mserver that needs to join or leave a group. These two messages simply register and de-register the mserver with the mcaster. A JOIN_GROUP must be sent before an mserver sends any messages to the mservers in the group. These messages are the third restriction of using mcaster, and are essential for the mcaster to be able to maintain its local group lists and memberships for these groups.

In summary, the restrictions placed by the use of mcaster in the MS are:

1. Multicast is simulated at on uc LANs by repetitive unicast transmissions by mcaster. Therefore, mservers running on unicast LANs must have a unicast socket to listen to "multicast" group messages.

2. Since every message delivered by IGMP carries the address of the host that sent it, every message propagated by mcaster, has mcaster's address. To be able to identify the original sender an additional header must be put on the message containing the address. This is done by mcaster. Receivers must identify if the message comes from mcaster, to extract this additional header.

3. Each member of the group must register itself into mcaster's internal list of groups. This is essential for the operation of mcaster. Therefore, each new member must join mcaster's internal group, before attempting to join the real mserver group. Leaving the mcaster's internal group after leaving the real mserver group is not essential, unless the same host is going to be used for a new copy of an mserver.

Every mserver that needs to join or create a group, sends initially a JOIN_GROUP to mcaster. Lines 9 through 17 show how this message is used by mcaster to maintain updated group lists. If the group needs to be created, mcaster does so, and then if it runs on a mc LAN, it actually joins IP multicast group specified by the class D address, even if the original sender is a uc mserver. This reserves the class D address group in the mc LAN for future mc mservers.

Line 25 is the most vital to mcaster's functionality. It is shown here as a call to a function, mcast. If a message is not sent for the mcaster specifically (i.e. JOIN_GROUP or LEAVE_GROUP) then it is propagated to the group it was sent for. Figure 8 shows the algorithm for function mcast. Mcaster extracts the group information from the message header, uses this information to locate the group in its own internal list, then adds the
additional header that will enable the final receiver to extract the sender's information. Finally, it sends out zero or more messages, according to the logic shown in Figure 8, so that the message will reach every member of that group.

```
1. /* MCASTER */
2. Initialize first (ms) socket  /* this socket is used for unicast communication */
3. Initialize second (mc) socket  /* this socket is used for multicast communication */
4. if on a multicast (mc) lan, then
5. use mc socket for multicast communication
6. else
7. mc socket is not used
8. for every message
9. if msg_type = JOIN_GROUP
10. if group exists
11. join member to group
12. else
13. create group
14. join member to group
15. if mcaster on a mc LAN
16. join mcaster to group
17. send reply to sender
18. else if msg_type = LEAVE_GROUP
19. if group exists and member of this group
20. delete member from group
21. if group becomes empty
22. delete group
23. mcaster leaves class D address port
24. else /* must send message to members*/
25. mcast message to group members
26. end for

Figure 7: Algorithm for Multicast Emulator (mcaster)
```

A group may have both uc and mc members. Also, the sender can be either uc or mc. Since mc member communication is taken care of in the IP multicast level, if the sender is mc, there is no need for mcaster to reroute the message to the same class D address (address of a multicast group). That is why after line 15 in Figure 8 there is no if statement for the mc members. For the same reason, if the sender is uc, and one member is found to be mc, then mcf flag ensures that only one message will be transmitted to this class D address. For all uc members in a group, a per member peer-to-peer transmission of the message must be used, so that the message arrives at all of them, as shown in lines 13 and 17.
1. /* function MCAST */
2. mcast (msg, sender)
3.  extract sender and group info from msg
4.  form extended header
5.  if group does not exist in list
6.    exit
7.  for each member in the list maintained by mcaster
8.    if sender is uc
9.      if this is uc member and mflag = 0
10.     send msg to class D address for this uc member
11.    set mflag = 1 /* no need to re send msg to mc socket */
12.   else if this is uc member
13.     send msg to uc member /* through uc socket */
14.   go to next member in list
15. else if sender is mc
16.   if this is uc member
17.     send msg to uc member /* through uc socket */
18.   go to next member in list

Figure 8: Algorithm for the Message Propagation Function mcast.

2. Code Description

The m caster code is very important for future work since the way it creates, maintains and updates internal state for the mserver groups is the same that will be used from MIs to maintain information about the application groups. In Figure 9 the initialization part of code for the m caster is shown.

1. /* test_addr will be used for testing the mc port */
2. test_addr = 0xe1f0f0f0;
3. /* initialize sockets */
4. ms = init_socket (Ssin, MS_PORT); /* unicast socket */
5. print_sock_info (ms, sin);
6. mc = init_socket (Smcsin, MC_PORT); /* multicast socket */
7. /* join a class D address to test mc capability */
8. reply = join_mc_grp (mc, test_addr);
9. if (reply == JOIN_ACK) {
10.   lan = 1;
11.   print_sock_info (mc, mc sin);
12.   leave_mc_grp (mc, test_addr);
13. }

Figure 9: Initialization Code for Mcaster

In line 4 the unicast socket is initialized. To initialize a socket for multicast communication, m caster must run on an mc host. To test if the host it runs is mc, it picks an arbitrary class D address in line 2 and tries to join a mc group for this address in line 8. If it succeeds, then flag variable lan is set to 1 in line 10 and m caster leaves the class D address group in line 12.
After initialization of mcaster is finished, it starts listening to the port(s) for possible messages. Figure 10 shows the loop of waiting, receiving and processing messages of mcaster. Whenever it receives a message in line 2, it extracts the message type in line 3 and goes to appropriate actions according to line 4 switch statement. The first two cases are the mcaster specific types of messages and are used to inform the mcaster about a change in the group. This way mcaster keeps its lists updated. These two cases are explained later.

The default case is executed whenever an ordinary message is trying to propagate to the group. Mcaster calls function mcast to retransmit the message as explained earlier in the algorithm section.

```c
1.  for (;;) {
    /* wait for incoming messages */

2.      if ((sent = receive_msg (ms, mc, &ws, &m, &from, recv_timeout)) > 0) {

3.          message_type = ntohs(m->msg_type);             /* check type of received message */

4.          switch (message_type) {

5.              case JOIN_GROUP:

6.              ...

7.              case LEAVE_GROUP:

8.              ...

9.              default:

10.             /* if sender is mc do not mcast to mc members */

11.             all = ws ? 0 : 1;

12.             mcast (ms, mc, m, from, all);

13.         } /* switch */ } /* if */ } /* for */
```

Figure 10: Main Waiting Loop of Mcaster

The operation of mcast function depends on whether mcaster runs on a mc LAN or not and whether there are both types (mc and uc) of members in the specified by the message group or not. When propagation of a message is finished, control is returned to main loop and mcaster waits for the next message to process.

**a. Internal Group Lists**

As mentioned above, mcaster keeps an internal list of mserver groups and lists of members for each group. These lists can change dynamically and are kept as linked lists. Figure 11 shows two structures defined in "msutil.h" header file, that are
used as elements for the group linked list and for the members' linked list. A graphical representation of these lists can be found in Chapter VI of [29].

```
1. /* element in list of members */
2. struct member {
3.       struct in_addr addr;
4.       u_char    lan;        /* lan = 0 if member on a unicast, 1 if on a multicast lan */
5.       u_char    loop;
6.       struct member *next;
7.    };
8. /* element in list of groups */
9. struct group {
10.      char      name[MAXGROUPNAME];  /* name of the group */
11.      struct in_addr grp_addr;       /* classD grp address */
12.      struct group *next;
13.      struct member *members;       /* first member */
14.      struct member *last;          /* last member */
15.    };
```

Figure 11: Structure Definitions for Members' and Groups' Internal Lists of Mcaster

A whole series of function calls was implemented to support proper searching, adding and deleting members and groups from these lists. All these functions are included in the "mcast.c" file, with a comprehensive description of their arguments and functionality.

b. Mcaster Specific Messages

As mentioned before, two messages that are specifically for the mcaster are: JOIN_GROUP and LEAVE_GROUP. Both of them have the standard message format used throughout the MS code, with the latter carrying an empty data section. JOIN_GROUP, in its data section, has a copy of a short integer, showing whether the sender of the message is mc or uc (as described in Figure 11 line 4). This is essential so that mcaster maintains a complete image of each member. Also, function mcast uses this field to avoid sending out unnecessary messages.

Mcaster replies to the above messages according to its internal state. Figure 12 summarizes the definitions of these types of messages as they are found in "msutil.h" header file.
Each of these messages, exchanged between mservers, who try to change their membership to mserver groups and mcaster, does not use the extended header format. To receive such messages, function receive_msg is used. All other messages, when received by mservers, have the possibility that they were propagated through mcaster. In such a case the message has an extended header and the receiver must check the sender's address. If it is the same as the mcaster's, it extracts from the extended header the address of the original sender. The function that makes such a discrimination is recv_msg. Both these receiving functions along with basic forming and sending message functions have their definitions in "msutil.h" header file and the code in the "msutil.c" utility file. An outline of these functions follows in the next section.

1. /* mcaster related message types */
2. #define JOIN_GROUP 120 /* request to join a group list kept by mcaster */
3. #define LEAVE_GROUP 121 /* request to leave a group list kept by mcaster */
4. #define JOIN_ACK 130 /* mcaster positive reply to JOIN_GROUP */
5. #define DUP_MEMBER 131 /* mcaster negative reply to JOIN_GROUP, member found in list */
6. #define NEG_JOIN 132 /* mcaster negative reply to JOIN_GROUP, problem with port */
7. #define LEAVE_ACK 140 /* mcaster positive reply to LEAVE_GROUP */
8. #define NO_GROUP 141 /* mcaster could not locate the group in its list */
9. #define NO_MEMBER 142 /* mcaster could not locate the member in the group's list */
10. #define NEG_LEAVE 143 /* mcaster negative reply to LEAVE_GROUP */
11. #define GROUP_EXISTS 150 /* mcaster found the group requested in its list */

Figure 12: Definitions for Mcaster's Related Types of Message.

B. BASIC MESSAGE FUNCTIONS

1. Function Receive_Msg

Function receive_msg receives a message from the buffer of a port and stores it into a variable. Since the multicast emulator and membership server (mserver) have at most two sockets, one each for unicast and multicast, the first attribute of function receive_msg is to be able to listen to both and receive from either of them whenever a message appears. Information about who sent the message and, of course, the message itself must be returned. Figure 13 lists the heart of the code for function receive_msg that actually monitors two ports for any incoming message and then uses the library function recvfrom to receive it.
1. int receive_msg (ms, mc, w, msg, frm, timeout)
2. ...
3. FD_ZERO (&fread);             /* Initialize for reception from multiple sockets */
4. FD_SET (ms, &fread);          /* Unicast socket */
5. if (mc >> 0) FD_SET (mc, &fread); /* Multicast socket */
6. if (select (32, &fread, 0, 0, &timeout) < 0) { /* Wait until either socket is ready */
7.     perror ("Select error\n"); return -1;
8. }
9. if (ready) {
10.    if (FD_ISSET (ms, &fread)) { /* Unicast socket receives */
11.        *w = 0;
12.        if (sent = recvfrom (ms, buf, MAXMSGLEN, 0, frm, &rlen) < 0) {
13.            perror ("Error in UC message received\n"); return -1;)
14.    } else
15.    if (FD_ISSET (mc, &fread)) { /* Multicast socket receives */
16.        *w = 1;
17.        if (sent = recvfrom (mc, buf, MAXMSGLEN, 0, frm, &rlen) < 0) {
18.            perror ("Error in MC message received\n"); return -1;)
19. }
20. ...

Figure 13: Function receive_msg Receives Messages from Two Sockets.

The first two arguments are the two socket numbers. If the calling program needs only one socket to read, it can set the second, mc, to -1. In line 5 this disables the second socket. Argument w returns 0 or 1 in correspondence with the socket that the message was read from. Argument msg returns a pointer to the message and argument frm a pointer to sender's address structure. Finally, the calling program defines a period of time during which a message maybe received using timeout. After time period expires and no messages received, receive_msg returns a NULL pointer. This makes the function non-blocking and gives the calling program the opportunity to regain control and decide what to do next even if no messages were received. Normally, the function is called with a small timeout, like one second, because messages are stacked in the ports, so that usually when the socket is read, the message is already there. Receive_msg is a blocking function and will wait for a message if the buffer of the port is empty until the timeout expires.

2. Function Recv_Msgg

This function is similar to receive_msg, except it checks the sender's address against mcasters. When a message is propagated through the mcasters, it arrives at the destination with an extended header, including the address of the original sender. If the
receiver calls receive_msg to read these messages, it will return with mcaster's address as
the sender's address, since the C library functions [31] read what IGMP puts as a sender
(and it is always the real sender - mcaster in this case).

Function recv_msg, shown in Figure 14, solves this problem by comparing
mcaster's address with sender's address and if the same, reads the extended header of the
message and replaces the sender's address with the one in the header. It is obvious that
this function cannot be used with messages listed in Figure 12. These messages are used
to communicate between mcaster and mservers and do not contain an extended message
header. The extended message header and its description can be found in [29, pages 102 -
103, Figure 53]. Function mcast of mcaster constructs and puts the extension to the mes-

**Figure 14: Function recv_msg Extracts Sender's Address from Extended Message.**

Recev_msg works as follows: it checks the sender's address against mcaster's,
which is passed as an argument, and if they match, it replaces the sender's address with
the one that it finds in the first len bytes of the message (where the extended header is
supposed to be). Since it uses the low level bcopy standard C function, if used to receive
mcaster generated (not propagated) messages, it may lead to unexpected results, without
necessarily showing an error like "core dumped".

3. Function Form_Messg

This is a relatively simple function. It takes as arguments all the components of a
regular message, as described in [29, Chapter IV, Section B] and shown in Figure 15 and
returns a pointer to a message structure. There is a deviation from the original message format: after the group_name field there is another field of type in_addr to hold the class D group address the message is going to. This was considered necessary as the tuple \{group_name, group_address\} defines a multicast group completely.

![Diagram of message format and corresponding data structure](image)

Figure 15: MS General Message Format and the Corresponding Data Structure

Pointers to exclude list, subject list and data along with the length of each field are also passed as arguments, so that the correct number of bytes is copied from each one. Finally, function \texttt{make_chksum} is called to evaluate the checksum field just before it is entered to the message structure. At this time, \texttt{make_chksum} is a dummy function, always returning a constant number.

4. Function \texttt{Send\_Messg}

Function \texttt{send\_messg} sends a message through a specified socket to the specified address. Its code was originally written by Neely [29] and was slightly modified to meet the changes described above. It copies the message, the exclude list, the subject list and the data to sequential bytes of a buffer and then calls the library function \texttt{sendto} to send that buffer to the socket port. The socket can be either a uc or an mc. The IP multicast extensions, described in [30], overload the library function \texttt{sendto}, making no difference whether the socket is uc or mc.
C. MEMBERSHIP SERVER (MSERVER)

1. Algorithm Design

The basic module in the MS is mserver. Each mserver process controls a whole LAN or subset of hosts on a LAN. Mservers join into mserver groups, monitoring each other and exchanging messages. Depending on the type of the LAN they run on, they use the multicast capability, or try to send messages to the group by a single transmission, using the multicast emulator, mcaster. Messages can be mserver-group specific or just received from the applications' Membership Interface processes, MIIs.

Figure 16 shows the basic communication diagram of mservers through IP multicasting or through mcaster. Two cases are demonstrated: on the left, the sender is mc, on the right the sender is uc. The types and related names (as defined in "msutil.h" header file) of the sockets used are also shown in this figure. To send and receive group messages, uc mservers rely on the mcaster. Monitoring messages are sent and received through the designated unicast sockets. Finally, as described in the previous section, communication between mserver and mcaster is done through the unicast socket. In all cases, the format of the message is the same. Destination of a group message is defined by the tuple \{group_address, group_name\}. The group address is always a class D address. Although the group address is sufficient to specify a group, the group name is reserved for future features such as multiplexing of groups with different names on a single address.

The algorithm for mserver consists mainly of three phases: the initialization phase, the new member join phase and the basic monitor and process message loop phase. The first two phases are executed once, when mserver comes to life. Mserver spends the rest of its life in a loop, monitoring its clockwise mserver in the mserver group (if it exists) and processing any messages read from its socket(s).
Figure 16: Communication Among Mservers

Figure 17 outlines the algorithm for mserver. As stated in line 3, some command line arguments are provided. A command line call to mserver looks like:

```bash
% mserver grpaddr, grpname, mcasterIP
```

where `grpaddr` is the mserver group class D address selected by the system administrator, `grpname`\(^1\) is a name for this group and `mcasterIP` is the IP address of the host on which the multicast emulator runs, provided in dot notation [31]. Next, sockets are initialized and finally mserver sends a message of type JOIN_GROUP to mcaster, whose address is known from the command line argument. The message to mcaster informs it about the mserver's intention to join a group. If no reply is received, mserver assumes that there is no mcaster available on that address. If mserver runs on a multicast LAN it assumes that the system administrator plans to create a group of mc mservers only and continues normally. If it runs on a uc LAN, then without mcaster the MS exits, as it needs some form of multicasting. It also exits if a reply received from the mcaster, is other than of type JOIN_ACK, implying there is some kind of a problem.

\(^1\) Up to 32 characters long or as specified by variable `MAXGROUPNAME`, global defined in "msutil.h" header file. `Grpname` must be enclosed in double quotes if it contains special characters like ' & ' or space(s).
Figure 17: Algorithm for Mserver

If everything goes normally in the initialization phase, mserver in lines 9 and 10 tries to join the mserver group, as specified by its command line arguments grpaddr and grpname. If it succeeds, it enters the main loop, where three basic operations are executed: 1) receive and process messages, 2) monitor and 3) update of the internal state. Processing messages is done in function process_msg described later. To make the loop faster, monitoring takes place only when no messages are heard from the other members. If the mserver group is busy sending change messages, then one of the change protocols will discover the failures, if any. Thus, additional monitoring is not required.

The procedure of a new member joining a group as well as function reliable_link are explained in the next subsections. A description of the internal state of the mserver, part of which is the fail_list, is given in the next section.
a. New Member Joins Group

The procedure of joining the group is described in Figure 18. The logic is simple: it tries to communicate with the other members of the mserver group, sending a request for join. As given in the protocol description [29], normally the group will process the change through a 2-phase protocol, resulting in the second phase of the multicast of a COMMIT message. This message is received by the new member also. The procedure of sending a message, waiting for a specific reply for a certain time and retrying all over again in case of wrong or no reply, is done by the function reliable_link, discussed later.

After certain retries, reliable_link returns the message (if any) it read. The new member's actions depend on this answer, as described in lines 5 through 13. If no answer was received, in line 5 mserver assumes there is no group yet and initializes its own internal state in line 6. The internal state now contains the new group with the mserver as the only member.

```plaintext
1. /* join_group */
2. time = T_M_JOIN_REQ;                  /* set timeout variable */
3. do loop
4.   answer = reliable_link(M_JOIN_REQ, COMMIT, group, time)
5.   if no answer was received            /* mserver is alone in this group */
6.     initialize internal state
7.     exit
8.   else if answer = COMMIT
9.     update internal state
10.  exit
11.  else                                /* answer was not the expected COMMIT */
12.    time = time * 6;                   /* group members are busy; increase wait time by 6 (arbitrary) */
13. end loop
```

Figure 18: New Member Join Algorithm

If a COMMIT answer is received, then in line 9 the new mserver updates its internal state according to the contents of the COMMIT message. This message is transmitted by the group coordinator of the join change and contains the internal state updated to include the new member. The new member just copies this internal state to its own.

There is also the possibility that the new mserver receives as an answer a message other than COMMIT. In this case the group exists but probably other higher
priority changes are being executed. If this is the case, the new mserver goes back to line 3 for a new round of reliable communication, but decides to wait a little longer than its previous attempt, as indicated in line 12.

b. Function Reliable_Link

This function is used to communicate by the sender that calls the function, with the specified receiver (or group) through a series of retries and timeouts to ensure reliable connection. The combination of sending a message to a specific member or group, waiting for a specific reply for a certain period of time and then trying all over again, up to specific number of retries, results in function reliable_link. Its algorithm is shown in Figure 19. It needs the message to be sent, the expected reply, the recipient and amount of time to wait for the reply (the number and specifications of arguments passed to the real function must not be misunderstood with the simplified explanation given here at algorithm level). The effect of this function is to make the sender attempt to establish a reliable connection or link with the receiver (member or group).

As shown in Figure 19, a message is received in line 6 using recv_messg. If it does not match the expected message type, it tries to receive a new message until timeout expires as controlled by line 5. Then it sends the original message again and resets the timer. The whole schema is repeated max_retries times. If at any time the expected message is received, the function returns with the reply at line 8. After the retries are exhausted, it returns with the last message read, at line 11. The return message can be null indicating no messages were received at all.

Unexpected messages are ignored while in the timeout loop. When timeout expires if an unexpected message is the last received, the function retries again by resetting the timer. This is repeated max_retries times. When last timeout expires and still the expected message is the not received, the last message received is returned. The caller has to check if this was the answer that it expected. If no message was received, a NULL message may be returned.
1. reliable_link (outmsg, expected_msg, recipient, timeout)
2.    max_retries = MAX_RETRIES, retries = 0
3.    while retries != max_retries
4.        send msg to recipient
5.        while timeout has not expired
6.            reply = recv_msg()
7.            if reply = expected_msg
8.                return reply
9.        reset timeout
10.       retries = retries + 1
11.    return reply

Figure 19: Algorithm for Function Reliable_link

2. Internal State

Each mserver must keep internal information about its mserver group and real state of itself and other members. This internal state must be kept updated to reflect the most recent changes. The internal state of an mserver, at this level of implementation, can be described as the core table, the fail list and the view. These components are described here.

a. Core Table

Core table is an array of structures as shown in Figure 20. Under normal conditions, each member has the same copy of information in its own table. Each member is assigned a rank when joining and its information lies at the same line of the table. It may seem that cw and ccw fields are redundant, but they are useful during the processing of one or more changes, where ranks and the table itself are not updated yet.

```
1. struct table_entry {
2.    u_long    addr;  /* IP address of member */
3.    u_short  rank;  /* rank (or gid) of member */
4.    u_short  cw;    /* gid of clockwise member (to left) */
5.    u_short  ccw;   /* gid of counterclockwise member */
6.    u_char   flag;  /* status flag for each member */
7. },
8.
9 struct table_entry cs tbl [MAXTBLSIZE];
10.
```

Figure 20: Core Table Structure and Definition

b. View

View is an integer kept by each member of the group internally. When the group is idling with no changes being processed, view is the same in every mserver
participating the group. View is incremented by one for every new change at the end of the change (commit phase). In this way, every change is marked uniquely by a number and can be identified with that number. View is used to put changes in sequence, give them priority or simply discard them because they came out of order, according to the state of the group and the members themselves. In normal conditions, view must be the same in every member of the mserver group.

3. Processing Messages - Function Process_Msg

This function is used to process any message according to the state of the caller and the type of the message. In mserver's main loop of operation, the incoming messages get processed by process_msg. This function actually implements both the two-phase and the three-phase protocols. Moreover, it can be called recursively to handle more than one change in the order specified by the priority and the phase of each change. Figure 21 shows a simplified algorithm of process_msg.

The function is divided into two parts: The part that is executed by the coordinator and the part that is executed by all other members. In phase I, the coordinator sends the AGREE message through a reliable link similar to that provided by function reliable_link. The new function, rel_mlink, establishes reliable links with all members in a group. Then, a reply is expected by everybody. If a group member does not reply after rel_mlink returns, it places the non-responding member(s) on the fail list to be processed later.

When AGREE is sent out by the coordinator, process_msg gets activated in the non-coordinators, resulting in their sending back the AGREE_ACK. The coordinator collects all acks and begins phase II by sending the COMMIT message. This gets received by the non-coordinators in line 20, committing the change in line 22. A timer is set at the non-coordinators from phase I to phase II, to trap a possible failure of the coordinator. If this is the case, lines 15 through 19 describe the actions of the detector of the failure, while lines 24 through 28 describe the actions of the rest of the members in response.
1. process_msg (msg)
2. select coordinator
3. if coordinator
4. /* phase */
5. rel_mlink (AGREE, AGREE_ACK, group)
6. /* phase II */
7. send COMMIT
8. update internal state
9. else /* non-coordinator */
10. check if AGREE is for same change and drop if so
11. send AGREE_ACK
12. set timer for COMMIT
13. do loop
14. if timer expired /* current coordinator assumed to have failed */
15. send CO_FAIL to group /* process coordinator's failure */
16. send and receive status of members
17. elect new coordinator
18. process_msg (msg)
19. exit
20. try to receive any message
21. if msg = COMMIT
22. update internal state to reflect change
23. exit
24. if msg = CO_FAIL
25. send and receive status of members
26. elect new coordinator
27. process_msg (msg)
28. exit
29. end loop

Figure 21: Algorithm for Function "process_msg"

While this algorithm is simplified to hide the specific details of each different change processed, it gives an idea on the implementation of the protocols. Lines 18 and 27 show how recursion is used, so that any number of subsequent changes (like coordinators failing one after the other) get handled. The number of changes are limited only by the system's available memory, since any new recursive call reserves new space for all variables used. More detailed description is given at the code description section.

4. Code Description

a. Initialization

As in mcaster, it is vital to mserver to learn in its initialization phase if it runs on a host with multicast capability. The technique used in mcaster applied here also. After initializing the uc socket in line 2 of Figure 22, it tries to join an IGMP group on the class D address provided by the command line arguments as shown in lines 4 to 10. If this fails, no multicast is available and it is replaced by the mcaster in lines 12 to 16. As shown in Figure 16, one unicast socket is devoted to monitoring. The other socket is
either multicast (multicasting is done in IP multicast extension layer) or unicast (multicasting is simulated by mcaster), according to the capability of the host mserver runs onto.

```
1. /* INITIALIZATION PHASE */
2. uc = init_socket(&ucsin, PORT_UC); /* Initialize unicast socket */
3. /* Join multicast IGMP group; if it cannot, then host is not MC capable */
4. mc = init_socket(&mcisin, MC_PORT); /* Initialize multicast socket */
5. reply = join_mc_grp(mc, grpaddr);
6. if (reply = = JOIN_ACK) {
7.     setsockopt(mc, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
8.     lan = 1;
9.     mcaddr = grpaddr;
10.    mcport = MC_PORT;
11. }
12. else {
13.     lan = 0;
14.     mcaddr = mcaster; /* mcaster replaces mc lan */
15.    mc = init_socket(&mcisin, MS_PORT);
16.    mcport = MS_PORT; /* direct mc port to mcaster's */
17. }
```

Figure 22: Code for Initializing Sockets of Mserver

**b. New Member Join Procedure**

After initializing the sockets and updating the internal list of mcaster (as described in Figure 17), mserver is about to send its first message to its mserver group. Figure 23 shows the code for joining the group. The communication with the group is done through a reliable link. New member is waiting for only one specific answer, a COMMIT message and uses reliable_link to send out its request and intercept the answer.

One common problem when a new member tries to join the mserver group is that the mserver group may be busy processing some other mserver group change of higher priority. If the network is slow (or a member is slow), then the current change may take some time to finish. Since this time is unpredictable, the solution of increasing the timeout time of the new member in line 3 does not cover all possible scenarios. A more intelligent code was implemented. The big loop between lines 9 and 34, suggests that the reliable link communication is tried two more times (making it a total of three, with the
one of line 8). In between, any intercepted message is examined for its origin. If it comes from the mserver group, this means that the group is alive. Then the new member goes for another round, with a little longer waiting time, as suggested in line 32. This makes the new member more patient. On the other hand if no messages are heard from the mserver group, then as described in lines 11 through 18, the new member creates its own group and initializes its internal state. It then exits the new member join procedure.

```
1. /*----------------NEW MEMBER JOIN PROCEDURE-------------*/
2. max_retries = 3; /* initialize retries and timeout */
3. tout = T_M_JOIN_REQ; /* form M_JOIN_REQ message and socket address */
4. form_messg (&msg, grpname, grpaddr, O, O, M_JOIN_REQ, O, NULL, O, NULL, O, NULL, O);
5. to.sin_family = AF_INET;
6. to.sin_port = htons(chport); to.sin_addr = htonl; /* either mc or mcaster */
7. mptr = reliable_link (&msg, to, uc, mc, COMMIT, max_retries, tout, mcaster, &f);
8. for (i = 0; i < max_retries - 1; i++)
9.   /* no msgs received at all, or no group msgs were listened */
10.  if (!mptr || (mptr->grp_addr.s_addr != grpaddr.s_addr)
11.    && strcmp(mptr->group_name, grpname))
12.    myrank = 0; myrow = 0; /* assign rank 0 to itself */
13.    bzero((char *)cs_tbl, tlen); /* put in core table itself */
14.    cs_tbl[0].addr = myaddr.s_addr;
15.    cs_tbl[0].rank = 0; cs_tbl[0].cw = 0;
16.    cs_tbl[0].ccw = 0; cs_tbl[0].flag = 1;
17.    break;
18. }
19. else {
20.    /* some msg was received */
21.    if (!f) {
22.      /* it was the expected COMMIT */
23.      /* extract core table from commit msg */
24.      extract_init_state (mptr, &who, &n_req, cs_tbl);
25.      myrank = mptr->subject_gid; /* find my rank and row of table i exist */
26.      for (i = 0; i < MAXTABLESIZ + 1; i++)
27.        if (cs_tbl[i].addr == myaddr.s_addr)
28.          myrow = i;
29.      view = mptr->group_view; /* adjust my view to change's view */
30.      break;
31.    }
32.    else {
33.      tout *= 6; /* wait a little longer */
34.      mptr = reliable_link (&msg, to, uc, mc, COMMIT, max_retries, tout, mcaster, &f);
35.    } /* else */
36. } /* for */
```

Figure 23: Code for New Member Join Procedure

Of course, if at any time the long awaited message of type COMMIT comes in, the new member is accepted into the group. The COMMIT message carries the
internal state of the coordinator updated to include the new member. Then the new member extracts this internal state and copies it to its own. Thus the new member is fully synchronized with the rest of the *mserver* group members.

**c. Monitoring - Processing Messages**

After an *mserver* becomes a member of an *mserver* group, the rest of its code is a loop, doing two major tasks: Monitor its clockwise member and process any incoming messages. Figure 24 shows this portion of the code. The loop starts in line 2 and ends in line 40. In line 4 *mserver* tries to read a message from the socket port. In lines 5 through 12 *mserver* processes any failed members recorded in the fail list. Monitoring is done in lines 13 through 28. There is an associated timer and when it expires, the member tries to monitor its clockwise member. Monitoring is not performed only when there is only one member in the group; There is no need for an *mserver* to look after itself. This is checked in line 14.

The incoming messages can be in one of three categories: 1) monitoring messages, 2) group change messages, and 3) application messages. Monitoring messages are part of the monitoring code. Group change messages are processed in the separate function *process_msg*. As shown in lines 35 and 37, application messages are also directed to the *process_msg* function which simply forwards them to the appropriate group.

There is also another important feature of the code in Figure 24: monitoring is executed only when no messages arrive at the ports of the *mserver*. This is achieved by inserting the monitor lines inside the *while* statement of line 4. If there are messages, this means that group members are active sending messages to each other and if there is any failure, it will be discovered through a reliable communication or a two-phase protocol. Therefore, additional monitoring is not needed.

**d. Function Process_Msg**

This function is the heart of the MS protocol. It includes a complete implementation of the two-phase protocol as well as the three-phase protocol to handle regular changes (joins - failures) and coordinator's failures. A separate function called by
process_msg (to avoid long sequential code and to enhance the recursion) is function
process_cfail that handles the coordinator's failure. Both function declarations exist in
"message.h" header file and their definitions in "message.c" file.

```
1. /*-----------START MONITORING AND PROCESS OF INCOMING MSGS-----------*/
2. for (;;) {
3.     /* big loop */
4.     while (recv_msg (uc, mc, &s, mcast, &mpt, &from, t) <= O) {
5.         while (fail_list) {
6.             /* see if there are failed members */
7.             form_msg (&msg, grpname, grpaddr, O, view, myrank, AGREE_L_0,
8.             fail_list->gid, NULL, O, NULL, O, NULL, O); /* form the AGREE_L msg */
9.             /* process the change (includes sending agree msg) */
10.            from.sin_addr.s_addr = O;
11.            /* deactivate from's address */
12.            process_msg (&msg, from);
13.            free (&msg);
14.        }
15.        if (timed_out (q, tout)) {
16.            /* see if it is time to tx a query */
17.            if (myrank != cs_tbl[mynrow].cw) {
18.                /* check if only one member in table */
19.                form_msg (&msg, grpname, grpaddr, O, view, myrank, QUERY, Q, Q, Q, Q, Q, Q, Q, Q),
20.                to.sin_family = AF_INET;
21.                /* construct address of cw member */
22.                to.sin_port = htons (PORT_UC);
23.                to.sin_addr.s_addr = htonl (cs_tbl, cs_tbl[mynrow].cw, O)->addr;
24.                mptr = reliable_link (msg, to, uc, uc, QUERY, max_retries + myrank * 2,
25.                              T_R_QUERY, mcast, &t);
26.                free (&msg); /* free space reserved for query msg */
27.            }
28.            if (mptr->msg_type == QUERY) /* see if someone is querying me */
29.                break;
30.        }
31.        if (f) /* no REPLY msg from cw member; fail it */
32.            add_gid_entry (&fail_list, cs_tbl[mynrow].cw);
33.            free (mptr); /* free space allocated for reply msg */
34.        }
35.        set_timeout (&q, tout, T_QUERY); /* set query (monitor) timer */
36.    }
37.    /* while */
38.    switch (mptr->msg_type) {
39.    case QUERY:
40.        /* reply to query */
41.        break;
42.    default:
43.        /* process the msg */
44.        process_msg (mptr, from);
45.    } /* switch */
46.    free (mptr);
47. } /* big loop */
```

Figure 24: Code for the Mserver's Main Loop
The algorithm for _process_msg_ was presented in Figure 21. In that algorithm, all changes were treated in the same way, i.e., there was only one AGREE message for any kind of change. Each change has its own details that prohibit them from being encoded the same way. For example, the COMMIT message for a join carries the _mserver_ group table as its data, because the new member depends on this message to extract the table and synchronize with the rest of the group. The COMMIT message for a fail does not have any data because the information for the failed member exists in the other fields of the message which is received by all surviving members.

In the current level of implementation, there are three kinds of change messages: a _join_, a _fail_ (or _leave_) and a _coordinator fail_. Their _initiate_ messages to start the change protocol are AGREE_J, AGREE_L and CO_FAIL correspondingly. There is only one AGREE_ACK to send as an acknowledgment for all of these types of initiate messages. Also, there is only one COMMIT for all changes although it may carry different data according to each case. To summarize the message descriptions (message fields not described are filled out normally):

- **AGREE_J** is the message issued by the lowest in rank active member upon reception of an M_Join_REQ group message as sent by a new member. This coordinator assigns a new rank to the new member, which copies to the _subject_gid_ field of the message. The address of the new member gets copied to the _data_ field and the _data_len_ is adjusted. This message is sent out with the current _view_ number.

- **AGREE_L** is the message issued by any member detecting (or suspecting) the failure of another member. In this case, the detector is the coordinator. So, if this message is originated from the _mserver_, this _mserver_ is the coordinator. If the _mserver_ received it, this _mserver_ is a non-coordinator. No _data_ is needed for this message. This message is sent out with the current _view_ number.

- **CO_FAIL** is the message sent originally by any member detecting (or suspecting) the failure of the coordinator of the current change. In the status exchange phase that follows, each member sends a _co_fail_ to exchange its status with the others. There is no _data_ in this message but the _data_len_ field is used to pass a 0 or 1 indicating that the status of the member sending the message is either "finished change (committed)" or still "processing the change" respectively.
- **AGREE_ACK** is the acknowledgment message sent at the end of phase I of the two-phase protocol. No special syntax is needed and it is common for every change. It is sent with the current **view**.

- **COMMIT** is the message sent always by the coordinator in the final phase of the change protocol. Although the same **msg_type** is used for all changes, there are some differences according to each case. If the change is a **Join**, then the current core-table of the coordinator (and of the group, since it is kept consistently in every member) gets copied to the **data** field of the message. Of course the **data_len** field is adjusted properly. If the change is a **Leave**, there is no need for any **data**. This message is always sent with a new **view** number (the current incremented by one).

Figure 25 shows the difference between the messages needed for a **join** and a **fail** change. In the first case, a two-phase protocol is initiated because an external message (**m_join_req**) gets received by the group. The decision for the coordinator is made in all **mservers** of the group. In the second case, the detector of a change (**fail**) initiates a two-phase protocol being itself a coordinator, which the other(s) accept.

![Diagram of Two-phase Protocol Time-lines for a Join and a Fail](image)

Figure 25: The Two-phase Protocol Time-lines for a **Join** and a **Fail**.

If the phase of receiving the **m_join_req** and deciding for the coordinator is omitted, then the logic between the two changes is similar. Next, the code for processing a **Join** is described pointing out the differences with the corresponding code for a **Fail**.

**e. Processing Join Requests**

Function **process_msg** has a "switch" as its first statement activating the appropriate portion of code according to the message type (**msg_type** field). As soon as an **m_join_req** is received by the group, all members execute the part of code shown in
Figure 26. The election of the coordinator is based on the rank. The lowest rank active member gets elected. The function to elect the coordinator, `elect_coord`, is also shown in Figure 26. All `mservers` not elected as coordinator exit the case statement and return to normal idling state. The elected coordinator continues and starts the two-phase protocol by forming and broadcasting the `AGREE_J` message to the group. The non-coordinators receive the message and process it. Figure 27(a) shows the code for the coordinator, while Figure 27(b) shows the code for the non-coordinators.

```c
1.    case M_JOIN_REQ:
2.        if (tblshr (cs_tbl, 0, from.sin_addr.s_addr))    /* check if it is a duplicate */
3.            break;
4.        if (myrank != elect_coord (cs_tbl))    /* see if I am coordinator, if not discard msg */
5.            break;
6.    u_short elect_coord (tbl)
7.        struct table_entry *tbl;  
8.    {  
9.        int i;
10.       u_short lr = 65535;
11.       for (i = 0; i < MAXTLSIZE; i++)
12.           if (tbl[i].addr)  
13.               if (lr > tbl[i].rank && tbl[i].flag) /* lowest rank active member in table is elected */
14.                   lr = tbl[i].rank;
15.        return lr;
16.    }
```

Figure 26: Receiving a New `Mserver` Request for Join and Electing Coordinator

It must be clear that a lot of error-checking code has been omitted here to clarify the specific points of discussion. Also part of the code is slightly modified so that it is presented in a more complete form.

The two-phase protocol code would work in all cases if we had a totally fail-proof environment. Unfortunately, one or more `mservers` of the group may fail while processing a change. If a non-coordinator member fails during phase I, then the coordinator finds it out in line 7 of Figure 27(a). Since it is discovered by the end of phase I of the current change, it is ensured that the failure of the member is of lower priority [28, 29]. The failed member is added to the failed list and processed later.

46
/* NON-COORDINATOR */

/* extract info about new member */

/* ack the join */

/* if acks received proceed to phase II */

/* wait for commit */

/* if commit */

/* commit */

/* process_co_fail */

Figure 27: Code for AGREE_J for the Coordinator and the Non-Coordinator.

f. Processing Coordinator’s Failure

The coordinator can also fail during a change. First, the other members must learn about the failure. This is done by timing out the waiting cycle for a commit message. If the coordinator fails, then at least one non-coordinator member times out while waiting for the commit and assumes that the coordinator failed. Then it forms and sends out a co_fail message, which includes its status. Then it enters the process_co_fail function which is described in Figure 28. The other members may be in the same position waiting for a commit or may have committed already (depending on how far into the change protocol the original coordinator has gone). Upon receipt of the co_fail message, they also enter the process_co_fail function.

In the function itself there are three sections. In the first section (lines 4 through 35), the exchange of co_fail messages and collection of status for all members of
the group takes place. In the second (lines 37 through 46), a new coordinator is elected and the members that did not reply are added to the fail list. The lowest rank member that completed phase I of the original change (sending an ack) becomes the new coordinator. The third section looks like an ordinary two-phase protocol, like the one for the join shown in Figure 27. It is omitted from Figure 28 for compactness.

In the exchange of the status section, each member sends its status with the broadcast of a co_fail message as in line 23 and then tries to receive the same message from every other member in the group. The co_fail message uses the data_len field of the message to carry an integer showing the status of the member with respect to the current change. All answers are placed into an array and they are processed later in the election phase.

The lowest rank active member that has finished phase I of the current change but not committed, is elected as the new coordinator. Then if at least one member has committed the current change, it is marked at line 40. Depending on this, the new coordinator will decide if it will go through a complete three-phase protocol if none of the group members had committed, or will use the compressed three-phase protocol if at least one of them had. After finishing with the original change the failure of the old coordinator is being processed as usual with the same new coordinator starting a two-phase change protocol.

The change described here in Figures 27 and 28 is a new member's Join. Almost the same code can be used for a Leave, occurring when a member fails. The slight differences are already described in subsection (d) of this section.

D. REMARKS

The current level of mserver implementation takes care of the Join, Leave and Coordinator's (of a Join) failure changes. Implementation of a coordinator's failure while processing a failure is expected to be similar. Additional changes may be added when the implementation will expand to include top-down hierarchy messages (parent-child relations). The code described here can be used as a guide to implement the processing of
any new messages. The compilation, running and testing of this code is described in the Appendix.
1. void process_co_fail (mptr, p_mptr)
2. 	struct message *mptr, *p_mptr;
3. ...
4. /* reset reply table and count active members in group */
5. 	n_mbrs = 0;
6. 
7. 
8. 
9. 
10. }
11. /* subtract detector and mark its answer */
12. 	n_mbrs -= 1;
13. 
14. /* if other than detector, subtract me also */
15. 
16. 
17. 
18. }
19. mptr->sender_gid = myrank; /* adjust sender's gid into msg to reflect me */
20. /* send co_fail msg and collect answers; re-send maxtries times */
21. 
22. 
23. 
24. if (n_mbrs < 0) break; /* if all answers received, stop */
25. 
26. while (n_ans < n_mbrs & & time_out (t)) /* receive until all reply or time expires */
27. 
28. if [recv_msgg (uc, mc, &mcast, &rmsg, &who, &t) > 0] 
29. 
30. 
31. 
32. /* update reply table */
33. 
34. }
35. /* if -if */
36. } /* for */
37. /* elect new coordinator based on answers */
38. 
39. /* mark if a member has committed */
40. 
41. /* new coord must have replied and been in agree phase */
42. 
43. /* member in agree phase */
44. 
45. 
46. }
47. /* process_co_fail */

Figure 28: Code for Function Process_Co_Fail.
V. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

This thesis presented an implementation of two basic components of the MS, the \texttt{mcaster} and the \texttt{mserver}. A set of useful MS-related utility functions was implemented and each one tested independently. These functions were organized in different files, providing good modularity and portability of the code.

A complete working \texttt{mcaster} was implemented and tested to work in uc-mc mixed as well as uc-only LANs, handling one or more groups. The present version of \texttt{mcaster} is capable of propagating correctly any kind of group messages.

The \texttt{mserver} implemented is capable of creating and maintaining an \texttt{mserver} group at one level. New member joins and failures are handled correctly and according to the protocol.

B. FUTURE WORK

There are a lot of possible paths for follow-up work. First, to demonstrate that the MS is truly scalable to global proportions, a parent-child model must be added to the \texttt{mserver}, and tested on progressively larger scales. To achieve this, the basic \texttt{mserver} model presented in this thesis must be enhanced to include handling of messages related to the parent-child model. Also the internal state of the \texttt{mserver} must be enriched to support these relations. The functions developed for the current implementation can be used as a basis for future development.

Second, the partition resolution protocol must be implemented and tested since it is one of the strong advantages of the MS. A strategy to test partition handling without failing the network will need to be designed.

Third, a complete performance analysis of the operation, overhead, network constraints, service latency and functionality of the MS must be accomplished. Also, an implementation of an MI is needed to provide the \texttt{mserver} group the necessary interface to applications groups.
Fourth, an Application Programmer's Interface (API) as described in [28], must be defined completely and implemented, to give a tool to programmers to use the MS. Using this API, the next step is to create some test applications to take advantage of the MS use.

Finally, the MS architecture must be revisited in the future, 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.
APPENDIX. COMPILe, RUN AND TEST THE MS

A. COMPILING

Table I provides the names of the files currently used in the MS implementation, and a short description of the contents and the names of the functions defined in each file.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Associated header</th>
<th>Description</th>
<th>Included Files</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>msutil.c</td>
<td>msutil.h</td>
<td>Definitions of global constants</td>
<td>None</td>
<td>leave_mc_grp, join_mc_grp, init_socket, addrcmp,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Definitions of structures</td>
<td></td>
<td>form_msg, send_msg, recv_msg, receive_msg,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Utilities collection</td>
<td></td>
<td>set_timeout, timed_out, search_gid_list, add_gid_entry,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>copy_gid_list, extract_gid_list, delete_gid_list,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print_in_addr, print_sock_addr,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print_sock_info, print_hostent, print_msg,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>printf_gid_list, printf_nos, logo, make_checksum,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>chk_checksum</td>
</tr>
<tr>
<td>mcaster.c</td>
<td>—</td>
<td>Multicast emulator program</td>
<td>msutil.h</td>
<td>search_group_list, search_member_list, add_group,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add_member, join_group, remove_group,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>remove_member, leave_group, mcast,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print_group_list, print_member_list</td>
</tr>
<tr>
<td>msutil2.c</td>
<td>msutil2.h</td>
<td>Utilities collection used by mserver program only</td>
<td>msutil.h</td>
<td>tbrth, add_table_entry, rm_table_entry, rm_fail_entry,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extract_int_state, form_int_state,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extract_change_info, form_change_info,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print_core_table, elect_coord</td>
</tr>
<tr>
<td>reliable.c</td>
<td>reliable.h</td>
<td>Definitions and declarations of reliable link functions</td>
<td>msutil.h</td>
<td>reliable_link, relmlink</td>
</tr>
<tr>
<td>message.c</td>
<td>message.h</td>
<td>Definitions and declarations of message processing functions</td>
<td>msutil.h</td>
<td>process_msg, process_co_fail</td>
</tr>
<tr>
<td>mserver.c</td>
<td>—</td>
<td>Mserver main program</td>
<td>msutil.h, msutil2.h, reliable.h</td>
<td>None</td>
</tr>
</tbody>
</table>

Table I: Description of the Files for the MS

To compile the above files, the ANSI C compiler (acc) was used over in CC department. There are a lot of different hosts in the CC network and not all of them have the IP multicast extensions [30] installed. If these extensions are not installed then it is not possible to compile the msutil.c file (specifically the join_mc_grp function) and subsequently, all the files that include it.
Table II gives some of the hosts on the NPS Computer Center LAN, their type and their Internet address. If the compilation takes place on a specific type of machine (SUN or SGI), then the programs can run only on the same type of machine.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Internet Address</th>
<th>IP Multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>aloft.oc.nps.navy.mil</td>
<td>SGI</td>
<td>131.120.53.2</td>
<td>Yes</td>
</tr>
<tr>
<td>momik.oc.nps.navy.mil</td>
<td>SGI</td>
<td>131.120.53.5</td>
<td>Yes</td>
</tr>
<tr>
<td>megrez.oc.nps.navy.mil</td>
<td>SUN</td>
<td>131.120.53.8</td>
<td>Yes</td>
</tr>
<tr>
<td>acener.oc.nps.navy.mil</td>
<td>SUN</td>
<td>131.120.53.80</td>
<td>Yes</td>
</tr>
<tr>
<td>sp25420x.oc.nps.navy.mil</td>
<td>SUN</td>
<td>131.120.254.20x</td>
<td>Yes</td>
</tr>
<tr>
<td>mx502yy2.oc.nps.navy.mil</td>
<td>SUN</td>
<td>131.120.50.2yy</td>
<td>No</td>
</tr>
</tbody>
</table>

1 \(x = 1, 2, \ldots, 8\)

2 \(yy = 11, 12, \ldots, 17\)

Table II: Host Machines on NPS Computer Center LAN

All files must be in the same directory at the compilation time. If a file is in a different directory, the path must be included in the `#include` directive. To compile and link successfully a file that includes other files, all those files must previously be compiled without problems. For example, if a change has been made to `mserver.c` file, the command line for the compilation should read:

\[
\text{%>acc -c mserver.o mserver.c} \\
\text{%>acc -o mserver mserver.o msutil.o msutil2.o reliable.o message.o}
\]

If a change has occurred in the `msutil.c` file then the sequence of commands should be:

\[
\text{acc -c msutil.o msutil.c} \\
\text{acc -c msutil2.o msutil2.c} \\
\text{acc -c reliable.o reliable.c} \\
\text{acc -c message.o message.c} \\
\text{acc -c mserver.o mserver.c} \\
\text{acc -o mserver mserver.o msutil.o msutil2.o reliable.o message.o}
\]

For this purpose, it is better to use a Makefile. A sample Makefile is in the `~mserver/compile` directory. To debug the files using the debugger, the `-g` option switch must be
included in all lines of compiling, so that the compiler generates information for debugging.

**B. RUNNING**

After successful compilation, it is time to run the *mserver*. As many command tool windows as the *mserver* that are going to run plus one more for the *mcaster* are needed. The remaining procedure is as follows:

1. Remote login (*rlogin*) to a different host from each command tool window.

2. Run *mcaster* at one of these hosts and mark its IP address printed at the end of the initialization of *mcaster*. The *mcaster* must run on a mc host, if there are going to be servers running on mc hosts. If there are no hosts running on uc hosts, then *mcaster* is not needed.

3. Start the first *mserver* as described on page 33. The creation of a new group takes a couple of seconds, since the *mserver* tries to find out if other members exist.

4. Start the rest of the *mservers*, delaying the second one 15 seconds after the first one. The *mservers* should join sequentially and each one should print its own core-set table, which should be the same at every place. At the same time, the *mcaster* shows the traffic of the messages going through its channels.

5. *Mservers* should monitor each other in a ring schema as described by the monitor protocol. Monitoring query takes place every one minute for each *mserver*.

**C. TESTING**

At this level of implementation, the following tests can be performed and their results can be observed:

1. **Add a new member:** Open a new command window, remote login to a host not in use by other *mservers* or *mcaster* and start a new *mserver* program. The join request shall appear in every member of the group. The lowest rank member is the coordinator. It proceeds with the two-phase protocol and finally the new *mserver* is accepted as seen in the core-table printed in every member, that contains the information for the new member.

2. **Leave (fail) a member:** With a group running, go to a command window and kill one of the *mservers*. Soon the *mserver* monitoring the failed one will discover the failure and start a two-phase protocol, with itself being the
coordinator. In the end, every member prints its own core-table in which the information for the failed member has been removed.

3. **Make a member slow:** In a command window running an `mserver`, suspend (CTRL-Z) the process. Wait until monitoring finds out. After the process of the fail, recover the suspended member. After a while, it tries to monitor its clockwise member (of the old group), which denies to accept the query and does not respond. Then sequentially, the suspended and recovered member fails all the members of the old group. In the end, it is alone in its own group, while the other members run their own group. The problem of naming the two different groups has not been solved yet.

4. **Add a new member and fail another while in the process of join:** Before starting the join, kill a member and immediately (before monitoring finds out about the fail) start a new `mserver` to join. The coordinator finds out about the fail from the two-phase protocol for the join and adds the failed member to the failed list. After completing the new member's join, the coordinator process the fail as usual.

5. **More than one groups running at the same time:** Start and setup two different groups of one or more `mserver` each. The two groups should differ in the class D address or the group name or both. After the setup, both groups function independently. All above scenarios can run in one or both groups at the same time.

6. **Failure of the coordinator of a join:** To run this test case, a special version of an `mserver` program is needed. Make a faulty `mserver` (called `fmserver`) that has some code lines that simply delay just before the second phase of the join protocol. Since the protocols are described in file `message.c`, such delay lines should be added there as shown in Figure 29. Set up a group as usual, running the `fmserver` as the first (lowest rank) member. Then start a new `mserver`. While the `fmserver` receives all responses from members and delays (end of phase I) kill it. The other members wait for the commit message. The first that times out, starts a three-phase protocol by broadcasting a `co_fail` message. After that, all members exchange status through subsequent `co_fail` messages. Then the join of the new member gets processed. In the end the old coordinator (`fmserver`) gets failed and removed from the group and core-table.
1. /* file MESSAGE.C */
2. ...
3. case M_JOIN_REQ:
4. ...
5.    /* if acks received proceed to phase II, form COMMIT */
6.    free (data);
7.    free (&msg);
8.    /* -- FAULTY SERVER GETS SLOW -- */
9.    sleep (3);
10. ...
11. /* send COMMIT */
12. ...

Figure 29: Delaying a Faulty Mserver (Fmserver) to Test Coordinator's Failure
LIST OF REFERENCES


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, California 93943-5101

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

4. Chairman, Code CS
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5118

5. Professor Shridhar B. Shukla, Code EC/Sh
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

6. Professor Gilbert M. Lundy, Code CS/Lu
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5118

7. LTJG John Kostrivas
   310, Hatten Rd.
   Seaside, California 93955

63