A NETWORK MANAGEMENT FACILITY FOR A FAULT-TOLERANT DISTRIBUTED INFORMATION RETRIEVAL SYSTEM

by

Allen Quoc-Luan Tran

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2000 by Allen Quoc-Luan Tran
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-53394-8

Canada
Abstract

A Network Management Facility for a Fault-Tolerant Distributed Information Retrieval System

Allen Quoc-Luan Tran

Master of Science

Graduate Department of Computer Science

University of Toronto

2000

This thesis presents the design and implementation of a Network Management Layer to provide a group membership facility and an atomic modification facility that is typically required by distributed applications. The types of applications targeted are distributed information retrieval systems. The layer is designed to facilitate the addition of nodes that wish to become part of the system as well as the removal of members that have failed and are no longer useful. Furthermore, an atomic commitment facility is provided to allow the members to accomplish atomic operations ensuring system consistency. The Network Management Layer has been designed to be embedded into a more comprehensive layer, the Reliable Storage Management Layer, that manages the data for information retrieval systems in a fault-tolerant manner.
Acknowledgements

I would like to acknowledge the Natural Sciences and Engineering Research Council (NSERC) of Canada for funding this research. I would also like to thank Charles Clarke for his insightful suggestions and recommendations in all aspects of the thesis. Those long walks were very productive. Finally, I would like to thank Daniel Tersigni for proof-reading this thesis in its most primitive form and providing a “theory person”’s perspective.
## Contents

1 Introduction 1
   1.1 Information Retrieval Systems .......................... 1
   1.2 The MultiText Project .................................. 3
   1.3 The Reliable Storage Management Layer .................. 5
      1.3.1 The Data Placement Scheme .......................... 5
      1.3.2 Managing Failures ................................ 6
      1.3.3 Managing Additions ................................. 9
      1.3.4 Managing Data Updates ............................. 10
   1.4 The Network Management Layer ........................... 10

2 Background 14
   2.1 Distributed Storage Systems ............................ 14
   2.2 Basic Distributed Theory ................................ 16
   2.3 Distributed Impossibility Results ....................... 17
   2.4 Atomic Commitment ................................... 19
   2.5 Group Membership ..................................... 20
   2.6 Cluster Computing .................................... 23
   2.7 Network Layer Performance ............................. 25

3 Design 27
   3.1 Objectives ............................................. 27
3.1.1 Behaviour Under Failures ........................................ 27
3.1.2 Degraded Systems ................................................. 29
3.1.3 Design Road Map ................................................... 30
3.2 Environment .......................................................... 31
3.3 Design Overview ....................................................... 32
  3.3.1 The Initiation Stage ............................................. 33
  3.3.2 Client Application Decision ................................... 34
  3.3.3 The Commitment Stage ......................................... 34
3.4 The Global State Machine ........................................... 35
  3.4.1 States ............................................................ 36
  3.4.2 Commit/Abort Indicators of Alive states .................... 39
  3.4.3 State Vectors ................................................... 39
3.5 State Transitions ..................................................... 40
  3.5.1 Initiation Transitions .......................................... 40
  3.5.2 Commitment Transitions ....................................... 43
  3.5.3 Consistency of Commitment Transitions ...................... 45
  3.5.4 From Failure-free to Failure-prone Services ................ 46
3.6 Accurately Detecting Failures ...................................... 48
  3.6.1 Alive Vectors .................................................. 50
3.7 Message Filtering .................................................... 50
  3.7.1 Modification Numbers .......................................... 51
  3.7.2 Last Commit Numbers ......................................... 52
  3.7.3 Message Sequence Numbers ................................... 56
3.8 Determining the Initiator ........................................... 57
  3.8.1 Steady Systems ................................................ 60
  3.8.2 Steady Systems With Failures ................................ 61
  3.8.3 Initiations ...................................................... 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8.4</td>
<td>Inconsistent Initiations</td>
<td>62</td>
</tr>
<tr>
<td>3.9</td>
<td>Transitions In The Presence of Failures</td>
<td>63</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Blocking Nature in the Presence of Failures</td>
<td>66</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Members Recovering From Soft Crashes</td>
<td>67</td>
</tr>
<tr>
<td>3.10</td>
<td>A Timing Analysis of <em>Steady</em>ₙ and <em>Failed</em>ₙ(r)</td>
<td>68</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Supporting A Modification</td>
<td>68</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Conflicting Initiations</td>
<td>68</td>
</tr>
<tr>
<td>3.10.3</td>
<td>Estimating Message Delay</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>75</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation Details</td>
<td>76</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Information Maintained by the GSM</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Member Timeouts and Broadcast Timeouts</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Determining <em>Comatose</em> and <em>Critical</em> Members</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>Evaluating <em>Steady</em>ₙ and <em>Failed</em>ₙ(r)</td>
<td>80</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Communication Protocol</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>GSM To GSM Message Protocol</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>GSM-Client Message Protocol</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>Client Connections</td>
<td>84</td>
</tr>
<tr>
<td>4.2</td>
<td>Performance</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>Future Work and Conclusions</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Integration of the NML with the RSML</td>
<td>87</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Node Controller</td>
<td>87</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Application Update Proxy</td>
<td>89</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Disk Management Layer</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Other RSML and NML Applications</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>Handling More Than One Failure Per Recovery Period</td>
<td>92</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.3.1 A Framework for a $k$-Failure Model</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>5.3.2 An Extension of the Crown Prince Concept</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>5.3.3 Expanding the States and Transitions</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>5.4 Summary and Contributions</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>

Bibliography                          96
## List of Figures

1.1 The MultiText architecture ........................................... 4
1.2 The perspective of the MultiText application ....................... 4
1.3 The data placement scheme for four members ......................... 6
1.4 The active data segments for four members during a failure ........ 7
1.5 The Network Management Layer as a component of the RSML and as a stand-alone facility ............................................ 11

3.1 An example of a degraded system ....................................... 30
3.2 The states and their relationships ...................................... 36
3.3 The substates within the modification states ......................... 38
3.4 The substate hierarchy. .................................................. 42
3.5 Commitment transitions for all modifications ......................... 47
3.6 A message delay problem ................................................ 56
3.7 Message filtering algorithm .............................................. 58
3.8 Change of support from a leader initiated removal to a crown prince initiated removal of the leader ............................... 70
3.9 Change of support from a crown prince initiated removal of the leader to a leader initiated removal ............................................ 70
3.10 Change of support from a leader initiated addition or update to a crown prince initiated removal of the leader ............................... 72
3.11 A method to calculate the upper bound on message delay ........ 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12</td>
<td>How the leader and crown prince determine support for initiations</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>The main event loop of the GSM</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>The client-GSM communication data structure</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>Average soft crash detection time</td>
<td>86</td>
</tr>
<tr>
<td>5.1</td>
<td>The RSML components and their interaction with the application</td>
<td>88</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Information Retrieval Systems

The realm of Information Retrieval (IR), including Digital Libraries and Web Search Engines, is perhaps one of the most suitable application areas for distributed and cluster computing. Transforming a centralized IR system to a distributed IR system is typically straight-forward and cost-effective. A common technique is to partition the data roughly evenly among the given number \((N)\) of commodity-constructed workstations, each effectively owning a part of the original data. A central process issues a query to all the workstations, then collects and organizes the results. The effect is an almost \(N\)-fold decrease in processing time per query or a linear speed-up with an increase in number of nodes [29].

In addition, the hardware to construct a distributed IR system using a network of workstations is relatively inexpensive. This is due to the low-cost commodity components such as high-capacity storage media, powerful processors, and fast communication networks. Today’s hard disks are as large as 75GB while at the same time, as cheap as $6/GB. This makes them well-suited for the large amounts of data handled by IR systems. Processing unit speeds have risen according to Moore’s Law providing ample
processing power, while prices continue to drop with each generation of processors. Commodity network components also provide extremely fast and inexpensive communication media such as Fast Ethernet and Gigabit Ethernet that provide a seamless transition from traditional Ethernet. Although query sizes and result sets tend to be small and require little bandwidth to communicate, a high-capacity network is useful for maintenance periods where potentially large amounts of data need to be transferred across the network. One bottleneck to IR systems has been disk access speeds. The amount of time it takes to retrieve a block of data has not improved significantly as have other components. Distributing an IR system would thus alleviate such performance bottlenecks. The benefits and simplicity of converting a centralized IR system to a distributed IR system outweigh the costs.

An alternative to this network-of-workstations architecture is a single n-CPU machine. However, a typical n-CPU machine is typically much more expensive than n single-CPU workstations connected using a high speed communication network. In addition, the advantage of a network-of-workstations approach over a single multi-processor machine is its flexibility and scalability. Additional workstations can be integrated to improve query execution times or to simply expand the data set. Individual workstations can be replaced upon failure. This can reduce down-time and enhance availability.

Nevertheless, the network-of-workstations architecture also has its disadvantages. There are many issues related to availability, reliability, and the overhead to manage a distributed IR system. Assuming that node failures are independent and that time to failure is exponentially distributed, the mean-time-to-failure (MTTF) of the system as a whole decreases in proportion to the inverse of the number of nodes it contains [11]. If not managed properly, this can have a profound negative effect on the availability and reliability once the system grows. There is a need for system management facilities to mask these node failures as well as node additions.

Load-balancing is another critical issue that surfaces when failures occur as well as
when nodes are added. In order to realize the benefits of all nodes, data must be distributed relatively evenly since the bottleneck of the query time is the slowest node. Automatic re-balancing after failures or additions is required as manual re-balancing can be tedious. Thus, the management of failures and additions must be handled transparently in order to provide acceptable levels of availability, reliability, and performance.

1.2 The MultiText Project

*MultiText* [12] is an example of a distributed IR system. The MultiText project, undertaken at the University of Waterloo, is an information retrieval system concerned with techniques used to index and retrieve very large amounts of data. The ultimate goal is to efficiently index and retrieve a significant fraction of all electronically available data.

The development of this project requires many important issues be addressed. One of these issues is the query response time. When users submit queries, the solutions should be returned quickly despite the potentially extensive amount of data that must be searched. The basic factor underlying the speed of searches is the design of the algorithm that the MultiText application must apply to the data. However, an orthogonal approach to enhancing the search speed is through distributing the data among multiple nodes and having the application on each node perform the algorithm in parallel (see Figure 1.1). Each node would contain a copy of the MultiText application responsible for processing query requests and updating data on its physical storage (see Figure 1.2). A *Query Marshaller/Dispatcher* acts as the central process that is responsible for accepting query requests from clients, issuing the queries to all the nodes, and integrating the responses. An *Update Marshaller/Dispatcher* is responsible for accepting data update requests and issuing these requests to the individual nodes.

However, the problem with the MultiText architecture in this form is its lack of resilience. This architecture is not fault-tolerant. The failure of even one machine (or
Figure 1.1: The MultiText architecture for $N$ nodes.

Figure 1.2: The perspective of the MultiText application. It may receive many query requests at a time, but only one update request at a time.
node) would result in the loss of data and thus query solutions would be incomplete until the failed node recovers. A challenge for MultiText is to ensure it is highly available while having a minimal impact on performance. A goal of MultiText is to transform the distributed IR system into a fault-tolerant distributed IR system.

### 1.3 The Reliable Storage Management Layer

The problem with MultiText in its current form is a lack of a transparent mechanism to handle failures. Hence, the *Reliable Storage Management Layer* (RSML) has been designed to handle this deficiency. The responsibility of the RSML is to provide reliable storage over multiple nodes, transparently handling single failures. In order to accomplish this, the RSML must provide data redundancy and management facilities to seamlessly handle node failures, node additions, and modifications to the IR system’s data. For example, if the data collection needs to be augmented in the IR system, the RSML manages the data is stored in the nodes, and how it is replicated. In the event a node becomes unavailable, the RSML would be responsible for making this data available.

#### 1.3.1 The Data Placement Scheme

The key to a successful RSML is the data placement scheme used. A *data placement scheme* or *redundancy scheme* describes the organization of data across the nodes. The choice of data placement scheme can directly affect management overhead as well as performance. The current RSML design uses a data placement scheme similar to the Tiger Video Server [6]. Data is first divided (approximately) equally among the $N$ nodes. This data is called the node’s *primary* data. Then each node’s primary data is distributed (approximately) equally among the remaining $N - 1$ nodes. These redundant pieces are called the node’s *secondaries*. The result of this data placement scheme is a doubling of the original amount of data. For example, if there are four nodes that form the system,
then the data is divided into four equal primaries. Then, each block of primary data is divided into three pieces and replicated, one piece on each other node. Node 0 would divide its primary into three pieces, and place one piece each on nodes 1, 2, and 3. These redundant pieces form the secondaries. See Figure 1.3:

1.3.2 Managing Failures

When no failures have occurred, each node only searches its primary for any query request. In the event a node is suspected to have failed, every other node must then activate its secondary for the (potentially) failed node. On any node, an application process is designated for each data segment (primary or secondaries). The application process for the primary data is always active; however, the application processes designated for the secondaries initially lay dormant. When a node is suspected to have failed, the secondary for that node is then activated (see Figure 1.4). The terms potentially failed and suspected are used because it may not be the case that the node in question had actually failed but rather other system or network issues could make it appear as if it had. However, it is important to detect and respond to this potential failure as quickly as possible instead of waiting for more conclusive evidence of a failure. Delaying the activation of the secondaries could result in query solutions being incomplete for that period of time.
Active region of data.  Inactive region of data.

(a) No failure has occurred.

(b) The failure of Member 1 (before its removal).

Figure 1.4: The active data segments when (a) no failure has occurred and (b) the failure of member 1 has occurred.
A problem can occur if the secondaries are activated for a node that is suspected to have failed when in fact, it has not. Duplicate query responses would be returned to the central process (the Query Marshaller/Dispatcher in the case of MultiText). However, it is the responsibility of the central process to reconcile any duplication. The only performance loss is the time, possibly wasted, searching through the secondaries. However, the amount of time required to search through the secondaries, whether it is wasted or actually required, is minimized. A consequence of this data placement scheme is that it maintains load-balance even during node failures. As mentioned, this aspect is imperative to performance as the query search time is only as fast as the slowest member. Thus, a redundancy scheme where the responsibility of a failure of one node rests completely with one other node would result in the doubling of the search time. With this scheme, since the amount of data from the failed node is distributed evenly, the load is still balanced among the remaining non-faulty nodes.

This redundancy scheme is able to sustain the failure of one node while at the same time, ensuring that the load is balanced among the remaining non-faulty nodes. However, the failure of a subsequent node would result in the loss of data. For example, in Figure 1.4(b), if member 2 now fails, the set of data from the union of members 0 and 3 would not equal the original set of data. Some data will then be unavailable. One way to handle an arbitrary number of failures would be to replicate all the data on all the nodes. However, this is infeasible considering the potential amount of data that must be indexed and retrieved.

The currently proposed RSML design handles this situation by attempting to reconstruct the data placement scheme with one less node. This can be accomplished because the data placement scheme allows for one node failure with no loss of available data. In the context of the previous example (Figure 1.4(b)), conceptually, the failure of member 1 would involve reconstructing the data placement scheme but with one less member. More specifically, members 0, 2, and 3 need to divide all the data into three equal primaries
each placed on one of the surviving nodes. Then, each primary is divided into two pieces placed at the two other surviving nodes. In practical terms, this would involve each member taking the secondary of the failed node, incorporating this data into its primary, and ensuring this data gets duplicated in the secondaries on the other two nodes. The result is that the redundancy scheme is restored and another failure is sustainable. In theory, if no additions occur and given enough disk space on a single member, the system would be able to sustain as many as $N - 1$ successive failures.

### 1.3.3 Managing Additions

Given the pace at which electronic data is growing, and the pace at which computer prices are declining, it is only natural to be able to extend the system by adding new nodes. The RSML is designed to seamlessly handle such events. Conceptually, this addition would involve reconstructing the redundancy scheme but with one more node included in the system. Practically, for each original node, this requires donating some of its primary data to the new node. The number of secondaries grows by one on each node; however, the size of each secondary shrinks, the converse of what occurs during a failure.

In a dynamic environment, failures and additions can occur frequently and the number of members in the system can be highly variable over a long enough period of time. Thus, the number of secondaries (directly related to the number of members) is not predictable. Fixing each secondary size can limit the scalability of the system. It is the responsibility of the RSML to manage this. In the current RSML design, the primary is allocated a fixed amount of space and the rest of the space is allocated to the secondaries. However, individual secondaries vary in size and are treated as virtual disks that may grow and shrink. A Disk Management Layer is responsible for managing the secondary space efficiently and allows for a highly dynamic secondary structure, theoretically allowing as many as required.
1.3.4 Managing Data Updates

Finally, data is rarely ever static. The RSML must also manage updates to the data collection. New data must be distributed to the primaries as well as replicated on the relevant secondaries. The opposite occurs when data becomes obsolete. These data updates must be done atomically in order to maintain consistency.

1.4 The Network Management Layer

The RSML is a large component responsible for the overall reliable management of data over multiple nodes. However, many of the facilities of the RSML can be modularized. One module is the network related facilities required by the RSML to complete its responsibilities. Hence, the Network Management Layer (NML) is introduced as an abstraction of the required network-related facilities. The overall objective of the NML is to provide a global view of the system for the RSML by facilitating node additions and failures. When a member fails, the NML is responsible for detecting the failure, initiating its removal, and finalizing the removal. Similarly, when a new node requests to join the system, the NML is responsible for facilitating its integration into the system. In addition, another objective of the NML is to provide the members with facilities to make an atomic decision. These atomic decisions are required by the RSML during data updates. For example, when the central process (the Marshaller/Dispatcher for MultiText) requires an update to the data collection, the RSML must ensure that this data is added atomically among all members, preserving the data placement scheme. The NML is responsible for guaranteeing this addition is accomplished atomically. The NML is essentially the glue that ensures system-wide consistency.

The features of the NML are designed to be part of a larger protocol such as the RSML. However, the NML is general enough to be complete in itself providing a plug-in to any application that would require these system management facilities (see Figure
Figure 1.5: The Network Management Layer as (a) a component of the RSML, or as (b) a service for applications requiring these system management facilities.

1.5. The view of Figure 1.5(b) is frequently used. However, whenever convenient, the context of the RSML in Figure 1.5(a) will be used.

More specifically, the facilities that the NML provides the client application are:

1. An early detection mechanism that notifies the local application in a timely fashion of potentially failed remote nodes. This service is important to allow the local application to make any time-critical response necessary to compensate for the potential failure. In the context of the RSML, this would involve activating the otherwise dormant application process associated with the secondary of the potentially faulty node. This is important because while the faulty node remains inoperable, solutions to queries can be incomplete. It does not indicate that the remote node has completely failed, but rather is the first phase in removing the node. If it turns out the potential failure was a mistake, the client application would be notified and it can tell the secondary process(es) to go back to sleep.

2. After a node has been suspected to have failed for a substantial period of time, the NML then proceeds to formally remove the node. The local application must be given the opportunity to indicate to the NML its ability to formally remove the
failed member, based on its ability to perform the necessary operations to recover from the failure. In the context of the RSML, this would involve the local node reconstructing the data placement scheme except with one less node. The decision from the local application is used by the NML to finalize the removal.

3. In the event a new node wishes to join the system, again the local application is notified by the NML to allow it to perform the necessary operations for the node's inclusion in the system. In the context of the RSML, this would involve all the nodes reconstructing the data placement scheme with one additional node. Similar to when failures occur, control is passed to the application for a decision and then back to the NML to finalize the addition.

4. A mechanism to allow the system to make an atomic decision. For example, in the RSML, periodic updates may be necessary to modify the data it stores. These updates must be performed atomically to maintain a consistent data set and redundancy scheme. The NML is responsible for ensuring updates are performed atomically. It is also responsible for providing a mechanism for a local client to indicate that a data update is required. It can then proceed to initiate the update, again, passing control to the local application for a decision about its ability to perform the update. Control is then passed back to the NML to finalize the data update.

5. A mechanism that would indicate the system is in a "degraded" state and only an addition would allow the system to return to normal. This feature may be somewhat unusual, but it is needed in the context of the RSML. In the RSML, the failure of one member could result in the inability of the remaining members to reconstruct the given data placement scheme and thus, the system would not be able to sustain another failure. This usually occurs because one or more members are not able to absorb the data of the failed member and/or are not able to distribute this new
primary to the saturated secondaries due to disk constraints. In this situation, it would be necessary for a new node to be added to alleviate the disk usage. Meanwhile, it would make no sense to attempt further removals or data updates.

The objective of this thesis is to focus on the design and implementation of the Network Management Layer. The NML is still intended as an embedded service for the RSML which itself attempts to provide a level of fault-tolerance for MultiText. An overview of the other components of the RSML is presented in Section 5.1. More specific details can be found in a thesis by Kevin Harris [28].
Chapter 2

Background

2.1 Distributed Storage Systems

There have been a number of designs for fault-tolerant distributed storage systems. Many reliable storage systems have been proposed in the same context as RSML has been for MultiText [26] [36] [6] [8]. Most of these proposed storage systems are able to handle single server failures; however, any further server failures would result in an incomplete available set of data.

Palladio [26] is a distributed storage system that provides a virtual data storage service for applications running on a cluster of hosts by providing virtual stores. It supports dynamic changes to a virtual store’s layout to enhance availability and resource utilization. The Tiger Video Server [6] is a fault-tolerant real-time file-server predominantly designed to support applications that require a constant sustained rate of data delivery, such as multimedia applications, although traditional file systems are also supported. It is devised to be resilient to one failure by using a similar data placement scheme as RSML. The replicated data is placed evenly over the other servers (known as cubs). Petal [36] is a distributed storage system much like Palladio. It utilizes virtual disks much like Palladio’s virtual stores, but unlike Palladio, it uses the same data placement strategy.
for all the virtual disks. A simple chained-declustered data placement scheme is used, which duplicates data at the server to the "right". The deficiency in this scheme is that the effect of one server failure causes an imbalance in its replicated data on the surviving servers. RSML and Tiger Video Server's data placement scheme does not suffer from this potential dilemma, although their schemes are slightly more complicated to maintain.

Nevertheless, the loss of data after one server failure suffered by these large scale storage systems is undesirable since this failure may take time to be externally detected and repaired. RSML attempts to improve upon this by being resilient to one failure for a given "period". A "period" is the time required to reconstruct the data placement scheme among the surviving servers. After a server has failed and the data redundancy scheme reconstructed, another failure is sustainable.

One fundamental, but challenging requirement of all distributed storage systems is to design a protocol to handle failures, whether it be server or disk failures. Disk failures are somewhat easier to handle while server failures can be quite difficult to accurately detect. The failure detection protocols are just as diverse as the data placement schemes. The Tiger Video Server detects failures using a deadman protocol [6]. Each cub is responsible for watching the cub to its left and sending pings to its right while Petal uses Leslie Lamport's Paxos algorithm [34] to maintain server membership. Palladio uses a layout control protocol [26] to detect manager and device failures.

RSML is no different. It requires a membership protocol to determine when a server has potentially failed and to compensate for it. This is the purpose of the NML. As a complimentary service, the NML provides an atomic commitment service to handle dynamic changes in data. Modifications to the RSML servers are required to appear atomic just like distributed database transactions. Atomic commitment and group membership have been studied in much detail in the academic community.
2.2 Basic Distributed Theory

Distributed algorithms depend heavily on an accurate characterization of the behaviour of processes and the underlying communication medium. Assumptions about these factors are necessary to correctly write an algorithm that satisfies the given requirements. Timing assumptions, and process failure assumptions are probably the most widely studied components affecting distributed algorithms. Distributed systems generally come in two (extreme) timing models: the synchronous model and the asynchronous model, and two (extreme) process failure semantics: crash failures and arbitrary failures.

The *synchronous* model is the simplest timing model and assumes that each node or process takes steps simultaneously. That is, time is sliced into equal units with each process simultaneously performing all required steps, including sending and receiving messages, at each time unit. These assumptions of the system are generally impossible or inefficient to implement as a global discrete clock would need to be accessible by all processes. A system can approximate a synchronous system if there is a known upper bound on the relative speeds of correct processes and on the maximum message delay time. However, if these bounds are violated, then there may be no defined behaviour.

The *asynchronous* model, on the other hand, assumes that each process takes steps in arbitrary order at arbitrary relative speeds. Message transmission times are not bounded and some algorithms handle the situation where the message may not arrive at all. In general, time is disregarded at the process and communication levels. This model has been proven to be too strong and limiting to solve fundamental distributed agreement problems. Even one of the most basic problems in distributed computing, Consensus, can not be solved in this system [24]. Between these two timing models lies a spectrum termed the *partially synchronous models* [38], where, generally, processes have access to some timing information, but not as complete as in the synchronous model. Partially synchronous models tend to make assumptions directly based on the underlying network the algorithm is intended for and thus, are probably the most realistic models.
Process failure semantics are also an important aspect of distributed algorithms. These semantics describe the behaviour of processes or nodes when they fail. At one extreme lies the crash failure semantics in which a faulty process simply stops and performs no more actions. There may be other assumptions that describe how faulty processes recover. At the other extreme, arbitrary failure semantics specify that upon failure, the process can exhibit any type of behaviour, even acting maliciously to break the system. The algorithmic complications that arise from handling arbitrary failures can be very involved and thus, this failure model is only tackled in critical systems. Again, between these two extremes, there are also other semantics for process failures such as a faulty process failing to send or receive messages (known as omission failures).

2.3 Distributed Impossibility Results

Before a proper treatment of atomic update and group membership can be discussed, a description of some fundamental properties in distributed computing is necessary. At the heart of distributed computing algorithms is a form of agreement protocol: the processes (or nodes) must agree on a value or course of action to take and follow through with it despite a dynamic environment.

At the core of these agreement problems lies a more fundamental problem of distributed systems: Consensus. The Consensus problem [38] is a basic abstraction of general agreement problems that carves out their salient properties. Thus, if Consensus in the given model is impossible then so are the many agreement problems that encompass these properties. Furthermore, the impossibility of Consensus in a given model implies the impossibility of other distributed algorithms such as Terminating Reliable Broadcast. This, in many respects, is similar to reducing a non-computable problem to a given problem to claim that this problem is also non-computable.

In the Consensus problem, each process starts with a value \( v \) and at the end, each
process must reach an irrevocable decision (value) based on the following constraints:

1. No two correct processes reach different decisions (Agreement).

2. Every correct process eventually decides a value (Termination).

3. If a process decides $v$ then $v$ is the initial value of some process (Integrity).

$v$ may not necessarily be a value but may also be an action or decision.

Based even on the simple synchronous timing model, in the presence of communication failures, an algorithm for Consensus is impossible [27], even if process failures are not allowed. In addition, another fundamental result in the theory of distributed systems is that the Consensus problem in a time-free asynchronous system is impossible to achieve if we allow even just one process to fail (and the process follows crash failure semantics) and there are no communication failures [24]. The rationale behind this impossibility result is that processes cannot accurately determine if another process has crashed or is just too slow in taking steps. However, this decision is necessary to achieve this basic agreement problem. It is important to note that Consensus does not form the core of all agreement problems, but certainly many of the essential ones.

In a more comprehensive analysis of the Consensus problem, the timing assumptions of process computation, the timing assumptions of message transmission, whether the underlying network preserves message ordering, whether sends and receives are able to be performed in the same step or not, and the broadcast capabilities play integral roles in the ability to solve Consensus [16]. Dolev, Dwork, and Stockmeyer precisely classify when Consensus can be achieved and when it can not with respect to these factors [16]. For example, if processes are synchronous, but communication is asynchronous, and message order is not preserved, then there is no Consensus algorithm that can tolerate one failure, even if sends and receives are able to be performed in the same step and there are broadcast facilities.
Thus, the time-free asynchronous timing model is too strong to construct the required services for NML. The assumptions or requirements will have to be altered slightly to accomplish our goals. In general, communication failures and the purely asynchronous timing model render fault-tolerant agreement problems unsolvable.

2.4 Atomic Commitment

Distributed atomic commitment was the focus of much research in the early 1980s because of its need in distributed database systems. Generally, an algorithm solving atomic commitment requires the following properties to be satisfied:

Every member starts with a vote $\in \{\text{yes}, \text{no}\}$ indicating its ability to commit.

1. No two processes reach a different decision.

2. If a process decides to commit, then all the processes must have voted to commit.

3. If all processes vote to commit and there are no failures, then no process decides to abort.

4. Either:
   
   (a) Each correct process eventually decides.
   
   (b) If no failure occurs, then each process eventually decides.

The choice depends on the requirements.

An algorithm satisfying 1, 2, 3, and 4a is said to solve strong atomic commitment while an algorithm satisfying 1, 2, 3, and 4b is said to solve weak atomic commitment.

Atomic commitment is similar to Consensus. However, they have different allowable solutions. For example, if every member started with a yes vote ($v = \text{yes}$ in Consensus), atomic commitment would allow for the final decision to be no because of point 3. Failures allow members to end up with the no decision. However, this would violate Integrity of Consensus.
The two-phase commit (2PC) protocol [27] [35] is a popular protocol that satisfies weak atomic commitment. Every node sends its readiness to commit vote to the coordinator who is determined a priori. The coordinator then collects the votes, puts in its vote, and issues the ultimate decision to the other nodes based on the votes. Since it only solves the weak commitment variant, the failure of just one node or process can result in all processes blocking. However, the use of timeouts and termination protocols help prevent processes from blocking during failures [4]. A variation of 2PC that attempts to reduce blocking is the decentralized 2PC devised by Skeen [47]. There does not exist only one single coordinator in the decentralized version but rather all processes are potential coordinators. Although the protocol adopts more measures to avert blocking, these measures can be expensive in terms of the number of messages exchanged. Another variation of 2PC includes a linear version [27] [45]. Nevertheless, regardless of the protocol, there still exist scenarios where 2PC can block.

An alternative to 2PC is a protocol devised by Skeen [48] [47] [46] known as the three-phase commit. The three-phase commit protocol takes 2PC and extends it to solve strong atomic commitment. It uses three phases of message exchange, as the name suggests, to accomplish the feat. However, to ensure that the processes do not block, the termination protocol could potentially require much more than three phases. To prevent blocking, as many as three phases are required to handle each failure. The cost of this protocol is considerably more than 2PC and thus, is typically not used in favour of 2PC.

2.5 Group Membership

Group membership has also attracted much attention since its first definition for synchronous systems in 1991 [13]. Since then, the focus of attention has shifted to defining and designing group membership services in asynchronous systems. In general there are two classes of group membership protocols: primary-partition services and partitionable
services. In \textit{primary-partition} services, the members must maintain one agreed upon view of the existing membership whereas \textit{partitionable} services allow more than one view of the group to co-exist. The intuition for this classification is that primary-partition group membership services are designed for networks that are not partitionable, or that allow the group membership to change in at most one partition, typically the majority partition.

However, despite much research, a satisfactory formal definition for primary-partition and partitionable group membership services in asynchronous systems has been elusive. Ricciardi and Birman [42] and Dolev, Malki, and Strong [17] introduced rigorous definitions for each problem respectively, but despite their efforts, and further efforts ([44], [18]), they still remain unsatisfactory [2]. Unlike Consensus and atomic commitment, there still does not exist a formal definition for group membership problems.

Not only has a definition for the primary-partition group membership problem been elusive, but any "realistic" solution in the purely asynchronous model is impossible [10]. The basis of this result directly relies on the Consensus impossibility result [24]. It may not be clear how the impossibility result applies. The group membership problem allows processes to remove a member even though it may not really have failed. Nevertheless, despite this additional seemingly stronger property, a \textit{Weak Group Membership (WGM)} was introduced [10] that has the properties that would be included in any reasonable group membership definition. It was then shown that even WGM is not solvable in time-free asynchronous systems with crash failure semantics, even if communication is perfectly reliable.

However, most primary-partition group membership services are time-oriented and use timeouts as opposed to being time-free [43] [33] [14] [39] [30]. Nevertheless, some partitionable services were designed to handle partitions and allow local groups to be formed [32] [1] [14] [50]. These services must also be careful when handling group merges. Many of these group membership services are found as part of a larger distributed system.
project and thus, rely on underlying protocols. For example, the protocol of Amir et al. [1] was built as part of the Transis system and relies on the message delivery subsystem to overcome arbitrary communication delays and message losses as well as to guarantee fast delivery of messages. Horus' [50] membership layer (VIEWS) relies on the Multicast Transport Service (MUTS). Horus was designed to be embedded into operating systems such as Mach or Chorus.

Furthermore, group membership services generally rely on a centralized leader to make decisions on behalf of the non-faulty group members. If the leader fails, however, then there must be some mechanism to determine a new leader. Ricciardi and Birman [43] used the notion of a leader known as the manager or coordinator. It executes a reconfiguration algorithm to select a new coordinator and, if necessary, re-establishes the group view. Jahaniana et al. [32] also uses the same concept of a leader and a crown prince to lead the group in the occurrence of a leader failure.

Aside from the asynchronous model, a network model more closely resembling existing network behaviour, introduced by Cristian and Fetzer, is called the Timed Asynchronous System Model [15]. In this model, a fail-aware datagram service is introduced [21] in which messages can be classified as "on-time" or "late". From this underlying structure, a class of fail-aware protocols has been devised including one for membership services [22] and leader election [23].

We have noted that group membership designs have neglected the surrounding environment. More specifically, clients who subscribe to the service typically do not have input into decisions made by the membership service. This passive nature may be desirable for certain applications in which members of the group are highly independent. For example, if we have a load-balancing application which allocates process execution based on the load within the distributed system, a service that simply detects faulty members is sufficient for the application to redirect subsequent processes to other processing nodes. A load-balancing server (the client to the membership service) may not need to partic-
ipate in the removal of the failed member or the addition of a new member; it simply makes use of the knowledge.

However, with many applications, it is necessary for the subscribing client application to have input in the decision-making process to maintain system consistency. For example, during the removal of a failed node or the inclusion of a new node, the RSML members must perform data redistribution amongst themselves to attain the desired data placement scheme. Furthermore, during atomic updates, each member of the system needs to ensure that the data is properly included into the primaries and distributed properly among its secondaries. Thus, the success or failure of the redistribution as a whole depends on the success or failure of the individual applications located at each member. The design of the aforementioned distributed services must provide a mechanism to allow the clients to provide their input into the decision of whether to proceed or not. This type of synchronization between the membership service and the application, and among the applications, is required to guarantee consistency in the system.

2.6 Cluster Computing

A cluster is a distributed processing system consisting of several interconnected stand-alone computers working together to give the illusion of one computing resource. Cluster computing has gained notoriety predominantly because of four trends [7]: high performance microprocessors, high-speed networks, standards for high performance distributed computing and the increased need of computing power that is inexpensive compared with traditional supercomputers. The advances in technologies and their availability as low-cost commodity components make clusters a more attractive alternative for high performance computing. Another advantage of clusters is the scalability they provide. A cluster can grow simply by attaching another computer to the communication interconnect. The benefits of cluster computing have already been demonstrated for Internet-server
workloads [25] allowing incremental scalability, providing "24x7" availability through fault-masking, yet still being cost-effective.

From a hardware standpoint, two main components can be identified for building clusters: the computational nodes and the communication medium. The computational nodes typically consist of the processors and the various levels of memory (cache, main memory, and disks). The computational power of processors continues to increase at a rapid rate while the price of memory has reached a point where it is almost negligible compared with other components. A more related concern of this thesis, given its nature, is the medium or network with which the computational nodes communicate. Some of the most common commodity network components are Ethernet, ATM, and Myrinet.

Perhaps the most popular communication medium among computers is Ethernet. A prominent reason is its simplicity and flexibility. Standard Ethernet has a bandwidth of 10 megabits per second (Mbps) which is relatively slow. Fast Ethernet was designed to be an upgrade path for Standard Ethernet and has a bandwidth of 100 Mbps. The most recent design in Ethernet is the Gigabit Ethernet which provides 1 Gbps (gigabits per second) bandwidth, a ten fold improvement over its predecessor.

Asynchronous Transfer Mode (ATM) is a switched virtual-circuit technology originally developed for the telecommunications industry. ATM works by dividing data into small fixed data size packets called cells that can be sent through the network on a virtual path. ATM has the characteristic of being able to maintain a constant rate of data flow. This is advantageous for multimedia applications that require a steady flow of video and sound data.

Myricom created a proprietary interconnect called Myrinet which provides a 1.28 Gbps full duplex communication bandwidth. Myrinet provides flow control, error control, and "heartbeat" continuity monitoring on every link. It uses low latency cut-through crossbar switches which handle fault-tolerance by automatically mapping the communication network in the event of failures. It also provides a mechanism for processes to bypass
the operating system and communicate directly with the network to send, receive, and
buffer packets. Its advantage over Ethernet is the low latency, high throughput, and a
programmable on-board chip that provides greater flexibility. One serious disadvantage
is the cost: approximately $1500 per host.

2.7 Network Layer Performance

An influential factor affecting the performance of distributed and cluster applications is
the overhead of the communication network. The network layer is typically the perfor-
mance bottleneck. The aforementioned communication media have variants that reach
bandwidths of 1 Gbps and more which can help alleviate this bottleneck. However, these
maximum capacities are only theoretical and may not be achievable in practice. More
importantly, it may not be achievable given the potentially demanding communication
requirements of the application.

The performance of the variants of ATM and Ethernet have been analyzed under
different conditions. Empirical experiments of Standard Ethernet under many situations
have been conducted by Boggs et al. [5]. The experiments involved stressing an Ethernet
by generating a total offered load that continuously exceeded the capacity of the network.
This attempts to gauge the performance in extreme circumstances. In general, as the
size of the packet increases, the Ethernet utilization also increases. However, this is
accompanied by an increase in the average transmission delay. Once packet sizes reached
approximately 512 bytes, Ethernet utilization reached approximately 95% even with 24
hosts in the cluster. An increase in the packet size beyond 512 bytes did not substantially
increase utilization. However, beyond 1024 byte packet sizes, the average transmission
delay begins to seriously suffer (with the same number of hosts). Below 1024 byte packets,
average transmission delay under a stressed network was less than 20ms even with a 24
host network vying for access.
The National Institute of Standards and Technology (NIST) conducted a more recent empirical study of a comparison between Asynchronous Transfer Mode (ATM) and Fast Ethernet (FE) [31]. Two identical 16 node clusters were used with the only difference being the communication medium. In general, with small message sizes, FE performs significantly better than ATM. Throughput is generally higher, transactions per second (using Netperf’s TCP/UDP request/response tests) was considerably greater, and latency of TCP and UDP was significantly lower with FE over ATM. However, as message sizes increase, ATM noticeably outperforms FE. With some benchmarks, this occurred once the message size reached as small as 512 bytes. However, with others, the noticeable difference occurred with a 100,000 byte message size.

Currently, there has been much research in improving network performance by decreasing the overhead of the software communication layer. Typically, to handle the transmission of data, system calls must be invoked through the operating system. When the data to transmit is large, this overhead can be negligible. However, for smaller messages, the overhead is much more pronounced. User-space communication models were introduced to overcome this bottleneck with small messages. The models basically describe mechanisms for applications to bypass the operating system and have direct access to the network. Some of the common user-space communication models include Generic Active Messages (GAM) [20], Basic Interface for Parallelism (BIP) [41], Fast Messages (FM) [40], PM [49], and Virtual Memory Mapped Communication (VMMC) [19]. All of these user-space communication designs have been found to provide low latencies for small messages, in the range of 5 to 17 \( \mu s \) [3].
Chapter 3

Design

3.1 Objectives

The XLIL provides a group membership and atomic commitment facility for the set of nodes that constitute the client application. The ultimate objective of the NML is to maintain consistency and ensure a homogeneous view of the system for all members in the presence of a potentially dynamic environment that may include node failures, partitions, and message losses. Many of the requirements have been outlined in Section 1.4. The NML works in conjunction with the application when making modifications. Group members are added and removed, and data updates are committed only in consultation with the client application. The description of the requirements for member addition and data update facilities are relatively straight-forward. However, member removals and the associated degraded state require a more precise discussion.

3.1.1 Behaviour Under Failures

The NML must facilitate the removal of a member in the event of its failure. Similar to any other type of modification, this is done in conjunction with the client application. The amount of time it would take the client application to perform the necessary operations
to formally remove a faulty member and for the NML service to finalize this removal is termed a recovery period ($\theta$). $\theta$ is not only application dependent, but can also be dependent on the number of members in the system. For the RSML, $\theta$ includes the time it takes for the members to reconstruct the data placement scheme with one less member.

However, if another failure occurs before the recovery period, then the RSML is essentially blocked from any more modifications. The current design of the RSML is only equipped to handle one failure per recovery period. The NML must be aware of this one-failure limitation. Thus, the NML must facilitate full recovery from one failure over a recovery period, and facilitate partial recovery if more than one failure occurs during this recovery period. Full recovery occurs when the non-faulty members of the client can perform the necessary operations to successfully remove the faulty members. Partial recovery occurs when the non-faulty members of the client can not perform these necessary operations because too many members have failed; however, the members are configured so that the recovery of the faulty member(s) can permit a full recovery.

The partial recovery strategy for the NML is simple. It waits for enough members before any modification is attempted. If, before the system can successfully remove a failed member, another member fails, then the number of failures would surpass the application's threshold of one failure. Specifically, the RSML/MultiText would no longer be able to function to its specifications: query solutions would be incomplete because of the loss of available data. The NML should not attempt to repeatedly remove the two failed members because the remaining members would never be able to commit their removal. Rather, it should wait until at least one of the failed members is revived and rejoins the system. In general, the NML will wait for enough failed members to recover before attempting any more modifications. Thus, to be able to fully recover, at least one of the two failed members will have to be revived so that the non-faulty members can attempt a removal. However, if both members recover, then no removal would be needed.

In addition, this mechanism can be used to handle possible system partitions. System
partitions are typically indistinguishable from a set of failures. For example, in an eight member system, a partition into two clusters of four would be indistinguishable from one cluster of members perceiving the other cluster members had failed. The members of one cluster will not attempt to remove the other cluster members because of the one failure threshold. However, if these members partitioned into a cluster of seven and one, then the seven member cluster would attempt to remove the singleton because this partition would be indistinguishable from the singleton failing.

3.1.2 Degraded Systems

In theory, if there was enough disk space on a single member and no new members were added, the client application would be able to handle \( N - 1 \) failures (assuming at least \( \theta \) time between failures), where \( N \) is the number of members. However, in reality, this may not be possible with the given client application. For example, in the RSML, the amount of disk space on one member is typically insufficient to store the amount of data of all members that fail. In fact, it is usually the case that once the number of members drops below a certain point, the data placement scheme can not be reconstructed anymore and thus, any further failures would result in an incomplete set of data. The client application on a node can remove the faulty member, but would not have enough space to absorb this member’s data. Although the available data set is still complete because of the duplication on the secondaries, the data placement scheme can not be reconstructed (see Figure 3.1). Hence, in the resulting system, no more failures are sustainable, no more data updates are permitted, and the only permissible modification would be a node addition that would effectively replace the previously failed member. The addition of a new member would allow the reconstruction of the data placement scheme and hence, the system can then again sustain one failure.

To summarize, at any time, the number of failures that is sustainable (\( f \)) can be either \( f = 1 \) or \( f = 0 \). When \( f = 1 \), the system is said to be stable: it can sustain another
Figure 3.1: Members 0, 1, and 2's disks are completely saturated at this point. If 2 now fails, then 0 and 1 would have to absorb 2's data ($S_{02}, S_{12}$). Also, to reconstruct the data placement scheme, this data will have to be replicated to the other member. However, there is not enough disk space to absorb 2's data and replicate the absorbed data. 0 and 1 require a new node or the recovery of 2 before the data placement scheme can be constructed.

f

failure and any modification is permitted. When $f = 0$, then the system is said to be degraded: it is only permitted to add new members to bring the system back to being stable. The NML will not attempt any removals or updates. If members still continue to fail in a degraded system, then no modification will be initiated, not even an addition. Any additions in a degraded system require all degraded members to participate.

3.1.3 Design Road Map

The rest of the chapter is organized as follows. A description of the environment in which the NML is to function is discussed first followed by an overview of its facilities. A detailed description of a protocol that handles one modification under a no-failure assumption is described as well as the details needed to achieve it. This protocol is then augmented to handle any number of modifications able to fully recover from one failure and partially recover from more than one failure during a recovery period. The mechanisms required to accomplish this are also described. The working client application example will be the RSML designed for MultiText.
3.2 Environment

The intended environment for this distributed service is a cluster architecture. There are multiple autonomous computing machines (nodes) each connected via a dedicated network.

The underlying network is assumed to closely follow the *Timed Asynchronous Model* [15]. The model is an abstraction of network properties typically exhibited by networks in practice. The network topology is arbitrary. No assumption is made about how the members are connected except that each node may send a message directly to any other node and may broadcast a message to every other node. The channels between any two members can be lengthy and lossy; that is, an individual message sent from one node to another may take an arbitrary amount of time and may not even reach its intended destination. This is known as an omission failure. However, in a cluster with an uncongested network, the probability of this occurring is very small. A timeout delay $\delta$ can be placed on the transmission time of messages. Messages are not guaranteed to arrive within time $\delta$, but rather $\delta$ can be set so that the probability of a message arriving within time $\delta$ is high. If a message arrives within time $\delta$, it is considered to be *timely*, otherwise, the message transmission has experienced a performance failure. Message duplication and out-of-order delivery can occur but message corruption is assumed to be handled by the network and can be ignored.

In addition, the system may partition. Since there is no real effective difference between a partition and multiple node failures, they are treated in the same way.

Each node has its own stable storage (disk) but no shared storage facilities between the nodes is assumed. In addition, each node is assumed to follow crash failure semantics rather than acting arbitrarily when faulty. A node fails by crashing but may also recover from crashes. There are two types of failures that can occur: soft crashes and hard crashes. *Soft crashes* occur when the node crashes but retains its stable storage and has the ability to recover its previous state and re-integrate itself back into the system.
**Hard crashes** occur when the node crashes and will not be able to re-integrate itself back into the system in its pre-crash state. The data on the stable storage may no longer be relevant or may be corrupted. When a crash failure occurs, it is not known for sure what type of failure had occurred. The NML must infer the type of failure in order not to remove members that have really soft crashed, but also remove members that have indeed hard crashed.

Furthermore, each node has access to its own hardware clock that has a bounded drift rate $\rho$. It is assumed that clocks of non-faulty nodes are monotonically increasing and drift within $\rho$. The assumption is used to keep clocks relatively synchronized over a small period of time. However, clock drift between two nodes over the entire lifetime of the clocks may grow without bound.

### 3.3 Design Overview

In general, the NML facilitates three different possible modifications: additions, removals, and data updates. An addition or removal occurs when a new node requests to be part of the system or when it has been concluded that an existing member has failed, respectively. This constitutes the group membership component. For the RSML, this affects the primaries and secondaries of every member and must appear atomic. External data updates occur when a logically external agent, the Update Marshaller/Dispatcher in the case of MultiText, issues a request to perform an update to the data collection. As with additions and removals, data updates affect the local primaries and the remote secondaries of every member and must also appear atomic.

Every system modification undergoes a two-stage process to maintain consistency: initiating the modification, and atomically committing (or aborting) the proposed modification.
### 3.3.1 The Initiation Stage

The first, or initiation, stage is required for the members to agree on a single modification for the system to perform at any point in time. In the given environment, there may be several "requests" to modify the system at the same time. For example, one member may have detected the possible failure of another member while at the same time a new node may be requesting addition to the system. Furthermore, a data update may be requested by the external agent. Nevertheless, no two members should attempt to modify the system differently. The initiation stage is required to ensure that the members of the system consistently make the same modifications.

The initiation problem is handled by implicitly designating a *leader* for the system. The leader is responsible for initiating modifications. It is important for each member to be able to locally determine the proper unique leader of the system. This is accomplished by assigning a unique index to each member when it is added. The list of members and their associated index is maintained by every member. The leader is the member with the smallest index. One complication that can occur in designating one member to initiate all modifications is when the leader itself has failed. If only the leader has the power to make modifications, its failure would lead to a stagnant system unless the leader recovers; however, it may never recover since it could have suffered a hard crash. Thus, it is important to allow another member limited modification capabilities. That is, if it has been detected that the leader has failed, then another member must be granted the authority to initiate its removal. Hence, a *crown prince* [32] is introduced that functions as the initiator when the leader fails. Similar to the leader, the crown prince is chosen such that it has the second smallest index in the system. In general, the leader is the member that initiates all modifications. Nevertheless, in the event it has failed, the crown prince has the limited power of removing the leader. It should be noted that the crown prince can not make any other modification. Specifically, it is not permitted to perform any addition, update, or removal of a member other than the leader. However, after the
crown prince of the original system successfully removes the leader, it may become the leader of the resulting system and may now initiate other modifications.

### 3.3.2 Client Application Decision

After the initiation stage, the NML passes control to the local client application so that it may perform the necessary operations to complete the modification. These operations are dependent on the application and the modification. For example, when the RSML wishes to augment the data collection, this would involve an individual node adding data to its local primary and distributing replicas to the remote secondaries. When the application on a node has completed its necessary operations, or if the application could not complete it as required, then it returns an appropriate response to the NML to finalize the modification. Control is effectively passed back to the NML. At this point, the second stage of the modification begins: the commitment stage.

### 3.3.3 The Commitment Stage

During the commitment stage, the system as a whole must collectively decide whether to commit or abort the modification. Once an individual member has received information from its local client application about its ability to commit the modification, the NML must now determine every other member’s ability to commit. In general, if one member had to abort the modification, then every member must abort the modification. Every member must have been able to complete the necessary operations for the modification in order for each individual member to commit the changes locally. To achieve this, a decentralized two-phase commit [48] is used. This involves each member broadcasting its readiness to commit as well as collecting other members’ readiness message. If a member receives one abort, then it can locally abort the modification. However, if it receives a message from every other member indicating they are all ready to commit the modification, then it locally commits the modification.
Every member moves through these two stages together which introduces some synchrony and reduces uncertainty. Thus, no member ever gets many steps ahead of another member in the modification process. Together, the initiation and commit stages ensure a single, global view using incremental modifications.

3.4 The Global State Machine

The group membership and atomic update facility has been designed as a state machine. This service is encapsulated by the Global State Machine (GSM) which is implemented as a process that resides on each member, or potential member, of the system. Client applications on the local member can subscribe to this service by establishing a connection with it. This GSM provides periodic information to the applications. It is the responsibility of the client application to monitor this information and react to it.

At any given time, each GSM resides in a state indicating the action it is performing and makes transitions from one state to another depending on applicable triggers. A trigger occurs when a message is received from another member of the system, when a message is received from the client application, or when a timeout occurs. Each GSM maintains a view of all the other members in the system by periodically broadcasting its state information (every $\beta$ time) to every other GSM while, at the same time, collecting these messages to update its state information. These periodic broadcast messages act as a heartbeat to indicate the health of the member. Absences of heartbeats indicate a possibility of failure. Although $\beta$ is application dependent, it should be relatively small to keep information about the members as current as possible. In the current implementation for the RSML application, this is set to 0.25s.
3.4.1 States

The states of the GSM indicate what situation the member is currently in. In general, the possible states are:

*Join*: the node is indicating it wishes to become a member of the system.

*Alive*: this comes in two forms:

  * Stable: the member is not undertaking any modification and it is able to sustain one failure, $f = 1$.
  
  * Degraded: the member is not undertaking any modification but it is not able to sustain a failure, $f = 0$. Only additions are possible.

*Remove*($r$): the member is removing $r$ from the system.

*Add*($a$): the member is adding $a$ to the system.

*Update*: the member is performing a data update to the system.

These states and their relations are shown in Figure 3.2.

![Figure 3.2: The states and their relationships. From Stable, we can attempt any modification (Add($a$), Remove($r$), and Update). However, from Degraded, we can only attempt an addition.](image)

Furthermore, the *Remove*($r$), *Add*($a$), and *Update* states have a *substate* indicating how far along in the modification the member has reached. As mentioned above, there are two stages to performing a modification: deciding what modification to perform and atomically committing to it. The substate reflects this dual stage:

*Propose*: the member is proposing a modification to the system, waiting for all members to propose it as well.
Prepare: the member is preparing the previously proposed modification. The GSM at this point passes control to the client application to undertake the necessary operations required to complete the proposed modification. At this point, it is up to the client application to indicate to its local GSM any further decisions about the status of the modification (either Scommit, Dcommit, or Abort).

Commit: this value comes in two forms:

Scommit: the member is able to commit the modification and return to a stable system. That is, the commitment of the modification will allow it, as part of the system, to again sustain one failure (i.e., \( f = 1 \)).

Dcommit: the member is able to commit the modification; however, the resulting system would be degraded. As part of the system, it would not be able to tolerate another failure without first adding a new member (i.e., \( f = 0 \)).

The role of Dcommit is only needed for the Remove\((r)\) modification: the removal of \( r \) would not allow the remaining members to sustain another failure, although they can remove \( r \). The Add\((a)\) and Update modifications do not require this extra state.

Again, it is up to the client application on the node to determine which Commit (if at all) it can perform.

Abort: the member is not able to commit the modification.

Together, the Scommit, Dcommit, and Abort substates make up the voting states of a modification.

A state is represented by a pair \(<state, substate>\) if there exists a substate, or \(<state, \ast>\) if there is no related substate, such as one of the Alive states. The state of a member is part of the information it periodically broadcasts in its heartbeat. Figure 3.3 illustrates the substates within a modification state. The state transitions are described in Section 3.5.
Figure 3.3: The substates of Remove(r), Add(a) and Update.
3.4.2 Commit/Abort Indicators of Alive states

When a member is in one of the *Alive* states, it is sometimes necessary to have information about the success of the previous modification. Thus, for notational purposes and where necessary, an indicator will be attached to an *Alive* state to indicate the success or failure of the last modification. For example, if a member committed the last modification and moved to \(\langle Stable, *\rangle\), this would be indicated by \(\langle Stable, *\rangle_{Commit}\). If it had aborted the modification, returning to \(\langle Stable, *\rangle\), this would be indicated by \(\langle Stable, *\rangle_{Abort}\). Similarly if it committed a removal, but this resulted in a degraded system, this would be indicated by \(\langle Degraded, *\rangle_{Commit}\).

The indicators are always maintained by the GSM. However, the indicators will not always be shown in the discussion except when necessary.

3.4.3 State Vectors

As mentioned, each GSM member \(i\) periodically broadcasts its state information to every other member. A component of this state information is \(i\)'s current state. Another component of this state information is \(i\)'s view of every other member in the system. Thus every GSM member \(i\) is not only required to keep information about its current state, it is also required to keep information about the most current state of every other member in the system. Thus, a state vector for \(i\), \(SV_i\), is used to keep this information.

The state vector records the last state of every member and is updated when \(i\) changes its state or when \(i\) receives a heartbeat from another member \(j\). When \(i\) receives a heartbeat from \(j\), it can extract \(j\)'s state and update \(j\)'s entry in the state vector, \(SV_i(j)\). \(SV_i(j)\) represents \(i\)'s most recent view of the state \(j\) was in. \(SV_i(i)\) represents \(i\)'s most recent view of the state \(i\) was in or simply \(i\)'s most current state. For example, if \(i\) receives a heartbeat from \(j\) indicating that \(j\) was in state \(\langle Stable, *\rangle\), then \(i\) would set \(SV_i(j) := \langle Stable, *\rangle\). Similarly, if \(i\) changed its state to \(\langle Remove(r), Propose\rangle\) it would
The state vector gives \( i \) a snapshot of the states or actions of all members in the system. This information is used predominantly for \( i \) to make state transitions.

### 3.5 State Transitions

The GSM of each member can make transitions depending on messages received. Members typically reside in the \( \langle \text{Stable}, \ast \rangle \) or \( \langle \text{Degraded}, \ast \rangle \) state, and wait for the initiator (either the leader or the crown prince) to initiate a modification \( m \) taking it into \( \langle m, \text{Propose} \rangle \). for \( m \in \{ \text{Add}(a), \text{Remove}(r), \text{Update} \} \). Recall from Figures 3.2 and 3.3, if the modification originated from \( \langle \text{Degraded}, \ast \rangle \), then \( m = \text{Add}(a) \). Most of the time, the client application takes a passive role. The GSM periodically provides it with state information. However, the client application must monitor this state information and react to any modifications initiated by the GSM.

Initiations are trivial in a system with no failures, but can pose serious problems when failures can occur. The initiation problem is deferred until Section 3.8 where it is given a rigorous treatment. The description of the state transitions that follow is for a single modification \( (m) \) with no member failures. This protocol will later be augmented to handle a more general case with member failures.

#### 3.5.1 Initiation Transitions

Once a modification has begun, the initiator (either the leader or crown prince) attempts to obtain agreement from every member to propose the same modification by placing its state \( \langle m, \text{Propose} \rangle \) in its heartbeat. This action starts the modification process.

If member \( i \) is in the state:

- \( \langle \text{Stable}, \ast \rangle \) or \( \langle \text{Degraded}, \ast \rangle \)

\( i \) can transition to:
1. \(\langle m, \text{Propose} \rangle\), if \(i\) receives a heartbeat message from \(j\) where the state of \(j\) is \(\langle m, \text{Propose} \rangle\).

If \(i\) is in \(\langle \text{Degraded}, \ast \rangle\), then \(m = \text{Add}(a)\) is the only possible modification.

It is not necessary to change to this state based only on a message directly from the initiator. If we can guarantee that initiations are generated by only one member, then any time \(i\) receives a \(\langle m, \text{Propose} \rangle\) message from any other member \(j\), it would transition to this state. This gossiping technique is used to speed communication as \(i\) does not need to hear directly from the initiator. Also, it is able to handle any non-partitioning communication problem.

- \(\langle m, \text{Propose} \rangle\)

\(i\) can transition to:

1. \(\langle m, \text{Prepare} \rangle\), if:

   (a) \(\forall k. SV_i(k) = \langle m, \text{Propose} \rangle\) or

   (b) \(\exists j. SV_i(j) = \langle m, \text{Prepare} \rangle \lor SV_i(j) = \langle m, \text{Abort} \rangle \lor SV_i(j) = \langle m, \text{Dcommit} \rangle \lor SV_i(j) = \langle m, \text{Scommit} \rangle\)

If \(SV_i(j) = \langle m, \text{Dcommit} \rangle\), then \(m = \text{Remove}(r)\).

In the \(\langle m, \text{Propose} \rangle\) state, each member waits until every other member reaches this state determined by its state vector \(SV_i\). Once this has occurred, \(i\) moves ahead to \(\langle m, \text{Prepare} \rangle\). However, we can also use an inference technique to move to \(\langle m, \text{Prepare} \rangle\). Instead of waiting to receive a message from all members indicating they are in state \(\langle m, \text{Propose} \rangle\), \(i\) can see if any other member \(j\) is ahead of it (see Figure 3.4). \(j\) is ahead of \(i\) if \(j\) is further along in the modification than \(i\). When \(i\) is in \(\langle m, \text{Propose} \rangle\), \(j\) could have moved ahead of \(i\) if either \(j\) already received a message from every other member indicating they were in state \(\langle m, \text{Propose} \rangle\), or if \(j\) had heard from another member which was ahead of \(j\). In general, if any member is ahead in \(\langle m, \text{Prepare} \rangle\), there must exist an \(l\) such that every entry in \(l\)'s state vector was \(\langle m, \text{Propose} \rangle\).

- \(\langle m, \text{Prepare} \rangle\)

\(i\) can transition to:

1. \(\langle m, \text{Abort} \rangle\), if indicated by the client application.
2. \( \langle m, Dcommit \rangle \), if indicated by the client application.

3. \( \langle m, Scommit \rangle \), if indicated by the client application.

In the \( \langle m, \text{Prepare} \rangle \) state, the GSM passes control to the client application and waits for a decision about how to proceed with the modification. No received messages will affect its state transition at this point. This is where the application takes an active role in the modification. Only the client can trigger the next GSM transition. At this point, the client application would perform the necessary operations to make the modification. When the client application is ready, it indicates to the GSM whether to proceed to \( \langle m, Scommit \rangle \), \( \langle m, Dcommit \rangle \), or \( \langle m, Abort \rangle \).

\[\text{Figure 3.4: Substate hierarchy within a modification } m \text{ representing being ahead or behind.}\]

The initiation stage is, in itself, a form of a two-phase procedure, much like the two-phase commit, to initiate a modification. The initiation stage is the same for all modifications regardless if the modification originated out of \( \langle \text{Stable}, \ast \rangle \) or \( \langle \text{Degraded}, \ast \rangle \). However, the commitment stage depends on the modification and the Alive state the GSM originated from. The next section discusses the commitment transitions for \( m = \text{Remove}(r) \), originating from \( \langle \text{Stable}, \ast \rangle \) since it is the most complicated one. However, for other modifications originating from an Alive state, the transitions are very similar and follow the same reasoning.
3.5.2 Commitment Transitions

The commitment stage for a member begins after it has entered a voting state, either the \( \langle m. Scommit \rangle \), \( \langle m. Dcommit \rangle \), or \( \langle m. Abort \rangle \) state (where \( m = \text{Remove}(r) \)), indicated by its local client application. The transitions out of a voting state take the member back to an Alive state.

If member \( i \) is in the voting state:

- \( \langle m. Abort \rangle \)
  
  \( i \) can transition to:

  1. \( \langle \text{Stable}, \ast \rangle \)\text{_{Abort}}, if:
     
     (a) \( \forall k, SV_i(k) = \langle m. Abort \rangle \) or
     
     (b) \( \exists j. SV_i(j) = \langle \text{Stable}, \ast \rangle \)\text{_{Abort}}

  If the state \( i \) resides in is \( \langle m. Abort \rangle \), then it waits for every other member to reach this state according to its state vector, \( SV_i \). Then it transitions to \( \langle \text{Stable}, \ast \rangle \)\text{_{Abort}}. Again, we can make use of information provided by other members to speed the process. If \( i \) receives a message from even just one member in state \( \langle \text{Stable}, \ast \rangle \)\text{_{Abort}} then it will immediately follow it to this same state.

- \( \langle m. Dcommit \rangle \)
  
  \( i \) can transition to:

  1. \( \langle \text{Degraded}, \ast \rangle \)\text{_{Commit}}, if:
     
     (a) \( \forall k, SV_i(k) = \langle m. Dcommit \rangle \) or
     
     (b) \( \exists j. SV_i(j) = \langle \text{Degraded}, \ast \rangle \)\text{_{Dcommit}}

  Similar to being in state \( \langle m. Abort \rangle \), \( i \) would wait for all members to achieve \( \langle m. Dcommit \rangle \), according to its state vector. If this occurs, then it moves into \( \langle \text{Degraded}, \ast \rangle \)\text{_{Commit}}, but setting its \( f := 0 \). It can no longer sustain another failure without a further addition. Recall, state \( \langle m. Dcommit \rangle \) only occurs when \( m = \text{Remove}(r) \).
Also, the GSM will follow any member that is ahead of it. For example, if a member receives a heartbeat from \( j \) that is in \( \langle \text{Degraded}, \ast \rangle_{\text{Commit}} \), then it will follow this lead and transition to that state. However, it will be shown that if \( i \) is in state \( \langle m, \text{Dcommit} \rangle \), then it would never receive a \( \langle \text{Stable}, \ast \rangle_{\text{Abort}} \) nor a \( \langle \text{Stable}, \ast \rangle_{\text{Commit}} \).

2. \( \langle m, \text{Abort} \rangle \), if \( \exists j, SV_i(j) = \langle m, \text{Abort} \rangle \).

If \( i \) receives a heartbeat from any member in \( \langle m, \text{Abort} \rangle \), then it will move to \( \langle m, \text{Abort} \rangle \).

- \( \langle m, \text{Scommit} \rangle \)

\( i \) can transition to:

1. \( \langle \text{Stable}, \ast \rangle_{\text{Commit}} \), if:
   
   (a) \( \forall k, SV_i(k) = \langle m, \text{Scommit} \rangle \) or
   
   (b) \( \exists j, SV_i(j) = \langle \text{Stable}, \ast \rangle_{\text{Commit}} \)

\( i \) faces much of the same transitions as in \( \langle m, \text{Dcommit} \rangle \). It will wait for all members to achieve \( \langle m, \text{Scommit} \rangle \), again, according to its state vector, \( SV_i \). Then it will transition to \( \langle \text{Stable}, \ast \rangle_{\text{Commit}} \) and \( f := 1 \).

Again, if the GSM receives a message from another member ahead of it in state \( \langle \text{Stable}, \ast \rangle_{\text{Commit}} \), it will follow this member transitioning to this state. However, again, it will be shown that if \( i \) is in \( \langle m, \text{Scommit} \rangle \), then it would never receive a \( \langle \text{Stable}, \ast \rangle_{\text{Abort}} \) nor a \( \langle \text{Degraded}, \ast \rangle_{\text{Commit}} \).

2. \( \langle m, \text{Abort} \rangle \), if \( \exists j, SV_i(j) = \langle m, \text{Abort} \rangle \).

If \( i \) receives a heartbeat from any member in \( \langle m, \text{Abort} \rangle \), then it will move to \( \langle m, \text{Abort} \rangle \).

3. \( \langle m, \text{Dcommit} \rangle \), if \( \exists j, SV_i(j) = \langle m, \text{Dcommit} \rangle \).

If \( i \) receives a heartbeat from any member in \( \langle m, \text{Dcommit} \rangle \), then it will move to \( \langle m, \text{Dcommit} \rangle \).
3.5.3 Consistency of Commitment Transitions

Given such an protocol, it might not be evident that consistency is maintained. It would be inconsistent, for example, if one member reached \((Degraded, \ast)_{\text{Commit}}\) while another member reached \((Stable, \ast)_{\text{Commit}}\) for the same modification. When a member follows another member from a voting state to one of the \textit{Alive} states (\((Stable, \ast)\) or \((Degraded, \ast)\)), then it must be the case that they had the same corresponding voting state. More specifically, if \(i\) is in \((m, \text{Abort})\) and receives an \textit{Alive} message from \(j\), it must be the case that this message was a \((Stable, \ast)_{\text{Abort}}\). If \(i\) is in \((m, \text{Dcommit})\) and receives an \textit{Alive} message from \(j\), it must be the case that this message was a \((Degraded, \ast)_{\text{Commit}}\). Similarly if \(i\) is in \((m, \text{Scommit})\) and receives an \textit{Alive} message from \(j\), it must be the case that this message was a \((Stable, \ast)_{\text{Commit}}\).

The intuition behind this is to consider why the very first \(j\) transitioned to an \textit{Alive} state. By the protocol, it must have been the case that \(j\) received a message from each member indicating it was in the same voting state as \(j\). Consider the three possible voting states:

1. If \(j\) was in \((m, \text{Scommit})\) before it transitioned to \((Stable, \ast)_{\text{Commit}}\).

   This would mean that every other member was in \((m, \text{Scommit})\) at some time and \(j\) received a message from them while they were in this state. The only way one of these members, \(i\), could move out of this state and into \((m, \text{Dcommit})\) or \((m, \text{Abort})\) is if \(i\) receives a message from another member \(x\) indicating \(x\) is in one of these two states. If \(x\) was in one of these states at some point in time, then there is no way it can possibly have transitioned to \((m, \text{Scommit})\). This is because there is no transition in the protocol that allows \(x\) to transition from \((m, \text{Dcommit})\) or \((m, \text{Abort})\) into \((m, \text{Scommit})\). Thus, there does not exist such a member \(x\). Thus, \(i\) could not have moved out of state \((m, \text{Scommit})\) and into \((m, \text{Dcommit})\) or \((m, \text{Abort})\) in this situation.

2. If \(j\) was in \((m, \text{Dcommit})\) before it transitioned to \((Degraded, \ast)_{\text{Commit}}\).

   This would again mean that every other member was in \((m, \text{Dcommit})\) at some time and \(j\) received a message from them while they were in this state. If one
member $i$ moves out of $(m, Dcommit)$ and into $(m, Abort)$, then that would mean there existed an $x$ at some point in time that was in $(m, Abort)$. If this is the case, then $x$ could not have transitioned from $(m, Abort)$ to $(m, Dcommit)$ because the protocol forbids such transitions. Thus, $j$ could not have received a message from each member indicating it was in $(m, Dcommit)$. Hence, a contradiction. It should be noted that $i$ could not possibly have transitioned out of $(m, Dcommit)$ and into $(m, Scommit)$ because the protocol does not allow such a transition.

3. If $j$ was in $(m, Abort)$ before it transitioned to $(Stable, \ast)_{Abort}$.

This would mean that every other member was, at one time, in $(m, Abort)$ and $j$ received a message indicating this. If this is the case, it could not possibly be that any member could make a transition out of this voting state and into another voting state because the protocol does not have such provisions.

The key to the protocol is to present a hierarchy $(Scommit \rightarrow Dcommit \rightarrow Abort)$ in which the members can traverse in only one direction. This argument works with three voting states for removals, but would also work for the two voting states $(m, Scommit), (m, Abort))$ of additions and updates.

This illustrates how transitions occur for a removal of a member originating out of $(Stable, \ast)$. Figure 3.5 shows the commitment stage for the other modifications: an addition originating from $(Stable, \ast)$, an update originating from $(Stable, \ast)$, and an addition originating from $(Degraded, \ast)$.

3.5.4 From Failure-free to Failure-prone Services

As mentioned, the protocol, in the form presented above, will work for a modification when there are no member failures. Furthermore, message losses and delays are handled. Each member periodically broadcasts its current state to handle the potential loss of messages. Allowing members to follow other members that are ahead of it can also compensate for communication failures. Similar to following a member that is ahead, the protocol can be slightly altered to handle message delays by allowing members to ignore
Figure 3.5: Commitment transitions for all modifications. The transitions discussed were for \( m = \text{Remove}(r) \). Similar transitions are used for \( m = \text{Update} \) and \( m = \text{Add}(a) \) originating from \((\text{Stable,} *)\), and \( m = \text{Add}(a) \) originating from \((\text{Degraded,} *)\). Note the slight difference between the two \( \text{Add}(a) \) transitions. If the members were originally in \((\text{Stable,} *)\) and an addition was attempted, then regardless if the modification was successful or not, this would return them to \((\text{Stable,} *)\) with different indicators. If the members originally were in \((\text{Degraded,} *)\), and an addition was successful, this would take them into \((\text{Stable,} *)\). However, if the addition was unsuccessful, this would take them back into \((\text{Degraded,} *)\). Only a successful addition in this situation will return the members to \((\text{Stable,} *)\).

\[
\begin{align*}
\text{Alive} & \quad \text{Alive} \\
\text{(Stable,} *)_{\text{Commit}} & \quad \text{(Stable,} *)_{\text{Commit}} \\
\text{(Degraded,} *)_{\text{Commit}} & \quad \text{(Degraded,} *)_{\text{Commit}} \\
\text{(Stable,} *)_{\text{Abort}} & \quad \text{(Stable,} *)_{\text{Abort}}
\end{align*}
\]

messages that are behind.

However, the protocol needs to handle more than a single modification as well as handle failures. It is important that messages that are substantially delayed or are no longer relevant, possibly due to their out-of-order delivery, do not affect the current modification in an inconsistent way. For example, if messages from modifications of the past get lost in the network but find their way into the current modification \( m \), then it should not affect the members making modification \( m \). Furthermore, it is desirable to not only fully recover from single failures, but also partially recover from multiple member failures during a recovery period \((\theta)\). The complications that failures introduce, including properly detecting failures, can be troublesome. Assumptions will be needed to guard against inconsistency.
The next sections describe how these problems are handled, culminating in a fault-tolerant group membership and atomic update service. Section 3.6 discusses how failures are detected. In Section 3.7, a message filtering technique is used to determine the relevance of an arbitrary message. Section 3.8 discusses how modifications are initiated. Finally, a discussion of the transitions in the presence of failures is discussed in Section 3.9.

3.6 Accurately Detecting Failures

The discussion of a service that handles failures first requires a mechanism of properly detecting failures: failure detectors [9]. The most common approach relies on the use of timeouts to approximate a failure. If member $i$ has not received a heartbeat from member $j$ after a given period of time, it might infer that $j$ is no longer available. However, this is only an approximation as it is impossible to detect if the lack of a heartbeat was the result of a member crashing, or if the network was simply too congested between $i$ and $j$ to allow a message to cross through, or if the network itself had failed. Generally, a failure detector is complete if a faulty process is eventually permanently suspected to be faulty by every other non-faulty member. A failure detector is accurate if no member is suspected of being faulty unless it really has crashed [9]. The use of timeouts typically has the property of being complete, but lacks accuracy.

There is no state in Section 3.4.1 that describes the lack of heartbeat from another member. Collectively, those states in Section 3.4.1 are referred to as Awake states. Thus, two new states can be introduced to detect failures. One state is used as a warning for the client application whereas the other state is used by the GSM indicating a more serious case. Recall, the client application must monitor the state information the GSM provides it. The state of $j$, according to $i$, $SV_i(j)$, can change to:

- $(Comatose,*)$, if, after a period of time ($\sigma$), $i$ has not heard from $j$. 
This indicates that \( i \) suspects that \( j \) may have soft crashed. The problem is handled at the local level. The GSM will indicate this as part of the state information it periodically sends to the client application. The client application would take action locally. For example, if a member of the RSML receives this notification, it activates the otherwise dormant secondary of the potentially faulty member. However, no action is taken on a global level. \( \sigma \) is usually set to a relatively small value depending on the application. In the current proposal of the RSML, \( \sigma = 4\beta \) (1s) because if the member really has failed, then queries would be incomplete until the non-faulty members activate the secondary of \( j \). This corresponds to four missed heartbeats. The timeout would result in \( SV_i(j) := (Comatose, \ast) \).

- \( (Critical, \ast) \), if, after a period of time \( (\lambda) \), \( i \) has not heard from \( j \) (where \( \lambda \gg \sigma \)). This indicates that \( i \) suspects that \( j \) may have hard crashed. This incident would provoke a more serious action on the part of the entire system. This typically involves an attempted removal of \( j \) by the non-faulty members. Again, the value of \( \lambda \) would be application dependent. For the RSML application, this value is set to \( \lambda = 1200\beta \) or 300s. \( \lambda \) should be set to a value that allows a member that has soft crashed to recover. Removals can be potentially expensive and thus performance could suffer if too many false attempted removals occur. The timeout would result in \( SV_i(j) := (Critical, \ast) \).

Individually, these two states represent a trade-off between accuracy and responsiveness. If \( i \) has not received a heartbeat from \( j \) after \( \sigma \) time, it may not be the case that \( j \) has failed. However, \( i \) must be responsive because the client application may be sensitive to failures. If \( i \) has not received a heartbeat from \( j \) after \( \lambda \) time, it is much more likely that \( j \) has indeed failed than after \( \sigma \) time. However, reliability could suffer if the client application were to respond after \( \lambda \) time. The use of two timeouts attempts to achieve quick responsiveness along with a high accuracy of detecting true failures.

A member itself will never transition into these states, but rather these states are perceptions one member has about another. It is evident that the potentially failed member would have to be perceived as \( (Comatose, \ast) \) before \( (Critical, \ast) \). As with the \( Stable \), \( Degraded \), and \( Join \) states, the \( Comatose \) and \( Critical \) states have no associated
substates.

3.6.1 Alive Vectors

The state vectors in Section 3.4.3 give the GSM member $i$ the ability to record a snapshot of what state every member is in. Sometimes, it will be useful for $i$ to know the state vector of another member $j$. This can at least be approximated since every member broadcasts its state vector in its heartbeat. $i$ could simply store the last state vector received from $j$'s heartbeat.

Thus, we can extend the concept of a state vector for $i$ to an alive vector for $i$. The alive vector, $AV_j(k)$, is the last known state of $k$ by $j$ on $i$. However, it will be sufficient to record each entry as Critical, Comatose, or Awake. The conversion of a state vector received from $j$ ($SV_j$) by $i$ to the corresponding entry in the alive vector of $i$ for $j$ ($AV_j$) is trivial. Now, when $i$ receives a heartbeat from $j$, it would convert $SV_j$ to the form of an alive vector entry and store it in $AV_j$. It should be noted that $AV_i$ is the converted vector entry of the local state vector $SV_i$. Thus, if $i$ makes changes to $SV_i$, it should make the corresponding changes to $AV_i$.

3.7 Message Filtering

In a system where messages can be arbitrarily delayed, there must be a mechanism to distinguish when a heartbeat received by a member is relevant. For example, if a member $j$ fails during a modification $m$, while the other members are making modification $m'$, then it is not desirable for messages of $m$ to affect the progress of $m'$ (if $m \neq m'$). Furthermore, upon the recovery of $j$, it must be able to determine if it is still relevant to the system or relevant to the current modification. Being relevant to the system implies that $j$ has seen the same set of committed modifications as the rest of the members. Being relevant to the current modification implies that $j$'s modification $m$ is the same
as \( m' \). In this example, if \( m' \) was a failed removal of \( j \), and \( j \) recovered, then \( j \) would still be relevant to the system. However, \( j \) would find that its modification \( m \) is not relevant to the current modification and should abandon \( m \). On the other hand, if \( m' \) was a successful remove of \( j \), then \( j \) would no longer be relevant to the system.

A message filtering approach to received messages can be used to augment the algorithm given in Section 3.5. In addition to maintaining the states of all the members in the system using the state vector, every member must also keep track of a modification number for every modification, both successful and unsuccessful, that the system has undertaken. Furthermore, each member must keep track of a last commit number which is the modification it last committed. In addition, a message sequence number is needed to handle out-of-order message delivery. Keeping information about these changes will allow any member, including failed members that recover, to correctly determine if it is still relevant to the system and relevant to the current modification and prevent confusion during modifications.

It should be noted that if \( j \) is relevant to the current modification, then it would also be relevant to the system. The converse is not necessarily true as the above example illustrates. The following sections discuss how to design a simple mechanism to achieve this desired effect and thus, define more precisely what it means to be relevant to the modification and relevant to the system.

### 3.7.1 Modification Numbers

Modification numbers are used to keep track of all modifications that the members have attempted. Every time an initiator initiates a modification, regardless if it succeeds (committed) or not (aborted), a unique modification number, greater than any other previously used modification number, is associated with it. Thus, every member keeps the modification number of the most recent modification it attempted.

Modification numbers are part of each heartbeat message. Any member \( i \) can possibly
receive a message with a modification number \((MN_j)\) from \(j\) that varies from its own \((MN_i)\). This can be used to determine its relevance to the current modification:

- \(MN_i > MN_j\)
  
  \(i\) is currently undergoing a modification that supersedes \(j\).

- \(MN_i = MN_j\)
  
  \(i\) and \(j\) are performing the same modification. \(i\) would see that it is still relevant to this modification according to \(j\).

- \(MN_i < MN_j\)
  
  \(j\) is performing a modification that supersedes \(i\). If \(i\) is still relevant to the system, it will try and join \(j\) in performing the modification.

### 3.7.2 Last Commit Numbers

Each member also maintains a last commit number which is the modification number the member last committed. Thus, if member \(i\) committed modification \(MN_i\), then it would set its last commit number \(LCN_i := MN_i\). The last commit number acts as an identifier for the global state of the system. If two members share the same last commit number, then they are part of the same system configuration and are relevant to the system according to one another.

Like the modification numbers, any member \(i\) can possibly receive a last commit number \((LCN_j)\) from \(j\) that varies from its own \((LCN_i)\). We can now integrate the modification and last commit numbers to specify the actions a member takes upon reception of a message.

If \(i\) receives a message from \(j\) such that:

1. \(LCN_i = LCN_j\)
   
   \(i\) is still relevant to the system according to \(j\)
This situation is effectively saying that $i$ has the same last commit number as $j$ and thus has the same global state.

$i$ can now check its modification number $MN_i$ against $j$'s ($MN_j$).

(a) $MN_i < MN_j$

The first thing $i$ must do is resolve the uncertainty of its modification $MN_i$. It is still relevant to the system according to $j$ but now must close $MN_i$ before proceeding. If $i$ was still in the middle of its modification of $MN_i$ (i.e., not in an *Alive* state), then no matter what state $i$ was in — including if it was in $(m, Scommit)$ or $(m, Dcommit)$ — it would move to an *Alive* state, aborting the modification. The state depends on $m$ and which *Alive* state the modification originated from; see Figure 3.5. Its $LCN_i$ would be unchanged. This is because $j$ had already proceeded and obviously did not commit the change. If it did, $LCN_j > LCN_i$. (Note, the modification $m = MN_i$ is the same number for all participants of modification $m$. Thus, if $j$ committed modification $m$, it would set its $LCN_j := m$. Since $i$ is still in modification $m$, $m = MN_i > LCN_i$. Thus $LCN_j > LCN_i$.) This is like $j$ being ahead of $i$ and $i$ following $j$. The result effectively returns $i$ back to an *Alive* state, thus resolving $MN_i$. If $i$ was already in this state, the above transition would not be necessary.

The second thing $i$ will try to do is join $j$, if possible. If $j$ is in $(m, Propose)$, where $m$ is the modification proposed by $j$, then

i. $MN_i := MN_j$

ii. $SV_i(i) := (m, Propose)$

Otherwise, $i$ would wait in the *Alive* state. $i$ can not catch up with $j$ because $j$ has gone too far in the modification. Thus $i$ should wait until $j$ finishes modification $MN_j$ and hope that it is still relevant to the system afterwards.

(b) $MN_i > MN_j$

$i$ is performing a modification that is later than $j$'s. No state change is necessary.

(c) $MN_i = MN_j$

$i$ and $j$ are performing the same modification. The message is accepted and $i$ would apply the state transitions described in Section 3.5.
2. $LCN_i < LCN_j$, then check $MN_i$.

This situation is where $j$ seems to have a more current global configuration than $i$, but this can be misleading. It could be the case that $i$ was performing the same modification as $j$ but $j$ had already committed the modification. For example, if $i$ and $j$ were committing a modification but $j$ committed first, then $i$ would find $LCN_i < LCN_j$. A closer examination of $MN_i$ would give insight into the relevance of $i$ according to $j$.

(a) $MN_i = LCN_j$

$j$'s last commit number ($LCN_j$) is $i$'s current modification ($MN_i$). Thus, $i$ should attempt to synchronize its configuration with $j$'s, if possible (as discussed in Section 3.5). If $i$ is successful, it will now be relevant to $j$'s system. (Note, if $i$ was able to commit modification $MN_i$, it would have set its $LCN_i := MN_i(= LCN_j)$.)

For $m = \text{Remove}(r)$, there are three sets of states that $i$ can be in:

i. $SV_i(i) = (m, Scommit)$

$i$ should commit the modification. We can check how $j$ committed (either $Scommit$ or $Dcommit$) by looking at its state. We could also have examined its $f_j$. If $j$ is in $\langle \text{Stable, *} \rangle$, ($f_j = 1$), it was an $Scommit$. If $j$ is in $\langle \text{Degraded, *} \rangle$, ($f_j = 0$), it was a $Dcommit$. Set $f_i := f_j$. move to the corresponding state, and commit the modification ($LCN_i := MN_i$).

ii. $SV_i(i) = (m, Dcommit)$

If $j$ is in $\langle \text{Degraded, *} \rangle$. ($f_j = 0$), then $i$ should commit the modification. If $j$ is in $\langle \text{Stable, *} \rangle$, ($f_j = 1$), then $j$ committed leading to a stable system. Since $i$ can not commit to this (it is committing but to a degraded system), it should re-initialize, become a new node, and try to rejoin the system in this manner.

iii. Otherwise $SV_i(i)$ is in an unrecoverable state; $i$ should re-initialize, become a new node node, and rejoin the system.

For $m = \text{Add}(a)$ or $m = \text{Update}$, there would only be two sets of states $i$ can be in:

i. $SV_i(i) = (m, Scommit)$
Chapter 3. Design

$i$ should commit the modification. It would never receive a \((Degraded, *)_{commit}\)
message the way remove modifications do. The situation of \(SV_i(i) = (m, Dcommit)\) would not be applicable. Recall, \((m, Dcommit)\) is only for removals.

ii. Otherwise \(SV_i(i)\) is in an unrecoverable state; \(i\) should re-initialize, become a new node node, and rejoin the system.

(b) \(MN_i < LCN_j\)

In this case, regardless if \(i\) committed or aborted its modification \((MN_i)\), its last commit number \((LCN_i)\) would be stale with regards to \(j\). (Note, if \(i\) was able to commit modification \(MN_i\), it would set its \(LCN_i := MN_i\). However, this would still make \(LCN_i < LCN_j\).)

\(i\) can not possibly catch up to \(j\), so \(i\) should re-initialize, become a fresh new node, and rejoin the system.

(c) \(MN_i > LCN_j\)

This would never happen. It implies that \(i\) is attempting to make a modification that would make \(j\) irrelevant to it when before, it was irrelevant to \(j\).

3. \(LCN_i > LCN_j\)

This is the situation in which \(i\)'s global configuration is more current than \(j\)'s. \(i\) should just ignore \(j\) until \(j\) brings its global configuration equal to \(i\)'s, if it ever does.

If \(LCN_i = MN_j\), \(j\) can potentially re-integrate with the rest of the system. Thus \(i\) should update that it has heard from \(j\). If \(LCN_i \neq MN_j\), then \(j\) can not re-integrate and we should ignore this message from \(j\).

It should be noted that not all these situations would be encountered. The list is exhaustive for completeness.

To summarize, if a message is received from \(j\) that does not have the same configuration (the same last commit number) as the local member \(i\), then it is discarded. Furthermore, messages that do not pertain to the same modification (the same modification number) are not completely ignored. \(i\) would still update that it has received a
heartbeat from \( j \).

The modification number and last commit number can be used to implement the indicator \( q \in \{\text{Commit, Abort}\} \) for \( \langle \text{Stable, } \ast \rangle_q \) and \( \langle \text{Degraded, } \ast \rangle_q \). A member \( i \) can take a look at its modification number and compare it with the last commit number of the received message from \( j \). This is done in point 2(a) above.

### 3.7.3 Message Sequence Numbers

Finally, to ensure messages received from a member provide current information, it is necessary to determine if a message received is the latest one sent. The modification and last commit numbers that are attached to a message provide information about how stale the receiver’s individual modification is and whether the receiver is itself stale in the system, respectively. However, out-of-order message delivery can cause problems related to timing even if we know a member is relevant to the system and to the modification.

For example, suppose \( j \) broadcasts two messages: \( m_x \) at time \( x_j \) and \( m_{x'} \) at time \( x'_j \) where \( x_j < x'_j \), and then \( j \) crashes. Suppose \( i \) receives \( m_{x'} \) at \( x'_i \), but not after \( m_x \) because of its delay. \( i \) will then stop receiving any messages from \( j \) broadcasted later than time \( x'_j \) (because of \( j \)'s failure). Eventually, at time \( x'_i + \sigma \), \( SV_i(j) = \langle \text{Comatose, } \ast \rangle \). However, now if \( m_x \) gets delivered at \( x_i \), it would be inappropriate for \( i \) to suddenly suspect \( j \) as having returned. See Figure 3.6.

![Figure 3.6: An example of a message delay problem.](image-url)

Thus, a mechanism can be used to only accept the most current messages. Part of
every message broadcast by member $j$ is a message sequence number ($SN_j$) identifying that message. Whenever a member broadcasts a message, it would attach a sequence number greater than any previously used sequence number. Thus, a higher sequence number implies a more recent system view. Now, if member $i$ receives a message from member $j$ with sequence number $SN_j$, it would not accept the message if $SN_j$ is not greater than the sequence number of the most recent message $i$ received from $j$.

Together, the sequence numbers, modification numbers, and last commit numbers, can be used to augment the state transitions of Section 3.5 as summarized in Figure 3.7.

### 3.8 Determining the Initiator

Accurately determining the initiator is fundamentally difficult in asynchronous systems. The variability in the network and uncertain timing conditions can lead to inconsistencies. This is very similar to the leader election problem [37]. The GSM protocol attempted to steer away from these problems by designating a leader instead of electing one. However, in determining the initiator, many of the same problems arise.

To initiate modifications, an initiator is chosen to select a modification to perform. Each member $i$ in the system is assigned a unique identification number ($Id_i$) when it has been successfully added to the system. This identification number is used to distinguish one member from another and a list of all member identification numbers is maintained by every member in the system. Thus, any member can determine the leader and crown prince locally simply by checking this list. There are many complications involved in determining which member is the real initiator, the leader or the crown prince.

In a non-faulty system, the leader is responsible for initiating all modifications. However, in a system in which failures can occur, the initiator can either be the leader or the crown prince. The problem that can occur is accurately detecting a failure. Because of potential communication failures, the leader could perceive that the crown prince has
Figure 3.7: Incorporating message sequence numbers, modifications numbers and last commit numbers into the algorithm. This summarizes what actions i would take upon reception of a message from j.
failed. and at the same time, the crown prince could perceive that the leader has failed.

For example, consider the following scenario:

- four members \( \{m_1, m_2, m_3, m_4\} \)

- \( \text{Id}_{m_1} < \text{Id}_{m_2} < \text{Id}_{m_3} < \text{Id}_{m_4} \)

with the following state vectors:

- \( SV_{m_1} = \{(\text{Stable}, \ast), (\text{Critical}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast)\} \)

- \( SV_{m_2} = \{(\text{Critical}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast)\} \)

- \( SV_{m_3} = \{(\text{Critical}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast)\} \)

- \( SV_{m_4} = \{(\text{Stable}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast), (\text{Stable}, \ast)\} \)

According to the leader \((m_1)\), \(m_2\) has not been heard from for at least \(\lambda\) time, but \(m_3\) and \(m_4\) are \(\langle\text{Stable}, \ast\rangle\). The crown prince \((m_2)\), on the other hand, has viewed \(m_1\) as \(\langle\text{Critical}, \ast\rangle\) and \(m_3\) and \(m_4\) as \(\langle\text{Stable}, \ast\rangle\). \(m_3\) also views the leader as being \(\langle\text{Critical}, \ast\rangle\) and would thus think the crown prince should lead the system out of this problem. \(m_4\) has the opinion that the system is perfectly healthy with no faulty members.

This heterogeneous view poses a problem when initiating modifications. In the previous example, it would be inconsistent for \(m_1\) to initiate a removal of \(m_2\) while at the same time, \(m_2\) is initiating the removal of \(m_1\). Thus, it is important for only one member to make a decision and for this member to make only one definitive decision. The GSM protocol must allow only one modification at any time and ensure the modification is consistent with the current view. Thus, even when one of the potential initiators views the other as faulty, more information is required before it should attempt to modify the system.
3.8.1 Steady Systems

In the previous example, the cause of such an imbalanced view is typically due to the network; it is unreliable. Messages are being lost or perhaps the network has partitioned in an unusual manner. In such a system, the leader or crown prince should not attempt a modification until the network becomes more reliable.

Ideally, to be able to perform additions or updates, every member must be present, with no faulty members. If there was a faulty member, it would have to be removed before an addition or update could be performed. Thus, the initiator would be the leader. It should not attempt to initiate an update or addition unless it has the view that every other member is alive. This can also be strengthened. The leader should not attempt to initiate an addition or update unless every member is also receiving heartbeats from every other member. This is our notion of a steady system with no failures. However, the leader would need an oracle in order to accurately determine if this was indeed the case. This can only be approximated. This is where the alive vector is used.

The alive vector of $i$ keeps, as current as possible, the state vectors of all other members. As mentioned, this is done by saving the converted state vector of a last heartbeat from another member $j$. Thus, the alive vector can be used to approximate every member's perception of the system, according to $i$. Now, the concept of being steady, with respect to $i$, can be formalized.

Let $M$ be the set of all the members of the group. If $i$ is the leader in a stable system, $\text{Steady}_i$ is true iff:

1. $\forall m \in M, \forall k \in M, AV_{im}(k) = \text{Awake}$

In general, in a steady fault-free system, every member is able to detect heartbeats from every other member.
3.8.2 Steady Systems With Failures

In a failure-prone system, where $i$ is either the leader or crown prince and $r$ is a faulty member. $Failed_i(r)$ is true iff:

1. $AV_i(r) = Critical \land \forall m \in (M - \{r\}). AV_i(m) = Awake$
2. $\forall m \in (M - \{r\})$,
   
   (a) $AV_{im}(r) = Critical$
   
   (b) $\forall k \in (M - \{r\}). AV_{im}(k) = Awake$

The first condition states that according to $i$’s alive vector, $i$ has the view that $r$ is in $\langle Critical, * \rangle$ and $i$ views every other member as being in one of the Awake states. The second condition ensures that, according to $i$’s alive vector, every member $\neq r$ has the same alive vector entry as $i$ (essentially $AV_i = AV_{im}$). $Failed_i(r)$ has the property that every member $\in (M - \{r\})$ views one member as the ultimate leader. Thus, in the example presented at the beginning of this section (3.8), neither the leader nor the crown prince would attempt to remove the other.

3.8.3 Initiations

With the $Steady_i$ and $Failed_i(r)$ predicates, the first initiation transition from $\langle Stable, * \rangle$ (or $\langle Degraded, * \rangle$) to $\langle m, Propose \rangle$ can now be described.

For the leader $i$, if it was in state:

- $\langle Stable, * \rangle$
  
  $i$ can transition to:

  1. $\langle Add(a), Propose \rangle$ or $\langle Update, Propose \rangle$, if $Steady_i$.
  2. $\langle Remove(r), Propose \rangle$, if $Failed_i(r)$

- $\langle Degraded, * \rangle$
  
  $i$ can transition to:
1. \((Add(a), Propose)\), if \(Steady_i\).

Recall, in a degraded system, an update or removal is not permitted.

For the crown prince \(i\), if it was in state:

- \(\langle Stable, \ast \rangle\)

\(i\) can transition to:

1. \(\langle Remove(r), Propose \rangle\), if \(Failed_i(r)\), where \(r\) is the leader.

In a degraded system, the crown prince has no role in initiations because even if the leader fails, no removals are possible.

### 3.8.4 Inconsistent Initiations

As mentioned in Section 3.5, it would be inconsistent for two modifications to be initiated. This could easily happen if, for example, the leader \(i\). evaluated \(Steady_i\) and transitioned to \(\langle m, Propose \rangle\). However, before communicating this in its heartbeat, it immediately crashes. Eventually, the crown prince \(j\), will evaluate \(Failed_j(i)\) and transition to \(\langle Remove(i), Propose \rangle\). Now, if \(i\) immediately recovers, then there becomes a race between \(i\) and \(j\) to draw other members to their state. Thus, we assume that once \(Failed_j(i)\) is evaluated at member \(j\), then \(i\) has indeed hard crashed and will never recover its pre-crash state:

- \(Failed_x(y) \supset HardCrashed(y)\)

The strength of the \(Failed_x(y)\) predicate gives us some justification of this assumption. As discussed in Section 3.6, the value of \(\lambda\) should be set to allow time for members that have soft crashed to recover. Thus, if one member has not received a heartbeat from another member for \(\lambda\) time, this should be sufficient to deduce its failure. However, \(Failed_x(y)\) is much stronger requiring that every member also has not received a heartbeat
for λ time. In addition, Failed\(_x(y)\) requires that every member has recently received heartbeats from every other member (except for \(y\)). From a graph theory perspective, this would represent a complete directed graph or clique with directed edges representing recently received heartbeats between members.

### 3.9 Transitions In The Presence of Failures

Now that solutions to the aforementioned problems to the distributed facility have been discussed, a full description of the protocol can be given. We have also filled in the missing piece to the previous state transitions by describing how initiations begin. Essentially, the protocol that handles failures is the same as the protocol that does not handle failures except that in each state, a description of the transitions are required when a failure is detected. Messages received are assumed to pass through the filtering mechanism described in Section 3.7.

Conditions for initiations have been discussed in Section 3.8.3. Initiations are always triggered by the leader or the crown prince. Thus, if a member \(i\) which is not the leader or crown prince detects a failure of another member \(j\), it would not attempt to initiate any modification. If \(j\) had indeed failed, then the initiator would eventually attempt to initiate a removal. If \(j\) had not failed and it was a communication problem with \(i\) then no action would take place which is the correct behaviour. It should be noted that if more than one member is perceived to have failed, then the leader and crown prince will never be able to evaluate Steady\(_i\) or Failed\(_i(r)\) to true because the predicates only allow for one failure. As described in Section 3.5, this \((m, Propose)\) will eventually propagate to the other members.

The following state transitions describe how a member would handle a failure within each substate of the modification. The transitions must also maintain consistency. Thus, a justification of the transitions is also given.
For a member \( i \) in state:

- \( \langle m. \text{Propose} \rangle \)

  \( i \) can transition to:

  1. \( \langle m. \text{Abort} \rangle \), if \( \exists j, SV_i(j) = \langle \text{Comatose}, * \rangle \).

     \( i \) may not be able to reach \( \langle m. \text{Prepare} \rangle \) if \( j \) really had failed. It could thus transition to \( \langle m. \text{Abort} \rangle \) unilaterally without causing inconsistencies. The reason for this is because even if \( i \) were incorrect in detecting this failure, and the other members managed to move to \( \langle m. \text{Prepare} \rangle \), and then proceeded to \( \langle m. \text{Scommit} \rangle \) or \( \langle m. \text{Dcommit} \rangle \), they would not be able to commit the modification without \( i \)'s commit vote. The fact that \( i \) never broadcasted a \( \langle m. \text{Scommit} \rangle \) or a \( \langle m. \text{Dcommit} \rangle \) message, means no other member can collect enough commit messages to return to \( \langle \text{Stable}, * \rangle \text{Commit} \), or to a \( \langle \text{Degraded}, * \rangle \text{Commit} \), respectively. Eventually, when the other members recognize that \( i \) is in \( \langle m. \text{Abort} \rangle \), then they will also move to this state as well. \( i \) may have made a mistake but this does not cause inconsistency. However, it does cause inefficiencies because the modification would have to be restarted. If \( \lambda \) is set properly, the number of false positives can be minimized.

- \( \langle m. \text{Prepare} \rangle \)

  then \( i \) still continues to broadcast and receive heartbeats. It still detects heartbeats and a lack of them. However, it does not make transitions based on this. As before, it only makes the transitions based on a message from the client application.

- \( \langle m. \text{Abort} \rangle \)

  \( i \) can transition to:

  1. \( \langle \text{Stable}, * \rangle \text{Abort} \), if \( \exists j, SV_i(j) = \langle \text{Critical}, * \rangle \) and \( m \) originated from a stable state.

  2. \( \langle \text{Degraded}, * \rangle \text{Abort} \), if \( \exists j, SV_i(j) = \langle \text{Critical}, * \rangle \) and \( m \) originated from a degraded state.

If a member \( i \) is in \( \langle m. \text{Abort} \rangle \), and there was a perceived failure of \( j \) from \( i \) then \( i \) would not be able to receive all the necessary votes to move to \( \langle \text{Stable}, * \rangle \text{Abort} \) (no
change to $LCN_i$ or $f_i$). However, since $i$ is in $\langle m, \text{Abort} \rangle$, it could unilaterally transition to $\langle \text{Stable}, * \rangle_{\text{Abort}}$. This transition does not cause inconsistencies nor any further inefficiencies. However, we need to show that if $i$ reaches state $\langle \text{Stable}, * \rangle_{\text{Abort}}$, then no other member could reach a $\langle \text{Stable}, * \rangle_{\text{Commit}}$ or $\langle \text{Degraded}, * \rangle_{\text{Commit}}$.

If $i$ reaches $\langle m, \text{Abort} \rangle$, then it can transition to this state in one of two ways:

1. $i$ went straight to $\langle m, \text{Abort} \rangle$, never having transitioned through $\langle m, \text{Scommit} \rangle$ or $\langle m, \text{Dcommit} \rangle$ (and thus, never broadcasted $\langle m, \text{Scommit} \rangle$ nor $\langle m, \text{Dcommit} \rangle$).

   If this is the case, then it is not possible for the other members to receive all necessary $\langle m, \text{Scommit} \rangle$ or $\langle m, \text{Dcommit} \rangle$ votes to transition to $\langle \text{Stable}, * \rangle_{\text{Commit}}$ or $\langle \text{Degraded}, * \rangle_{\text{Commit}}$.

2. $i$ was in either the $\langle m, \text{Scommit} \rangle$ or $\langle m, \text{Dcommit} \rangle$ state before transitioning to the $\langle m, \text{Abort} \rangle$ state.

   We must be careful because $i$ could have broadcasted that it was in one of these two commit states before proceeding to $\langle m, \text{Abort} \rangle$. Thus, we look at why $i$ moved to $\langle m, \text{Abort} \rangle$ in the first place. $i$ must have proceeded to $\langle m, \text{Abort} \rangle$ after receiving a message from another member $j$ who was in that state. Consider the first member $x$ that moved into $\langle m, \text{Abort} \rangle$. Since no other members are in $\langle m, \text{Abort} \rangle$, it must be the case that it transitioned to $\langle m, \text{Abort} \rangle$ from $\langle m, \text{Prepare} \rangle$ (via a timeout) or from a notification received from the client application indicating it could not perform the necessary operations to complete the modification. Thus, $x$ never sent out a $\langle m, \text{Scommit} \rangle$ nor a $\langle m, \text{Dcommit} \rangle$ message and thus, the other members could not possibly receive the required votes to move to $\langle \text{Stable}, * \rangle_{\text{Commit}}$ or $\langle \text{Degraded}, * \rangle_{\text{Commit}}$.

- $\langle m, \text{Dcommit} \rangle$ or $\langle m, \text{Scommit} \rangle$

   If a member $i$ is in one of the Commit states, $\langle m, \text{Scommit} \rangle$ or $\langle m, \text{Dcommit} \rangle$, and perceived a failure of $j$ but still has not reached a unanimous verdict on the modification, then $i$ is effectively blocked. It can not unilaterally move to $\langle m, \text{Abort} \rangle$ because it already had broadcasted its willingness to commit $m$ which $j$ might have used to locally commit $m$ but crashed before notifying anyone else of this result. However, because of the decentralized commitment approach, the only way $i$ would be deadlocked is if $j$'s vote (or implied vote if it had transitioned to an Alive state) was not received by any other member, and $j$ received all other members' commit
messages. However, the next section discusses another approach used to further prevent blocking in this scenario.

3.9.1 Blocking Nature in the Presence of Failures

One dilemma this protocol introduces is the potential of blocking. Again, the reason a member \( i \) might block is if it enters one of \( (m, Scommit) \) or \( (m, Dcommit) \), but cannot collect enough votes to transition to \( (Stable, *)_{commit} \) or \( (Degraded, *)_{commit} \). This typically occurs because the failed member \( s \) may have taken its vote to its grave when it crashed so the rest of the members never knew \( s \)'s vote. However, if, at this point, we assume that \( s \) has hard crashed and will never recover, the resulting members can prevent blocking.

An alternative to blocking would be to now take an approach similar to initiations to handle this failure. The initiation transition that begins a modification is centralized with one definitive initiator while the rest of the transitions are decentralized. However, to overcome the blocking dilemma, a centralized approach will again be needed. The key is to look at what the members can do in the resulting system if they committed the modification.

For the leader \( i \), if it was in state:

- \( (m, Scommit) \)

\( i \) can transition to:

1. \( (Stable, *)_{commit} \), if \( Failed_i(s) \)

If every other member except \( s \) was in \( (m, Scommit) \), then these members are able to commit the modification and in the resulting system, they would still be able to sustain one failure. The resulting system would be stable. Thus the members, less \( s \), could commit the modification and immediately attempt to remove \( s \). Again, this situation would not cause inconsistencies with the assumption that \( s \) has hard crashed (implied by \( Failed_i(s) \)).
For the crown prince $j$, if it was in state:

- \(S\text{commit.} \ast\)

$j$ can transition to:

1. \(\text{Stable.} \ast\)\text{commit.} \text{if Failed}_j(s)$, where $s = i$

This is the same as the previous situation except where the leader has failed and the crown prince has taken charge.

However, if either the leader or the crown prince was in state \(D\text{commit.} \ast\), then these transitions can not be made. The state \(D\text{commit.} \ast\) indicates that the member is able to commit the modification; however, it would not be able to sustain another failure in the resulting system. Thus, in the resulting system, the removal of $s$ would not be permitted. In this situation, the members would be effectively blocked until $s$ recovers and its vote is known.

Nevertheless, if further failures occur, blocking will be necessary. This is due to the one-failure requirement of the client application. In the resulting system, the client application would never be able to remove two or more members at a time. Thus, even a three-phase commit protocol, which handles an arbitrary number of failures would not be very useful here.

### 3.9.2 Members Recovering From Soft Crashes

A member recovering from a soft crash can take a look at the messages on the network to determine if it is still relevant to the system and relevant to the modification. Section 3.7 describes how this is done. It can then perform the necessary actions to resolve any differences, including restarting as a new node if the differences can not be resolved.
3.10 A Timing Analysis of \textit{Steady}_i \textbf{and} \textit{Failed}_i(r)

This section shows some interesting timing analysis that might be interesting for future work. This analysis is intended to show how one might overcome the required assumption associated with the \textit{Failed}_i(r) predicate; the required assumption being \textit{Failed}_i(r) implies that \( r \) has hard crashed. The following is an observation and is not part of the protocol.

3.10.1 Supporting A Modification

When the leader or the crown prince wants to initiate a modification, it must get \textit{support} from every member (except the member being removed for removals).

The \textit{support} by \( j \) for a modification initiated by \( i \) depends on the modification. For example, if \( i \) (the leader) wanted to initiate an addition, then it would require that, according to the last state vector received by \( j \) (stored in \( AV_{ij} \)), every member is in \( \langle \text{Stable}, \ast \rangle \) (or \( \langle \text{Degraded}, \ast \rangle \) for degraded additions). Thus, if \( j \) broadcasted a heartbeat with this state vector, then it is supporting an addition (or update) initiated by the leader \( i \). Similarly, if \( i \) wanted to initiate a removal of \( r \), then it would require that, according to the last state vector received by \( j \), every member is in \( \langle \text{Stable}, \ast \rangle \) and \( r \) is in \( \langle \text{Critical}, \ast \rangle \). Thus, if \( j \) broadcasted such a state vector, then it would be supporting a removal of \( r \). It should be noted that support is not a different message that is sent from one member to another but rather support is inferred from the state vector that is part of the heartbeat message.

3.10.2 Conflicting Initiations

There are two possibly conflicting initiations:

1. The leader \( i \) is initiating the removal of a member \( r \), where \( r \) is possibly the crown prince \( (\text{Failed}_i(r)) \), while the crown prince \( j \) is initiating the removal of \( i \), \( (\text{Failed}_j(i)) \).
2. The leader $i$ is initiating an addition or update ($\text{Steady}_i$) while the crown prince $j$ is initiating the removal of $i$ ($\text{Failed}_j(i)$).

It is important that $\text{Failed}_i(r)$ and $\text{Failed}_j(i)$ do not occur at the same time as each other, similarly for $\text{Steady}_i$ and $\text{Failed}_j(i)$. We now show that this could happen in certain conditions which can be rectified by augmenting $\text{Steady}_i$ and $\text{Failed}_i(r)$.

Consider the conflicting initiation of situation 1. We need to consider how a member $k$ would change support from a removal by the leader to a removal by the crown prince and vice-versa.

1a. $k$ changing support from a removal of $r$ by the leader to the support of a removal of the leader by the crown prince.

If $k$ could support a removal of $r$ (where $r$ could be the crown prince), $i$ must have $r$ as $(\text{Critical, } *)$ and every other member is $(\text{Stable, } *)$, in $k$'s state vector. To switch support to the crown prince, its state vector must reach the point where the leader is in $(\text{Critical, } *)$ while every other member (including the member originally being removed, $r$) is in state $(\text{Stable, } *)$. The shortest time this could occur is if $k$ stopped receiving messages from the leader for $\lambda$ time and then immediately receives a heartbeat from $r$. See Figure 3.8.

1b. $k$ changing support from a removal of the leader by the crown prince to the support of a removal of $r$ by the leader.

This is very similar to the previous situation. If $k$ was supporting a removal of the leader by the crown prince, then $k$'s state vector must have the leader as $(\text{Critical, } *)$ and every other member as $(\text{Stable, } *)$. To change support to a removal initiated by the leader, $k$'s state vector would need to show that the member being removed ($r$) is in $(\text{Critical, } *)$ and every other member is in $(\text{Stable, } *)$. The quickest way this could happen is if $k$ experienced a $\lambda$ timeout on $r$, and immediately received a message from the leader (thus, making it $(\text{Stable, } *)$ according to $k$). See Figure 3.9.

In either case, there exists a window of $\geq \lambda - \sigma$ time for which $i$ does not support either modification.
Figure 3.8: k's change from a leader initiated removal of r to the crown prince initiated removal of the leader. i.e. t₁ is the last time k has heard from the leader. It continues to support the leader (Δt₁) until it experiences a short timeout on i at t₂ or if it starts to receive heartbeats from r again. Regardless, during Δt₂, it does not support any modification. However, after time t₃ (a λ timeout on i), k could now support the crown prince if it hears from r. Note: Δt₁ = σ, Δt₂ = λ − σ.

Figure 3.9: k's change from a crown prince initiated removal of the leader to the leader initiated removal of r. t₁ is the last time k has heard from r. It continues to support the crown prince (Δt₁) until it experiences a σ timeout on r (SVᵢ(r) = (Comatose,*)) at t₂ or if it starts to receive heartbeats from the leader again. Regardless, during Δt₂, it does not support any modification. However, after time t₃ (a λ timeout on r and SVᵢ(r) = (Critical,*)), i.e. k could now support the leader if it hears from the leader. Note: Δt₁ = σ, Δt₂ = λ − σ.

Now consider the conflicting initiation of situation 2. We need to consider how a member k would change support from an addition or update by the leader to a removal by the crown prince and vice-versa.

2a. k was supporting an update or addition by the leader and switched to supporting the crown prince for the removal of the leader.

If this were the case, then every entry in k's state vector must have originally been (Stable,*). Now, if k were to switch support to the crown prince (a removal of the leader), its state vector must reach the point where the leader is in (Critical,*) while every other member still remains in (Stable,*). The shortest time this could occur is if k stopped receiving heartbeats from the leader and this continues for λ time. See Figure 3.10.

2b. k was supporting the crown prince for the removal of the leader and switched to supporting the leader's update or addition.

This requires a more careful analysis. The description of changing support so far indicates that there exists a fairly long time period (λ − σ) between any two supports. This is not necessarily the situation in this case.
If $k$ was initially supporting the crown prince at time $t$, and immediately received a message $\epsilon$ time later from the leader, it would have a state vector that would support the leader for an addition or update at time $t + \epsilon$. $\epsilon$ can potentially be very small.

However, consider how the crown prince would change its support from its removal of the leader to an update or addition by the leader. Then it must be the case that its state vector has the leader as \(\langle \text{Critical,} \ast \rangle\) while every other member is \(\langle \text{Stable,} \ast \rangle\). To switch support, the crown prince would have to start receiving heartbeats from the leader again. If the crown prince’s condition for the removal of the leader, \(\text{Failed}_j(i)\), has not been satisfied yet, it can switch support without any inconsistencies.

However, if it had initiated the removal of the leader, the leader could not get the required support from the crown prince for the addition or update because the crown prince’s state vector does not indicate this support. Recall, the leader requires that all members support it before an addition or update can take place. In this situation, the last time the leader could possibly have received a message of support from the crown prince was before the crown prince experienced a $\lambda$ timeout on the leader. Hence, for at least one member (the crown prince), there exists a $\lambda - \sigma$ window for which the crown prince changes support. It should be noted that if this situation occurred, the leader would realize the crown prince has initiated a removal of it and the leader would wait in \(\langle \text{Stable,} \ast \rangle\) in hope that the crown prince’s modification fails. If it succeeded, then every other member $i$’s $LCN_i$ would be greater than the leader’s and thus the leader would need to reset itself. This is an example of a false removal that can be minimized if the value of $\lambda$ is set properly. However, if the modification failed, $LCN_i$ would be unchanged and the leader could integrate itself back into the system.

Thus, in both situations, there exists a time window of $\geq \lambda - \sigma$ for at least one member for a support message to reach both the crown prince and the leader. Thus, the crown prince and the leader could ignore support that is not recent. This is discussed in the next section.
Figure 3.10: k’s change from a leader initiated addition or update to a crown prince initiated removal of the leader. i.e., $t_1$ is the last time $k$ has heard from the leader. It continues to support the leader ($\Delta t_1$) until it experiences a $\sigma$ timeout on $i$ ($SV_k(i) = (Comatose, *)$) at $t_2$. During this time ($\Delta t_2$), it does not support any modification. However, after time $t_3$ (a $\lambda$ timeout on $i$ and $SV_k(i) = (Critical, *)$), $k$ now supports the crown prince. Note: $\Delta t_1 = \sigma$, $\Delta t_2 = \lambda - \sigma$.

### 3.10.3 Estimating Message Delay

As mentioned, to prevent both the crown prince and leader from initiating two different modifications, a mechanism is required to determine how recent the support is from other members. This is analogous to estimating message delay. If there is a bound on how long it took for a support message to arrive, it can be known if a support message is recent.

Estimating message delay time can be difficult primarily because of the different clocks on each member and a dynamic system. However, a crude way of estimating message delay requires a two message exchange. Node $i$ could send a message $m_i$ to $j$ and record the time, $t_i$, that this message was sent. $j$ would then send message $m_j$ back to $i$ indicating that it had received $m_i$. When $i$ receives this message at $t'_i$, in the worst case, it would have taken $\Delta t_i = t'_i - t_i$ time to receive message $m_j$. See Figure 3.11.

There are more sophisticated methods to achieve a more accurate upper bound on the message delay time [21]. However, this typically requires much more overhead and this crude method would be sufficient.

We can now apply this to our situation in which the crown prince and leader can use this mechanism to determine the recentness of support messages. Every member would send back, as part of the heartbeat, the last message sequence number that it had received from every other member. For example, if $k$’s last received message from $l$ was sequence number $SN_{m_l}$, then when $k$ is ready to broadcast its heartbeat, it would indicate that $SN_{m_l}$ was the last received message from $l$ in its message.
Figure 3.11: A crude method of estimating the upper bound on message delays.

Now, when the crown prince or leader $i$ receives a message $m$ from $j$, it can check the last message (based on the sequence number) $j$ had received from $i$, since $j$ now includes this as part of its message $m$. The crown prince or leader can keep track of the last message it sent out that would mark the time interval ($\gamma$) for which a support message would be considered recent (where $\gamma < \lambda - \sigma$). This is not exact given the clock drift rates $\rho$. However, $\gamma$ can be normalized to take into account $\rho$. Nevertheless, if $\gamma$ is sufficiently small ($\gamma \ll \lambda - \sigma$), the clock drift rate would be compensated.

Now, suppose member $i$ sent out a message $m$ with sequence number $SN_m$ at time $t_m$ such that $t_m = t_{curr} - \gamma$, where $t_{curr}$ is the current time. Then all messages $m'$ sent out by $i$ with sequence number $SN_{m'}$ at time $t_{m'}$ where $SN_{m'} > SN_m$, would be accepted as recent support. See Figure 3.12. $t_m$ and $SN_m$ would have to be updated as the current time passes. It should be noted that the messages sent by $j$ must also satisfy the support condition as well as being recent.
Figure 3.12: Suppose \( i \) sent out message \( m \) at time \( t_m = t_{curr} - \gamma \), and \( m_1 \) at time \( t_1 \). If \( i \) received a message from \( j \) indicating that the last message \( j \) had received from \( i \) was at least message \( m \), then \( i \) would accept the message as recent support. Thus, if \( j \) received \( m \) and sent out \( m_2 \), then this would be accepted as support from \( j \) if received at time \( t_2 \). However, if \( i \) received \( m_2 \) at time \( t_4 \), then it would not count this as recent support from \( j \). Also, if \( j \) sent out \( m_3 \) indicating that the last message received from \( i \) was \( m_1 \) and this is received by \( i \) at \( t_3 \), then the message would also be counted as recent support. This "indication" is accomplished by the sequence numbers.
Chapter 4

Implementation

The GSM was implemented in the C programming language in the *Linux* operating system environment. The *TCP/IP* suite of protocols were used for all communication, UDP for communication between GSMS and TCP for communication between the GSM and the client applications. The GSM is highly configurable. All configurable parameters are placed in a configuration file (*config.h*). For example, the values of the broadcast interval ($\beta$), the soft and hard crash parameters ($\sigma$ and $\lambda$), the number of client connections, and the maximum number of members can be configured.

The rest of the chapter describes some of the implementation decisions. Information maintained by the GSM, including the structures it uses, is discussed in detail. From this information, we discuss how some of the concepts are evaluated. Specifically, a discussion of how a member knows when to broadcast a message, how it calculates when $\sigma$ or $\lambda$ is reached, and how $\text{Steady}_i$ and $\text{Failed}_i(r)$ are evaluated. Furthermore, the communication between GSMS and the communication between the GSM and the client applications are discussed including the communication protocol used between them. Finally, the chapter ends with a short discussion of system performance.

75
4.1 Implementation Details

The primary actions for the GSM are receiving messages from other GSMs and acting upon them, receiving messages from client applications such as a vote message (an indication to commit or abort a modification), broadcasting its state information, and handling member timeouts. The secondary action the GSM is responsible for is client requests to establish a connection.

The GSM is driven by these events, and thus, remains idle unless an event occurs. A select statement can be used to trigger the GSM when one of these events occurs. It can then take the necessary actions based on the event.

Pseudo-code for the main loop is illustrated in Figure 4.1.

4.1.1 Information Maintained by the GSM

Each GSM maintains a list of information about the current members and potential members. This information is periodically persistently stored to allow a crashed GSM to attempt a recovery and re-join the system. This information allows the GSM to maintain up-to-date information about the system. Each GSM $i$ maintains:

$id$: an unsigned int uniquely identifying $i$ in the system. Currently a configurable maximum number of members ($n$) are allowed. The unique identification number is chosen between 0 and $n - 1$, inclusive.

$members$: an unsigned int representing the actual number of members that are currently in the system, $0 \leq members \leq n$. This field has the value 0 when the node is in $(Join, \ast)$ indicating that it is not yet part of the system.

$modid$: an unsigned int representing the index of the member $i$ is currently modifying (either adding or removing), or $GSM\_NO\_MOD$ if it is not making any modification related to another GSM such as data updates or when $i$ is in $(Stable, \ast)$.

for each member $j$ in the system (including $i$ itself)

$ip$: a struct in_addr representing the IP address of member $j$. 
Figure 4.1: The event driven main loop of the GSM. There are four major events that can occur: establishing client connections with the GSM, a timeout, a heartbeat message received from another GSM, and a message received from a connected client.

```plaintext
loop
  wait for an event to occur

  if the event is a client application connection request
    establish a connection with this new client
    add it to the list of clients to broadcast to

  if the event is a timeout on the local member
    broadcast its state information to the GSMs and clients
    update the next time to broadcast the state information
  else if the event is a sigma timeout of the remote member
    notify the client application
  else if the event is a lambda timeout of the remote member
    a removal may be necessary

  if the event is a message from another GSM
    update the local information based on this message

  if the event is a message from a client application
    handle this message based on the type
endloop
```
state: a stateType for the perceived state of \( j \) (according to \( i \)). A stateType is the representation of states described in Section 3.4.1 and 3.6 which consists of two components: a state and a substate, if necessary.

sn: an unsigned int for the message sequence number. If \( i = j \) then this is the sequence number of the next message \( i \) is to broadcast. If \( i \neq j \), then this is the last sequence number \( i \) has received from \( j \).

mn: an unsigned int for \( j \)'s current modification number.

lcn: an unsigned int for the modification number of the modification \( j \) last committed.

f: an unsigned int for the number of failures \( j \) can sustain. This corresponds to \( j \) being in either Stable (\( f = 1 \)) or Degraded (\( f = 0 \)).

nextmsg: a struct timeval whose value depends on whether \( i = j \). If \( i = j \), this is the next time \( i \) is expected to broadcast its state. If \( i \neq j \), this value is the next time \( i \) expects to hear a message from \( j \). If \( i \) receives a message from \( j \) then it would reset this field to the time of the next expected message (\( \text{currTime} + \beta \), where \( \text{currTime} \) is the current time).

missedmsg: an unsigned int to record the number of times we have timed out on this member. When \( i = j \), this field is not relevant. Thus, if \( i \) times out on \( j \) (nextmsg[j] expires) it would increment this field by one and reset nextmsg[j] to be \( \text{currTime} + \beta \) indicating it is still expecting messages from \( j \). If \( i \) receives a message from \( j \), this field is reset to 0.

steady: a bool indicating if the last message received from \( j \) has the same state vector as \( i \). It is set to TRUE if so, FALSE otherwise. If, according to \( i \), \( j \) becomes \( \langle \text{Comatose}, \ast \rangle \) or \( \langle \text{Critical}, \ast \rangle \). then this field is set to FALSE if it was not already this value.

Member Timeouts and Broadcast Timeouts

Timeouts are used to determine if a member should be considered \( \langle \text{Comatose}, \ast \rangle \) or \( \langle \text{Critical}, \ast \rangle \). We make use of the hardware clocks that reside on the underlying system.
For GSM $i$, the $\text{nextmsg}[j]$ field is used to keep track of the next time we expect to receive a heartbeat from $j$. When $i = j$, the $\text{nextmsg}[j]$ field is the next time $i$ should broadcast its state.

The $\text{nextmsg}$ array forms the list of timeout events. The next expected timeout event is $T = \{k|\forall l. \text{nextmsg}[k] \leq \text{nextmsg}[l]\}$. In the case where there is more than one choice, when $|T| > 1$, an arbitrary $k \in T$ can be chosen. The amount of time until the next timeout is $\text{timeLeft} = \text{currTime} - \text{nextmsg}[k]$. If $\text{timeLeft}$ is negative, it is just set to 0. The $\text{select}$ function uses $\text{timeLeft}$ as its timeout trigger.

$k$ can either be the local member $i$ or it could be a remote member. If $k \neq i$, two things can happen with respect to $k$: the $\text{select}$ can either receive a message from $k$, or timeout on $k$.

1. If we receive a message from $k$ before $\text{timeLeft}$ expires, then the $\text{nextmsg}[k]$ field would be incremented to the next time we expect to hear from $k$. More specifically, $\text{nextmsg}[k] := \text{currTime} + \beta$. The $\text{missedmsg}[k]$ is set to 0 (if it is not already), indicating that we have recently heard from $k$ again.

2. If we do not receive a message from $k$, then $i$ indicates that it has missed an expected message by incrementing $\text{missedmsg}[k]$ by one. $i$ accumulates the number of missed messages of $k$ until $\sigma$ or $\lambda$ thresholds are met. The $\text{nextmsg}[k]$ field gets set again to $\text{currTime} + \beta$ and the process of choosing the next expected timeout event described above is again invoked.

If $k = i$, then essentially only one event can happen with respect to $i$. When $i$ times out in the $\text{select}$ statement, it will broadcast its state to all GSM members of the system including to all connected clients. The $\text{nextmsg}[i]$ field will then be reset to $\text{currTime} + \beta$.

**Determining Comatose and Critical Members**

The $\text{missedmsg}$ field is used to indicate the number of missed messages $i$ has experienced for any particular member. The amount of time since it has last received a message from $k$ can be calculated by taking $\text{missedmsg}[k] \times \beta$. 
If missedmsg[k] exceeds the threshold $\sigma$, then the member is considered Comatose. If it exceeds $\lambda$, then the member is considered Critical. The local member would then take the necessary actions to handle each failure. A Comatose member is only handled at the local level and clients are immediately notified of such an occurrence. A Critical member is handled at the global level and the leader will attempt to get enough support to remove $k$.

**Evaluating Steady, and Failed,$(r)$**

The steady field is used to determine if, according to $i$, the system is Steady, or Failed,$(r)$. The alive vector ($AV_i$) of each member does not need to be maintained. Instead, $i$ only needs to maintain this steady field for all members. steady[i] is set to TRUE if, according to $i$’s state vector, it has every member in an Awake state (attempting to achieve Steady,$i$) or it has every member except $r$ in an Awake state and $r$ is in $\langle$Critical, $\ast$$\rangle$ (attempting to achieve Failed,$i$(r)). Now, when $i$ receives a heartbeat from another member $j$, it can evaluate to see if $j$ has the same converted state vector as it does. If so, then steady[j] on $i$ is set to TRUE. If every member has this field set to TRUE, then the corresponding predicate is true. If $i$ and $j$ do not have the same converted state vector or if $i$ views $j$ as $\langle$Comatose, $\ast$$\rangle$ or $\langle$Critical, $\ast$$\rangle$, then steady[j] is set to FALSE.

Only the leader or crown prince would attempt to evaluate the either predicate. However, since every member can potentially become the leader or crown prince, every member maintains this information.

### 4.1.2 Communication Protocol

The communication used between the client application and the GSM is the connection-oriented Transmission Control Protocol (TCP). However, the communication used between GSMs is the connectionless User Datagram Protocol (UDP). Client applications make a connection with their local GSM by establishing a TCP connection at a well-
known port. TCP is used between the client and the GSM because this connection is local. Thus, reliability and message transmission delays are negligible. However, because communication between GSMS is remote, messages may suffer unpredictable transmission delays. Although TCP ensures the reliability of message transmission whereas UDP uses a "best efforts" approach, TCP does not handle lost connections such as node failures in a timely manner. Thus, the periodic message heartbeats over UDP and timeouts are used to deduce potential failures in a timely manner.

**GSM To GSM Message Protocol**

Messages sent from one GSM to another reflect the information necessary to make state changes. Messages are simply snapshots of the system information of the GSM encapsulating the imperative details. The heartbeat message broadcasted from member \( i \) takes the following form:

- **id**: an unsigned int of the unique identification number of the sending node \( i \) (used to identify members).
- **sn**: an unsigned int indicating the sequence number of this message.
- **mn**: an unsigned int of the current modification number of \( i \) to indicate the modification it is currently performing.
- **lcn**: an unsigned int of the last modification \( i \) last committed.
- **members**: an unsigned int for the number of members that are in the system according to \( i \).
- **f**: an unsigned int of the number of failures it can further sustain. Again, this corresponds to member \( i \) being in state \( \langle Degraded, * \rangle \) or \( \langle Stable, * \rangle \).
- **pb**: piggybackType information with which client applications can have the GSM transparently piggyback across the network to other client applications. It may be convenient for the client applications to communicate information using the GSM. Thus, the GSM is also equipped with the ability to receive data from its local client
application and transport it in its heartbeat message to the other GSM members and ultimately to the client application on the remote nodes. The user can define a *piggybackType* to specify this data in the configuration file.

**modid:** an *unsigned int* for the member \(i\) is currently modifying (either adding or removing), or *GSM_NO_MOD* if it is not modifying any member.

**ip:** a *struct in_addr* for the IP address of the member \(i\) is currently adding (if any).

for each member \(j\) in the system (including \(i\) itself)

**ip:** a *struct in_addr* for the IP address of the member \(j\).

**state:** a *stateType* for the state of member \(j\). Collectively, this field forms the state vector of \(i\).

The message is created based on the current configuration of the local member described in Section 4.1.1.

**GSM-Client Message Protocol**

Communication between client applications and the GSM takes a similar form to communication among GSMs. When the GSM broadcasts its local state to the other GSMs, it also broadcasts its state to the clients in a slightly different representation. It is the responsibility of the client application to listen to this information and take actions accordingly. For example, when the local GSM has indicated that another member is in \(\langle Comatose, *\rangle\), the RSML for MultiText would take the action of activating the secondaries to quickly compensate for the potential failure.

Clients can also communicate with the GSM by indicating their preparedness to commit an operation. When the GSM passes control to the clients (during \(\langle m, \text{Prepare} \rangle\)), the client needs to make the necessary operations to perform the indicated modification and then inform the GSM of its status. In addition, the client application may also wish to transparently send data to other members via the GSM. This is packaged in the
user-defined piggybackType. Finally, the client application may request for the GSM to facilitate an atomic update on its behalf.

Thus, there are four different client-GSM message types. The first is sent only from the GSM to clients, while the others are only sent from clients to the GSM. Thus, messages take the following form:

type: the type of message being sent. There are three different types:

STATE: this type indicates a local state message sent from the GSM to the client.

COMMIT: this type is a message sent from the client to the GSM about its readiness to commit the current modification.

PIGGYBACK: this type is the information the client wishes to send transparently over the GSM.

UPDATEREQ: this type is a request by the client to facilitate an atomic update. The GSM would initiate the update at the earliest possible time based on the Steady, predicate.

data: the relevant data being sent based on type. This information is a union of three possible structures:

state: for STATE message types. This structure is exactly the same structure as those messages sent from GSM to GSM described in Section 4.1.2.

commit: for COMMIT message types. This is a single field with three possible values corresponding to the client application's status:

CLIENT_ABORT: the client can not commit the modification.

CLIENT_DCOMMIT: the client can commit the modification, but it could not sustain any more failures.

CLIENT_SCOMMIT: the client can commit the modification unconditionally.

pb: for PIGGYBACK message types. This is the piggyback structure defined by the user application.

It should be noted that there is not associated data for UPDATEREQ. The GSM only facilitates the update and ensures it is performed atomically.

This structure is illustrated in Figure 4.2.
typedef union {
    gsmMsgType state;    /* same as messages sent from GSM to GSM */
    cltCommitType commit; /* either CLIENT_ABORT, CLIENT_SCOMMIT, or
                                CLIENT_DCOMMIT */
    piggybackType pb;     /* application user defined */
} cltMsgUnion;

typedef struct {
    cltTypeOfMsg type;  /* either STATE, COMMIT, PIGGYBACK, UPDATEREQ */
    cltMsgUnion data;
} cltMsgType;

Client Connections

Clients can establish a connection with the GSM at a well-known port on the local host. The number of clients that can connect to the GSM is user definable; however, the number of client connections to the GSM with the ability to vote must be limited to one. The extra client applications must be passive and simply use the information the GSM provides. For example, remote TCP reads and writes can make use of the service by establishing a connection and monitoring the GSM to determine the status of the remote member it has the TCP connection with. If the client finds that the remote member has potentially failed, it can abort the read and reset the TCP connection. If the client notices from the GSM that the remote member has recovered, it can attempt to re-establish the connection again.

4.2 Performance

Measuring the performance of the NML can be difficult since much of the performance depends on the client application. Once the client application has been implemented, one could measure the percentage of the total execution time for a given modification
that is attributable to the NML.

However, one of the main goals of the NML is to quickly notify the client applications of a potential crash. Thus, we can measure how much time it takes before the NML responds to failures. A proper measure must ensure that notification is achieved at all members. This measure must be done with respect to the soft crash parameter $\sigma$. The NML should notify all the clients within some $\epsilon$ of $\sigma$.

The cluster environment used is the Department of Electrical and Computer Engineering (ECE) cluster at the University of Toronto. It consists of 16 individual workstations. Each workstation is constructed with off-the-shelf components consisting of dual Pentium-II processors, between 128MB and 1GB of main memory, and 20GB of disk space (on two disks). The workstations communicate using an Ethernet hub operating at 100Mbps. In addition, some workstations have Myrinet cards installed.

The experiment was conducted with a value of $\sigma = 1s$. A killer rests on one of the members while the other members have witnesses. The killer is responsible for terminating the GSM on the local member while the witnesses act as the clients on the other members and wait for their local GSM to respond to the crash on the killer's member. This information is then relayed to the killer who calculates the time between the kill and the last witness to notify the killer. The response times for various broadcast periods ($\beta$) were measured for a various number of members. The results are presented in Figure 4.3.

Before the test was conducted, we expected the response time to be $\kappa = \sigma + \epsilon$ for some $\epsilon > 0$. In addition, intuition suggested that as the broadcast period increased, this would be accompanied by a decrease in $\kappa$. However, this was not the case. A careful analysis of the implementation revealed this discrepancy. Consider the experiment with a broadcast period of 0.5s and member $i$ having a witness while $j$ having the killer. Let the heartbeat $i$ last receives from $j$, before it is killed, be at time $t$. According to $i$, the next expected message from $j$ would be at time $t + \beta$. The GSM at $i$ would notify its client application
of a failure at time $t + 2\beta$ or after two missed heartbeats. However, the failure of $j$ occurs at some time between $t$ and $t + \beta$ on average being $t + \frac{\beta}{2}$. Thus, the expected response time would be: $(\text{notification time} - \text{failure time}) = (t + 2\beta) - (t + \frac{\beta}{2}) = \beta + \frac{\beta}{2}$. In general, a more accurate estimate of the expected response time is $\kappa = (\sigma - \beta) + \frac{1}{2}\beta + \epsilon$.

Essentially, the smaller the number of missed heartbeats we are willing to tolerate, the shorter the response time.
Chapter 5

Future Work and Conclusions

5.1 Integration of the NML with the RSML

The NML constitutes a major component of the RSML. There are three other main components to an RSML node: the Disk Management Layer (DML), the Node Controller (NC), and the Application Update Proxies (AUP). From an application standpoint, it is given a virtual disk to work with and receives requests to update information on its virtual disk, as well as requests to perform queries on the data stored on it. This is a similar application environment as in Figure 1.2 except for the interface with the (virtual) disk. Figure 5.1 shows the components of the RSML and its interaction with the application.

5.1.1 Node Controller

Every member has one Node Controller. The NC is responsible for co-ordinating all activities that involve the local primary and its remote secondaries. One specific responsibility is organizing the data in its primary and remote secondaries. Thus, when a data update is requested by the Update Marshaller/Dispatcher, when a node fails, or when a node is added, the NC is responsible for shuffling the data around so that the data placement scheme is intact. For example, if an augmentation to the collection is requested by
Figure 5.1: The RSML components and their interaction with each other and the application. This illustrates how node $N - 1$ would look like, where $N$ is the number of nodes in the system.
the Update Marshaller/Dispatcher, it will divide this data among the NCs, taking into account node balancing. Each NC will then place its portion of data on its local primary storage, and then replicate and divide its portion on each of the remote secondaries, again with node balancing in mind. In the event a remote node has potentially failed, the NC will activate an application on its node to manage the secondary of the failed node.

Another duty of the NC is to manage the virtual disk of the system, which is the disk interface available to each copy of the application running on the node. Operations include creating a new or removing an old virtual disk and resizing one or more virtual disks. For example, during an addition, a new virtual disk will be needed for the secondary of this new node. However, the virtual disks of the existing secondaries no longer need to be as big and can be shrunk.

Furthermore, the NC is responsible for communicating with the local GSM to accomplish any modification the GSM may have initiated. This involves performing the necessary operations to make the modification, indicating its ability to commit it, and then waiting for the final outcome of the modification. The NC takes its cue from the GSM and does not attempt any updates, node additions, or node removals unless the GSM indicates so.

### 5.1.2 Application Update Proxy

Every secondary and primary on a node has an associated Application Update Proxy which results in \( N \) AUPs per node, where \( N \) is the number of nodes. Each AUP connects to its corresponding NC on the remote node and receives data update instructions from that NC. For example the secondary AUP of node 2 that resides on node 0 would establish a connection to the NC on node 2. However, the primary AUP connects to the NC on the local node. The AUP is responsible for receiving data update requests from its NC and then relaying them to the application. From the point of view of the application, the AUP acts as the Update Marshaller/Dispatcher.
The need for the AUP is to ensure the environment of the application is as simple as possible (as in Figure 1.2). Each NC may need to communicate with each of its remote secondaries during a modification. This does not happen in an unmoderated way. One of the goals of the RSML is to provide a framework such that the database application itself could be written without knowledge of the distributed nature. Unfortunately, there is important information about the secondaries that cannot be stored in the NC itself (since it cannot afford to be lost in case of a node failure). To remedy this problem, all communication between a NC and a remote copy of the application takes place through the Application Update Proxy, which filters the requests passed from the NC in such a way that the distributed nature of the system is completely transparent to the application itself.

5.1.3 Disk Management Layer

The DML is a simple layer which interfaces with both the local NC and all copies of the application being run on the node. Each application is given a virtual disk on which to store all its data, which is then mapped, by the DML onto the physical addresses of the disk. The number of virtual disks and their sizes can also be changed by the NC, as the requirements of the system change with the number of nodes.

5.2 Other RSML and NML Applications

The interface of RSML with the application is designed to be simple so that an application writer only needs to worry about two external operations: updating data on its virtual disk and replying to query requests. The internal operations of adding or removing data on the virtual disk follow a well-specified interface with the DML. RSML's interface is well-suited for search-and-retrieve applications such as Digital Libraries (DL) and Web Servers (WS). RSML is already designed to integrate completely with these applications.
to provide the fault-tolerant management of data.

However, there are applications that are very similar to IR systems but have other requirements on top of those specified by the RSML. An example is Multimedia Servers (MS) which need to store and retrieve data; however, it also has a timing element. A constant rate of data is required for MS which the RSML lacks. Extensions to RSML may include a time element to handle this constant rate of data flow. In addition, a MS typically needs to service one (typically large) multimedia file at a time that, under RSML, would be stored in one member's primary. Thus, a MS could only make use of the fault-tolerant storage of RSML, but could not realize performance benefits of load-balancing using RSML. RSML divides all the data across the members and ensures its replication; however, it does not divide an individual file across the members. An extension to RSML may include dividing all individual files, or at least large files, to accommodate applications such as MS. Further investigation is still needed to determine the performance impact of such extensions.

At the NML level, it can also be used to handle the system management of a cluster. Potential failures are reported and facilities are present to remove and add members to the cluster. Individual nodes can make use of the cluster state information to make time-critical decisions if needed. The NML also has facilities for degraded systems to limit the possible modifications to those that will allow the system to return to a stable system. In addition, atomic update facilities are useful for any distributed system. The use of the GSM by the RSML for TCP/IP reads and writes is another simple example. Clients can use the facilities to determine if any member has potentially failed and take the necessary actions.
5.3 Handling More Than One Failure Per Recovery Period

The NML described a service that was capable of facilitating one failure ($f = 1$) every recovery period. This was due to the RSML’s data placement scheme which is only tolerant to one failure. However, another storage management system may attempt a solution that may be tolerant to two failures, or even $k$ failures. Such an aggressive storage management system might be required since a recovery period may vary and could be potentially long. In addition, there may be applications in general that require the handling of potentially $k$ failures during a recovery period. A more general solution would thus be needed to handle $k$ failures ($f = k$) every recovery period, where $k < \frac{n}{2}$. This generalization would be able to manage any level of fault-tolerance placed by the application.

5.3.1 A Framework for a $k$-Failure Model

A $k$-failure model can be an augmentation to the presented model. The $k$-NML model would still provide an early detection mechanism of potential failures for the local client applications, facilities to allow members to join the system, perhaps more than one at a time, and facilities to allow the members to perform an atomic operation. However, the extension the $k$-NML would provide over the original NML is the facility to allow the removal of up to $k$ members for every recovery period.

5.3.2 An Extension of the Crown Prince Concept

The concepts of a leader and a crown prince will need to be extended to include up to $k+1$ possible initiators, since up to $k$ members can fail. This can be denoted $\text{leader}_0, \ldots, \text{leader}_k, \text{leader}_{k+1}$. $\text{leader}_0$ can make any modifications while $\text{leader}_i$, where $1 \leq i \leq k + 1$, would only have limited modification abilities. As with the current implementation, unique
identification numbers can be used to accomplish this ordering.

The introduction of more potential initiators requires a more rigorous treatment of the initiation procedures. Only leader_0 is permitted to initiate additions or updates. Thus the Steady predicate does not need to be modified. However, the Failed_t predicate must be changed to handle a set of up to \( k \) members: \( \text{Failed}_t(R) \) where \( |R| \leq f_i \).

If leader_0 has \( f = k \), it can remove up to \( k \) members. In a system where \( f < k \), then the members are still able to sustain as many as \( f \) failures. Thus, removals are also permitted in systems where \( f < k \) as long as \( f > 0 \). In general, a member \( i \) is allowed to initiate the removal of members \( R \) if:

1. \( \text{Failed}_t(R) \)
2. \( \{\text{leader}_j\} \subseteq R \) where \( 0 \leq j < i \)
3. \( |R| \leq f_i \).

Thus, a member \( i \) is allowed to initiate removals as long as after the removal, this would result in \( i \) becoming the leader_0.

5.3.3 Expanding the States and Transitions

The NML had two states, \( \langle \text{Stable}, \ast \rangle \) and \( \langle \text{Degraded}, \ast \rangle \), to make the distinction between \( f = 1 \) and \( f = 0 \), respectively. A \( k \)-NML model would need to be more general.

An extension of the states is necessary. The original NML had three voting states: \( \langle m, \text{Abort} \rangle \), \( \langle m, \text{Scommit} \rangle \), and \( \langle m, \text{Dcommit} \rangle \) to indicate if the member could abort the operation, commit the operation, or commit the operation but resulting in no more sustainable failures, respectively. The client application is responsible for this initial decision. In a \( k \)-failure model, these states need to be augmented. For example, if \( k = 3 \), it could be the case that the removal of two members resulted in a system where the surviving members could only sustain one more failure (\( f = 1 \)). A voting state would be necessary to indicate this type of degraded vote. One proposal is to
extend the \( (m, S\text{commit}) \) and \( (m, D\text{commit}) \) into \( k + 1 \) possible states: \( (m, \text{Commit})_0 \),

\[
... \ (m, \text{Commit})_k
\]

If a member transitions to \( (m, \text{Commit})_l \), where \( l = k \), this indicates a vote for a stable commit. If \( l < k \), this would indicate a form of degraded commit. Again, the value of \( l \) would need to be indicated by the client application. Also, state transitions are needed to ensure that all members commit to only one form of the resulting systems. However, a hierarchy can be presented \( ((m, \text{Commit})_k \rightarrow (m, \text{Commit})_{k-1} \rightarrow ... \rightarrow (m, \text{Commit})_0) \) to ensure consistency. In order to be general enough, these voting states would be needed not only for removals, but also for additions and updates. The need for these voting states in additions and updates are discussed later.

Furthermore, \( Alive \) states corresponding to voting states are necessary as well. The state \( (\text{Degraded}, \ast) \) could be augmented to indicate how degraded the member is. For example, if \( f = 1 < k \), then this might be indicated by \( (\text{Degraded}, 1) \). If \( f = k \) this would just be \( (\text{Stable}, \ast) \). It should be noted that in degraded systems where \( f \neq 0 \), any modification may still be permitted. Again, it is up to the client application to indicate the failure level \( f \) of the resulting system during a modification. For example, if \( k = 3 \) and \( f = 1 \), the addition of one member may not bring the system back to being \( (\text{Stable}, \ast) \). However, the client application may indicate to the GSM that the resulting system is able to sustain one more failure \( (f = 2) \). This illustrates the need for the voting states in additions.

The general idea is that for any modification, whether it be an addition, removal, or update, it is up to the client application to indicate how many failures are sustainable in the resulting system. The GSM members commit to the modification with the smallest number of sustainable failures (or abort) because the system as a whole would only be able to sustain as many failures as the node with the smallest number of sustainable failures.

This discussion is only a sketch of a \( k \)-failure model. A more rigorous treatment is necessary for a complete model and is left as future work.
5.4 Summary and Contributions

The design of a Reliable Storage Management Layer contains many challenges for providing fault-tolerance for information retrieval systems. One of these challenges involves managing the network and communication issues. The Network Management Layer describes the services and related facilities needed by the RSML to provide one-failure fault-tolerance to IR applications. The main services are group membership and atomic commitment facilities to handle incremental system modifications. These modifications must be conducted in conjunction with the RSML application. The Global State Machine describes the design and implementation of such a facility.

An important aspect of the NML is that it interacts with the application when making any type of modification. The client applications that are targeted require their own operations before the modification can be committed. The two levels of timeouts allow for early detection of potential failures as well as the removal of members. The first level timeout allows time-critical operations to be performed to compensate for potential failures. The second level timeout is used for removals. The Failed\(_i\)(r) predicate is based on the second level timeout to infer a hard crash. A member does not unilaterally attempt to remove another member based just on local information, but also information from other members. Finally, the degraded state allows for only certain operations to be performed to allow the system to be brought back to a normal (stable) state. These features of the NML provide the foundation for developing fault-tolerant distributed IR applications.
Bibliography


