GOSSIP-BASED FAILURE DETECTION AND CONSENSUS FOR TERASCALE COMPUTING

By

RAJAGOPAL SUBRAMANIYAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

ACKNOWLEDGMENTS

First and foremost, I thank Dr. George for having confidence in me and for allowing me to pursue this research. I also thank him for his guidance and encouragement throughout this work.

I thank Matt Radlinski for being a very co-operative leader. He provided many technical suggestions and also helped polish my writing. I especially thank Pirabhu Raman, labmate and good friend, for his support, encouragement, and technical assistance.

I express my gratitude to my parents, who have been with me during the ups and downs of my life. I especially thank my mother for her moral support throughout my career. I would be nowhere without her.

I thank my roommates Kishore, Ninja and Arun for being friendly and supportive and for helping me to graduate successfully. I also thank my friends Charles, Venkat, KB, Rama, Rajesh, Muthu, Sridhar and others for making life at UF memorable.

I also thank my colleagues at the HCS Lab for their support and peer reviews. Special thanks go to Burt, Julie, Ian and Kyu Sang for being friendly and entertaining. Finally, I thank all the sponsors of this work. This work was supported by Sandia National Labs on contract LG-9271; and by equipment grants from Nortel Networks, Intel, and Dell.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

GOSSIP-BASED FAILURE DETECTION AND CONSENSUS FOR TERASCALE COMPUTING

By

Rajagopal Subramaniyan

May 2003

Chair: Alan D. George
Department: Electrical and Computer Engineering

One promising avenue of research on failure detection for large systems is the use of gossiping and other epidemic techniques to disseminate availability information. Gossip protocols and services provide a way to detect failures in large, distributed systems in an asynchronous manner without the limitations associated with reliable multicasting for group communications. However, the performance of these failure-detection protocols in terms of metrics like correctness, completeness and scalability need to be improved for use in terascale clusters. The consensus algorithm to detect failures in a system-wide fashion should be resilient enough to detect both individual node and group failures. Gossiping with consensus can take place throughout the system via a flat structure, or it can be hierarchically distributed across cooperating layers of nodes. This thesis presents a scalable multilayered gossip protocol, with features to strengthen the completeness and correctness of the failure-detection service. We examine the performance of the layered gossip protocol in terms of scalability on an experimental

testbed. Also we develop analytical formulae to project performance beyond our testbed and to determine optimal configurations for systems of arbitrary sizes. Finally, we advance a node-insertion scheme to support dynamic logical reconfiguration of the gossip service and include timing results that establish the efficiency of the method.

CHAPTER 1
INTRODUCTION

Today, high-performance computers and applications are key to scientific and technological development. The claim has been made that "all science is computer science," as evidenced by the tremendous increase in the need for high-speed, high-performance computing resources. Clusters, ensembles of computers maintained as a single unified resource, cater to the processing demands of such high-performance applications. Clusters built as a network of workstations from commercial off-the-shelf components exhibit a simplicity and cost effectiveness their conventional supercomputing brethren lack and, as such, have seen their popularity soar in the recent past. Regardless, the heterogeneity and rapidly increasing size of such clusters exacerbates the chore of maintaining them. As the system size increases, so increases the likelihood of failures and the difficulty in monitoring and managing the resources.

A number of challenges in the field of scalable, fault-tolerant computing must be overcome for heterogeneous, high-performance clusters to achieve their full potential. Distributed applications require a reliable, fast and scalable low-level failure-detection service. Speed is critical, as very low failure-detection times minimize the effect of failures on the system and enable quick recovery from faults with strategies like checkpointing and process migration. However, minimizing failure-detection time is a nontrivial issue, as a system-wide consensus on failures must be reached in a scalable

fashion. As such, scalability constraints limit failure-detection services based on traditional group communication mechanisms.

Gossip-style failure detectors exploit the inherent advantages of gossip communication. Gossip protocols are very responsive, have no single point of failure, and are more efficient than classical group-communication methods. Research involving random gossiping to disseminate liveness information [1-4] has demonstrated high-speed, low-overhead dissemination of system state information.

Gossip protocols form a potent platform to build highly responsive failure-detection services. This thesis presents a reliable and scalable gossip-style failure-detection service, designed as an extension of the simple hierarchical approach developed at the University of Florida by Sistla et al. [4]. The service is capable of serving systems of arbitrarily large size including terascale clusters. We experimentally analyzed and mathematically modeled the service to provide performance projections for very large systems beyond the scope of our testbed. A node-insertion mechanism that improves dynamic scalability further enhances the service. We attempt to demonstrate a near-ideal service in terms of correctness, completeness and scalability.

The remainder of this thesis is organized as follows. Chapter 2 provides background on failure models, performance metrics, and gossip-style failure-detection services. Chapter 3 discusses the shortcomings of previous designs in terms of these performance metrics. We developed a multilayered design, suitable for terascale clusters that addresses solutions to these shortcomings; it is experimentally analyzed in Chapter 4. Chapter 5 presents the node-insertion mechanism that enables dynamic reconfiguration of

the gossip service. Chapter 6 offers conclusions and suggests directions for future research.

CHAPTER 2
FAILURES AND FAILURE DETECTION SERVICES

A failure can be defined as a deviation of a service without conforming to the given system specifications. Many researchers have investigated various failure models to comprehensively classify failures in computing systems and occasionally suggest solutions to specific types of failures. Such classifications contribute to our understanding of the fundamental nature of failures and the design of fault-tolerant systems.

2.1 Failure Classifications

Bondavalli and Simoncini [5] classified failures from a detection standpoint. The following is a brief classification of failures relevant to the failure-detection service discussed in this thesis.

Halting failure: Halting failures occur whenever a process required to produce messages or change certain variables ceases to do so on a permanent basis, leading to irreversible changes. Such failures are also termed crash failures. If a halting failure is reversible, and a faulty process may be repaired and restarted, it is dubbed a napping failure.

Value failure: A failure in the value domain is known as a value failure and is classified as follows based on the type and intensity of the failure:

- **Omission value:** Failure to provide a required output for an input.
- **Coarse incorrect value:** An output different from NIL is produced by the system and detected as incorrect by the user based on knowledge of system specifications.

- **Subtle incorrect value:** Similar to coarse incorrect failures, but undetectable by the user.

Timing failure: A failure that causes the system to provide a correct output, but at a different time than expected. Based on the time when the output is issued, timing failures can be classified as follows:

- **Early:** Output delivered from the system earlier than expected.
- **Late:** Output delivered from the system later than expected.

Transient failure: Brief failure in any part of the system (e.g., node, link, etc.) causing temporary effects on the global state of the system in an arbitrary way. The effect of the agent inducing the failure is not perceived thereafter.

Byzantine failure: The weakest of all failure models, Byzantine failures correspond to completely arbitrary failure patterns. Failure could be in any part of the system and could even affect the functioning of the entire system.

2.2 Performance Metrics of Failure-Detection Services

Failures that could affect the normal functioning of the failure-detection service were classified in Chapter 1. The design of the service should be resilient enough to overcome such failures and perform well. It would be appropriate to consider a few performance metrics that help characterize the reliability of the service and its ability to scale.

Completeness: The number of nonfaulty members that suspect faulty members in the system. Completeness, like many other problems in distributed systems, can be reduced to the consensus problem. The issue of consensus is discussed in detail in the latter part of this chapter.

Correctness: Correctness can be defined informally as the ratio of the number of actual failures to the total number of failures detected by the failure detection service. The correctness measure is always < 100% with the assumption that the service when not functioning correctly, declares properly functioning entities to have failed. The reliability of a service is highly dependent on its correctness measure.

Scalability: Measure of the ability of the failure detection service to detect failures in the system while consuming minimal system resources irrespective of the system size. Modern distributed systems may span thousands or even tens of thousands of nodes; scalability is critical to the success of any failure detection service.

The failure detection service should be high performance-wise on these metrics and should not succumb to any of the failures discussed earlier. Discussed in terms of these performance metrics, research has shown that gossip-based protocols are more efficient than group communication and individual ping-based detection services [6 – 9]. Gossip-style failure detectors provide a fully distributed solution without the limitations of any central hotspot or single point of failure.

2.3 Gossip-Style Failure Detection

Early on, gossip concepts were primarily used for consistency management of replicated databases, reliable broadcast and multicast operations. Van Renesse et al. [1] first investigated gossiping for failure detection at Cornell University. In their paper, they present three protocols: a basic protocol, a hierarchical protocol and a protocol that deals with arbitrary host failures and partitions. Researchers at the University of Florida extended the preliminary work at Cornell to build a full-fledged failure detection service. Burns et al. performed high-fidelity, CAD-based modeling and simulation of various gossip protocols to demonstrate the strengths and weaknesses of different approaches.

Ranganathan et al. introduced several efficient communication patterns and simulated their performance [3]. Also, Ranganathan and colleagues proposed a distributed consensus algorithm which formed the basis for the experimental analysis of various flat and hierarchical designs of gossip-based failure detection by Sistla et al. [4]. Sistla and colleagues' design [4] is the latest and most efficient design of a gossip-style failure detection service. It forms the basis of this work as well, which is an extension of Sistla and colleagues' work [4] to provide a better, near-ideal service. The remainder of this chapter discusses the basic design structures developed by Ranganathan et al. [3] and Sistla et al. [4], which are used and extended in the design proposed in our work.

Three key parameters involved in failure detection and consensus are the gossip time, the cleanup time, and the consensus time. Gossip time, or T_{gossip}, is the time interval between two consecutive gossip messages. Cleanup time, or $T_{cleanup}$, is the interval between the time liveness information was last received for a particular node and the time it is suspected to have failed. That is, if node 1 receives no fresh liveness information about node 2 in $T_{cleanup}$ time, then node 1 will suspect that node 2 has failed. Finally, consensus time, or $T_{consensus}$, is the time interval after which consensus is reached about a failed node. The first two are input parameters configured for a particular gossip-based failure detection system. The cleanup time is some multiple of the gossip time, and the time required for information to reach all other nodes sets a lower bound for $T_{cleanup}$. When gossip time and cleanup time are relatively small, the system responds more quickly to changes in node status. When they are relatively large, response is slower but resource utilization decreases. The third parameter, $T_{consensus}$, is a performance metric determining how quickly failures are detected and consensus is reached.

2.3.1 Flat Gossiping

The important data structures maintained in each node of the system are the gossip list, suspect vector, suspect matrix and live list. The gossip list is a vector containing the number of T_{gossip} intervals since the last heartbeat received for each node. The suspect vector's i^{th} element is set to '1' if node i is suspected to have failed, otherwise it is set to '0.' The suspect vectors of all n nodes in the system together form a suspect matrix of size $n \times n$. Finally, the live list is a vector maintaining the liveness information of all the nodes in the system.

Every T_{gossip}, a node chooses another node in the system at random and transmits a gossip message to it. A gossip message consists of the sender's gossip list and suspect matrix and various headers. The suspect matrix sent by node i has the suspect vector of node i as the i^{th} row. On receipt of a gossip message, the local suspect vector and suspect matrix are updated based on the heartbeat values provided by the gossip list. Low values in the gossip list imply recent communication.

Figure 2-1 illustrates how the data structures in a node are updated upon receipt of a gossip message from another node in the system. The $T_{cleanup}$ for the 4-node system has been set to a value of 20 (i.e., $20 \times T_{gossip}$). In its initial state, node 0 suspects nodes 2 and 3 to have failed, as the heartbeat entries in gossip list corresponding to nodes 2 and 3 indicate values greater than $T_{cleanup}$, the suspicion time. The entries corresponding to nodes 2 and 3 in the suspect list reflect the suspicion. Likewise, node 1 also suspects node 3 to have failed as indicated by a corresponding entry in the suspect list. Node 1 does not suspect node 2, as only five gossip intervals (lesser than $T_{cleanup}$) have elapsed since the receipt of a message from node 2. The entries in the suspect matrix of node 1

also indicate that node 2 suspects the failure of node 3, which would have been indicated by the gossip message from node 2 to node 1.

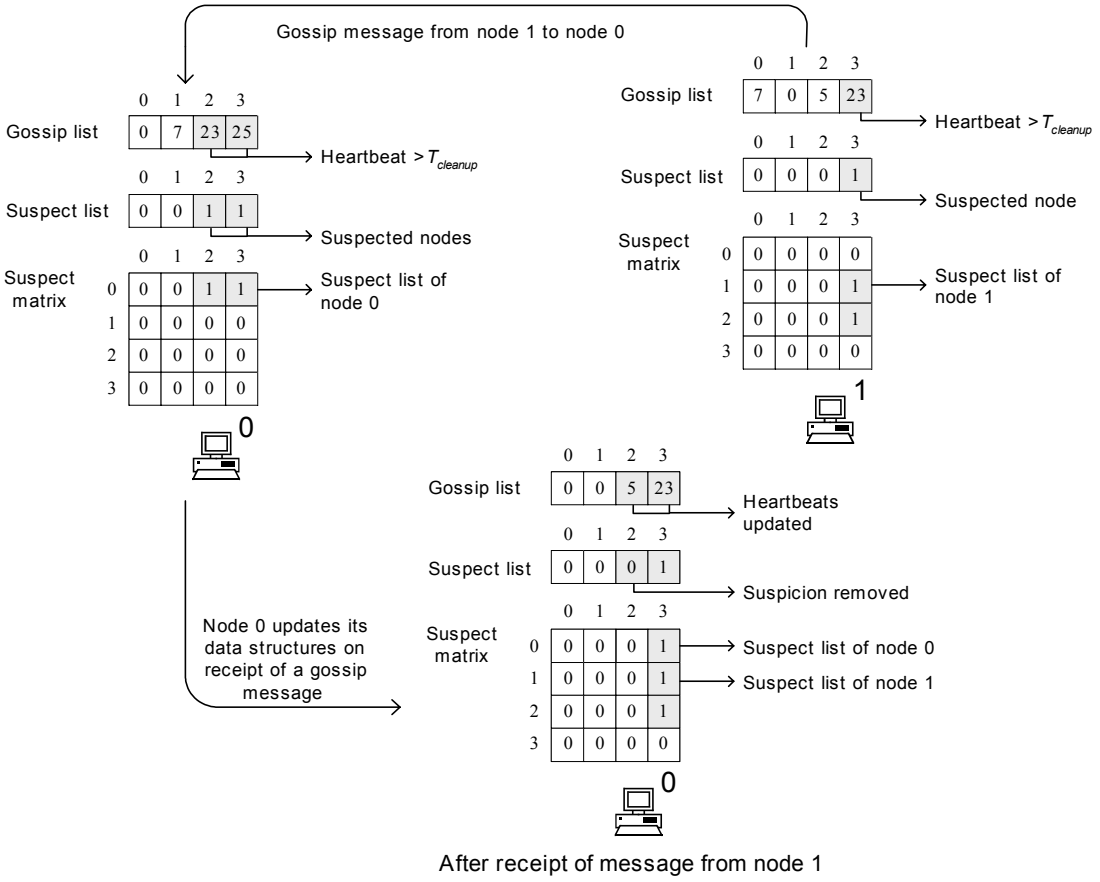


Figure 2-1. Data structure updates in a 4-node system

On receipt of a gossip message from a node, say node Y, node 0 compares the heartbeat entries in the received gossip list with those in the local gossip list. A smaller value in the received gossip list corresponding to a node, say node X, implies more recent communication by node X with node Y. Node 0 subsequently replaces the entry in its gossip list corresponding to node X with the value in the received gossip list. A larger value in the received gossip list corresponding to node X implies that node X has communicated with node 0 more recently than with node Y. Then, the entry in the local gossip list is not modified. In Figure 2-1, node 2 has communicated five gossip intervals

ago with node 1, and 23 gossip intervals ago with node 0. On receipt of a message from node 1, node 0 updates the gossip list to reflect the fact that node 2 was alive until five gossip intervals ago.

The suspect vector is next updated based on the modified gossip list. The suspicion entries are modified to reflect the present heartbeat values. In Figure 2-1, the suspicion on node 2 is removed, while node 3 is still suspected. The suspect matrix is next updated, based on the modifications done to the gossip list. A new smaller heartbeat value corresponding to any node, say node Z, implies that the received message has node Z's more recent perspective of the system. In Figure 2-1, heartbeat values of nodes 1, 2 and 3 have been changed, implying that node 1 has a more recent version of nodes 1, 2 and 3's perspective of the system than node 0. Subsequently, node 0 replaces the entries in the suspect matrix corresponding to nodes 1, 2 and 3 with those in the received suspect matrix. The entries corresponding to node 0 itself, are also modified to reflect node 0's present vision of the system as given by its new suspect vector.

2.3.2 Layered Gossiping

Scalability constraints limit flat gossiping. As the system size increases, so does the size of gossip messages. Failure detection time then increases, as the number of gossip cycles required to spread information throughout the system is very high. An immediate solution to the growing message size is to localize information exchange within a group of nodes by dividing the network into many logical groups. Nodes gossip frequently with other nodes in the same group while gossip messages between groups are passed less frequently. Nodes in each group take turns communicating group information to other groups. Gossip list and suspect list are maintained only for the nodes within a group. Separate gossip list and suspect list are maintained for the groups. The notation

'L#' will be used hereafter to denote a layer; 'L' abbreviating layer, followed by layer number. Figure 2-2 shows the communication pattern between L1 nodes and between L2 groups. L2 messages include the gossip list and suspect matrix of the groups. L1 messages include the gossip list and suspect matrix of the nodes within the same group in addition to the gossip list and suspect matrix of the groups.

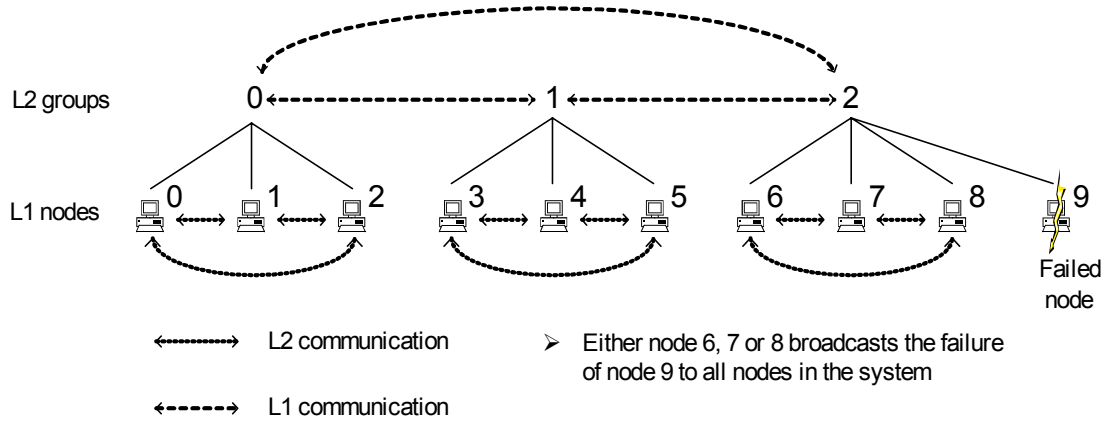


Figure 2-2. Communication structure in a two-layered system

2.4 Consensus

Primitive failure detection services work on basic timeout mechanisms. However, such services are vulnerable to network failure, delays and message losses. Gossip-style failure detection, though fully distributed, is also not immune to false failure detections especially with the random pattern of gossip messages. In order to obtain a consistent system view and prevent false failure detections, it is necessary for all the nodes in the system to come to a consensus on the status of a failed node.

2.4.1 Algorithm Design

Whenever a node suspects that any other node in the system may have failed, it checks the corresponding column of its suspect matrix to consult the opinions of other nodes about the suspected node. If a node is suspected by more than half the number of

live nodes (i.e., a majority of the live nodes), then the node is not included in the consensus. The opinion of the masked node is discarded. The majority check prevents false detections from affecting the correctness of the algorithm by ensuring that only faulty nodes are masked. Should all the other nodes agree with the suspicion, the suspected node is declared failed and the information is broadcasted to all the other nodes in the system. In the case of layered gossiping, the consensus is localized within a group and the failure is broadcasted to all the nodes in the system.

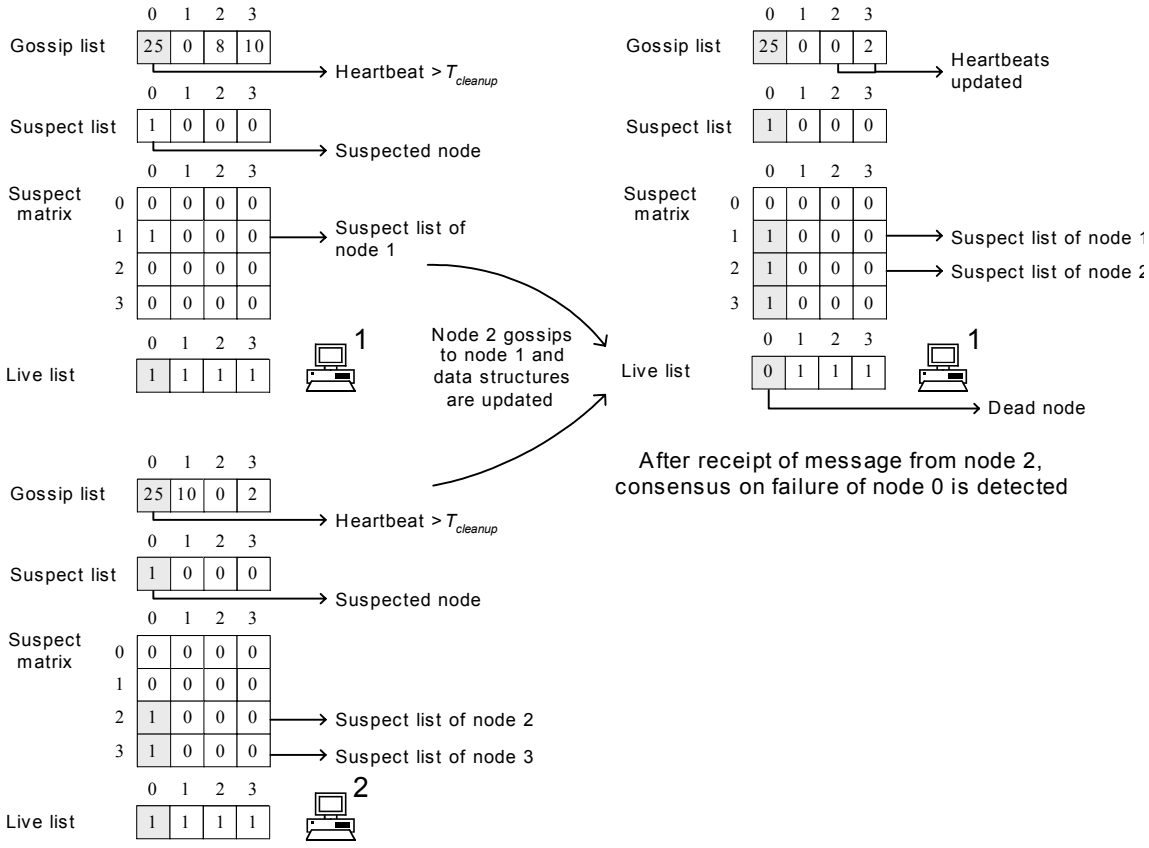


Figure 2-3. Consensus algorithm on a 4-node system

Figure 2-3 illustrates the working of the consensus algorithm in a 4-node system with one failed node, node 0. Node 2 sends a suspect matrix to node 1 indicating that it and node 3 suspect node 0 may have failed based in part upon earlier messages received from node 2-3. Node 1, which already suspects node 0, updates its suspect matrix on

receipt of a gossip message from node 2 and finds that every other node in the system suspects that node 0 has failed. The suspect matrix is updated as explained in the previous chapter. Thus, a consensus has been reached on the failure of node 0 and this information is broadcasted throughout the system. All the nodes subsequently update their live lists to indicate the status of node 0.

2.4.2 Theoretical Limits of Consensus

Fischer et al. theoretically demonstrated that distributed consensus is not possible even with one faulty process [10]. It might seem that the proof calls into question the validity of the consensus algorithm used in this failure detection service. However, it must be understood that the proof by Fischer et al. is based on the assumptions that timeout-based algorithms cannot be used and that no node can know about the status of any other node in the system (dead or alive). The assumptions are reasonable as the primary target of the proof is applications managing distributed databases, where nodes do not communicate with each other in between commit points.

Proofs on the impossibility of consensus by other researchers are also based on the same grounds and assumptions [11 – 13]. Ironically, the primary purpose of the consensus algorithm used in the gossip-style failure detection services discussed in the earlier chapters and in the design proposed in this work is to determine the status of other nodes. The nodes communicate with each other at regular intervals of time. When a faulty node stops communicating, other non-faulty nodes suspect the faulty node after a user-configurable cleanup time. The consensus algorithm is timeout-based, and every node in the system is aware of the status of the other nodes in terms of the timeouts. Since suspected nodes are not included in the consensus, only the non-faulty nodes are required to come to a consensus on the status of any node, and the “impossibility” of

distributed consensus may be discounted. It is again emphasized that the assumptions made for proving the impossibility of consensus are not valid for a gossip-style service and that the proper functioning of the consensus algorithm on a node failure under normal system conditions has been shown [3,4].

CHAPTER 3
PERFORMANCE DEFICIENCIES OF PREVIOUS DESIGNS

In this chapter, we address the pitfalls and shortcomings of previous designs, which motivated us to develop a new design. We discuss the resilience of the earlier designs to the failure models and their performance in terms of the metrics discussed in the previous chapter. Although Sistla et al. [4] experimentally demonstrated their service's efficient use of resources, it is far from ideal and fails to achieve high performance.

3.1 Correctness

Correctness describes the reliability of the service when the system suffers any kind of failure. The basic working of the gossip protocol overcomes value failures. The use of the consensus algorithm and maintenance of a consistent view of the system avoid incorrect value failures, which could occur when an application using the failure detection service queries the service to determine the system status. Omission value failures could occur only if node failures go undetected, and the failure is not intimated to the application. It has been demonstrated by Sistla et al. [4] that gossip detects all node failures under normal working conditions of the system, overcoming omission value failures.

Timing failures, if any, can be suppressed by a barrier synchronization embedded into the service. The barrier is called when the failure detection service is started, ensuring the reset of logical counters used in all the nodes to keep track of gossip cycles.

However, depending upon the speed of the network and the load on it, late timing failures can occur. This condition is the detection of a failure after a noticeable delay. Early timing failures, which would require the detection of a failure before it occurs, violate causality. Transient failures do not affect the service, as gossiping is totally independent of any particular message or link. Transient node failures are overcome by $T_{cleanup}$, the suspicion time factor. However, Byzantine failures such as permanent link failures and network partitions counteract the failure detection service. Simultaneous failures of nodes are also undetected by the service in some cases. The next two chapters discuss the effect of such Byzantine failures on the service.

3.1.1 Network Partitions and Link Failures

The requirement of a majority vote to mask faulty nodes from the consensus implies that the algorithm operates under the assumption that half or more of the nodes in the system will not experience a failure during a single consensus cycle. Figure 3-1 shows how a partition in the network could prevent the consensus algorithm from functioning properly and, hence, break the failure detection service. Figure 3-1 shows a flat system with four nodes partitioned due to a link failure into two parts, consisting of one and three nodes respectively. The nodes in the larger partition, nodes 1-3, will suspect node 0 after time $T_{cleanup}$. When three nodes suspect the fourth node has failed in a 4-node system, a consensus on the status of fourth node is appropriately reached. Node 0 is declared as failed by all the nodes in the system.

On the other side of the partition, node 0 starts to suspect node 1, node 2 and node 3. However, the suspect matrix in node 0 is not updated, as there is no communication across the partition. Due to a lack of majority, nodes 1, 2 and 3 cannot be masked from consensus. The consensus algorithm fails in node 0. All the other nodes are suspected

but still considered alive, and node 0 tries repeatedly and fails to communicate with these other nodes. The application is not terminated in node 0, leading to a waste of resources. The application may gracefully degrade in all the nodes in the larger partition, but might be inconsistent in the smaller partition.

The situation is worse yet when a link partitions a large number of nodes. The worst case would be when the system partitions exactly into groups of equal size. For example, consider a system with 60 nodes partitioned into two with 30 nodes in each partition. Successful consensus requires at least 31 nodes in a partition, to mask off rest of the nodes from participating in the consensus. Since neither partition has 31 nodes, there would be no successful consensus and the failure goes undeclared. The application continues inconsistently in every node, making no progress without communication between the nodes. A similar situation occurs when the system partitions into three or more partitions, with no partition having the required number of nodes for a successful consensus. Any later failure n the partition will not be detected either.

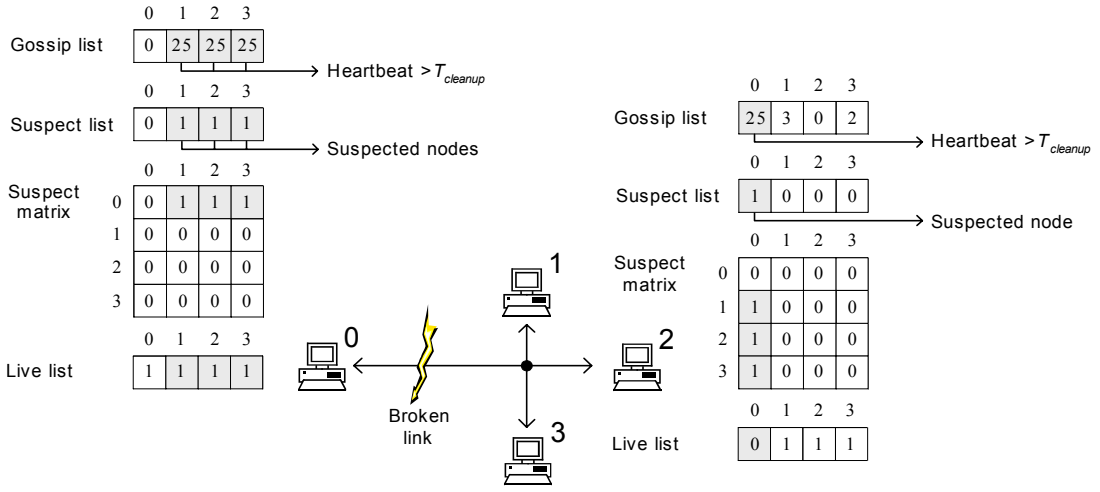


Figure 3-1. Flat system with four nodes, partitioned into 1 and 3 nodes

3.1.2 Simultaneous Node and Link Failures

Sequential failures of nodes are detected by the service without any complications. On a node failure, the system size, or the group size in the case of layered gossiping, is reduced to match the number of live nodes. However, simultaneous failures of nodes and links may lead the system to an inconsistent state in some cases. For example, consider the same system as shown in Figure 3-1 but without any link failure. When two nodes, say node 1 and node 2, fail at the same instance of time, then the other two nodes suspect them. But there will not be a consensus due to lack of majority (i.e., only two out of four) and failures go undeclared. This case is similar to a link failure partitioning the system into two even groups. In situations where the system is divided into many partitions because of multiple link failures occurring simultaneously, there are chances that all the partitioned groups lack majority. In such cases, the application may continue in an inconsistent state in each node.

3.2 Completeness

Chandra and Toueg state that in a system which exhibits strong completeness, "eventually every faulty participant is permanently suspected by every correct participant" [11]. The consensus algorithm requires that every correct participant properly suspects faulty nodes. However, in layered gossiping, only the nodes within a group are required to suspect a node for consensus. Thus, completeness in a layered failure detection service relies on how failures in a group are intimated to the rest of the system.

The previous design used a UDP broadcast informs all the nodes in the system when a node failure occurs within a group. But, UDP is an unreliable service, so the success of a UDP broadcast may not be guaranteed. As such, a few nodes in the system

may be totally unaware of the failure of a node and relevant fault-tolerance measures will be neglected. Inconsistent system state information results in an unreliable service which exhibits weak completeness.

The earlier designs do not address the failure of groups as well. Although the suspect matrix and gossip lists of groups, which would be required to detect a group failure, are included in second-layer communication, group failure is not considered. Undetected group failures again weaken the completeness of the service since, while live nodes in other groups will recognize that each node in a group has failed, they will be left unaware of the failure of the group as a whole.

3.3 Scalability

With the size of real-world systems marching towards tens of thousands of nodes, the scalability of a failure detection service plays a pivotal role in the development of high-performance distributed applications. A gossip service with two layers does provide improved scalability as compared to a flat service. However, considering the fact that the failure detection service does not provide any computational assistance to the application and merely consumes resources, the overhead imposed by the service should be meager.

The network utilization or bandwidth required per node in the two-layered service is on the order of one-hundred Kbps while CPU (Central Processing Unit) utilization reaches double-digit percentages, even for systems with a few-hundred nodes. These values represent per node use; the scalability, then, of a two-layered failure detection service is very limited.

CHAPTER 4
DESIGN OF MULTILAYERED FAILURE DETECTION AND CONSENSUS

Chapter 3 addressed the various shortcomings of the previous designs of gossip-style failure detection service. The service requires improved correctness, stronger completeness, enhanced scalability, and dynamic reconfiguration facilities. In this work, we propose a new multilayered design to provide solutions to the problems addressed in the previous chapter. This chapter discusses the design and development of the multilayered gossip-style failure detection service, which is intended to support systems with a very large number of nodes including terascale clusters.

Sistla et al. [4] improved scalability of a two-layered service over a flat service. In a flat service, the network bandwidth requirement per node scales as $O(n^2)$, exceeding 1 Mbps per node in a 96-node system. A two-layered service with restricted group size scales much better compared to the flat service. However, as the system size grows beyond a few hundred nodes, the resources used by the service increase tremendously, encumbering the application, network and system as a whole. In real-world systems with thousands of nodes, the gossip-style failure detection service would be unusable.

4.1 Multilayered Gossip Service

The multilayered design proposed here, a divide and conquer approach, extends the simple hierarchical design provided by Sistla et al. [4]. The system is divided into groups, which in turn are clubbed to form groups in different layers, with the number of layers dictated by performance requirements.

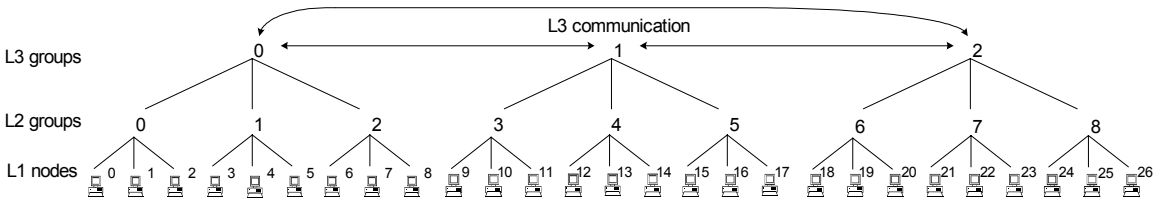


Figure 4-1. Sample multilayered system with 27 nodes and three layers

Figure 4-1 shows the communication structure and system configuration of the multilayered design in a sample system having 27 nodes divided into three layers. The nodes are divided into groups, with three nodes in each L2 group. L2 groups are grouped themselves to form L3 groups. The system shown in the figure has three L2 groups in each L3 group. The gossip messages sent between nodes in the same group include the gossip list and suspect matrix of the nodes in the group, as in a flat system, along with the data structures involving the groups in all the upper layers. In general, gossip message in layer k encompasses the information of the groups in all the layers above layer k. Consensus about the failure of a node is restricted to the L2 group in which the failure occurred and is propagated to the rest of the system by both broadcast and live list propagation, as described in Section 4.3.

4.2 Modified Consensus Algorithm

Byzantine failures like network partitions, link failures and simultaneous failures challenge the correctness of the service, as do the lack of facilities to detect group failures. The requirement of masking faulty nodes from the consensus is reasonable only in cases when no bizarre failures occur and the system is running smoothly, except for a few minimal failures. The consensus algorithm requires modifications to function correctly under all circumstances.

When a failure divides a system into two or more partitions, the partition with more than half the number of nodes in the entire system can always use the consensus algorithm successfully to detect the failure of the nodes in the other partitions. However, the nodes in smaller partitions cannot detect the failure of nodes in larger groups due to a lack of the majority required for consensus.

Timeout, the basis behind the idea of suspicion and failure detection, can be used to modify the consensus algorithm to overcome such Byzantine failures. Along with the normal procedure for detecting the failure of nodes, a user-definable timeout is used on the duration for which a node is suspected. For any node j suspected, the j^{th} column of the suspect matrix is repeatedly checked at periodic intervals to verify if the entries are updated. A change in any entry $[i, j]$ of the suspect matrix indicates communication with node j indirectly via node i. On the other hand, if the node is suspected even after the timeout period and none of the entries in the j^{th} column have been updated, a communication failure has likely occurred. Then the live list is updated to reflect the failure of communication with node j. Whenever there is a failure of any network link and henceforth a partition in the network, a few nodes might end up in the smaller partitions. Such nodes in the smaller partition use the enhanced consensus algorithm, to identify the other nodes in their partition without being affected by the requirement for a majority check.

Figure 4-2 illustrates the consensus algorithm modified to use timeouts. The figure depicts a system with five nodes, partitioned into two groups of three and two nodes. The group with three nodes (i.e., nodes 2, 3 and 4) will use the consensus algorithm as usual to detect the failure of the other two nodes (i.e., nodes 0 and 1). In the

other group, the majority requirement will not be satisfied and normal consensus will not work with just two nodes. However, the suspect matrix in node 1 will indicate the suspicion of the other three nodes, while the update check indicates no update. Node 0 will also indicate through its suspect vector that it has not received any updates for the three nodes in question. Thus node 0 and node 1 detect the network partition and consider the other nodes as failed. The same timeout mechanism is used by node 0, which detects that node 1 has broken communication with the other nodes in the system. Nodes 0 and 1 can now form a new logical system with size two, if their services are still required.

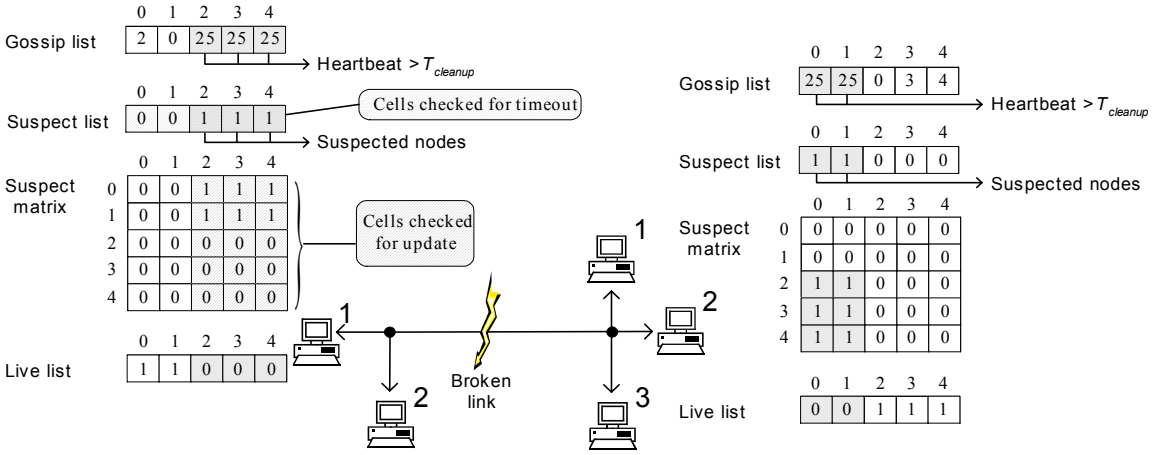


Figure 4-2. Flat 5-node system, with a broken link and using modified consensus algorithm

When many links fail simultaneously creating multiple partitions in the network, the consensus algorithm with timeout is used by all the nodes in the partitions that do not have the required number of nodes to use regular consensus. When the system is divided into two or more partitions of equal size, the timeout mechanism will identify the partition for all the nodes. The timeout-based algorithm is used to detect the partition of groups as well if a hierarchical failure detection service is used.

When a failure occurs, regardless of the nature of the failure, the modified algorithm helps the application using the service degrade gracefully or terminate, saving system resources. The deadlock resulting from a partition no longer occurs, and system-wide consistency is maintained.

Failures of groups could be due to a partition in the network, splitting the network into two or more components, or due to the individual failures of each node in a group. Bulk failures go undetected if group failures are not addressed, calling into question the correctness and completeness of the service.

The solution is to detect such failures with a group-level consensus in the layer corresponding to the failure. On a group failure, the other groups with the same parent group in the next higher layer as the failed group use the standard consensus algorithm to detect the failure. All the subgroups and nodes within the failed group are also declared failed. However, it should be understood that a group is considered dead, only when all the nodes in the group die. A group with even one live node is considered alive, since from the application's point of view, that one live node can share resources with the other groups. As an example of consensus on a group failure, in Figure 4-1, detection of failure of group 2 in L3 requires the consensus of L3 groups 0 and 1, which involves consensus of L2 groups 0-5, which requires the agreement of nodes 0 through 17.

4.3 Failure Notification

The completeness of the gossip-style failure detection service is primarily determined by the mechanism through which knowledge of a detected failure is disseminated to all the nodes of the system. Using the standard consensus algorithm, strong completeness is achieved within a group. However, in a hierarchical service, the completeness of the system is questionable. New methods to promulgate the liveness

information and detected failures throughout the system need to be designed to keep all the entire the system informed of all failures.

Since UDP broadcasts can be unreliable, an alternate method of intimating the failure of a node is necessary. One solution is to propagate the live list of a group by appending it to inter-group gossip messages. Every node in the system maintains the live list of the nodes within its own group, that is, a local live list. The local live list is generally consistent within a group, thanks to the consensus algorithm. The local live list is propagated to other groups and is also consistent irrespective of which node in the group sends the gossip message. On receipt of a gossip message from a different group, a node identifies the sender's group and updates the live list corresponding to the group. So the L2 messages will now include "(gossip list + suspect matrix) of the groups + live list of the group"**.** In the multilayered design, the first layer live list is appended to all higher layer gossip messages.

Propagating a failure message through transmission of the live list as described above takes much more time than a broadcast. However, live list transmission is done in addition *to* a broadcast, so only the nodes that missed the broadcast need to be updated. Thus the speed with which the nodes are updated is not an important factor here. The frequency with which second and higher layer gossip messages are transmitted may be the same as the frequency with which first layer gossip messages are transmitted, thus propagating the live list more frequently, with little increase in overhead. Ultimately, every node will know the status of every other node in the system, regardless of the reliability of UDP broadcasts, and the system will exhibit strong completeness.

4.4 Dynamic System Reconfiguration

The scalability and efficiency of any service is primarily dependent on the reconfiguration software, which is a part of the service. Likewise, the dynamic scalability of the gossip service depends on the ability to join a new node into the system on the fly. Presently, none of the previous designs of a gossip-style failure detection service support the reconfiguration of the system or the easy inclusion of a new node into the system. The addition of a new node requires a service restart, and subsequently the abrupt termination and restart of an application using the service. Thus, the service is unusable with applications which may require or benefit from an incremental allocation of resources.

Dynamic scalability of a service, the ability to expand the size of the system without a restart of the application, is largely dependent on the abilities and performance of the reconfiguration software embedded in the service. This requires the expansion of the failure detection service as well, to support the addition of new nodes into the system dynamically without any hindrance to failure detection. Support for node-insertion also enables the service to be easily used with load balancing and scheduling services, wherein new nodes frequently join and leave the system. The rest of this chapter discusses a new mechanism to insert nodes into the system during execution, followed by an analysis of insertion time to study the efficiency of the node-insertion mechanism.

4.4.1 Node Re-initialization

Node insertion can be coarsely defined as the addition of a new node into the cluster or inclusion of a node already part of a cluster into a group of nodes running a specific application. The node insertion mechanism could be used to add a new node into the system, or to re-initialize a node which failed, and has since recovered.

Re-initialization is faster compared to inserting a new node, as no data structures must be rebuilt.

To re-initialize a node, the gossip service must be restarted on the node after it has recovered. The node starts to communicate gossip messages with other nodes as usual. The node which receives the first gossip message from the restarted node is called the eternal node. The eternal node realizes from its live list that a failed node has come back to life again, and it takes the responsibility of broadcasting the information to all the other nodes in the system. On receipt of the broadcast, each node's gossip list is modified to indicate recent communication. Their suspect vectors and suspect matrices are modified to remove suspicion, and the live lists are changed to show the live status of the node. The broadcast avoids inconsistency in the system when two or more dead nodes come back alive simultaneously. The restarted node determines the status of all other nodes through normal gossiping.

4.4.2 Related Research

Re-initialization is relatively simple compared to the insertion of a new node, which would require many changes in the data structure. Many general node-insertion and reconfiguration mechanisms relevant to gossip-style service designed in the past were considered to provide deadlock-free and consistent node-insertion software. Approaches used in mobile adhoc network were also studied as they involve very frequent reconfiguration. We discuss the node-insertion mechanism proposed in Microsoft Cluster Service (MSCS) [14] because of its adaptability to suit gossip-style service with few modifications.

MSCS proposes a join algorithm involving six different phases. During the discovery phase, a search is made to identify a sponsor node to sponsor the join.

Concurrent threads are used for the search. The threads terminate on a timeout or the establishment of a connection. Phase 2 occurs when the sponsor acquires a global lock using a global update (GUP) method. However, the GUP method is highly dependent on the number of nodes in the system and a dynamically centralized lock manager. In Phase 3, "enable network," a series of RPC calls are made to the sponsor to retrieve system information. This is followed by the establishment of connections with active nodes, again by RPC calls. In Phase 4, the joiner petitions the sponsor for membership and the sponsor joins the node. In Phase 5, the joiner performs a database synchronization to synchronize its configuration database with the sponsor. The global lock is released in the Phase 6, and the overall time required to insert a node is on the order of ten seconds, with high dependency on system size.

4.4.3 Design of the Node-insertion Mechanism

An insertion mechanism designed for use with a failure detection service should have very low insertion time. Realizing an insertion time as short as possible and leaving the system uninterrupted is critical. The insertion schemes discussed in the previous section incur a very high insertion time and are not scalable for large systems. However, we have developed a fully distributed approach with low insertion times.

The joining algorithm involves six different phases. During Phase 1, the joiner node tries to identify a sponsor node. Multiple requests to sponsor may be done sequentially, should the first attempt fail due to timeout or a negative response. Steps 1 and 2 in Figure 4-3 form Phase 1. During Phase 2, the joiner requests the sponsor to allow it to join the sponsor's group and goes into a wait state. During Phase 3, the sponsor acquires a global lock to ensure that only a single join is in progress at any instant of time. The locking mechanism is inspired by the GUP-based lock provided by

MSCS [14]. One node in the system plays the role of a lock manager, with its identity known system wide. The sponsor node queries the lock manager for the lock, and proceeds to insert the new node after obtaining the lock. The lock manager reclaims the lock after the insertion of the new node. When multiple nodes attempt to the join the system simultaneously, the lock requests are maintained in a queue at the lock manager, and the lock is released to the sponsors in the order in which their requests were received. Should the lock manager fail, the next node in the system takes up the role, providing fault-tolerance. There could be speculations that the hot spot at the lock manager will limit the scalability of the mechanism. Considering the fact that node insertions are infrequent, and the chances of simultaneous insertions of large number of nodes which create the hot spot, is very low, the locking mechanism is scalable.

During Phase 4, the sponsor broadcasts the identity of the new node to all the nodes in the system. The various data structures (gossip list, suspect vectors and matrices) are modified in all the nodes to include the recent addition. The sponsor also takes the responsibility of modifying the shared configuration file to reflect the inclusion of the new node. Steps 5, 6 and 7 in the figure form Phase 4. During Phase 5, the sponsor sends current system information to the joiner and waits for the joiner's acknowledgement. On receipt of an acknowledgement, the sponsor broadcasts restart instructions to all nodes including the new node and releases the lock in Phase 6. Whenever a sponsor node fails during the process of insertion, another node in the system takes up the responsibility of sponsoring the join. This avoids deadlocks during the join. The choice of the new sponsor is made based on the order of nodes in the configuration

file, which avoids unnecessary contention and race conditions. The design has been tested to be robust under all critical conditions.

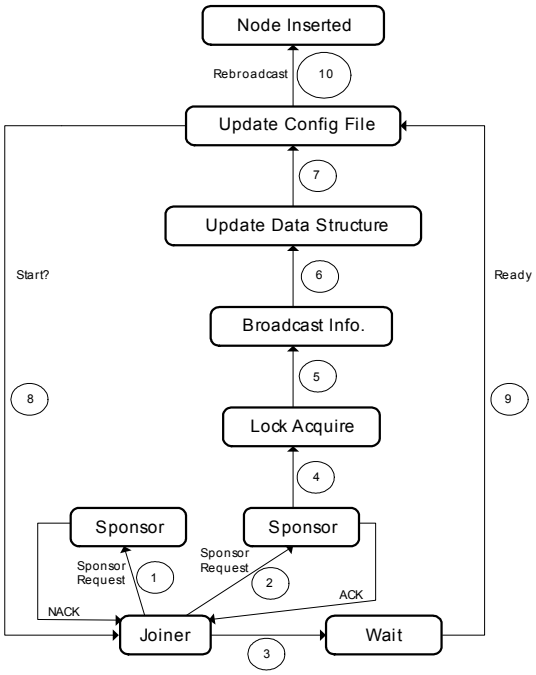


Figure 4-3. Sequence of steps for joining a new node

CHAPTER 5
EXPERIMENTS AND ANALYSIS

Sistla et al. experimentally demonstrated faster node failure detection in a two-layered rather than flat service [4]. Consensus/failure detection time was proved to be dependent only on group size. The smaller the group size, the faster the failure detection, since fewer nodes are required for consensus. However, the size of L2 gossip messages increases with the number of groups, so a tradeoff exists between overhead and failure detection time when dealing with two-level gossiping. Rather than increasing the number of groups, the number of layers may be increased, thereby mitigating this effect.

We experimented with the multilayered gossip service to verify its scalability, the overhead imposed on the system, and to determine failure detection speed for multilayered systems. Experiments were performed on the CARRIER cluster at the High-performance Computing and Simulation Research Laboratory at the University of Florida. The cluster is comprised of over 200 nodes featuring a control network of switched Fast Ethernet and a variety of high-speed data networks including SCI, Myrinet, ATM and Gigabit Ethernet.

5.1 Failure Detection Time Experiments

Figure 5-1 illustrates the dependence of failure detection/consensus time on the size of a group in a two-layered system. The figure validates the results provided in [4], as well as extends the experiments for larger systems sizes. For a given system, the best consensus time is achieved by setting the $T_{cleanup}$ parameter to the lowest possible value,

called optimal cleanup time [4], below which true consensus cannot be reached. Selecting a value below this minimum for $T_{cleanup}$ will increase false failure detections and make consensus impossible. For a fixed group size, consensus time is entirely independent of system size: large group sizes result in long consensus times, while small group sizes yield lower consensus times. However, group size cannot be too small for a larger system using two-layered gossiping due to resource utilization constraints. Using a multilayered design, the group size can be kept small without overhead constraints by increasing the number of layers. Thus a multilayered service can detect node failures in as little as 130ms irrespective of system size.

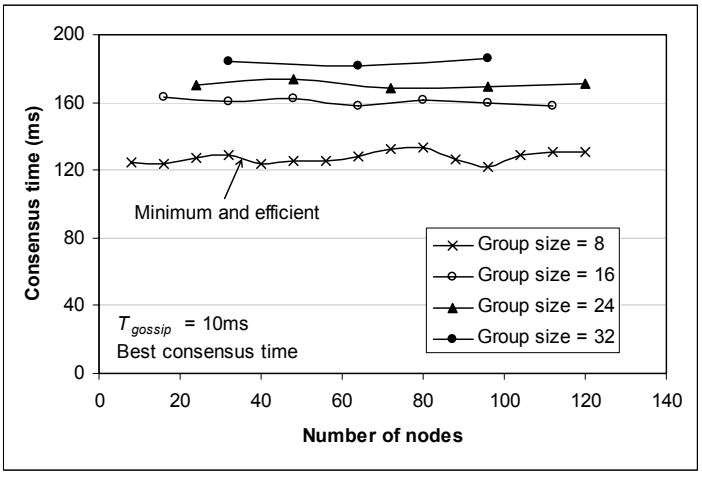


Figure 5-1. Detection time of node failures in a two-layered system

Failure detection of groups is similar to that of nodes, except that subgroups participate in consensus instead of nodes. Figure 5-2 compares the detection time of a L2 group in two-layered and three-layered systems of the same size. Increasing the number of layers can keep detection time for group failures low. With the L2 group size fixed at four, the number of L2 groups must increase with system size. This increases the detection time of a group failure in a two-layered system, as more groups participate in consensus. In the case of a three-layered system with a L3 group size of three, every four

L2 groups form a L3 group. When a L2 group fails, only three other groups must come to a consensus, speeding up the process.

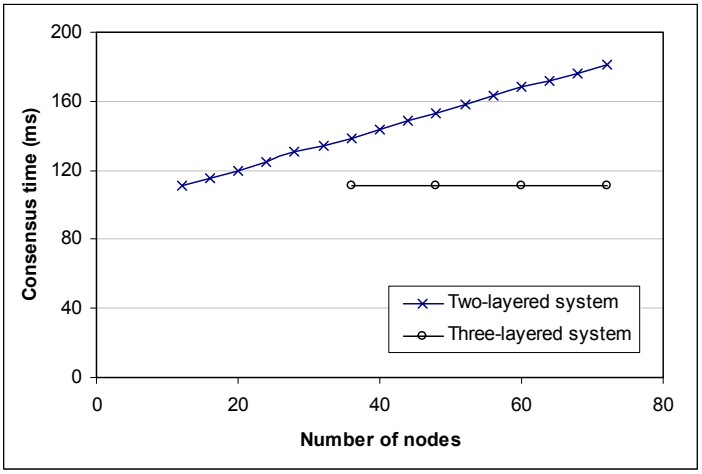


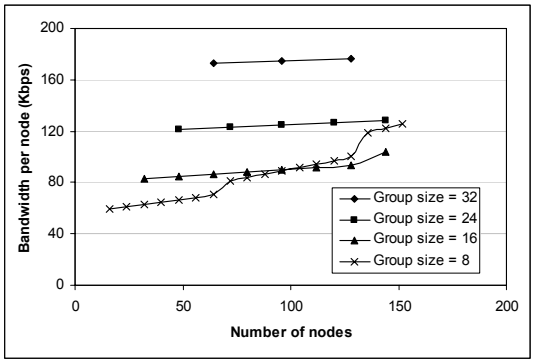
Figure 5-2. Comparison of detection time of group failures in layered systems

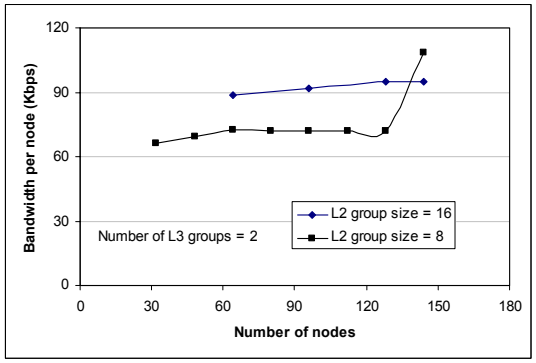
5.2 Network Utilization Experiments

What are the performance costs of using this service, and is this service scalable? We measured the required network bandwidth using a packet-capturing tool called Ethereal available under the GNU public license.

Figure 5-3 A shows the variation of bandwidth requirement per node for various group sizes. With a fixed L1 group size, an increase in system size increases the number of groups, increasing the L2 component of network utilization while the L1 component remains constant. The L1 network utilization component dominates the L2 component for small systems, while network use due to L2 traffic stands out for larger systems. With larger group sizes, the L2 component remains small, even for larger systems. Note the sharp increase in bandwidth whenever the number of groups (system size ÷ group size) crosses a multiple of eight. Steep increases can be seen when system size moves from 64 to 72 and 128 to 136 for a group size of eight, and 128 to 144 for a group size of 16. Since gossip data is a sequence of bits, and packets are transmitted as bytes, an entire

additional byte is required whenever the number of bits required crosses a multiple of eight.

A 



B

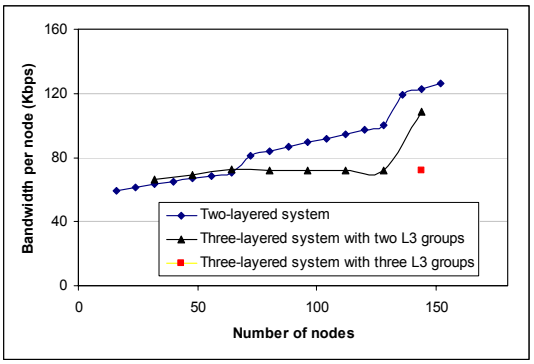
Figure 5-3. Variation of bandwidth requirement per node with group size. A) In a two-layered system. B) In a three-layered system.

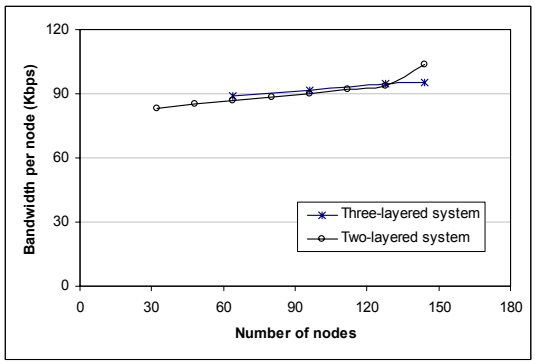
Figure 5-3B shows the same behavior in a three-layered system for various L2 group sizes. When the number of L3 groups is fixed at two, each L3 group contains half the L2 groups. For example, with a group size of eight, a system with 128 nodes will have 16 L2 groups and two L3 groups, each of size eight. In a three-layered system, the L2 component also remains constant for a fixed L3 group size. When the system size increases from 128 to 144, while the L2 group size remains fixed at eight, the L3 group size increases from eight to nine with a steep rise in required bandwidth.

Figure 5-4 shows the improved scalability of the three-layered system over the two-layered system, justifying the development of a layered service supporting any number of layers. In Figure 5-4A, the bandwidth requirement per node is compared for two and three-layered systems with L2 group size fixed at eight. For smaller systems ($<64 = 8\times8$, here), additional layers increase bandwidth due to unnecessary additional communication in the higher layers. For larger systems (>64 here), bandwidth required per node in a three-layered system is less than that in a two-layered system, since fewer L2 groups communicate frequently. Reducing the size of a L3 group further reduces the

bandwidth, resulting in an even greater improvement over a two-layered system. Figure 5-4B illustrates the situation when the L2 group size fixed at 16. Here, the three-layered system performs better than the two-layered system for system sizes greater than 128, when the number of L2 groups increases beyond eight. However, the efficiency of the multilayered service is evident from network utilization measurements for very large system sizes.

A





B

Figure 5-4. Comparison of bandwidth requirement per node in layered systems. A) L2 group size = 8. B) L2 group size = 16.

5.3 Processor Utilization Experiments

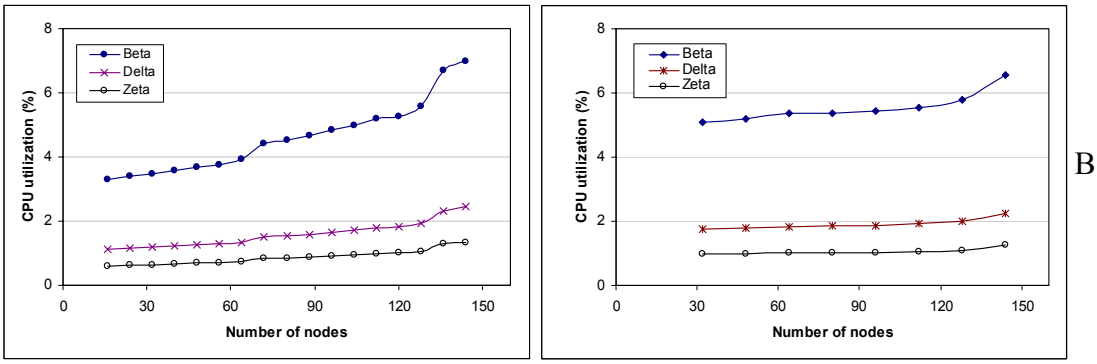
CPU utilization of the service is computed by measuring the number of CPU cycles consumed per Tgossip. Experiments were conducted in three different node architectures, each with a different CPU configuration. The three node architectures are summarized in Table 5-1.

Table 5-1. Summary of node architectures

Name	Configuration
Beta	400 MHz Intel Pentium-II processor
Delta	600 MHz Intel Pentium-III processor
Zeta	733 MHz Intel Pentium-III processor

Figure 5-5 is a verification of the results reported in [4], with the newly developed multilayered service. L2 group size is fixed at eight in Figure 5-5A. Significant CPU

utilization is experienced with two-layered gossiping for larger systems with slower nodes (beta). A steep increase in CPU utilization can be seen when system size crosses a multiple of 64 (8×8) and the number of L2 groups crosses a multiple of eight. L2 group size is fixed at 16 in Figure 5-5B. For system sizes greater than 128, a system with group size 16 has lower CPU utilization than a system with group size eight. A higher increase in CPU utilization can be seen when system size crosses a multiple of 128 (16×8). L2 CPU utilization varies as $O(g^2)$, where g is the number of groups.

A 

B

Figure 5-5. CPU utilization of two-layered system. A) L2 group size = 8. B) L2 group size=16.

CPU utilization is insignificant with three-layered gossiping even for larger systems with slower nodes (delta). An increase in CPU utilization is reduced by introducing additional layers or by using an optimum group size in the third layer, restricting the total number of layers to three. The CPU utilization does not vary much for smaller systems as demonstrated in Figure 5-6, with three-layered gossiping. With L2 group size fixed at 16, we expect an increase in CPU utilization will be significant only when system size goes beyond 1024 (8×8×16) nodes.

Scalability in terms of CPU utilization makes a strong case for multi-layering. Figure 5-7 compares the CPU utilization of two and three-layered systems for L2 group sizes of eight and 16. For a 144-node system, three-layered gossiping requires only

about 50% of the CPU utilization required by two-layered gossiping with a L2 group size of eight. For a larger L2 group size, improvement in CPU utilization is highly pronounced for very large systems. However with L2 group size fixed at 16, the crossover can be seen when system size crosses 128 and the number of L2 increases to nine.

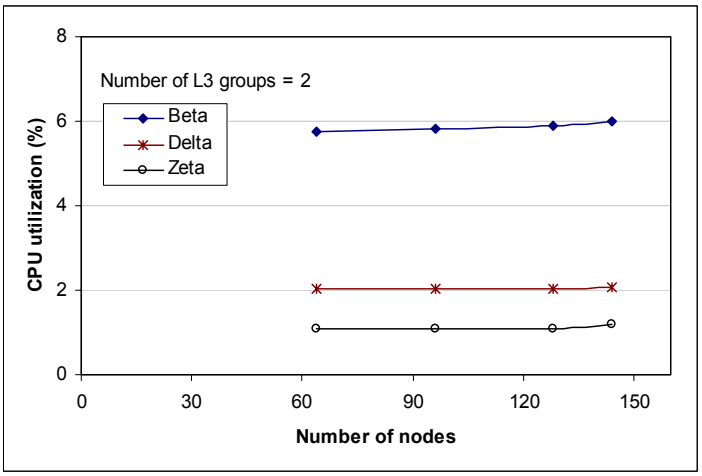


Figure 5-6. CPU utilization of three-layered system.

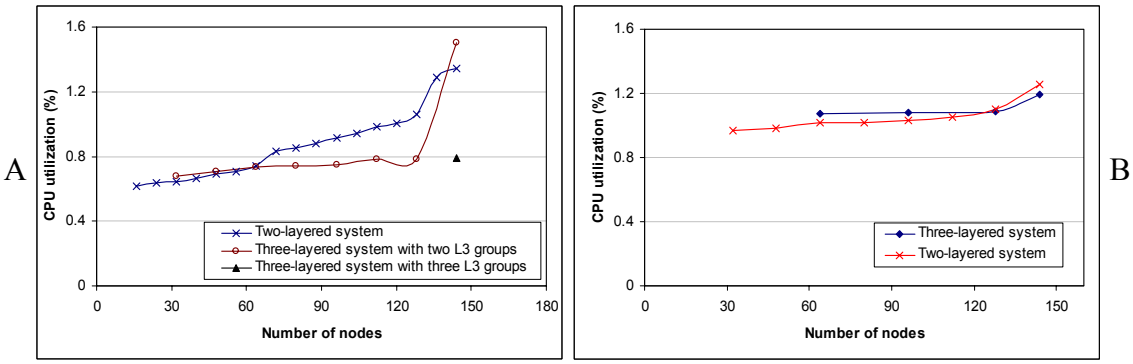
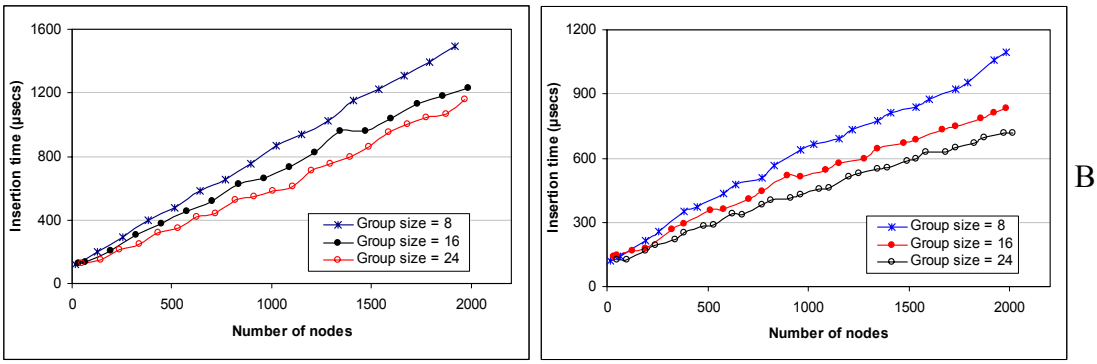


Figure 5-7. Comparison of CPU utilization in two and three-layered systems. A) L2 group size = 8. B) L2 group size = 16.

5.4 Node-Insertion: Timing Results and Analysis

During the insertion of a node, gossip communication is stopped between nodes to avoid inconsistency, when a few nodes in the system will use the newly modified

configuration file, while others use the older unmodified version of the configuration file. If a failure occurs during this time, it will be detected and reported throughout the system only after communication resumes. Thus minimizing this communication stoppage time, which depends on node-insertion time is critical to maintain faster failure detection. The time to insert a new node was measured under various setups to demonstrate the efficiency of the mechanism designed.

A 

B

Figure 5-8. Node-insertion time for various system sizes and group sizes. A) Node inserted into first group of the system. B) Node inserted into last group of the system.

Figure 5-8 gives the time to insert a new node in a system with two layers for various group sizes. The time to insert a node is the time from when the sponsor got the request to insert the node to the moment when the communication is restarted after the node is inserted. The insertion time, thus is basically the time taken at the sponsor node to execute the function for inserting a node. Due to the limitations in the size of the testbed, the insertion time is measured by simulating the insertion function. The time taken to execute the function is measured by changing the input parameters like system size and group size. The measurements were made on the zeta nodes, where each point in the plot represents an average of 10 different trials. Small noise-like small jitters in the pattern are attributed to the very small magnitude of the insertion time.

Insertion times are in the microseconds even for very large systems as compared to other similar schemes addressed earlier, where insertion times are on the order of tens of seconds. We observe that insertion times decrease with an increase in group size, because the number of groups decreases as the group size increases. Figure 5-8B illustrates that the pattern followed by the insertion time is the same irrespective of the group in which the node is inserted.

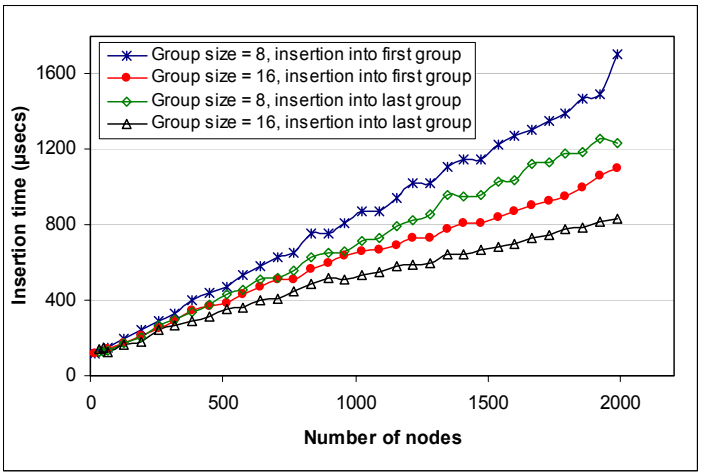


Figure 5-9. Comparison of insertion of a node in the first and last group of a system

Figure 5-9 compares the time to insert a node in the first and last group of a system for various group sizes. Changes in data structure and memory reallocation involved with global lists take up a major part of the insertion time, as do those related to groups when the number of groups outnumber the group size for large systems. A node inserted into the first group requires large modifications to the group structures and global lists like the global live list and global name list, increasing the insertion time. Inserting a node into the last group would require only minor changes, and only in the group structure.

5.5 Optimizing the Multilayered Gossip Service

This section presents an analytical model of the failure detection service enabling resource utilization projections for system sizes not available in the testbed. The projections and analytical model are subsequently used to determine the optimum configuration and setup of the service for use in large, real-world systems.

5.5.1 Analytical Formula

We have extended the formulae provided by Sistla et al. [4] to model a service using an arbitrary number of layers. The bandwidth requirement per node is a function of the gossip message size, including a gossip header of 4 bytes, gossip list and suspect matrix, Ethernet UDP overhead of 42 bytes, and the message transmission frequency. In an l-layered system, g_l is the number of groups in the l^{th} layer.

B : bandwidth utilization per node	g_k : group size in layer k
L_j : length of j^{th}-layer gossip packet	l : number of layers
f_i : frequency of i^{th}-layer gossip message	

$$B = \sum_{i=1}^{l} L_i \times f_i \tag{5-1}$$

$$L_1 = 46 - (l+1) + 1 + \sum_{k=1}^{l} \left(g_k + 1\right)\left(\left\lceil \frac{g_k}{8} \right\rceil + 1\right) \tag{5-2}$$

Without live list propagation for intimation of failure:

$$L_j = 46 - \left(l+1\right) + j + \sum_{k=j}^{l} \left(g_k + 1\right)\left(\left\lceil \frac{g_k}{8} \right\rceil + 1\right); j \neq 1 \tag{5-3}$$

With live list propagation for intimation of failure:

$$L_j = 46 - \left(l+1\right) + j + \left(g_1 + 1\right)\left(\left\lceil \frac{g_1}{8} \right\rceil + 1\right) + \sum_{k=j}^{l} \left(g_k + 1\right)\left(\left\lceil \frac{g_k}{8} \right\rceil + 1\right) \tag{5-4}$$

5.5.2 Overhead Analysis for Live List Propagation

With the addition of live list propagation, the gossip-style failure detection service exhibits stronger completeness, but additional data must be transmitted resulting in additional bandwidth consumption. We can determine the additional overhead required for this modification with a simple formula, which determines the bandwidth requirements from the size of the packets and the frequency of their transmission. The percent increase in bandwidth resulting from the addition of the live list to a gossip packet is given as

$$B_{increase} = \frac{(g_1+1)\times\left(\left\lceil\frac{g_1}{8}\right\rceil+1\right)}{g_1\times\left\{44+(g_1+1)\times\left(\left\lceil\frac{g_1}{8}\right\rceil+1\right)+(g+1)\left(\left\lceil\frac{g}{8}\right\rceil+1\right)\right\}+\left\{45+(g+1)\left(\left\lceil\frac{g}{8}\right\rceil+1\right)\right\}} \quad (5\text{-}5)$$

where g_1 is the group size, and g is the number of groups.

For example, the overhead required for a 12,000-node system due to the additional live list is approximately 100 bps per node, which is reasonable for such a large system. Generally, the group size g_1 is 8 or 16 nodes, as previous research has found small group sizes result in lower overhead and an efficient use of resources. When the group size is kept small, any additional network or CPU overhead is very low. With this addition, the gossip-style failure detection service exhibits strong completeness, and the additional cost is miniscule.

5.5.3 Optimizing System Configuration

The formula to calculate the bandwidth can always be used to optimally configure the service for a given system size. The number of layers and the group size of each layer required to achieve minimum bandwidth utilization may be calculated for any system size. However, this requires the formula be differentiated to find the minima, but

an equation with a ceiling function is discrete and not easily differentiable. As such, a generalized formula which produces an optimum system configuration is not possible. Using Matlab, we calculated the optimum group size given the system size and number of layers by calculating the bandwidth required for each possible group size and selecting the value which results in minimum bandwidth utilization. By analyzing patterns within the equation, we developed generalizations for calculating the group size which results in minimum bandwidth. Figure 5-10 illustrates the generalized formula.

'L' layered system with 'n' nodes
- g_1 = next higher multiple of 8 of ('L^{th}' root of system size)
- g (for all other layers) = 'L-1'th root of $n \div g_1$

Figure 5-10. Generalized formula for calculating group size giving the approximate minimum bandwidth

As an example, for a system with 812 nodes and four layers, g1 should be eight and g2 and g3 should be five for minimum bandwidth utilization. However, the generalization requires a small modification for two-layered systems alone, wherein g1 should be the closest multiple of 8 of the square root of system size, instead of the next higher multiple.

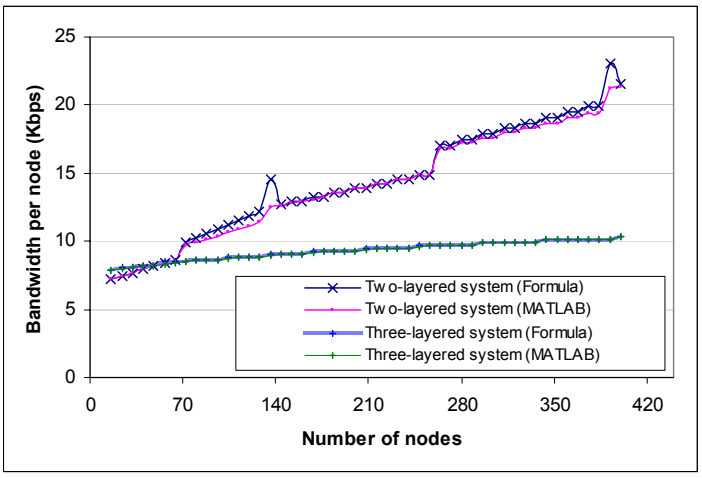


Figure 5-11. Verification of generalized formula for minimum bandwidth utilization

Figure 5-11 shows the bandwidth requirement per node calculated for a system configuration based on both the generalizations and actual minima calculated using the Matlab. The results validate the generalization, as bandwidth calculations based on generalized group sizes closely match the actual possible minima. The figure also illustrates the bandwidth overhead for two-layered and three-layered services for small systems with fewer than 400 nodes. For systems with 64 or greater nodes, a three-layered service is better than a two-layered service, based on bandwidth overhead.

The generalized formula may be used to determine the bandwidth requirements per node for very large systems with various numbers of layers. Figure 5-12 illustrates the minimum bandwidth for systems up to 6,000 nodes with different numbers of layers. This may be used to determine the optimum configuration to achieve the best performance with minimum cost in terms of resource utilization. Table 5-2 gives configurations for minimum bandwidth utilization for various system sizes.

Table 5-2. Optimum system configuration for minimum resource utilization

System size		Number of layers	Group size in each layer
Lower bound	**Upper bound**		
8	63	2	8
64	511	3	8
512	4,095	4	8
4,096	32,767	5	8

Projections also demonstrate improvement in network utilization provided by a multilayered gossip service. For example, the minimum possible bandwidth consumption per node in a 6,000-node system is 175 Kbps with a two-layered service while it is just 11 Kbps with a five-layered service.

The CPU utilization measurements follow the same pattern as network utilization measurements, but the development of an analytical formula for CPU utilization is

impractical. Regardless, it stands to reason that CPU utilization should scale with network utilization, as the CPU must perform work whenever a gossip packet is transmitted or received. Therefore, minimum network utilization breeds minimum CPU utilization, and the general rules of Figure 5-10 applied to 5-4 yield minimum overall resource utilization.

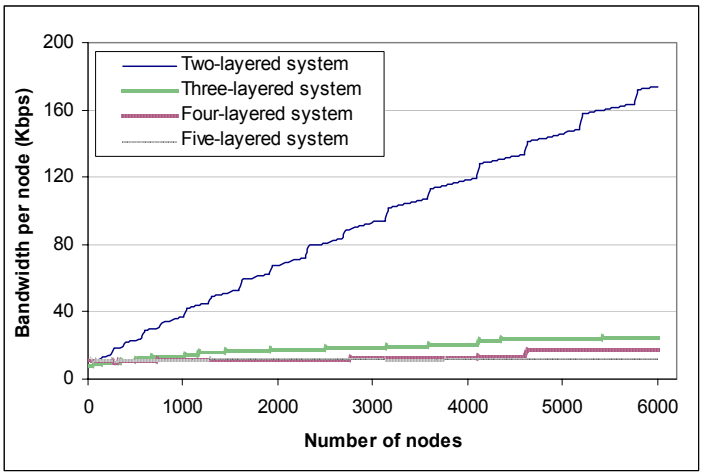


Figure 5-12. Projection of minimum bandwidth requirement per node for large systems

The multilayered design developed here, has been demonstrated to be scalable to systems of any arbitrarily large size. The design also addressed solutions to overcome all failures hindering the correct working of the service and provided new mechanisms to improve the completeness of the service. This new design, thus attempts to take gossip-style failure detection to a higher level of correctness, completeness and scalability, moving it closer to ideality.

CHAPTER 6
CONCLUSIONS

In this thesis, we have developed an efficient and scalable gossip-style failure detection service to address the needs of high-performance applications, high availability compute centers. We have made several extensions, resulting in a more complete, correct, and scalable service than previous efforts. We have performed a comprehensive analysis with a full implementation of the service on a cluster testbed of 160 nodes. The additional overhead incurred due to the improvements provided has been analytically investigated, and confirmed to be negligible. We have compared the performance of the multilayered gossip service to the earlier design supporting only two layers in terms of resource utilization and failure detection times. Performance projections and optimal system configuration have been presented through an analytical model to promote efficient use of the service in real-world systems of any arbitrary size including terascale clusters. Finally, a new node-insertion mechanism has been embedded into the service to enable dynamic system reconfiguration. The efficiency of the node-insertion mechanism has been analyzed based on experiments measuring node-insertion timings.

Increasing the number of layers with system size increases the efficiency of the service. Optimum use of the service is achieved when a flat service is used for systems with fewer than eight nodes, a two-layered service is used for systems with fewer than 64 nodes, a three-layered service for those with fewer than 512 nodes, a four-layered for those with fewer than 4,096 nodes, and a five-layered for those with fewer than 32,768

nodes. Failure detection time can be as low as 130ms for systems of any size with a group size of eight. Bandwidth used by the service can be kept as low as 11 Kbps even for systems as large as 25,000 nodes with a five-layered service.

The modified consensus algorithm with timeouts can detect any type of failure in the system, which, along with the changes in the failure announcement methods, highly improves the reliability of the service. The node-insertion mechanism improves dynamic scalability, and provides insertion times in microseconds even for large systems. The time to insert a new node into a system with 1,000 nodes and group size of eight is just 625 μs.

REFERENCES

1. R. Van Renesse, R. Minsky, and M. Hayden, "A Gossip-style Failure Detection Service," *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing Middleware*, England, September 15-18, 1998, pp. 55-70.

2. M. Burns, A. George, and B. Wallace, "Simulative Performance Analysis of Gossip Failure Detection for Scalable Distributed Systems," *Cluster Computing*, Vol. 2, No. 3, 1999, pp. 207-217.

3. S. Ranganathan, A. George, R. Todd, and M. Chidester, "Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters," *Cluster Computing*, Vol. 4, No. 3, July 2001, pp.197-209.

4. K. Sistla, A. George, R. Todd and R. Tilak, "Performance Analysis of Flat and Layered Gossip Services for Failure Detection and Consensus in Scalable Heterogeneous Clusters," *Proc. of IEEE Heterogeneous Computing Workshop at IPDPS*, San Francisco, CA, April 23-27, 2001.

5. A. Bondavalli and L. Simoncini, "Failure Classification with respect to Detection," *Proc. 2nd Workshop on Future Trends of Distributed Computing Systems in the 90s*, IEEE, Cairo, September 1990, pp. 47-53.

6. F. Christian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Information and Computation*, Vol. 118, No. 1, April 1995, pp.158-179.

7. M. Lin, K. Marzullo, and M. Stefano, "Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks," *UCSD Technical Report* TR CS99-0637.

8. M.J. Lin and K. Marzullo, "Directional Gossip: Gossip in a Wide-Area Network," *Third European Dependable Computing Conference*, Springer-Verlag, Berlin, Sept. 1999, Vol. 1667 of Lect. Notes Comput. Science, pp. 364—379.

9. R. Van Renesse, "Scalable and Secure Resource Location," *Proc. of the IEEE Hawaii International Conference on System Sciences*, Maui, Hawaii, January 4-7, 2000.

10. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, Vol. 32, No. 2, April 1985, pp.374–382.

11. T.D. Chandra, S. Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems,” *J. ACM*, Vol. 43, No. 6, March 1996, pp.225-267.

12. T. D. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost, “On the Impossibility of Group Membership,”*Proc. Of the 15th Annual ACM Symp, on Principles of Distributed Computing (PODC96)*, New York, USA, 1996, pp.322-330.

13. C. Dwork and N. Lynch, “Consensus in the Presence of Partial Synchrony,” *Journal of ACM*, April 1998, Vol. 35, No. 2, pp. 288–323.

14. W. Vogels, D. Dumitriu, A. Agarwal, T. Chia, and K. Guo, “Scalability of Microsoft Cluster Service,” *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, August 3-4, 1998.

BIOGRAPHICAL SKETCH

Rajagopal Subramaniyan was born in Tanjore, India. He completed his schooling at Carmel Garden matriculation higher secondary school. He went on for his bachelor's degree in electronics and communication engineering with a scholarship at Anna University, Chennai, which is one of the most prestigious engineering institutions in India. He successfully completed his bachelor's degree with distinction and joined University of Florida to pursue his master's in electrical and computer engineering. In Spring 2001, he joined the High-performance Computation and Simulation (HCS) Research Laboratory as a graduate research assistant under the guidance of Dr. Alan George. His research at the HCS Research Lab forms the basis of this work. He completed his master's degree in May 2003 and plans to pursue his Ph.D. at University of Florida.